Documentation for `data_models.py`

A comprehensive guide to the data models used in the project.

Overview

This file, `data_models.py`, defines several data models using Pydantic's `BaseModel` and Python's `TypedDict`. These models are crucial for structuring and validating data within the application, especially for the Resume Analysis Agent (RAR), Cross-Job Comparison Agent (CJC), and the overall multi-agent system. They also define the structure for API requests and responses. Let's dive in!

ResumeFeedback (RAR Agent)

This model represents the feedback generated by the Resume Analysis Agent (RAR) for a candidate's resume. It includes an analysis of the resume, scores in different categories, overall score, key strengths, and areas for improvement.

Fields:

candidate_name (`AnyStr`): The name of the candidate.
analysis (`AnyStr`): A textual analysis of the resume.
scores (`Dict[AnyStr, int]`): A dictionary containing scores for different aspects of the resume. For example: `{"formatting": 8, "skills": 9}`.
total_score (`int`): The overall score of the resume.
key_strengths (`List[AnyStr]`): A list of key strengths identified in the resume.
areas_for_improvement (`List[AnyStr]`): A list of areas where the resume can be improved.

Example:

Imagine the RAR agent analyzes John Doe's resume. The `ResumeFeedback` model might look like this (in Python dictionary form for illustration):

```
        {
            "candidate_name": "John Doe",
            "analysis": "The resume is well-structured and highlights relevant skills.",
            "scores": {"formatting": 9, "skills": 8, "experience": 7},
            "total_score": 80,
            "key_strengths": ["Strong technical skills", "Clear and concise formatting"],
            "areas_for_improvement": ["Quantify achievements", "Add more detail to project descriptions"]
        }
```

JobResumeMatch (CJC Agent)

This model is used by the Cross-Job Comparison (CJC) agent to represent the matching score and explanation between a job description and a candidate's resume.

Fields:

job_description_name (`str`): The name of the job description.
candidate_name (`str`): The name of the candidate.
match_score (`float`): The matching score between the job description and resume (e.g., 0.85 for 85%).
match_explanation (`str`): An explanation of why the resume and job description were matched.

Example:

Consider matching John Doe's resume to a "Software Engineer" job. A possible `JobResumeMatch` could be:

```
        {
            "job_description_name": "Software Engineer",
            "candidate_name": "John Doe",
            "match_score": 0.85,
            "match_explanation": "The candidate's skills and experience align well with the requirements
of the Software Engineer role."
        }
```

CrossJobMatchResult

This model aggregates the results of matching multiple jobs and resumes, providing the best matches and an overall recommendation.

Fields:

job_resume_matches (`List[JobResumeMatch]`): A list of `JobResumeMatch` objects, representing all job-resume pairings.
best_matches_per_job (`Dict[str, str]`): A dictionary mapping each job name to the name of the best-matching resume.
best_matches_per_resume (`Dict[str, str]`): A dictionary mapping each resume name to the name of the best-matching job.
overall_recommendation (`str`): An overall recommendation based on the job-resume matches.

Example:

An example of `CrossJobMatchResult` could look like this:

```
        {
            "job_resume_matches": [
                {"job_description_name": "Software Engineer", "candidate_name": "John Doe",
"match_score": 0.85, "match_explanation": "Skills align."},
                {"job_description_name": "Data Scientist", "candidate_name": "Jane Smith", "match_score":
0.90, "match_explanation": "Experience matches."}
            ],
            "best_matches_per_job": {"Software Engineer": "John Doe", "Data Scientist": "Jane Smith"},
            "best_matches_per_resume": {"John Doe": "Software Engineer", "Jane Smith": "Data Scientist"},
            "overall_recommendation": "John Doe is a strong candidate for Software Engineer, and Jane
Smith is a great fit for Data Scientist."
        }
```

MultiJobComparisonState

This `TypedDict` represents the overall state of the multi-agent system. It includes the job openings, resumes, all candidate rankings per job opening, final recommendations, and processed job descriptions.

Fields:

job_openings (`Annotated[List[Dict[str, Any]], "List of Job Openings"]`): A list of dictionaries, each representing a job opening.
resumes (`Annotated[List[Dict[str, Any]], "List of Candidate's Resumes"]`): A list of dictionaries, each representing a candidate's resume.
all_rankings (`Annotated[Dict[str, List[ResumeFeedback]], "Ranking of all Candidates per Job Opening"]`): A dictionary where keys are job opening names and values are lists of `ResumeFeedback` objects, representing the ranking of candidates for that job.
final_recommendations (`CrossJobMatchResult`): The final recommendations generated by the system, using the `CrossJobMatchResult` model.
processed_job_description (`Annotated[List[AnyStr], operator.add]`): A list of processed job descriptions.

Example:

A simplified example of `MultiJobComparisonState` :

```
        {
            "job_openings": [{"name": "Software Engineer", "description": "Develop software."}, {"name":
"Data Scientist", "description": "Analyze data."}],            "resumes": [{"name": "John Doe",
"content": "Software skills."}, {"name": "Jane Smith", "content": "Data analysis skills."}],
            "all_rankings": {
                "Software Engineer": [{"candidate_name": "John Doe", "analysis": "Good skills", "scores":
{}, "total_score": 90, "key_strengths": [], "areas_for_improvement": []}],
                "Data Scientist": [{"candidate_name": "Jane Smith", "analysis": "Good experience",
"scores": {}, "total_score": 95, "key_strengths": [], "areas_for_improvement": []}]
            },
            "final_recommendations": {
                "job_resume_matches": [],
            "best_matches_per_job": {},
            "best_matches_per_resume": {},
            "overall_recommendation": "Recommend John and Jane."
        },
        "processed_job_description": ["software engineer description", "data scientist description"]
    }
```

AnalysisRequest

This model defines the structure for the API request to start the analysis process. It contains lists of job openings and resumes.

Fields:

- job_openings (`List[Dict[AnyStr, Any]]`): A list of dictionaries, each representing a job opening. Each dictionary contains the job details.
- resumes (`List[Dict[AnyStr, Any]]`): A list of dictionaries, each representing a resume. Each dictionary contains the resume details.

Example:

An example of `AnalysisRequest` structure:

```
    {
        "job_openings": [{"name": "Software Engineer", "description": "Looking for a skilled software engineer."}, {"name": "Data Scientist", "description": "Seeking a data scientist with experience in machine learning."}],
        "resumes": [{"name": "Alice", "content": "Experienced in software development."}, {"name": "Bob", "content": "Proficient in data analysis and machine learning."}]
    }
```

StatusResponse

This model defines the structure for the API response when requesting the status of an analysis. It includes a trace ID, status, progress (optional), and results (optional).

Fields:

- trace_id (`str`): A unique identifier for the analysis request.
- status (`str`): The current status of the analysis (e.g., "running", "completed", "error").
- progress (`Optional[Dict[str, str]]`): An optional dictionary containing progress information (e.g., {"step": "Analyzing resumes", "percentage": "50%"}).
- results (`Optional[Dict[str, Any]]`): An optional dictionary containing the results of the analysis.

Example:

An example of a `StatusResponse` when the analysis is running:

```
    {
        "trace_id": "123e4567-e89b-12d3-a456-426614174000",
        "status": "running",
        "progress": {"step": "Analyzing resumes", "percentage": "75%"},
        "results": None
    }
```

An example of a `StatusResponse` when the analysis is completed:

```
    {
        "trace_id": "123e4567-e89b-12d3-a456-426614174000",
        "status": "completed",
        "progress": {"step": "Analysis complete", "percentage": "100%"},
            "progress": {"step": "Analysis complete", "percentage": "100%"},
            "results": {"recommendations": "Alice is recommended for Software Engineer."}
        }
```

StartResponse

This model defines the structure for the API response when starting an analysis. It includes a trace ID and a message.

Fields:

- trace_id (`str`): A unique identifier for the analysis request.
- message (`str`): A message indicating the status of the start request (e.g., "Analysis started successfully").

Example:

An example of a `StartResponse` :

```
        {
            "trace_id": "123e4567-e89b-12d3-a456-426614174000",
            "message": "Analysis started successfully"
        }
```