project-root/

│── backend/

│ │── main.py # FastAPI entry point

│ │── config.py # Configuration settings

│ │── models.py # Database models using SQLAlchemy

│ │── database.py # Database connection setup

│ │── api/

│ │ │── auth.py # User authentication APIs

│ │ │── jobs.py # Job search and details APIs

│ │ │── recommendations.py # AI-powered job recommendation APIs

│ │ │── resume.py # Resume processing APIs

│ │── ai/

│ │ │── matching.py # Job matching logic using FAISS

│ │ │── preprocessing.py # Resume text extraction & feature engineering

│ │── utils/

│ │ │── security.py # Authentication helpers

│ │ │── data\_loader.py # Functions to load and update job data

│── frontend/

│ │── src/

│ │ │── components/

│ │ │ │── JobList.js # UI for displaying job listings

│ │ │ │── Profile.js # User profile management UI

│ │ │ │── ResumeUpload.js # Resume upload form

│ │ │── pages/

│ │ │ │── Home.js # Homepage

│ │ │ │── Login.js # Login page

│ │ │ │── Dashboard.js # User dashboard with recommendations

│ │ │── api/

│ │ │ │── api.js # Handles API requests to backend

│ │ │── App.js # Main React App entry point

│ │── package.json # Frontend dependencies

│ │── vite.config.js # Vite configuration

│── requirements.txt # Python dependencies

│── README.md # Project documentation

│── .env # Environment variables

│── Dockerfile # Docker setup for backend

# Backend (FastAPI) - Example Basic Functions

# main.py

from fastapi import FastAPI

from api import auth, jobs, recommendations, resume

app = FastAPI()

app.include\_router(auth.router, prefix="/auth", tags=["Authentication"])

app.include\_router(jobs.router, prefix="/jobs", tags=["Job Search"])

app.include\_router(recommendations.router, prefix="/recommendations", tags=["AI Recommendations"])

app.include\_router(resume.router, prefix="/resume", tags=["Resume Processing"])

# database.py

from sqlalchemy import create\_engine, MetaData

from sqlalchemy.orm import sessionmaker

DATABASE\_URL = "sqlite:///./job\_matcher.db"

engine = create\_engine(DATABASE\_URL, connect\_args={"check\_same\_thread": False})

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

metadata = MetaData()

# models.py

from sqlalchemy import Column, Integer, String, ForeignKey

from sqlalchemy.orm import relationship

from database import Base

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True, index=True)

name = Column(String, index=True)

email = Column(String, unique=True, index=True)

password\_hash = Column(String)

class Job(Base):

\_\_tablename\_\_ = "jobs"

id = Column(Integer, primary\_key=True, index=True)

title = Column(String, index=True)

company = Column(String)

description = Column(String)

location = Column(String)

class Application(Base):

\_\_tablename\_\_ = "applications"

id = Column(Integer, primary\_key=True, index=True)

user\_id = Column(Integer, ForeignKey("users.id"))

job\_id = Column(Integer, ForeignKey("jobs.id"))

status = Column(String)

# auth.py (Example API)

from fastapi import APIRouter, Depends

from pydantic import BaseModel

from sqlalchemy.orm import Session

from database import SessionLocal

from models import User

import bcrypt

router = APIRouter()

class UserRegister(BaseModel):

name: str

email: str

password: str

@router.post("/register")

def register\_user(user: UserRegister, db: Session = Depends(SessionLocal)):

hashed\_password = bcrypt.hashpw(user.password.encode('utf-8'), bcrypt.gensalt())

new\_user = User(name=user.name, email=user.email, password\_hash=hashed\_password)

db.add(new\_user)

db.commit()

return {"message": "User registered successfully"}

# Frontend (React) - Example API Call

# api/api.js

import axios from 'axios';

const API\_URL = 'http://localhost:8000';

export const login = async (email, password) => {

return await axios.post(`${API\_URL}/auth/login`, { email, password });

};

export const fetchJobs = async () => {

return await axios.get(`${API\_URL}/jobs`);

};

export const fetchRecommendations = async (userId) => {

return await axios.get(`${API\_URL}/recommendations/${userId}`);

};

# App.js

import React from 'react';

import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';

import Home from './pages/Home';

import Login from './pages/Login';

import Dashboard from './pages/Dashboard';

function App() {

return (

<Router>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/login" element={<Login />} />

<Route path="/dashboard" element={<Dashboard />} />

</Routes>

</Router>

);

}

export default App;

# README.md

# Job Matcher AI

## Overview

An AI-powered job recommendation system using FastAPI and React.

## Installation

1. Clone the repository: `git clone <repo\_url>`

2. Set up backend:

```sh

cd backend

pip install -r requirements.txt

uvicorn main:app --reload

```

3. Set up frontend:

```sh

cd frontend

npm install

npm run dev

```

4. Access the app at `http://localhost:3000`

## Features

- AI-powered job recommendations

- Resume parsing and job matching

- User authentication and profile management

# Job Matcher AI - System Design Document

## 1. Overview

The Job Matcher AI project is an intelligent job recommendation system leveraging AI agents, LLMs, and recommendation algorithms to help job seekers find the best-matched positions. The system consists of a backend (FastAPI), a frontend (React + Vite), and an AI module handling job matching, search, and recommendations.

## 2. System Architecture

### Components:

- \*\*Frontend:\*\* React.js for UI, Vite for fast local development

- \*\*Backend:\*\* FastAPI for handling API requests, SQLite as a local database

- \*\*AI Module:\*\* Machine learning models, LLM integration, and vector-based search using FAISS

- \*\*Database:\*\* SQLite (can be upgraded to PostgreSQL if needed)

### Architecture Diagram:

```

[Frontend (React)] <--> [Backend (FastAPI)] <--> [Database (SQLite)]

|

v

[AI Module: ML & LLM]

```

## 3. Features & Functionalities

- \*\*Job Search:\*\* Search for jobs based on keywords and filters

- \*\*Job Recommendation:\*\* AI-based personalized job matching

- \*\*Resume Analysis:\*\* Extract key skills and recommend jobs

- \*\*Profile Management:\*\* Store user preferences and job history

- \*\*Application Tracking:\*\* Track applied jobs and recommendations

## 4. API Endpoints

### Authentication & User Management

- `POST /register` - Register a new user (`register\_user()`)

- `POST /login` - Authenticate user and generate token (`authenticate\_user()`)

- `GET /profile/{user\_id}` - Get user profile details (`get\_user\_profile()`)

- `PUT /profile/{user\_id}` - Update user profile information (`update\_user\_profile()`)

### Job Search & Recommendations

- `GET /jobs?query=keyword&location=city` - Search jobs by keyword & location (`search\_jobs()`)

- `GET /job/{job\_id}` - Get job details (`get\_job\_details()`)

- `GET /recommendations/{user\_id}` - Get personalized job recommendations (`get\_recommendations()`)

### Resume & Application Tracking

- `POST /resume/upload` - Upload and process resume (`upload\_resume()`)

- `GET /applications/{user\_id}` - Get user job applications history (`get\_user\_applications()`)

- `POST /apply/{job\_id}` - Apply for a job (`apply\_for\_job()`)

## 5. AI & Recommendation Logic

- \*\*Keyword-based Search:\*\* Retrieve jobs using traditional search

- \*\*Vector-based Matching (FAISS):\*\* Encode job descriptions & resumes for similarity search

- \*\*ML-based Recommendations:\*\* Personalized job suggestions based on user behavior

### AI Pipeline Workflow:

1. \*\*Data Ingestion:\*\* Collect job listings and user resumes

2. \*\*Preprocessing:\*\* Tokenization, vectorization, feature extraction

3. \*\*Embedding Generation:\*\* Convert job descriptions and resumes into vector space using FAISS

4. \*\*Matching Algorithm:\*\* Compare job vectors with user profile vectors

5. \*\*Ranking & Filtering:\*\* Rank results based on similarity score and relevance

6. \*\*Recommendation Output:\*\* Return top-N job recommendations to the user

## 6. Database Schema

### Users Table

| id | name | email | skills | experience |

|----|------|-------|--------|------------|

| INT | TEXT | TEXT | TEXT | TEXT |

### Jobs Table

| id | title | company | description | location |

|----|-------|---------|-------------|----------|

| INT | TEXT | TEXT | TEXT | TEXT |

### Applications Table

| id | user\_id | job\_id | status |

|----|---------|--------|--------|

| INT | INT | INT | TEXT |

### Resumes Table

| id | user\_id | file\_path | parsed\_text |

|----|---------|-----------|-------------|

| INT | INT | TEXT | TEXT |

## 7. Tech Stack

- \*\*Frontend:\*\* React.js, Tailwind CSS

- \*\*Backend:\*\* FastAPI, SQLite, SQLAlchemy

- \*\*AI Module:\*\* LangChain, FAISS, scikit-learn

- \*\*Deployment:\*\* Local (Docker optional)

## 8. Working Flows

### Job Search Flow

1. User enters search criteria in the frontend

2. Request is sent to `/jobs` API endpoint

3. Backend queries job listings in the database

4. AI module enhances search results with ranking

5. Results are returned to the frontend

### Job Recommendation Flow

1. User logs in and accesses recommendations

2. Request is sent to `/recommendations/{user\_id}` API

3. Backend retrieves user profile and past interactions

4. AI module runs similarity matching using FAISS

5. Top-N recommended jobs are returned

### Resume Upload Flow

1. User uploads resume via `/resume/upload`

2. Backend stores the file and extracts text

3. Extracted skills and experience are stored in the database

4. AI module updates user profile for better recommendations

## 9. Future Enhancements

- Add support for real-time job alerts

- Upgrade to PostgreSQL for better scalability

- Implement deep learning models for enhanced recommendation

- Multi-language support for job matching

## 10. Conclusion

This system provides an efficient and AI-powered way to connect job seekers with relevant opportunities through intelligent search and recommendation algorithms.

**JOBSEEKERAI Project Summary**

**Overview**

JOBSEEKERAI is an AI-powered job search assistant that combines conversational AI with intelligent job matching capabilities. The system uses Llama2 for natural language understanding and multiple job search APIs to help users find relevant job opportunities through a chat interface.

**Technical Architecture**

**Backend (FastAPI/Python)**

 **FastAPI Framework**: RESTful API server with WebSocket support

 **Ollama Integration**: Local Llama2 model for conversational AI

 **Job Search APIs**:

 JSearch API (RapidAPI) for comprehensive job search

 Remotive API for remote tech jobs

 **Job Matching System**: Intelligent parameter extraction from conversations

**Frontend (React/TypeScript)**

 React-based chat interface

 Real-time message updates

 TypeScript for type safety

 Responsive design for multiple devices

**Key Features**

**1. Conversational AI**

 Natural language processing using Llama2

 Context-aware responses

 Job-focused conversation guidance

**2. Intelligent Job Matching**

 Extracts job preferences from conversation:

 Position/Title

 Location

 Salary requirements

 Experience level

 Job type (remote, hybrid, etc.)

 Maintains conversation history for context

**3. Multi-Source Job Search**

 Aggregates results from multiple job boards

 Standardized job listing format

 Real-time job data fetching

**4. Smart Response System**

 Combines AI responses with job listings

 Provides formatted job details

 Supports search refinement

**API Endpoints**

POST /chat

- Handles chat messages

- Returns AI responses and job matches

GET /health

- Health check endpoint

**Data Flow**

 User sends message

 Ollama LLM processes message

 Job intent detection

 Parameter extraction

 Job API searches

 Response combination

 Formatted reply

**Environment Setup**

# Backend dependencies

pip install fastapi uvicorn python-dotenv aiohttp requests langchain

# Required API keys (.env)

JSEARCH\_API\_KEY=xxx

OPENAI\_API\_KEY=xxx

**Running the Project**

**Backend**

cd backend

uvicorn main:app --reload --port 8000

**Ollama**

# Separate terminal

ollama run llama2

**Future Improvements**

 Resume parsing and matching

 User preferences storage

 Job application tracking

 More job source integrations

 Enhanced conversation context

 Personalized job recommendations

**Technologies Used**

 **Backend**: Python, FastAPI, Llama2

 **Frontend**: React, TypeScript

 **APIs**: JSearch, Remotive

 **AI**: Ollama, LangChain

 **Database**: (Future implementation)

**Security Features**

 CORS protection

 API key management

 Error handling

 Rate limiting (via APIs)

This project demonstrates the power of combining conversational AI with practical job search functionality to create a more natural and effective job search experience.