Awesome, Ragul! Let's begin your **Chapter 1: Project Structure** — this is your foundation. Once you understand this, everything else will click easily.

📁 Chapter 1: Project Structure — Understand the Folder Tree

Your project has two main parts:

🔵 Backend (Flask API)

🔷 Frontend (React UI)

🔍 Overall Structure

```
resume-evaluator/
│
├── client/          ← Frontend: React app
├── server/          ← Backend: Flask app
├── .gitignore       ← Files Git should ignore
└── README.md        ← Project documentation
```

Let's now explore each folder.

🧠 1. `client/` — Your React Frontend

React is used to build **interactive websites**. It's written in JavaScript.

📁 `client/` contains:

```
client/
├── public/
│   └── index.html         ← Base HTML file
│
├── src/
│   ├── App.js             ← Main UI logic (upload form, display results)
│   └── index.js           ← Entry point that loads <App />
```

```
│
├── package.json           ← List of all frontend libraries
├── .gitignore             ← Ignore build files
└── README.md              ← React app instructions
```

✅ React lets you:

- Upload resume and JD PDFs
- Submit them to the backend
- Show score and missing skills from response

We'll go deep into React in **Chapter 4**.

🧠 2. `server/` — Your Python Flask Backend

Flask is a **micro web framework** in Python that lets you create APIs (routes that do work behind the scenes).

📁 `server/` contains:

```
server/
├── app.py                 ← Main Python file with API logic
├── requirements.txt       ← List of Python libraries to install
├── uploads/               ← (Optional) folder to save uploaded PDFs
└── venv/                  ← (Optional) Python virtual environment folder
```

✅ Flask does the heavy lifting:

- Accepts uploaded files
- Extracts text from PDFs
- Uses **spaCy** to analyze skills
- Compares with job description
- Returns result to frontend

We'll explore this line by line in **Chapter 2**.

🔁 3. How They Work Together

Here's the data flow:

```
User → React Frontend → Flask Backend → NLP Analysis → Score → Back to React → Sh
```

Simple idea:

- Frontend = The face (form, buttons, result)
- Backend = The brain (analysis, logic)
- Together = A complete web app

🔄 BONUS: Development Environment

You run both apps **separately**:

- React runs on: `http://localhost:3000`
- Flask runs on: `http://localhost:5000`

React sends API requests to Flask using:

```
fetch('http://localhost:5000/evaluate', ...)
```

That's what you setup during development.

✅ **You now understand the complete folder structure and architecture of your project!**

👉 Next Step:

Do you want to go to **Chapter 2: Backend (Flask)** where we'll break down how your Python code works line by line?

Or would you like a short quiz to test what you've learned so far?

chapter 2

Great choice, Ragul! Welcome to **Chapter 2: Backend (Flask)** — the *brain* of your Resume Evaluator project. Here, you'll understand exactly what happens when you upload resumes

and job descriptions.

🧠 Chapter 2: Understanding the Backend — `Flask + Python + NLP`

✅ What is Flask?

Flask is a lightweight **Python web framework** that helps you build **APIs** (routes) to handle requests, process data, and return responses.

📄 The Key File: `server/app.py`

This file is where all the backend logic lives.

Let's go **step-by-step**.

🔹 1. **Import Required Libraries**

```
from flask import Flask, request, jsonify from flask_cors import CORS import
spacy import fitz # PyMuPDF for PDF parsing
```

Explanation:

- `Flask` : Creates your API app
- `request` : Lets you get data from the frontend
- `jsonify` : Returns data in JSON format
- `CORS` : Allows frontend to talk to backend
- `spacy` : For NLP (Natural Language Processing)
- `fitz` : Used to read PDF files

🔹 2. **Set up the Flask App**

```
app = Flask(__name__) CORS(app)
```

- `Flask(__name__)` : Initializes your app
- `CORS(app)` : Enables cross-origin requests (lets React talk to Flask)

🔹 3. **Load the NLP Model**

```
nlp = spacy.load("en_core_web_sm")
```

✅ This loads a small English model from spaCy which helps detect **skills**, **entities**, and **tokens** in the resume/JD.

🔹 4. **Function to Extract Text from PDFs**

```
def extract_text_from_pdf(file_stream): with
fitz.open(stream=file_stream.read(), filetype="pdf") as doc: text = "" for
page in doc: text += page.get_text() return text
```

What this does:

- Reads each page of the PDF
- Extracts text
- Combines it all into one string

🔹 5. **Skill Extraction Function**

```
def extract_skills(text): doc = nlp(text) skills = set() for token in doc: if
token.pos_ in ["NOUN", "PROPN"]: skills.add(token.text.lower()) return skills
```

✅ This analyzes the text and keeps words that are **nouns** or **proper nouns** — most **skills** fall into these categories (e.g., Python, Java, Machine Learning, Django).

🔹 6. Route to Evaluate Resume vs JD

```
@app.route("/evaluate", methods=["POST"]) def evaluate_resume(): resume_file = request.files["resume"] jd_file = request.files["jd"] resume_text = extract_text_from_pdf(resume_file) jd_text = extract_text_from_pdf(jd_file) resume_skills = extract_skills(resume_text) jd_skills = extract_skills(jd_text) matched_skills = resume_skills.intersection(jd_skills) missing_skills = jd_skills - resume_skills score = len(matched_skills) / max(len(jd_skills), 1) * 100 return jsonify({ "score": round(score, 2), "matched_skills": list(matched_skills), "missing_skills": list(missing_skills) })
```

🔍 What Happens Here:

1. Gets both files from the frontend
2. Extracts text using `fitz`
3. Finds skills using `spaCy`
4. Compares resume skills with job description skills:
   - `intersection` : skills present in both
   - `difference` : what's missing
5. Calculates a **score**
6. Returns a **JSON response** like this:

```
{ "score": 68.75, "matched_skills": ["python", "nlp", "sql"], "missing_skills": ["docker", "tensorflow"] }
```

🔹 7. Run the Flask App

```
if __name__ == "__main__": app.run(debug=True)
```

- Starts your server at `http://localhost:5000`
- `debug=True` shows helpful errors if something breaks

✅ Final Result

When you run this, your Flask backend becomes an **intelligent scoring engine**.
It:

- Reads uploaded PDFs
- Detects skills
- Scores how good the resume matches the JD

🧪 Want to Test It Without Frontend?

You can use **Postman** or **cURL** to test the `/evaluate` API by uploading two PDFs and seeing the result.

🛠️ Bonus: Virtual Environment Recap

You created a virtual environment so all these Python libraries (Flask, spacy, fitz) are installed **only for this project**.

Run it once when setting up:

```
python -m venv venv venv\Scripts\activate # for Windows pip install -r requirements.txt
```

After that, you only need to activate the `venv` when working on this project.

✅ You're Ready for Chapter 3

Would you like to go to:

👉 **Chapter 3: How spaCy + NLP Works (Behind the Scenes of Skill Extraction)**
or
👉 **Chapter 4: Frontend (React Code Step-by-Step)**

Which one?

chapter 3

Great! Let's dive into **Chapter 3: Frontend (Client) – React.js Explained Step by Step**.

🧠 Chapter 3: Frontend (Client) – React.js Step by Step

Your project uses **React.js** for the frontend – the part users interact with (uploading resume, viewing score, etc.).

🧾 What is React?

React is a **JavaScript library** for building **user interfaces** using reusable components. It helps you create fast and interactive web applications.

🔍 Project Folder Structure

```
client/
├── public/
│   └── index.html          ← Main HTML file
├── src/
│   ├── App.js              ← Main component
│   ├── index.js            ← Entry point of React app
│   ├── components/         ← You may place UI components here (if you add more)
├── package.json            ← Dependencies & scripts
└── README.md
```

🚀 How React Works (In Simple Words)

1. **index.js** is the entry point.

```
import React from 'react'; import ReactDOM from 'react-dom/client'; import App from './App'; const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<App />);
```

- It tells the browser: "Render the App component into the `<div id='root'>` inside `index.html`."

2. **App.js** contains your main frontend logic.

```
function App() { return ( <div className="App"> <h1>AI Resume Evaluator</h1> {/* Upload buttons, Evaluate button, Display result */} </div> ); } export default App;
```

🛠️ Core React Concepts in Your App

Concept	Simple Explanation
Components	Blocks of UI (like LEGO bricks). `App.js` is one big component.
JSX	Looks like HTML in JS files. It's how you write UI in React.
State	Stores data like uploaded files or match score.
Hooks (`useState`)	React's way of handling state in function components.
Events	Like `onClick` for button, handles user actions.
fetch API	Used to call your Flask backend to send PDFs and get results.

📦 `package.json` and Dependencies

This file lists all frontend dependencies. For example:

```
{ "dependencies": { "react": "^18.0.0", "axios": "^1.6.0" } }
```

- Run `npm install` to download these dependencies.
- You start the frontend with `npm start` (this uses the `"start"` script from this file).

📁 public/index.html

This is the only HTML file. React puts all its UI into this:

```
<div id="root"></div>
```

Everything you build in `App.js` is injected here!

🌐 Connecting to Backend

React (frontend) sends your PDFs to Flask (backend) using `fetch()` or `axios` like this:

```
const formData = new FormData(); formData.append("resume", resumeFile);
formData.append("job", jobFile); fetch("http://localhost:5000/evaluate", {
method: "POST", body: formData }) .then(res => res.json()) .then(data =>
setMatchScore(data.match_score));
```

✅ Summary

You Learned	Example
React folder structure	`src/` , `public/` , `App.js` , etc.
How React renders UI	JSX, Components, `render()`
State and Hooks	`useState` , `useEffect`
Communicating with Flask backend	`fetch` or `axios`
How to start it	`cd client && npm start`

Would you like me to walk you through **Chapter 4: Backend Code Explanation (Flask + Python)** next?

chapter 4

Awesome! Let's get into **Chapter 4: Backend – Flask + Python Step by Step**.

🔥 Chapter 4: Backend (Server) – Flask + Python Explained Step by Step

This is where your **resume parsing, keyword comparison**, and **match score calculation** happen. The backend receives files from the frontend, processes them, and sends back results.

🗂️ Project Structure (Backend Folder)

Assuming your backend is inside a folder like `server/` or root:

```
server/
├── app.py                    ← Main Flask app
├── parser.py                 ← Handles PDF text extraction
├── requirements.txt          ← Python packages
└── (uploads/, temp/, etc.)   ← Temporary storage for files
```

🚀 Step 1: What is Flask?

Flask is a lightweight Python **web framework** that lets you build web servers (APIs) easily.

Think of it like this:

- **React** = user interface
- **Flask** = handles the logic and data behind the scenes

🔁 How the Backend Works (Overview)

1. **Frontend** sends PDF files (resume & job description) using `fetch()` or `axios`.
2. Flask receives the files.
3. PDF text is extracted.
4. Keywords are compared using NLP (spaCy or NLTK).
5. A **match score** is calculated.
6. Result is sent back to the frontend as JSON.

⚙️ Step 2: `app.py` – Main Flask App

Here's a simplified breakdown:

```
from flask import Flask, request, jsonify from flask_cors import CORS from
parser import extract_text, calculate_match_score app = Flask(__name__)
CORS(app) # Allow frontend to access backend (Cross-Origin)
@app.route("/evaluate", methods=["POST"]) def evaluate(): resume =
request.files["resume"] job = request.files["job"] resume_text =
extract_text(resume) job_text = extract_text(job) score =
calculate_match_score(resume_text, job_text) return jsonify({"match_score":
score}) if __name__ == "__main__": app.run(debug=True)
```

🔍 What's Happening:

- `@app.route("/evaluate")` : Defines an API route.
- `request.files` : Gets uploaded PDFs.
- `extract_text()` : Extracts text from PDFs.
- `calculate_match_score()` : Compares resume vs. job description using NLP.
- `jsonify(...)` : Sends response to frontend as JSON.

📄 Step 3: `parser.py` – Text Extraction and Scoring

Here's what `parser.py` might look like:

```
import fitz # PyMuPDF import spacy nlp = spacy.load("en_core_web_sm") def
extract_text(file): doc = fitz.open(stream=file.read(), filetype="pdf") text =
"" for page in doc: text += page.get_text() return text def
calculate_match_score(resume_text, job_text): resume_doc = nlp(resume_text)
job_doc = nlp(job_text) resume_words = set(token.lemma_.lower() for token in
resume_doc if token.is_alpha) job_words = set(token.lemma_.lower() for token
in job_doc if token.is_alpha) matched_words = resume_words & job_words score =
len(matched_words) / len(job_words) * 100 return round(score, 2)
```

🧠 Key Concepts:

Function	Purpose
`fitz`	Used for extracting text from PDFs
`spacy`	NLP library for lemmatization, tokenization
`lemma_`	Gets base word form (e.g., "running" → "run")

Function	Purpose
Set Intersection	Calculates how many words match
`round(score, 2)`	Keeps score to 2 decimal places

📦 Step 4: `requirements.txt`

This file lists Python libraries needed:

```
Flask
flask-cors
spacy
PyMuPDF
```

Install them using:

```
pip install -r requirements.txt
```

And download the spaCy model:

```
python -m spacy download en_core_web_sm
```

🧪 Step 5: Testing the Backend

Run the backend server:

```
python app.py
```

It will start on:

```
http://localhost:5000
```

You can test it with Postman or by using the frontend.

✅ Summary

You Learned	Description
Flask app	Handles HTTP requests
PDF upload	Comes from frontend as `request.files`
Text extraction	Uses PyMuPDF (fitz)
NLP comparison	Uses spaCy to analyze keywords
Response	JSON data sent back to frontend

🔗 Chapter Flow So Far:

✅ Chapter 1 – Overview of the Project
✅ Chapter 2 – GitHub, Setup, and Dependencies
✅ Chapter 3 – Frontend (React.js)
✅ Chapter 4 – Backend (Flask + Python)

Would you like to continue to **Chapter 5: Resume Scoring Logic (NLP Explained Simply)** next?

chapter 5

Great! Let's dive into **Chapter 5: Resume Scoring Logic – NLP Explained Simply**.

📚 Chapter 5: Resume Scoring Logic – NLP (Natural Language Processing)

This is the **brain** of your Resume Evaluator.

It decides **how well a resume matches a job description** using AI-based text processing.
Here's a step-by-step breakdown.

🔍 What is NLP?

> **Natural Language Processing (NLP)** is a field of AI that helps computers understand and process human language.

In our case:

- NLP helps **analyze the resume and job description**.
- It identifies **important keywords**, their **meanings**, and checks for **similarity**.

We use **spaCy**, a powerful NLP library in Python.

🧠 Step-by-Step Scoring Logic

Step 1: Extract Text from Both PDFs

From Chapter 4:

```
resume_text = extract_text(resume.pdf) job_text =
extract_text(job_description.pdf)
```

Step 2: Process the Texts with spaCy NLP

```
nlp = spacy.load("en_core_web_sm") resume_doc = nlp(resume_text) job_doc =
nlp(job_text)
```

spaCy will:

- Tokenize the text (split into words)
- Lemmatize (convert “running”, “ran” → “run”)
- Remove punctuations, etc.

Step 3: Extract Meaningful Words

We want only **useful keywords** (skip “and”, “the”, numbers, etc.)

```
resume_words = set(token.lemma_.lower() for token in resume_doc if
token.is_alpha and not token.is_stop) job_words = set(token.lemma_.lower() for
token in job_doc if token.is_alpha and not token.is_stop)
```

- `is_alpha` : includes only alphabetic words.
- `is_stop` : removes common stopwords like "the", "is", "a".

Step 4: Match Keywords

```
matched_keywords = resume_words & job_words
```

This finds the **intersection** between both sets — the shared keywords.

Example:

Resume Words	Job Description Words
python, flask, ai	flask, django, python

✅ Matches: python, flask

Step 5: Calculate Match Score

```
score = len(matched_keywords) / len(job_words) * 100
```

This gives a **percentage** match.

- If `job_words = 20` and `matched = 10` → Score = 50%
- The higher the score, the better the resume fits the job.

📈 Final Function Recap

```
def calculate_match_score(resume_text, job_text): resume_doc =
nlp(resume_text) job_doc = nlp(job_text) resume_words =
set(token.lemma_.lower() for token in resume_doc if token.is_alpha and not
token.is_stop) job_words = set(token.lemma_.lower() for token in job_doc if
token.is_alpha and not token.is_stop) matched = resume_words & job_words score
= len(matched) / len(job_words) * 100 return round(score, 2)
```

📌 Bonus: You Can Also Display Matched Keywords

To show the user **which keywords were matched**, return them too:

```
return jsonify({ "match_score": score, "matched_keywords": list(matched) })
```

🧠 Improvements You Can Add Later

Feature	What It Does
Skill weightage	Give more weight to skills like "Python", "React", etc.
Fuzzy Matching	Match similar words like “ML” and “machine learning”
Section extraction	Separate "Education", "Experience", "Skills" sections
Named Entity Recognition (NER)	Extract things like tools, company names, etc.
Word Embeddings	Use vector similarity for smarter matching (using spaCy or Sentence Transformers)

✅ Summary

You Learned	Description
NLP Basics	spaCy processes text and removes noise
Lemmatization	Groups similar words (run, running, ran → run)
Keyword Matching	Finds common, meaningful words between resume & JD
Scoring	% of JD keywords that exist in the resume
Bonus	Return matched keywords to show to user

🚀 Next Up: Chapter 6 – Displaying Results and UX Tips in React

Shall we move on to that?

Yes

Awesome, Ragul! Let's move on to:

🎨 Chapter 6: Displaying Results & UX in React

This chapter explains **how your frontend (React)** takes the data from the backend and **displays** it nicely to the user.

📦 1. What Your Backend Sends

From Chapter 5, your Flask backend returns:

```
{ "match_score": 76.5, "matched_keywords": ["python", "flask", "api", "ml"] }
```

This is the **response JSON** your React app will receive after uploading the resume and job description.

🧪 2. React Receives the Result

In your React component (probably in `App.js` or `Evaluator.js`), after form submission:

```
const handleSubmit = async (e) => { e.preventDefault(); const formData = new
FormData(); formData.append("resume", resumeFile);
formData.append("job_description", jdFile); const response = await
fetch("http://localhost:5000/api/score", { method: "POST", body: formData, });
const data = await response.json(); setScore(data.match_score);
setKeywords(data.matched_keywords); };
```

🎯 3. Displaying the Score

Below your file input form, show the score:

```
{score !== null && ( <div className="result"> <h3>Match Score: {score}%</h3>
</div> )}
```

🧠 4. Displaying Matched Keywords

```
{keywords.length > 0 && ( <div className="keywords"> <h4>Matched Keywords:
</h4> <ul> {keywords.map((word, index) => ( <li key={index}>{word}</li> ))}
</ul> </div> )}
```

This makes your app **interactive and informative** for users.

💄 5. Make It Pretty (Optional UI Styling)

You can style it using CSS or a framework like Tailwind or Bootstrap.

Example with plain CSS:

```
.result { margin-top: 20px; font-size: 20px; color: #2e7d32; } .keywords ul {
list-style-type: none; padding: 0; } .keywords li { display: inline-block;
background-color: #e0f7fa; margin: 5px; padding: 5px 10px; border-radius: 4px;
}
```

🧪 6. Optional: Add Loading State

You can add a spinner or message while waiting for a response:

```
{loading && <p>Evaluating resume...</p>}
```

Set loading in your submit function:

```
setLoading(true); // fetch... setLoading(false);
```

📌 Final Result – Example Output

```
Match Score: 76.5% Matched Keywords: [ python ] [ flask ] [ api ] [ ml ]
```

✅ Summary

Feature	What You Did
Form submission	Used FormData to send files
Fetch backend response	Called the Flask `/api/score` route
Show score	Displayed % match to user
Show matched keywords	Helped users improve resumes
Styled UI	Made your app user-friendly

🚀 Next Up: Chapter 7 – Deploying Your Project (Frontend + Backend)

Ready to learn how to host it online and show it to the world?

Yes

Great, Ragul! Let's dive into:

🌐 Chapter 7: Deploying Your Resume Evaluator Project (Full Stack)

This chapter teaches you how to **host your project online** so others (like recruiters, friends, or interviewers) can try it!

🚧 Your Project Structure

You have two main folders:

```
resume-evaluator/
├── client/       ← React frontend
├── server/       ← Flask backend
```

We'll deploy both:

- **Frontend** → on **Vercel** (easy for React)
- **Backend** → on **Render** (free and supports Python easily)

🟢 Step 1: Deploy Backend (Flask) to Render

1.1. 🧪 Make sure your `server/` has:

- A `requirements.txt`
- A `Procfile` with this:

```
web: gunicorn app:app
```

- In `app.py`, make sure CORS is allowed:

```
from flask_cors import CORS app = Flask(__name__) CORS(app)
```

1.2. ✅ Push `server/` to GitHub

If not done already:

```
cd server git init git remote add origin
https://github.com/YourUsername/resume-evaluator-backend.git git add . git
commit -m "Initial backend commit" git push -u origin main
```

1.3. 🚀 Deploy on Render

1. Go to https://render.com
2. Sign up / log in
3. Click "**New Web Service**"
4. Connect your GitHub
5. Select the `resume-evaluator-backend` repo

6. Set:
    - **Build command:** `pip install -r requirements.txt`
    - **Start command:** `gunicorn app:app`
    - **Python version:** 3.10+
7. Click **Deploy**

➡️ After success, copy your Render backend URL (e.g., `https://resume-api.onrender.com`)

🔵 Step 2: Deploy Frontend (React) to Vercel

2.1. ✅ Push `client/` to GitHub

```
cd client git init git remote add origin
https://github.com/YourUsername/resume-evaluator-frontend.git git add . git
commit -m "Initial frontend commit" git push -u origin main
```

2.2. 🌐 Deploy to Vercel

1. Go to https://vercel.com
2. Sign in with GitHub
3. Click "**Add New Project**"
4. Select your `resume-evaluator-frontend` repo
5. Click **Deploy**

2.3. 🔁 Fix the API URL in React

In your React code (`App.js` or `.env` file):

```
fetch("https://YOUR-BACKEND-URL.onrender.com/api/score", ...)
```

Or add an environment variable:

```
REACT_APP_API_URL=https://your-backend.onrender.com
```

Then use:

```
fetch(`${process.env.REACT_APP_API_URL}/api/score`, ...)
```

Rebuild & redeploy the frontend if you change env vars.

✅ Done! Your project is now live.

Example:

- Frontend (React): `https://resume-evaluator.vercel.app`
- Backend (Flask): `https://resume-api.onrender.com`

🔒 Bonus Tip: Handle CORS Correctly

Ensure Flask backend allows frontend access:

```
CORS(app, origins="*") # or better: frontend URL only
```

🎉 You've Deployed Your Full Stack Project!

Part	Status
Flask backend	✅ Render
React frontend	✅ Vercel
GitHub repos	✅ Versioned
Live URL	✅ Ready to share!

Next Up 👉 **Chapter 8: Learning the Code Internals (Line-by-Line Understanding)**
Would you like to go line by line through the backend or frontend first?

Chapter 8

Awesome, Ragul! Let's start:

🧠 Chapter 8: Understanding Your Resume Evaluator – Line by Line

You'll now **understand how your full-stack AI Resume Evaluator actually works**, both **backend (Flask)** and **frontend (React)** – step by step.

🔴 Part 1: Backend (Flask + Python + NLP)

Let's break down your `server/app.py` — this is where the **resume gets scored**.

✅ Code Walkthrough – `app.py`

```
from flask import Flask, request, jsonify from flask_cors import CORS import
spacy
```

🔍 **Explanation**:

- `Flask` : the web server
- `request` : to get incoming data (like resume text)
- `jsonify` : to return results as JSON
- `CORS` : allows frontend to talk to backend
- `spacy` : the NLP model to process resumes

```
app = Flask(__name__) CORS(app)
```

✅ Creates a Flask app and allows it to receive requests from the React frontend (important for deployment).

```
nlp = spacy.load("en_core_web_sm")
```

💡 Loads the **small English NLP model** from spaCy to analyze text (keywords, nouns, verbs, etc.).

```
@app.route("/api/score", methods=["POST"]) def score_resume():
```

🎯 Defines an API route `POST /api/score` that your frontend will call.

```
data = request.get_json() resume_text = data.get("resume", "") job_desc_text = data.get("job_description", "")
```

📩 Reads the **resume text** and **job description text** sent from frontend via JSON.

```
resume_doc = nlp(resume_text) job_desc_doc = nlp(job_desc_text)
```

🔍 Processes both texts using spaCy NLP. Now we can extract things like:

- Tokens
- Named Entities
- Keywords
- Lemmas

```
resume_tokens = set([token.lemma_.lower() for token in resume_doc if token.is_alpha and not token.is_stop]) job_desc_tokens = set([token.lemma_.lower() for token in job_desc_doc if token.is_alpha and not token.is_stop])
```

✅ Extracts only the **meaningful words** (lemmas), removes stopwords (like “the”, “is”), and makes lowercase.

```
common_tokens = resume_tokens.intersection(job_desc_tokens) score =
round((len(common_tokens) / len(job_desc_tokens)) * 100, 2)
```

🧠 Calculates the **match score** as a percentage:

- More overlapping keywords → higher score

```
return jsonify({"score": score, "matched_keywords": list(common_tokens)})
```

📤 Sends back a JSON result to frontend with:

- `score` : keyword match %
- `matched_keywords` : words found in both resume and JD

✅ That's your full backend logic!

💡 Summary of Backend Flow

Step	What happens
1️⃣	Receive resume + job description from frontend
2️⃣	Clean & process text using spaCy
3️⃣	Find matching keywords
4️⃣	Calculate match %
5️⃣	Return score + keywords

Do you want to continue to **Frontend Code (React)** in Chapter 9 next?

Or pause here and do a quick **backend recap quiz or diagram**?

Chapter 9

Great, Ragul! Let's move to:

🎨 Chapter 9: Understanding Your Frontend (React) – Step by Step

This is where the **user uploads their resume**, enters the **job description**, clicks a button, and sees a **score**.

Your frontend files are inside the `client` folder.

🗂️ Key Files in Frontend (`client` folder):

File / Folder	Purpose
`App.js`	Main logic and components
`ResumeForm.js` (if used)	Handles file/text inputs
`package.json`	Keeps track of React packages
`index.js`	Entry point for React app
`public/`	HTML shell file
`src/`	Your source code (JSX, CSS)

We'll focus on `App.js` (your core logic).

⚙️ Code Walkthrough – `App.js` (or main component)

Let's say your `App.js` looks like this:

```
import React, { useState } from "react"; import axios from "axios";
```

📦 Loads React hooks and `axios` for sending requests to backend.

```
function App() { const [resumeText, setResumeText] = useState(""); const
[jobDescText, setJobDescText] = useState(""); const [score, setScore] =
useState(null); const [keywords, setKeywords] = useState([]);
```

🧠 Initializes React **state**:

- `resumeText` : text input of resume
- `jobDescText` : job description input
- `score` : match score from backend
- `keywords` : matched keywords from backend

📝 User Form Input

```
const handleSubmit = async (e) => { e.preventDefault(); try { const response =
await axios.post("http://localhost:5000/api/score", { resume: resumeText,
job_description: jobDescText, }); setScore(response.data.score);
setKeywords(response.data.matched_keywords); } catch (error) {
console.error("Error:", error); } };
```

✅ When user clicks "Evaluate":

- Sends `resumeText` and `jobDescText` to Flask
- Receives score and keywords
- Updates frontend with the results

👷 JSX Form UI

```
return ( <div className="App"> <h1>Resume Evaluator</h1> <form onSubmit=
{handleSubmit}> <textarea value={resumeText} onChange={(e) =>
setResumeText(e.target.value)} placeholder="Paste your resume text here" />
<textarea value={jobDescText} onChange={(e) => setJobDescText(e.target.value)}
placeholder="Paste job description here" /> <button
type="submit">Evaluate</button> </form>
```

🎯 This displays:

- 2 input boxes (Resume, Job Description)
- Button to submit
- React hooks handle changes

✅ Showing the Results

```
{score && ( <div> <h2>Score: {score}%</h2> <h3>Matched Keywords:</h3> <ul>
{keywords.map((word, index) => ( <li key={index}>{word}</li> ))} </ul> </div>
)} </div> );
```

🟢 If score is available:

- Show score
- List all matched keywords from backend

🔄 End-to-End Flow (Frontend + Backend)

```
User → Enters Resume + JD → Clicks Evaluate → Frontend (React) calls Flask API
→ Backend (Python) processes NLP → Sends score + keywords back → React shows
score + matches
```

💡 Technologies Used in Frontend

Tool	Purpose
React.js	UI Framework
axios	To send HTTP POST request
useState	React state management
JSX	For UI markup