`rar_endpoint.py` Documentation

Your friendly guide to understanding the Resume Analyzer and Reranker (RAR) API endpoint!

Overview

This file, `rar_endpoint.py`, is the heart of the AI-powered Resume Analyzer and Reranker tool's API. It uses FastAPI to create a robust and efficient API for analyzing resumes and matching them to job descriptions. It handles API requests, manages background tasks for analysis, and provides endpoints for checking job status. The API is secured with API keys and includes rate limiting to prevent abuse. Let's dive in!

Imports

The code starts by importing necessary modules. Here's a breakdown:

- INTERNAL IMPORTS:
  - `utils.CacheManager`: Manages caching of resources, like the compiled LangGraph.
  - `data_models.AnalysisRequest`, `data_models.StartResponse`, `data_models.StatusResponse`, `data_models.ResumeFeedback`, `data_models.JobResumeMatch`, `data_models.CrossJobMatchResult`: Data models defining the structure of requests and responses.
- REQUEST PROCESSING MODULE:
  - `langchain.schema.Document`: Represents a document (e.g., a resume) for processing by LangChain.
  - `uuid.uuid4`: Generates unique identifiers for tracking analysis jobs.
- API HANDLING MODULE:
  - `fastapi.FastAPI`: The core FastAPI class for creating the API application.
  - `fastapi.BackgroundTasks`: Manages tasks that run in the background, like the actual analysis.
  - `fastapi.HTTPException`: Used for raising HTTP exceptions (e.g., 404 Not Found).
  - `fastapi.Depends`, `fastapi.Security`: Dependency injection and security utilities.
  - `fastapi.Request`, `fastapi.Response`: Request and response objects.
  - `fastapi.middleware.cors.CORSMiddleware`: Handles Cross-Origin Resource Sharing (CORS).
  - `fastapi.security.APIKeyHeader`: Implements API key-based security.
  - `fastapi.responses.JSONResponse`: Creates JSON responses.
  - `slowapi.Limiter`, `slowapi._rate_limit_exceeded_handler`: Implements rate limiting.
  - `slowapi.util.get_remote_address`: Gets the client's IP address for rate limiting.
  - `slowapi.errors.RateLimitExceeded`: Exception raised when rate limit is exceeded.
  - `contextlib.asynccontextmanager`: Utility for asynchronous context management.
- UTILITIES:
  - `os`, `sys`: Standard library modules for interacting with the operating system.
  - `typing_extensions.Callable`, `typing_extensions.Dict`, `typing_extensions.List`, `typing_extensions.AnyStr`, `typing_extensions.Any`: Type hints for better code readability and maintainability.
  - `dotenv.load_dotenv`: Loads environment variables from a `.env` file.
  - `logging`: For logging events and errors.

Global Variables and Initialization

Let's look at some important global variables and initial setup:

- `cache_manager = CacheManager()`: Initializes the `CacheManager`, used for caching the compiled LangGraph to improve performance.
- `validate_extract_agent = None`: Placeholder for a validation agent (currently unused).
- `logging.basicConfig(level=logging.INFO)` and `logger = logging.getLogger(__name__)`: Configures logging to record events and errors.

`create_langgraph_app()` Function

This function is responsible for importing and initializing the LangGraph application. LangGraph is used for orchestrating the analysis workflow.

It attempts to import `create_multi_job_comparison_graph` from the `graph` module. If the import fails (e.g., during development when the graph isn't fully set up), it returns `None` and prints a warning, allowing the API to run in "mock mode".

`lifespan` Context Manager

The `lifespan` function is an asynchronous context manager used by FastAPI to manage the application's lifecycle. It's decorated with `@asynccontextmanager`.

Specifically, it attempts to initialize and cache the compiled LangGraph application using the `create_langgraph_app()` function. If the initialization fails, it logs a warning and allows the API to start in a "mock mode".

FastAPI Application Initialization

Here's where the FastAPI application is created:

Here's where the FastAPI application is created.

- `app = FastAPI(...)` : Creates a FastAPI instance with a title, description, version, and a `lifespan` context manager.

Rate Limiting Configuration

The API implements rate limiting to prevent abuse:

- `limiter = Limiter(key_func=get_remote_address, application_limits=["5/minute"])` : Initializes a `Limiter` that allows 5 requests per minute from each client IP address.
- `app.state.limiter = limiter` and `app.add_exception_handler(RateLimitExceeded, _rate_limit_exceeded_handler)` : Configures the FastAPI app to use the limiter and handle `RateLimitExceeded` exceptions.

Security Headers Middleware

The `add_security_headers` middleware adds security headers to every response to protect against common web vulnerabilities:

- `X-Content-Type-Options: nosniff` : Prevents MIME-sniffing.
- `X-Frame-Options: DENY` : Prevents clickjacking.
- `X-XSS-Protection: 1; mode=block` : Enables XSS protection.
- `Strict-Transport-Security: max-age=31536000; includeSubDomains` : Enforces HTTPS.

CORS Configuration

CORS (Cross-Origin Resource Sharing) is configured to allow requests from any origin:

- `app.add_middleware(CORSMiddleware, allow_origins=["*"], allow_credentials=True, allow_methods=["*"], allow_headers=["*"])` : Allows all origins, credentials, methods, and headers. In a production environment, you'd want to restrict `allow_origins` to specific domains.

In-Memory Job Tracking

The `jobs` dictionary is used to track the status of analysis jobs:

- `jobs = {}` : An empty dictionary to store job information, keyed by `trace_id`.

API Key Security

The API is secured with API keys. Let's break down how that works:

- `API_KEY_NAME = "DIREC-AI-RAR-API-KEY"` : Defines the name of the API key header.
- `api_key_header = APIKeyHeader(name=API_KEY_NAME, auto_error=True)` : Creates an `APIKeyHeader` instance to extract the API key from the request header.
- `API_KEYS = {os.getenv("DIREC_RAR_API_KEY"): "USER-RHYLIIEEE"}` : A dictionary mapping valid API keys to user identifiers. The API key is read from the environment variable `DIREC_RAR_API_KEY`.
- `async def get_api_key(api_key_header: str = Security(api_key_header))` : A dependency function that verifies the API key. If the key is invalid, it raises an `HTTPException`.

`run_analysis` Function

This function is the core of the analysis process. It's run as a background task.

1. It updates the job status to "running" in the `jobs` dictionary.

2. It converts the `AnalysisRequest` data models (`job_openings` , `resumes`) into the format expected by the LangGraph. Resumes are converted into `Document` objects.

3. It updates the progress tracking for each job opening to "pending".

4. It retrieves the compiled LangGraph from the cache (or compiles it if it's not cached).

5. It executes the LangGraph with the input data.

6. It stores the results and updates the job status to "completed".

7. If any error occurs, it updates the job status to "failed" and logs the error.

API Endpoints

Now, let's explore the API endpoints:

`/ai` - Root Endpoint

A simple endpoint that returns a welcome message. It requires a valid API key.

Example:

`GET /ai?api_key=[your_api_key]`

Response:

`{"Welcome": "You are now inside DBTI's AI ENDPOINT!"}`

`/ai/rar/v1/health` - Health Check Endpoint

A health check endpoint that returns a status of "ok". It requires a valid API key.

Example:

`GET /ai/rar/v1/health?api_key=[your_api_key]`

Response:

`{"status": "ok"}`

`/ai/rar/v1/analyze_and_rerank` - Analysis Endpoint

This is the main endpoint for starting an analysis. It accepts an `AnalysisRequest` (containing job openings and resumes), starts a background task to run the analysis, and returns a `trace_id` for tracking the job.

It's rate-limited to 5 requests per minute.

Example:

`POST /ai/rar/v1/analyze_and_rerank`

Request body (example):

```
{
  "job_openings": [
    {
      "name": "Software Engineer",
      "page_content": "Job description for a software engineer..."
    }
  ],
  "resumes": [
    {
      "metadata": {"filename": "resume1.pdf"},
      "page_content": "Resume content of candidate 1..."
    }
  ]
}
```

Response (example):

`{"trace_id": "a1b2c3d4-e5f6-7890-1234-567890abcdef", "message": "Analysis started"}`

`/ai/rar/v1/status/{trace_id}` - Status Endpoint

This endpoint retrieves the status of an analysis job, given its `trace_id`. It returns the job's status, progress, and results (if the job is completed).

Example:

`GET /ai/rar/v1/status/a1b2c3d4-e5f6-7890-1234-567890abcdef`

Response (example, job completed):

```
{
  "trace_id": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
  "status": "completed",
  "progress": {"Software Engineer": "completed"},
  "results": { ... }
}
```

Response (example, job pending):

```
{
  "trace_id": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
  "status": "pending",
  "progress": {"Software Engineer": "pending"},
  "results": null
}
```

Error Handling

The `http_exception_handler` function handles `HTTPException`s, returning a JSON response with the error details.

Conclusion

That's a wrap! Hopefully, this documentation has given you a solid understanding of the `rar_endpoint.py` file and the RAR API. Happy coding!