

# PROJECT INTERNSHIP REPORT

Dated :- 31-12-2024

## Project Details

Project Name	Technology
Resume Parser	Python, MongoDB, Node.js , React.js, HTML and CSS



**Chola**  
*Enter a better life*

Prepared By	Aravind A
	Jeffrey G
	Rajkamal S
Project Coordinator	Choudam Venkata Sivaiah
Mentored By	Suresh S
	A G Sriram
Reviewed BY	Santhi Swaroop Mohanty

## TABLE OF CONTENTS

Section	Title		Pg no
	<b>Abstract</b>		2
1	<b>Resume Parser</b>		3
	1.1	Introduction	3
	1.2	Objectives	3
	1.3	Tech Stack	4
	1.4	Implementation Details	4
		1.4.1 Workflow Details	4
		1.4.2 Dependencies Details	8
		1.4.3 Frontend Details	10
		1.4.4 Resume Parser Details	19
		1.4.5 Backend Details	31
		1.4.6 Resume Matcher Details	41
		1.4.7 Database Details	47
	1.5	Conclusion	49

## ABSTRACT

The Project Report is for the Resume Parser Project which aims to automate the Resume Shortlisting process which in turn eases the Job recruitment process. The Key Components include The Resume Parser, which extracts the relevant information such as skills, location, contact details using various Python Modules and Stores all the extracted information in the NoSQL based database management software “MongoDB”. The web application allows the user to upload the resumes files (in DOCX and PDF formats only) either individual files or as a bulk folder using API calls via Flask. The web application allows the user to create Job Roles with input fields which allow the user to enter the Job Description using API calls via Flask. We are able to match these Job Roles with the skills extracted in the MongoDB and these files are fetched and Displayed in a table format using Node module – AG Grid. These processes fast track the resume shortlisting process and improves operational efficiency

# 1. Resume Parser

## 1.1 INTRODUCTION

The Resume Parser Project has been developed to assist in the employee recruitment process using a web application which has been developed in Flask API, a high-performance web framework for building APIs in Python and it utilizes MongoDB as the backend for storing the user and the extracted information. The web application is accessible via the frontend developed using React.js and Node.js module- A G Grid which is a user friendly and easy to read Node module used to display extracted information in a table format. The user will Login to the application and will be able to create Job roles based on the requirements and match those requirements to the extracted information from the resumes which are stored in the database. These Job roles will be displayed as Job cards which can be used to access and will show the resumes which match the description provided . The web application is also used to upload the resumes and process them to extract the information from those resumes and store them in the database as various fields.

## 1.2 OBJECTIVES

- **Extract Required Information:** To extract the necessary information from the list of resumes given as input and store them in the database
- **Create Job Roles:** Allow users to create job roles based on the description entered into the respective fields
- **Resume Matching:** Match the extracted information of the resumes with the description of the job role and store them in the database for ease of access
- **View Job Roles:** Allow the user to view the resumes which have been matched to the given job role
- **Filters:** Allow the user to apply filters on the Job roles
- **View Logs:** Allow user to see the logs to know the status of the process and whether the resumes have been processed or not
- **Edit and Delete Job Roles:** Allow the user to edit the job roles based on the change in description as well as delete them once it has no more use
- **Create User:** Allow the admin to create more user profiles on demand

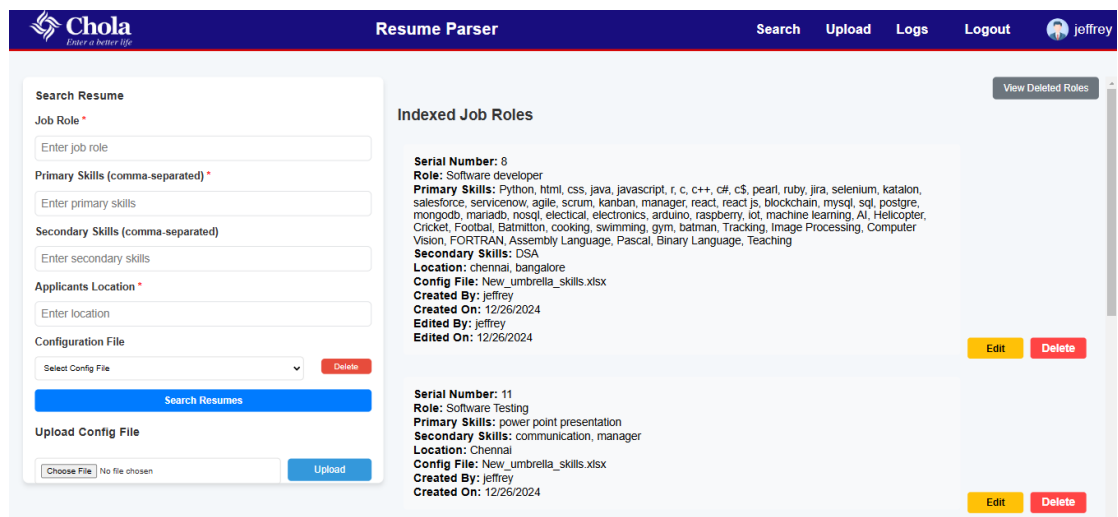
## 1.3 TECH STACK

- **Framework** : Flask 2.3.2
- **Backend** : Python 3.12
- **Frontend** : React.js 18.3.1  
: Node.js v22.9.0
- **Database** : Mongo DB 8.0.0

## 1.4 Implementation Details

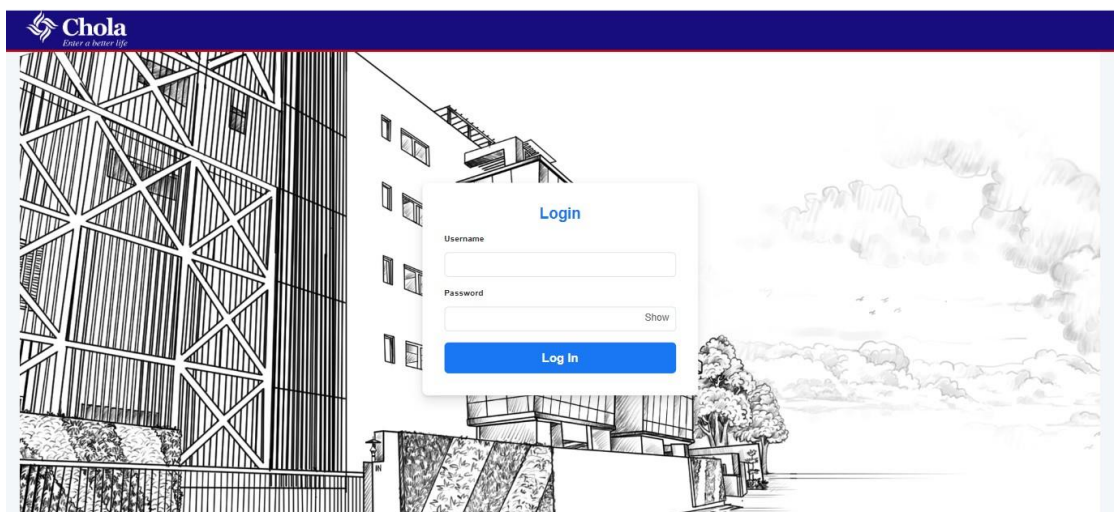
### 1.4.1 Workflow Details

1. We start in the login page of the web application and the login credentials should be entered



The screenshot shows the 'Resume Parser' application interface. The header includes the 'Chola' logo, navigation links (Search, Upload, Logs, Logout), and a user profile for 'jeffrey'. The main content area is divided into two sections. On the left, the 'Search Resume' section contains input fields for 'Job Role', 'Primary Skills (comma-separated)', 'Secondary Skills (comma-separated)', and 'Applicants Location', along with a 'Configuration File' dropdown and a 'Search Resumes' button. On the right, the 'Indexed Job Roles' section displays two job roles with their details: Serial Number, Role, Primary Skills, Secondary Skills, Location, Config File, Created By, Created On, Edited By, and Edited On. Each job role entry has 'Edit' and 'Delete' buttons. A 'View Deleted Roles' button is also present in the top right corner.

2. If the login credentials are given for the user it will land on the search job roles page



- Here we type in the required primary skills, secondary skills, the job role and the location of the posting of the job role

The screenshot shows the Chola Resume Parser application interface. The top navigation bar includes the Chola logo, the title 'Resume Parser', and links for Search, Upload, Logs, Create User, Logout, and a user profile icon labeled 'admin'. The main content area is divided into two sections. On the left, the 'Search Resume' form contains fields for Job Role (Software Testing), Primary Skills (Jira, selenium), Secondary Skills (javascript, python), Applicants Location (chennai), and Configuration File (a dropdown menu). Below these fields are buttons for 'Search Resumes' and 'Upload Config File'. On the right, the 'Indexed Job Roles' section displays a list of roles. The first role is 'Software Testing' with details: Serial Number: 1, Role: Software Testing, Primary Skills: javascript, c++, Secondary Skills: communication, team player, Location: Bangalore, Config File: New\_umbrella\_skills1.xlsx, Created By: admin, Created On: 12/24/2024, Reverted By: admin, Reverted On: 12/26/2024. The second role is 'Software Engineer' with details: Serial Number: 2, Role: Software Engineer, Primary Skills: wireframe, Secondary Skills: CSS, JAVA, Location: Bangalore, Config File: New\_umbrella\_skills1.xlsx, Created By: admin, Created On: 12/24/2024, Edited By: admin, Edited On: 12/24/2024. Each role entry has 'Edit' and 'Delete' buttons.

- Then we upload and select the configuration file which holds all the umbrella terms of the provided primary and secondary skills, the uploaded configuration files will get saved.

This screenshot shows the same Chola Resume Parser application interface as the previous one, but with the 'Configuration File' dropdown menu open. The dropdown menu lists several files: 'umbrella\_skills.xlsx', 'New\_umbrella\_skills.xlsx', 'New\_umbrella\_skills1.xlsx', 'UM.xlsx', 'skills.xlsx', and 'skills1.xlsx'. The 'Search Resume' form fields are now filled with: Job Role (editor), Primary Skills (photoshop, adobe premier), Secondary Skills (figma), and Applicants Location (chennai). The 'Indexed Job Roles' section remains the same as in the previous screenshot.

- Then we submit the form to make the search and once the search is done the job role will be indexed and displayed on the indexed list
- We then can access the list of shortlisted resumes by clicking on the indexed job role
- Once the job role is opened, we can see that all the shortlisted resumes are showed in order of score given to them based on the count of the mention of skills in the resume

Chola Resume Parser									
Resumes for Software Testing									
Resume	FileName	Name	Matched Skills	Secondary Skills	Expanded Skills	Location	Email	Phone	
<a href="#">Open</a>	Resume 5.pdf	Resume	c++, javascript		web development	no 137 jenkins street,vg...	ig0832@srmist.edu.in		94
<a href="#">Open</a>	ELAYARAJAN_Resume...	ELAYARAJANResume	c++, javascript	communication	jquery	of resources, program b...	elayaraj11@gmail.com		80
<a href="#">Open</a>	Test_User.pdf	TestUser	c++, javascript	team player		plot no :119, 7th cross s...	agasthian_r@yahoo.com		95
<a href="#">Open</a>	Naukri_PAVITHRANR[4...	NaukriPAVITHRANR	c++, javascript	communication		, aes encryption.	vadugalpavi@gmail.com		98
<a href="#">Open</a>	MathiyazhaganP[6_0].pdf	MathiyazhaganP	c++, javascript	communication		tamil nadu, chennai, tn, in...	mathiyazhaganece@g...		99
<a href="#">Open</a>	Magesh5_11_3.docx	Magesh	c++, javascript		react		magesh1.1.1994@gmail...		94
<a href="#">Open</a>	FSD_MohammedAthar[...	FSDMohammedAthar	c++, javascript	team player	web development, jquery	new no 15 new street,	athu091@gmail.com		97
<a href="#">Open</a>	Ebenezer Daniel[3y_9m...	Ebenezer Daniel	c++, javascript	communication		collision assistance.			
<a href="#">Open</a>	DASWANI[3y_0m].docx	DASWANI	c++, javascript			iripadu, spsr nelloe (dist)	ashwanidandboina@g...		75
<a href="#">Open</a>	Devarajan[10_0].docx	Devarajan	c++, javascript	communication	web development, jquery	pan-america,phpunit, lo...	devarajanm@hotmail.com		99
<a href="#">Open</a>	Mithun_R.pdf	mithun	c++, javascript	communication			mithun1.2@gmail.com		82

8. There are filters present in the top of the table which can be used to access or search specific things in the job roles

9. Click upload button in the top bar, this will redirect to the Upload page

Chola

Resume Parser

Search

Upload

Logs

Create User

Logout

admin

Search Resume

Job Role \*

Enter job role

Primary Skills (comma-separated) \*

Enter primary skills

Secondary Skills (comma-separated)

Enter secondary skills

Applicants Location \*

Enter location

Configuration File

Select Config File

Delete

Search Resumes

Upload Config File

Choose File

No file chosen

Upload

Indexed Job Roles

Filter by User:

All Users

Serial Number: 1

Role: Software Testing

Primary Skills: javascript, c++

Secondary Skills: communication, team player

Location: Bangalore

Config File: New\_umbrella\_skills1.xlsx

Created By: admin

Created On: 12/24/2024

Reverted By: admin

Reverted On: 12/26/2024

Edit

Delete

Serial Number: 2

Role: Software Engineer

Primary Skills: wireframe

Secondary Skills: CSS, JAVA

Location: Bangalore

Config File: New\_umbrella\_skills1.xlsx

Created By: admin

Created On: 12/24/2024

Edited By: admin

Edited On: 12/24/2024

Edit

Delete

10. Click choose files, this will open file selection window. Select files to upload.

Chola

Resume Parser

Search

Upload

Logs

Create User

Logout

admin

Bulk Upload Resumes

Select Folder

Upload Folder

Choose Files

Upload Files

Process Uploaded Resumes

Process

Number of resumes to parse : 124

Status: idle

View Results

Upload Status

Open

This PC > Downloads

Organise

New folder

Last week (3)

Earlier this month (11)

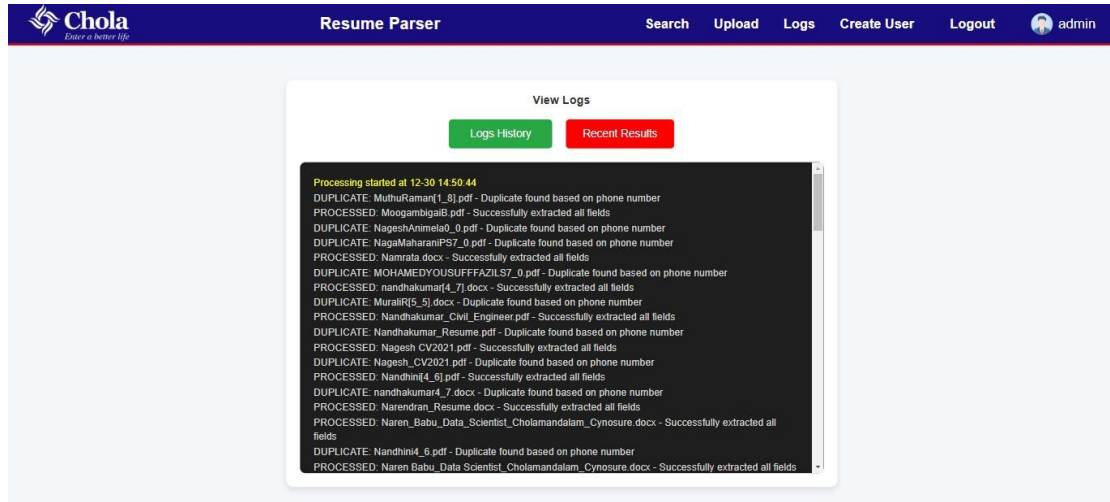
File name:

Custom Files

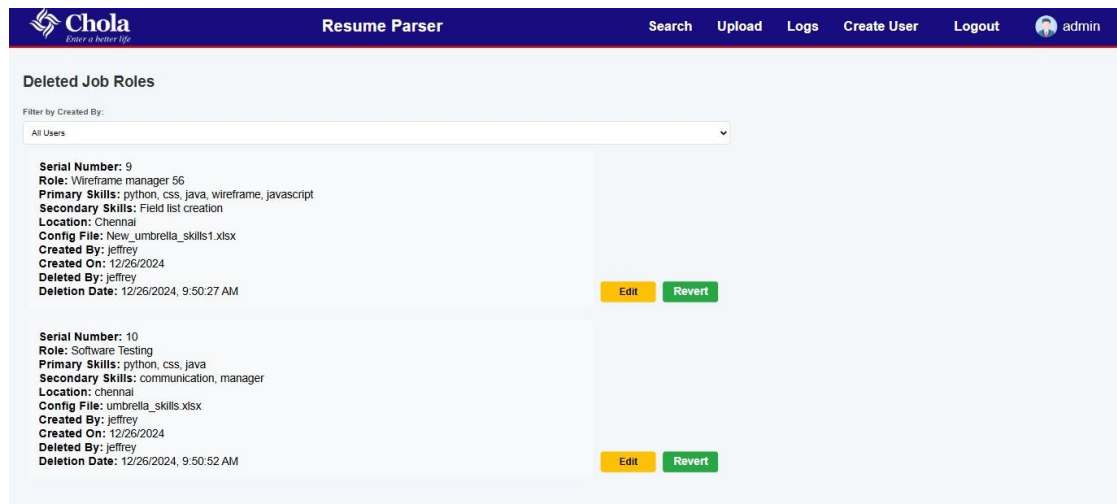
Open

Cancel

11. After upload, click on parse resumes, the remaining resumes will get displaced.  
Once the parsing is completed there will an Alert message
12. The user can click on view processing logs, which will contain processed files and skipped files Logs.



13. The deleted job roles can be reaccessed using the View Deleted Roles button and then they can be reverted or edited and reverted in that web page itself





### 1.4.2 Dependencies Details

- **Import os:** Used to interact with the operating system. In this script, it is used for path manipulations like setting upload folders and creating directories dynamically.
- **Import shutil:** Provides a higher-level interface to file operations. It is used here to delete or move files and directories.
- **import datetime:**Used to work with date and time. This can help in handling timestamps like createdOn values in your application.
- **Import subprocess:** Allows for spawning new processes, connecting to their input/output/error pipes, and obtaining their return codes. It seems included for executing external scripts or commands.
- **Import sys:** Provides access to system-specific parameters and functions. Typically used for command-line arguments or error handling.
- **import signal:** Used to handle asynchronous events and define signal handlers, which could manage termination signals for the application.
- **import threading:** Facilitates concurrent execution of threads within the program. It might support asynchronous or background tasks in the application.
- **from contextlib import contextmanager:** Provides utilities for writing concise and reusable resource management code, such as handling file locks or database connections.
- **import atexit:** Enables the execution of cleanup code (like clearing cache) when the program terminates.
- **from flask import Flask, jsonify, request, current\_app, g, send\_from\_directory**
  - Flask: Creates the application instance.
  - jsonify: Converts Python dictionaries to JSON responses.
  - request: Used to access incoming HTTP request data.

- `current_app`: Provides access to the application context.
- `g`: A namespace object for storing application-specific data during a request lifecycle.
- `send_from_directory`: Serves static files from a specified directory.
- **`from flask_pymongo import PyMongo`** : Simplifies the integration of MongoDB with Flask, allowing for interaction with the database.
- **`from flask_cors import CORS`**: Enables Cross-Origin Resource Sharing (CORS), allowing the API to be accessed from different domains or frontends.
- **`from werkzeug.local import LocalProxy`**: Provides a thread-safe proxy for lazy evaluation of database connections or objects.
- **`from werkzeug.utils import secure_filename`**: Ensures uploaded filenames are secure by stripping special characters to prevent security issues like path traversal attacks.
- **`from resumematcher import find_and_store_matched_resumes`**: A custom module (or library) for matching resumes against job criteria and storing the results in MongoDB.

### 1.4.3 Frontend Details

#### **Framework:** React Js (Vite)

We used ReactJs for its versatility and modular based building approach. React was selected because it suited the requirements for creating a dynamic app. React uses a `index.html` page as entry point to dynamically render components in a single page rather than using multiple static pages. The application uses **React Router** for navigation and implements role-based access control for different routes. All routes except the login page require user authentication.

#### **Route Breakdown**

##### 1. Root Route (/)

- Component: `<Login />`
- Purpose: Main entry point for user authentication
- Access: Public
- Functionality: Handles user login and authentication

##### 2. User Creation Route (/user\_creation)

- Component: `<AdminDashboard />`
- Purpose: Manage user accounts and permissions
- Access: Restricted to admin role only
- Security: Redirects to root if non-admin user attempts access

##### 3. Deleted Roles Route (/deleted\_roles)

- Component: `<DeletedRoles />`
- Purpose: View and manage deleted job roles
- Access: All authenticated users
- Security: Redirects to root if not logged in

##### 4. Search Route (/search)

- Component: `<Search />`

- Purpose: Resume search functionality
- Access: All authenticated users
- Security: Redirects to root if not logged in

#### 5. Upload Route (/upload)

- Component: `<Upload />`
- Purpose: Resume upload functionality
- Access: All authenticated users
- Security: Redirects to root if not logged in

#### 6. Logs Route (/logs)

- Component: `<LogViewer />`
- Purpose: View system activity logs
- Access: All authenticated users
- Security: Redirects to root if not logged in

#### 7. Job Role Route (/job\_role/:role)

- Component: `<ShowTable />`
- Purpose: Display specific job role information
- Access: All authenticated users
- Security: Redirects to root if not logged in

#### 8. Navbar:

- Centralized navigation through Navbar component

## Web Pages

### 1.4.3.1 Login.jsx (returns `<Login/>` component)

The Login component serves as the authentication gateway for the Resume Parser application, featuring a secure login interface with automatic redirection based on user roles.

## Key Features

### 1. Authentication Management

- Implements form handling using React Hook Form
- Maintains session storage for user authentication state
- Automatic role-based redirection after successful login
- Admin users → /user\_creation
- Regular users → /search

### 2. Security Features

- Password visibility toggle functionality
- Form validation with error handling
- Secure API communication using environment variables
- Session-based authentication storage

### 3. User Interface Elements

- Chola logo integration
- Background with corporate sketch image
- Responsive form design
- Clear error messaging system

### 4. Authentication Flow

- Form submission handling (username and password)
- API communication with backend
- Response processing
- Session storage update
- Role-based navigation
- Data Storage Structure

### 5. Security Measures

- Protected API endpoints
- Input validation

- Error handling for failed authentication
- Secure password field management
- Session-based authentication
- Auto-Redirect Logic
  - Checks for existing authentication
  - Prevents authenticated users from accessing login page
  - Directs users to appropriate dashboard based on role

This login implementation provides a secure, user-friendly entry point to the application while maintaining proper authentication and authorization controls

#### **1.4.3.2 adminPage.jsx (returns <AdminDashboard/>)**

The Admin Dashboard serves as a privileged control panel for user management, allowing administrators to create and manage user accounts within the system.

##### **1. User Creation Management**

- Creates new user accounts with specified roles
- Requires admin authentication for user creation
- Validates admin credentials before processing
- Provides immediate feedback on creation status

##### **2. Security Implementation**

- Role-based access verification
- Admin password verification for actions
- Unauthorized access prevention
- Session-based authentication checks

##### **3. Authentication Flow**

- Validates admin session on component mount
  - If not redirects to login page(failsafe if unauthorised access)
- Verifies admin credentials for each action
- Processes user creation requests
- Manages response handling
- Updates UI based on operation status

## 5. Security Measures

- Admin-only route protection
- Double authentication (session + password)
- Automatic redirection for unauthorized access
- Secure API communication

### **1.4.3.3 searchResume.jsx (returns <Search/>)**

#### 1. Job Role Management

- Creation of new job roles with detailed specifications
- Edit functionality for existing job roles
- Role deletion with confirmation
- Historical tracking of changes (created, edited, reverted)
- Serial number-based tracking system
- Timestamp management for all operations

#### 2. Configuration Management

- Upload system for Excel-based configuration files
- Configuration file deletion capabilities
- Validation of file formats (.xlsx, .xls)
- Dynamic configuration file selection
- Secure file handling system

#### 3. Search Parameters

- Job role specification
- Primary skills requirements (comma-separated)
- Secondary skills options
- Location preferences
- Configuration file association

#### 4. Admin-Specific Features

- User filtering capabilities
- View all users' job roles
- Access to delete, revert all job roles
- User-specific role management

## 5. Data Organization

- Serial number-based ordering
- Role status tracking

## 6. API Integration

- Job role fetching
- Configuration file management
- Search processing
- Role deletion handling
- Edit and revert functionality

### **1.4.3.4 logViewer.jsx (returns <LogViewer/> component)**

The LogViewer component displays and manages application logs with features like month-based categorization and color-coded log lines.

#### 1. Log Retrieval and Processing

- Fetches log data.
- Groups logs by.
- list of available months for filtering.

#### 2. User Interface Elements

- history and recent results.
- Dropdown menu lists available months for selection.
- Color codes logs
- Displays error messages for failed API calls.

#### 3. Log Content Handling

- Renders raw logs for recent entries and month-specific logs for history.
- Displays appropriate messages if no logs or month data is available.

#### 4. State Management



- Uses React states for log content, loading status, error messages, active button, selected month, available months, and grouped logs.

#### **1.4.3.5 ShowTable.jsx (returns <ShowTable/> component)**

1. Data Fetching and Handling
  - Fetches job role details from /job\_roles endpoint.
  - Retrieves resumes for specified job roles using /job\_role/:role endpoint.
2. Dynamic Columns
  - Defines columns for Resume, FileName, Name, Matched Skills, Secondary Skills, Expanded Skills, Location, Email, Phone, Days Since Upload, Date of Upload, and Score.
3. User Interface Features
  - Integrates Ag-Grid for sorting, filtering, pagination, and responsive design.
  - Dynamically displays job role based on backend data.
4. AgDataGrid Overview
  - Binds rowData dynamically to grid for real-time updates.
  - Defines columns with specific behaviors and interactivity.
  - Uses custom cellRenderer functions for tailored content display (e.g., clickable resume links, formatted phone numbers).
  - Supports sorting, custom comparators, and multi-select filtering.
  - Enables pagination for easier navigation of large datasets.

#### **1.4.3.6 DeletedJobRoles.jsx (returns<DeletedRoles/> )**

1. Fetching Deleted Roles
  - Retrieves deleted roles using fetchDeletedRoles and updates roles state.
  - Logs errors for network or API issues.
2. Revert Functionality
  - Restores deleted roles with revertJobRole.
  - Displays confirmation dialogs for user actions.
  - Re-fetches roles after a revert to keep the UI in sync.

3. Edit Functionality
  - Uses useForm hook for managing form states.
  - Populates form dynamically with setValue when editing.
  - Submits edited data and re-fetches roles to update UI.
4. User Role-Based Filtering
  - Admins can filter roles by creator.
  - Non-admins only see their own roles.
5. Dynamic Form Rendering
  - Displays form when a role is selected for editing.
  - Resets form when editing is canceled.
  - Highlights selectedJobRole in yellow to denote the role being edited.

#### **1.4.3.7 Upload.jsx (returns<Upload/> )**

1. File and Folder Selection
  - Users can select individual files or entire folders for upload.
  - Only .pdf and .docx files are allowed; invalid files are skipped.
2. Bulk Upload
  - Supports bulk uploading of multiple files at once.
  - Displays feedback on successful and failed uploads.
3. Resume Parsing
  - Initiates resume parsing via server requests.
  - Tracks remaining files and parsing status in real-time.
4. Persistent State
  - Uses localStorage to persist upload counts, status, and messages across sessions.
5. User Feedback
  - Provides visual indicators (messages, color-coded feedback) for upload and parsing status.
6. Major Functions
  - handleFolderSelection: Opens dialog for folder selection, filters valid files.
  - handleFileChange: Manages manual file selection and validation.
  - handleBulkUpload: Uploads files iteratively to the server.
  - handleFilesUpload: Uploads all files in a single request to the bulk\_upload endpoint.

- `handleParseResumes`: Starts resume parsing process.
- `handleStopProcessing`: Stops the parsing process.
- `checkProcessingStatus`: Polls server for parsing progress updates.
- `initializeComponent`: Fetches and updates the current parsing progress from the server.

#### **1.4.3.8 topBar.jsx (returns<Navbar/> )**

1. Component Overview
  - Renders in App.jsx
  - Imports CSS, logos
  - Tracks dropdown and mobile states
2. Responsive Design
  - Detects screen size
  - Shows dropdown in mobile view
3. Role-Based Features
  - Displays admin links
  - Restricts features based on role
4. User Interaction
  - Clears session on logout
  - Shows user profile or placeholder
  - Collapsible hamburger menu for mobile

## 1.4.4 Resume Parser Details

**Tech Stack:** Python,MongoDB

**Main.py** is the resume parser function which is used to parse the resumes and extract the information from those resumes and store the extracted info in the MongoDB .The following is the workflow extraction on how each process is done along with the function's explanation

### 1.4.4.1 Imports of functions

A. **From components.imports Import Everything (\*)**

- This line imports all the necessary modules and libraries included in the components/imports.py file.
- **Reason:** To centralize and manage external library imports and utility functions in one place for easier maintenance.

B. **from transformers import pipeline**

- **transformers** are a library from Hugging Face that provides pre-trained models for Natural Language Processing (NLP) tasks such as text classification, question answering, translation, and more.
  - **pipeline:** A simple way to run various NLP tasks. It abstracts the complexity of loading models, tokenizing data, and making predictions. Common tasks you can perform using the pipeline include sentiment analysis, named entity recognition (NER), and summarization.

C. **from pymongo.mongo\_client import MongoClient**

- **pymongo** is a Python library used to interact with MongoDB, a popular NoSQL database.
- **MongoClient:** This is the main entry point for connecting to a MongoDB database from a Python script. It allows you to connect to a local or remote MongoDB server, query databases, and manipulate documents stored within them. It handles communication between the Python application and the MongoDB server.

D. **From `components.text_extractor` Import `extract_info_from_document`**

- Imports a function to extract text content from resume files.
- Resumes might be in .pdf or .docx formats, requiring specialized parsing logic.

**Text\_extractor.py**

**1. `extract_text_from_pdf(pdf_path)`**

- **Purpose:** Extracts text from a PDF file.
- **Method:** First tries pdfplumber for text extraction; if it fails, falls back to PyPDF2.
- **Error Handling:** Logs unreadable files to unreadable\_files.txt and prints error messages.
- **Output:** Returns text in lowercase or None if extraction fails.

**2. `extract_text_from_docx(docx_path)`**

- **Purpose:** Extracts text from a DOCX file.
- **Method:** Uses the docx library to extract text from paragraphs in the document.
- **Error Handling:** Logs errors to unreadable\_files.txt and prints error messages.
- **Output:** Returns text in lowercase or None if extraction fails.

**3. `extract_info_from_document(file_path)`**

- **Purpose:** Routes the file extraction to the appropriate function based on file type (PDF or DOCX).
- **Method:** Checks file extension and calls either `extract_text_from_pdf()` or `extract_text_from_docx()`.
- **Error Handling:** Raises a ValueError if the file format is unsupported.
- **Output:** Returns extracted text or raises an error for unsupported formats

E. **From `components.entity_extractor` Import Specific Functions**

- Imports the following functions for extracting key entities from the resume text:
  - `extract_name`: Identifies the candidate's name from the text.
  - `extract_phone`: Extracts phone numbers using regex.
  - `extract_email`: Extracts email addresses using regex.
  - `extract_location_with_spacy`: Uses NLP to detect locations.

Entity\_extractor.py

### 1. `extract_name(text)`

- **Purpose:** Extracts a person's name from the given text.
- **Method:** The function processes the first 100 characters of the text, converts it to lowercase, and uses the spaCy NLP model to identify named entities. Specifically, it looks for entities labeled as 'PERSON', which indicate a person's name.
- **Output:** Returns the extracted name if found; otherwise, returns None.

### 2. `extract_phone(text)`

- **Purpose:** Extracts a phone number from the given text.
- **Method:** This function uses regular expressions to match both mobile numbers (including those with country codes like +91) and landline numbers (e.g., +044 1234 5678). It also removes unnecessary whitespace from the text.
- **Output:** Returns the first phone number match found in the text or None if no number is found.

### 3. `extract_email(text)`

- **Purpose:** Extracts an email address from the given text.
- **Method:** The function uses a regular expression to match email addresses in standard formats (e.g., user@example.com).
- **Output:** Returns the first email address match found in the text or None if no email is found.

#### 4. `extract_location_with_spacy(text)`

- **Purpose:** Extracts the location information from the given text in multiple steps.
- **Method:** The function follows a detailed multi-step process:
  1. **Step 1:** Tries to match phrases like "*Current Location*", "*Preferred Location*", and "*Address*" using regular expressions. If any of these matches, it captures the location after the keyword and returns it.
  2. **Step 2:** If no location is found in Step 1, the function checks for a 6-digit *pincode* in the text and tries to match it to a location in a CSV file (`output.csv`). If a match is found, it returns the location.
  3. **Step 3:** If no location is found in the previous steps, the function looks for an address after the keyword "*Address*". If found, it returns the address.
  4. **Step 4:** The function checks for street-related keywords (like "*street*", "*road*", "*lane*") and returns the street name if found.
  5. **Step 5:** As a final fallback, the function uses spaCy's named entity recognition (NER) to capture geographical locations (labeled as 'GPE' in spaCy) from the text.
- **Output:** Returns a list of extracted location(s) if found, or `None` if no location could be identified.

F.     **From**     **`components.skills_extractor`**     **Import**     **`extract_skills`**     **and**  
**`generate_skill_strins`**

- Identifies and extracts relevant skills from the resume text.

`Skills_extractor.py`

#### 1. `extract_skills(text)`

- **Purpose:** This is the main pipeline function for extracting both single-word and multi-word skills from the text.
- **Method:**
  - Preprocesses the text using the `preprocess_text()` function to clean the input.

- Extracts single-word skills (nouns and adjectives) by calling `extract_skills_from_text()`.
- Extracts multi-word skills by calling `extract_skills_from_skillner()`.
- Combines both single-word and multi-word skills, removes duplicates (using a set), and counts the occurrences of each skill (using Counter from the collections module).
- **Output:** Returns a list of unique extracted skills, sorted by frequency.

## 2. `generate_skill_string(text)`

- **Purpose:** Cleans the input text and generates a skill-oriented string by removing stopwords and special characters, while keeping key punctuation marks (commas, periods).
- **Method:**
  - Preprocesses the text using `preprocess_text()` to remove unnecessary elements.
  - Splits the text into words, filtering out stopwords (common, unimportant words) using the NLTK stopwords corpus.
  - Rejoins the remaining words into a single string.
- **Output:** Returns a clean string containing only relevant words for skill extraction.

## Helper functions

### 1. `normalize_string(text)`

- **Purpose:** This function cleans the input text by removing non-printable characters (such as control characters).
- **Method:** It uses Python's `string.printable` to filter out any characters from the text that are not printable, effectively sanitizing the text.
- **Output:** Returns the normalized version of the input text.

### 2. `preprocess_text(text)`

- **Purpose:** Cleans the input text by removing elements that are not useful for skill extraction (such as emails, phone numbers, URLs, special characters, and excessive whitespace).



- **Method:**
  - It uses regular expressions to remove:
    - Email addresses (e.g., user@example.com)
    - Phone numbers (e.g., 1234567890)
    - URLs (e.g., http://example.com)
  - It then normalizes the text using `normalize_string()` to remove non-printable characters.
  - The entire text is converted to lowercase for uniformity.
- **Output:** Returns the cleaned and lowercase version of the input text.

### 3. `extract_skills_from_text(text)`

- **Purpose:** Extracts potential skills (nouns and adjectives) from the preprocessed text using SpaCy.
- **Method:**
  - It processes the text with the SpaCy NLP model to extract linguistic features.
  - It filters for tokens that are either nouns (NOUN) or adjectives (ADJ), excluding stop words (common words like "the", "is", etc.).
  - The function uses lemmatization (getting the base form of words) to normalize the words (e.g., "running" becomes "run").
- **Output:** Returns a list of lemmatized skills (nouns and adjectives).

### 4. `extract_skills_from_skillner(text)`

- **Purpose:** Extracts multi-word skills using a custom skill extraction model, SkillExtractor from the SkillNer library.
- **Method:**
  - The SkillExtractor is used to annotate the text with potential skills.
  - The function retrieves:
    - **Full match skills:** Skills that are found as exact matches in the SkillNer database.
    - **Ngram scored skills:** Skills identified using n-grams (multi-word expressions) with a score above 0.8.

- It combines both sets of skills (full match and ngram scored), removes duplicates, and returns them.
- **Output:** Returns a list of multi-word skills, or an empty list if no skills are found.

G. **From components.db\_connector Import connect\_to\_mongo**

- Imports a function to establish a connection with a MongoDB database.
- Simplifies database operations by centralizing MongoDB connection logic.

H. **from concurrent.futures import ThreadPoolExecutor**

- **Purpose:** Enables concurrent (multi-threaded) processing of resume files.
- **Why:** Speeds up processing by handling multiple resumes simultaneously

#### 1.4.4.2 Functions present in main.py

**MongoDB Setup:**

- A connection to the MongoDB database is established using the uri = "mongodb://localhost:27017".
- The database resume\_db and collection resumes are accessed for storing and retrieving resume data.

**1. get\_file\_creation\_date(file\_path):**

- This function fetches the creation date of the resume file and formats it into a YYYY-MM-DD format. It uses os.path.getctime(file\_path) to extract the timestamp of when the file was created.

**2. insert\_resume\_data(collection, resume\_data):**

- This function inserts or updates a resume entry in MongoDB based on the resume's name and email/phone. It performs an upsert operation to either insert new data or update existing data if a record with the same name and email/phone exists.
- The function uses update\_one() to modify or add the resume data.

### 3. `clean_filename(filename)`:

- Strips the filename of unwanted elements such as special characters and text inside square brackets (e.g., JohnDoe [cv]). It ensures that the filename is clean and only contains alphanumeric characters and spaces.

### 4. `count_skill_occurrences(resume_text, skills)`:

- This function counts the occurrences of each skill from the list of skills in the resume's text. It compares the skills in the resume text and returns both the unique skills and their respective counts.

### 5. `log_message(message, log_file="logs.txt", results_file="results.txt")`:

- The function logs messages about the processing, errors, or duplicates to both logs.txt and results.txt. Each log entry is timestamped for better tracking.
- logs.txt captures the details of each operation, while results.txt contains the summary of results.

### 6. `process_resume(resume_file_path, processed_folder, skipped_folder, duplicates_folder)`

This is the core function responsible for processing each resume file individually. It takes the path of the resume file and the paths of folders for processed, skipped, and duplicate resumes.

#### 6.1 Check File Type:

- **Purpose:** The first task is to verify that the file is either a PDF or DOCX file, which are the supported formats.
- **Logic:**
  - The file extension is extracted using `os.path.splitext(resume_file_path)[1].lower()` and compared against the allowed file extensions (.pdf, .docx).

- If the file is not a PDF or DOCX, the function logs the file as "skipped" and moves it to the `skipped_folder` to avoid processing unsupported files.

## 6.2 Extract Text from the Resume:

- **Purpose:** After confirming the file type, the function attempts to extract the textual content of the resume.
- **Logic:** `extract_info_from_document(resume_file_path)` is called to extract the text from the resume file. If no text is extracted (e.g., if the resume is a scanned document or an image-based PDF), the function raises an exception and logs the error, moving the file to the `skipped_folder`.

## 6.3 Extract Key Resume Information:

- **Purpose:** After successfully extracting the resume's text, the function proceeds to extract relevant details.
- **Logic:**
  - **Name:** The `extract_name` function is called to extract the candidate's name.
  - **Phone:** The `extract_phone` function is used to extract the phone number from the resume text.
  - **Email:** Similarly, `extract_email` extracts the email address.
  - **Location:** The `extract_location_with_spacy` function identifies location details (e.g., current or preferred address).
  - **Skills:** The `extract_skills` function extracts both single-word and multi-word skills from the resume text.
  - **Skill String:** `generate_skill_string` produces a string of relevant skills after cleaning the text.

## 6.4 Handle Missing Name:

**Purpose:** If the name is not extracted (which can sometimes happen due to formatting issues), the function attempts to clean the filename by calling `clean_filename(resume_file)`. This ensures that even if the name is missing, the resume is still processed and stored with a meaningful identifier.

## 6.5 Prepare Resume Data:

- **Purpose:** After extracting all necessary details, the function organizes the data into a dictionary, `resume_data`, that includes:
  - Name
  - Phone number
  - Email
  - Location
  - Skills
  - Skill counts (occurrences of each skill in the resume)
  - Filename
  - File path
  - Date of upload (extracted via `get_file_creation_date`)
  - Skill string (a clean list of extracted skills).

## 6.6 Check for Duplicates:

- **Purpose:** To avoid processing the same resume multiple times, a duplicate check is performed.
- **Logic:**
  - If the resume has a phone number, it checks the database for an existing document with the same phone number.
  - If the phone number exists, the resume is considered a duplicate, and the file is moved to the `duplicates_folder`. A log entry is created, and processing for that resume stops.
  - If the resume doesn't have a phone number, it checks for a combination of name and email to determine duplicates.

## 6.7 Insert Resume Data into MongoDB:

- **Purpose:** If no duplicate is found, the resume data is inserted into the MongoDB collection `resumes`.

- **Logic:** The `insert_resume_data` function is called to either insert or update the document in the database. This function uses an upsert operation, meaning it will update an existing document if the name and email (or phone) match, or insert a new document if there's no match.

#### 6.8 Move Processed Resume:

- **Purpose:** After successfully processing the resume, it is moved to the `processed_folder`.
- **Logic:** The file is moved from its original location to the `processed_folder` using `shutil.move`.

#### 6.9 Handle Errors:

- **Purpose:** If any error occurs during the processing (e.g., text extraction fails or a required field is missing), the error is logged, and the file is moved to the `skipped_folder` for further review.

### 7. `parse_and_store_resumes(resume_folder, processed_folder, skipped_folder, duplicates_folder)`

This function is responsible for managing the overall process of parsing and storing resumes from a specific folder. It ensures that the necessary folders are set up and then processes each resume concurrently.

#### 7.1 Ensure Folders Exist:

- **Purpose:** Before starting, the function checks whether the `processed_folder`, `skipped_folder`, and `duplicates_folder` exist. If any of them are missing, they are created using `os.makedirs(folder, exist_ok=True)`.

#### 7.2 Log Start Time:

- **Purpose:** To track when the resume processing starts, the function logs the current timestamp in both `results.txt` and `logs.txt` files.
- **Logic:**

- The start time is recorded in results.txt to give a clear indication of when the processing began.
- The start time is also logged in logs.txt for traceability.

### 7.3 Get Resume Files:

- **Purpose:** The function retrieves a list of all resume files (PDF and DOCX) from the resume\_folder using os.listdir(resume\_folder). This list is then passed to the processing logic.

### 7.4 Concurrent Processing Using ThreadPoolExecutor:

- **Purpose:** To speed up the processing of multiple resumes, the function uses ThreadPoolExecutor from the concurrent.futures module to process the resumes concurrently.
- **Logic:**
  - Each resume file path is submitted to a thread pool, where it will be processed by the process\_resume function.
  - The executor.submit(process\_resume, resume\_file\_path, processed\_folder, skipped\_folder, duplicates\_folder) command is used to submit each file for processing.
  - The future.result() is called to ensure that each resume is processed before moving on to the next.

### 7.5 Log Completion Time:

- **Purpose:** After all resumes have been processed, the function logs the completion time.
- **Logic:** The completion time is recorded and logged in both results.txt (as a summary) and logs.txt (for detailed tracing).

### 1.4.5 Backend Details

#### **Framework:** Flask API (Python)

We used Flask for its lightweight and modular architecture, making it an ideal choice for building the backend of the application. Flask was selected due to its flexibility and ability to handle API calls efficiently, which suited the requirements for creating a robust server-side framework. It serves as the backbone for managing data flow between the frontend and the MongoDB database. Flask provides a single-entry point for routing API endpoints, ensuring seamless communication between the client and the server. The application implements role-based access control in Flask to secure API endpoints, with all routes except the login route requiring user authentication. This ensures a secure and efficient backend infrastructure that supports the dynamic features of the application. The following are the API Routes used for various different function

#### **Routes and Functions**

##### **Route: /welcome**

- **Function Name:** `welcome`
- **Core Functionality:** General Information

This route provides a welcome message to the users of the API to ensure the service is up and running.

##### **Route: /content/resume/<path: filename>**

- **Function Name:** `serve_processed_resume`
- **Core Functionality:** Resume Handling

Serves processed resume files stored in the `processed_resume` directory. Useful for providing processed resume data to users.

##### **Route: /content/duplicate\_resume/<path:filename>**

- **Function Name:** `serve_duplicate_resume`



- **Core Functionality:** Duplicate File Management

Serves duplicate resumes stored in the `duplicate_resume` directory, helping identify and retrieve files flagged as duplicates.

### **Route: /content/skipped\_resume/<path:filename>**

- **Function Name:** `serve_skipped_resume`
- **Core Functionality:** Skipped Resume Handling

Serves resumes that were skipped during processing, stored in the `skipped_resume` directory.

### **Route: /job\_roles**

- **Function Name:** `get_job_roles`
- **Core Functionality:** Job Role Information

Fetches job role details from the `job_roles_info` collection in the database, returning a list of available roles.

### **Route: /resumes**

- **Function Name:** `get_resumes`
- **Core Functionality:** Resume Retrieval

Retrieves all resumes stored in the database for further processing or review.

### **Route: /upload\_config**

- **Function Name:** `upload_config`
- **Core Functionality:** Configuration File Upload

Allows users to upload configuration files to the server for resume matching or job role processing.

## **Route: /config\_files**

- **Function Name:** `list_config_files`
- **Core Functionality:** Configuration File Management

Lists all configuration files uploaded to the server, enabling users to review available files.

## **Route: /delete\_config/<filename>**

- **Function Name:** `delete_config_file`
- **Core Functionality:** File Deletion

Moves a specified configuration file to the `deleted_configurations` folder instead of permanently deleting it.

## **Route: /process\_job\_search**

- **Function Name:** `process_job_search`
- **Core Functionality:** Resume Matching

Processes job searches based on user-provided details (e.g., job role, skills) and finds matching resumes.

## **Route: /login**

- **Function Name:** `login`
- **Core Functionality:** Credentials Management

Validates user credentials and returns a success response with the user's role or an error if authentication fails.

## **Route: /createuser**

- **Function Name:** `createuser`
- **Core Functionality:** User Account Management

Creates a new user account after validating that the username is unique and the password meets security criteria.

### **Route: /upload\_file**

- **Function Name:** `upload_file`
- **Core Functionality:** File Upload

Handles uploading individual files to the server and stores them in the `content/resume` directory.

### **Route: /bulk\_upload**

- **Function Name:** `bulk_upload_files`
- **Core Functionality:** Bulk File Upload

Handles uploading multiple files at once to the server, storing them in the `content/resume` directory.

### **Function Name: clear\_cache**

- **Core Functionality:** Cache Management

Deletes all `__pycache__` directories in the project to ensure the application runs without stale compiled files.

### **Function Name: get\_db**

- **Core Functionality:** Database Connection Management

Provides a thread-safe way to access the MongoDB instance within the application. Ensures a single connection per request lifecycle.

## **Function Name: validate\_password**

- **Core Functionality:** Password Validation

Validates passwords based on specific criteria, including length, uppercase, lowercase, numeric, and special character requirements. Returns validation results and an appropriate message.

## **Function Name: find\_and\_store\_matched\_resumes (Imported)**

1. **Core Functionality:** Resume Matching

Matches resumes against job search criteria (e.g., skills, location, job role) and stores results in the database.

- a. Function is passed with details from the frontend via route, the function is passed with these parameters excel\_file\_path, primary\_skills, secondary\_skills, location, job\_role, serial\_number.

## **Route : /parse\_resumes and /get\_resume\_progress**

**API Route:** /parse\_resumes

**Method:** POST

**Purpose:** Manages the subprocess responsible for parsing resumes. It allows the user to start, stop, or check the status of the process.

### **Flow:**

1. **Action: Start**

- Invokes the start\_process method of the ProcessManager class.
- If successful, it launches the subprocess to execute the main resume parsing script (main.py).

- Returns the current parsing status, including the total and remaining resumes (reads the file count of the resume folder in /app/Resumeless/backend/content/resumes).

## 2. **Action: Stop**

- Calls the `stop_process` method of the `ProcessManager` class.
- Terminates the subprocess if it is running.
- Resets the process state to idle.

## 3. **Action: Status**

- Retrieves the current status of the parsing process via the `get_status` method of the `ProcessManager` class.
- Provides details about whether the process is running, idle, or completed.

# ProcessManager Class

The `ProcessManager` class is the part of `/parse_resumes` route, it contains the logic for process management and status tracking.

## Key Attributes:

- **current\_process**: Stores the subprocess instance for the ongoing parsing process.
- **is\_processing**: Boolean flag indicating whether a process is running.
- **total\_files**: Tracks the total number of resumes in the folder.
- **remaining\_files**: Tracks the number of resumes yet to be processed.
- **process\_completed**: Indicates if the process has successfully finished.
- **just\_completed**: Flag to signal the completion of a process, used for status reporting.

## Key Methods:

### 1. **start\_process()**

- Checks if a process is already running. If not, starts the subprocess using `subprocess.Popen` to execute the **main.py** script, which contains the logic for resume parsing.

- Initializes file counts by invoking `_update_file_counts`.
- Launches a monitoring thread (`monitor_process`) to handle process completion tracking.
- Returns success status and a message indicating whether the process was started successfully.

## 2. **stop\_process()**

- Terminates the currently running subprocess gracefully using `terminate()`.
- If termination fails, it forcefully kills the process using `kill()`.
- Resets internal flags and counters to indicate an idle state.
- Returns success status and a message indicating the operation result.

## 3. **get\_status()**

- Provides a snapshot of the current processing state, including the number of total, remaining, and processed files.
- Updates file counts dynamically by calling `_update_file_counts`.
- Determines the process state (idle, processing, or completed) and includes it in the response.

## 4. **\_update\_file\_counts()**

- Scans the content/resume directory to calculate the total number of files.
- Updates the `total_files` and `remaining_files` attributes accordingly.

## 5. **monitor\_process()**

- Waits for the subprocess to complete execution using `wait()`.
- Checks the subprocess return code to confirm successful completion.
- Updates flags (`process_completed`, `just_completed`) and resets the `current_process` attribute upon completion.

## **Signal Handling:**

- The class integrates with OS signals (`SIGINT`, `SIGTERM`) to handle shutdowns gracefully. If a shutdown signal is received, it ensures the subprocess is terminated properly to avoid orphan processes.

## **API Route: `/get_resume_progress`**

**Purpose:** Tracks the progress of resume processing.

**Flow:**

1. Reads the content/resume directory to count the total number of resumes.
2. Separately identifies processed resumes by checking for files with a specific extension (e.g., .processed).
3. Calculates the number of remaining resumes by subtracting the count of processed files from the total.
4. Returns the progress summary.

#### **API Route: /view\_duplicate\_files**

- **Function Name:** view\_duplicate\_files
- **Core Functionality:** Duplicate File Viewer

Lists all duplicate resumes stored in the duplicate\_resume directory for user review.

#### **API Route: /view\_skipped\_files**

- **Function Name:** view\_skipped\_files
- **Core Functionality:** Skipped File Viewer

Lists all skipped resumes stored in the skipped\_resume directory for further analysis.

#### **API Route: /view\_processed\_files**

- **Function Name:** view\_processed\_files
- **Core Functionality:** Processed File Viewer

Lists all processed resumes stored in the processed\_resume directory for verification and review.

#### **API Route: /get\_next\_serial**

- **Function Name:** get\_next\_serial

- **Core Functionality:** Serial Number Generator

Determines the next available serial number by comparing the highest serials in active and deleted job roles.

### **API Route: /job\_role/<serial\_number>**

**Purpose:** This route is designed to manage job role information based on a unique serial\_number. It supports two operations: fetching job role data (GET) and deleting a job role (DELETE).

### **API Route: /deleted\_roles**

- **Function Name:** get\_deleted\_roles
- **Core Functionality:** Deleted Roles Viewer

Fetches all job roles marked as deleted for record-keeping or restoration.

### **API Route: /revert\_job\_role/<serial\_number>**

- **Function Name:** revert\_job\_role
- **Core Functionality:** Job Role Restoration

Restores a previously deleted job role along with its data by moving it back to the active state and renaming collections.

### **API Route: /edit\_job\_role/<int:serial\_number>**

- **Function Name:** edit\_job\_role
- **Core Functionality:** Job Role Editing and Re-Matching

Updates job role information and triggers a re-matching of resumes based on the new criteria.

### **API Route: /edit\_and\_revert\_job\_role/<serial\_number>**

- **Function Name:** edit\_and\_revert\_job\_role
- **Core Functionality:** Edit and Restore Deleted Job Role

Edits the details of a previously deleted job role and moves it back to the active state with updated configurations.



### **API Route: /recent\_results**

- **Function Name:** get\_recent\_results
- **Core Functionality:** Processing Results Viewer

Reads and displays the latest resume parsing results from the results file.

### **API Route: /logs**

- **Function Name:** get\_logs
- **Core Functionality:** Log Viewer

Retrieves and displays logs from the application log file for debugging and monitoring.

## 1.4.6 Resume Matcher Details

### **Tech Stack: Python, MongoDB**

`Resumematcher.py` contains `find_and_store_matched_resumes()` function which is imported in `server.py` code. This function gets passed with the search data from the frontend and then it queries the search result in PyMongo to find resumes with right match.

Important variables:

1. **Matched Skills:** These are skills explicitly mentioned in the resume and directly correspond to the required primary or secondary skills.
2. **Umbrella Skills:** These are skills related to a broader category (umbrella term). For example, the term "MERN" might include MongoDB, Express, React and NodeJs as sub-skills so we have to search them too.
3. **Expanded Skills List:** This involves normalizing and expanding the list of primary and secondary skills using mappings and reverse mappings to ensure variations, synonyms, and alternate forms of the same skill are captured.

### **1.4.6.1 Matched Skills: Identifying Direct Skill Matches**

Matched skills are those explicitly present in the resume that align with the job role's primary or secondary skills. This is the most straightforward and fundamental step in skill matching.

#### **Method:**

- The script extracts the skills field from resumes stored in the MongoDB collection.
- Each skill in the resume is normalized (converted to lowercase and stripped of whitespace) to ensure consistency with the job role's skill list.

- The script compares these normalized resume skills against the expanded primary skills and expanded secondary skills lists (explained in Section 3 below).
- A match is recorded if any skill in the resume corresponds to one in the job role's skill list (directly or through mappings).

**Example:**

- **Input:**
  - Job Role Primary Skills: ["JavaScript", "React"]
  - Job Role Secondary Skills: ["Node.js", "TypeScript"]
  - Resume Skills (1 sample resume): ["React", "JavaScript", "HTML", "CSS"]
- **Process:**
  - Normalize both job role skills and resume skills:
    - Primary Skills → ["javascript", "react"]
    - Secondary Skills → ["node.js", "typescript"]
    - Resume Skills → ["react", "javascript", "html", "css"]
  - Compare normalized resume skills with expanded job role skills:
    - Matched Primary Skills: ["javascript", "react"]
    - No Matched Secondary Skills.
- **Output:**
  - Matched Skills = ["javascript", "react"]

### **1.4.6.2 Umbrella Skills: Capturing Broader Skill Categories**

Umbrella skills refer to broader categories that encompass multiple related sub-skills. This matching step helps identify candidates whose resumes may not list the exact skills specified in the job description but include relevant sub-skills or parent categories.

## **Method:**

### 1. Umbrella Terms Database:

- The script loads umbrella terms from an Excel file. Each umbrella term is associated with a list of related sub-skills.
- Example:
  - Umbrella Term: "Frontend Development"
  - Sub-Skills: ["JavaScript", "React", "CSS", "HTML"]

### 2. Reverse Lookup:

- The script also creates a **reverse mapping**, where each sub-skill points back to its umbrella term(s).
- Example:
  - Sub-Skill: "React" → Umbrella Term: "Frontend Development"

### 3. Matching:

- For each skill matched in the resume, the script checks if it corresponds to an umbrella term or its sub-skills.
- If a sub-skill (e.g., "CSS") is found in the resume but not explicitly mentioned in the job role, it is still counted under the umbrella term.

## **Example:**

- **Input:**
  - Umbrella Term: "Frontend Development"
  - Sub-Skills: ["JavaScript", "React", "CSS", "HTML"]
  - Resume Skills: ["CSS", "HTML", "MySQL"]
- **Process:**

- Check resume skills against sub-skills of "Frontend Development":
  - Matched Sub-Skills: ["CSS", "HTML"]
- Umbrella Term Match: "Frontend Development"
- **Output:**
  - Umbrella Skills = ["CSS", "HTML"]

### 1.4.6.3. Expanded Skills List: Ensuring Comprehensive Coverage

The expanded skills list accounts for variations, synonyms, and alternate forms of skills to ensure comprehensive matching.

#### Method:

##### 1. Skill Mappings:

- The script loads a JSON file (skill\_db\_relax\_20.json) containing mappings of:
  - Canonical Skill Names: The standard form of a skill (e.g., "JavaScript").
  - High Surface Forms: Frequently used synonyms (e.g., "JS, JavaScript").
  - Low Surface Forms: Variations that may appear less frequently.
- Example:
  - "JavaScript" → ["js", "javascript", "ecmascript"]

##### 2. Reverse Mappings:

- For every canonical skill, reverse mappings are created to associate all its variations with the original skill.
  - Example:
    - "js" → "JavaScript"

### 3. Normalization:

- All primary and secondary skills are normalized to their canonical form and expanded to include variations.
- Example:
  - Input: ["JavaScript", "React"]
  - Output (Expanded): ["javascript", "js", "react"]

### 4. Matching:

- The script compares expanded job role skills with resume skills.
- Matches are recorded for any variation or synonym found in the resume.

#### Example:

- **Input:**
  - Primary Skill: "JavaScript"
  - Mappings: ["js", "javascript", "ecmascript"]
  - Resume Skills: ["js", "html", "css"]
- **Process:**
  - Expanded Primary Skills: ["javascript", "js", "ecmascript"]
  - Resume Skills: ["js", "html", "css"]
  - Matched Skills: ["js"]

## 1.4.6.4 Scoring Candidates

The final score is calculated by combining:

1. **Matched Primary Skills:** Each match adds to the score. The count is capped at a maximum value (e.g., 2) to prevent over-weighting.
2. **Matched Secondary Skills:** These contribute slightly less than primary skills but are still valuable.

3. **Umbrella Skills:** Skills under umbrella terms contribute less weight than direct matches but ensure comprehensive evaluation.

After matching the skills, the relevant data is structured into a dictionary (matched\_data) with fields like:

- Candidate details (name, contact, email).
- Resume location (resume\_location, filename).
- Skills breakdown:
  - Matched skills.
  - Secondary skills.
  - Umbrella skills.
- Skill counts.
- Scoring.

This data is then ready to be pushed into the database.

## 1.4.7 Database Details

The logic for saving matched resumes into MongoDB collections is carefully designed to ensure that:

1. **Resumes are stored systematically** in a job-role-specific collection.
2. **Existing records are updated**, while new records are inserted to avoid redundancy.

The database used is named `resume_db`. This database acts as the central repository for all resume and matching data.

### Resumes Collection (`resumes`):

- This is the main collection where all resumes are initially stored via `main.py`.
- Each document represents a candidate's resume and contains fields such as:
  - `skills` (list of skills in the resume),
  - `skill_counts` (number of times each skill appears),
  - `location` (candidate's preferred location),
  - Other metadata (e.g., name, phone, email, `file_path`).

### Matched Collection (`job_role_<serial_number>`):

- A separate collection is created for each job role, named using the pattern `job_role_<serial_number>` where `<serial_number>` uniquely identifies the role.
- This ensures that matched resumes are organized by job role, making it easy to query relevant matches for any specific position.

Before pushing data to the `job_role_<serial_number>` collection, the script first queries the `resumes` collection to identify matching resumes based on:

- **Expanded Skills** (primary and secondary).
- **Location** (optional, if specified in the job role).

The query is constructed using MongoDB's `$or` operator to match resumes with:

- At least one primary skill.
- Optional secondary skill.
- The specified location.

The `job_role_<serial_number>` collection is used to store the matched resume data. The logic for insertion and updating works as follows:

#### a. Upsert Logic

- **Condition:**
  - If a resume with the same name and `resume_location` already exists in the collection, it is **updated**.
  - If no matching record is found, a new document is **inserted**.



- Ensures that updated information (e.g., new scores or skills) is reflected without creating duplicate entries.

## **b. Fields Stored**

For each matched resume, the following fields are stored:

- **Personal Information:**
  - name, phone, email.
- **File Metadata:**
  - filename, file\_path, date\_of\_upload.
- **Matched Skills:**
  - Lists of matched\_skills, secondary\_skills, and umbrella\_skills.
- **Skill Counts:**
  - Count of each matched skill categorized into primary, secondary, and umbrella skills.
- **Score:**
  - Calculated score indicating the strength of the match.
- **Job Role Information:**
  - The job role for which this resume was matched, tied to the collection name.

## 1.5 CONCLUSION

The development of the Resume Parser Project is used to ease the process of recruitment of new employees and assist the HR team in the shortlisting process. The manual method of going through each and every resume and analysing it against the job description is a time taking process and this application aims to reduce the time by implementing efficient methods for extraction of information, compiling and showcasing the said information in a presentable manner, makes it easier for the HR team.