Assessment Task: Resume Link and Content Scraper API

Objective:

The primary objective of this task is to assess your ability to build a multi-stage data processing pipeline. This involves creating an API that can parse a PDF document, identify all embedded hyperlinks, and then intelligently scrape the textual content from each of those web pages. This will evaluate your skills in file handling, link extraction, robust web scraping, and API development.

Task Description:

You are required to build a RESTful API that accepts a PDF resume file as an input. The API must extract every hyperlink from the PDF, visit each URL, scrape the main textual content from the destination page, and return a structured JSON response with the results.

Key Requirements

1. API Endpoint:

- Create a single API endpoint that accepts a POST request using the multipart/form-data content type.
- The request should accept a file upload with the form field name resume_pdf.

Example Request (using a tool like curl):

```
downloadcontent_copyexpand_less
  curl -X POST -F "resume_pdf=@/path/to/your/resume.pdf" - > This should be your AWS link .
http://localhost:5000/scrape-resume-links
```

2. Core Functionality:

This is a two-step process:

- **Step 1: PDF Hyperlink Extraction**
  - Your service must be able to parse an uploaded PDF file.
  - It must extract **all** hyperlinks present in the document. This includes:
    - **Visible Links:** Plain text URLs (e.g., https://github.com/johndoe).
    - **Embedded Links:** Text that is hyperlinked (e.g., the text "My Portfolio" which links to a website).
  - You should compile a unique list of all URLs found.
- **Step 2: Intelligent Website Scraping**
  - For each unique URL extracted from the PDF, your service must visit the web

page.
    ○ You are required to perform **intelligent content extraction**. This means you should scrape the primary textual content of the page (like an article body, a project description, or an "About Me" section) while making a best effort to **exclude boilerplate** such as headers, footers, navigation menus, sidebars, and advertisements.
    ○ Simply extracting all text from the <body> tag is not sufficient. The goal is to isolate the meaningful content.

3. API Response:

- The API should return a single JSON object.
- This object should contain a key, scraped_data, which holds a list of results—one for each URL scraped.
- Each result object in the list must detail the outcome of the scraping attempt for that specific URL.

Example Response Structure:

code JSON
downloadcontent_copyexpand_less
IGNORE_WHEN_COPYING_START
IGNORE_WHEN_COPYING_END

```
  {
 "scraped_data": [
   {
     "source_url": "https://github.com/janedoe",
     "status": "success",
     "scraped_text": "Jane Doe - Overview. Passionate developer with a focus on building scalable web applications. Pinned Repositories: project-alpha, awesome-scraper, portfolio-website...",
     "error_message": null
   },
   {
     "source_url": "https://janedoe-portfolio.com/project-x",
     "status": "success",
     "scraped_text": "Project X: A Deep Dive. This project was developed to solve the problem of... We used a technology stack of Python, Django, and React. The main challenge was integrating the real-time data feed...",
     "error_message": null
   },
   {
     "source_url": "http://some-broken-link.com",
     "status": "error",
     "scraped_text": null,
     "error_message": "Failed to resolve host or received a 404 Not Found error."
   }
  ]
}
```
4. Error Handling & Robustness:

- Your API must handle a variety of potential failure points gracefully.
- Implement robust error handling for scenarios such as:
  - The uploaded file is not a valid PDF.
  - The PDF is corrupted or cannot be parsed.
  - The PDF contains no hyperlinks.
  - A URL is invalid, dead (404), or protected (403).
  - A website is slow and causes a timeout.
  - A website's content is not standard HTML (e.g., it links directly to an image or another file).
-
- For each URL, the status and error_message fields in the response should accurately reflect the outcome.

Technical Stack

- You are free to use any programming language and libraries you are comfortable with (e.g., Python with PyPDF2/pdfplumber, BeautifulSoup/Scrapy, and Flask/FastAPI; or Node.js with pdf-lib, Cheerio/Puppeteer, and Express.js).
- For the "intelligent content extraction" part, consider libraries designed for this purpose (e.g., Python's trafilatura or readability-lxml) or implement your own logic to identify and extract the main content block of a page.

Submission Guidelines

1. **Source Code:** Provide a link to a Git repository (e.g., on GitHub or GitLab) with the complete, well-organized source code.
2. **Documentation (README.md):** Your repository must include a README.md file that clearly explains:
   - The project's purpose and your technical approach.
   - All prerequisites and dependencies.
   - Step-by-step instructions for setting up the environment and running the API server.
   - An example of how to use the API using curl or a similar tool.
3.

4. **(Optional but Recommended) Concurrency:** For a higher-level implementation, consider how you would process multiple links from the PDF concurrently (in parallel) to improve performance, and briefly explain your approach in the README.md.

Evaluation Criteria

- **PDF Parsing:** How accurately and completely are hyperlinks (both visible and embedded) extracted from the PDF?
- **Scraping Quality:** How effectively does the scraper isolate the main textual content and filter out boilerplate?
- **Robustness & Error Handling:** How well does the API handle various errors, from bad file uploads to broken links?
- **Code Quality & API Design:** Is the code clean, well-structured, and maintainable? Is the API endpoint and JSON response well-designed?
- **Documentation:** Is the project clearly documented and easy for another developer to set up and use?

Disclaimer: This is an assessment to evaluate your technical skills. Please ensure your scraping logic is respectful to websites by setting a proper User-Agent and not making an excessive number of rapid requests.