Backend Developer Technical Assignment: Bulk Document Conversion Service

Objective

The goal of this assignment is to design and build a robust, scalable, and asynchronous web service that converts DOCX files into PDF format. The service must be able to handle a large number of files uploaded in a single batch request and expose its functionality through a clean RESTful API.

This project will assess your ability to:

- Design and implement a clean, well-documented REST API.
- Handle file uploads and downloads efficiently.
- Architect a system for handling long-running, asynchronous tasks.
- Manage local file storage and state within a containerized environment.
- Write clean, maintainable, and testable code.
- Containerize an application for easy deployment and setup.

Core Problem Statement

You are tasked with building a "Docx-to-PDF" conversion microservice. Users will **upload a batch of DOCX files** directly to the service. The service will then convert them to PDF in the background. Once all conversions for a job are complete, the resulting PDFs will be bundled into a single zip archive, which the user can download. The key requirement is that the system must be designed to handle bulk uploads (e.g., 1000 files at once) without blocking the client or timing out.

Functional Requirements

1. **API for Job Submission:**
    - Create an API endpoint to submit a new conversion "job"t.
    - This single request will contain all the DOCX files as a zip for the job.
    - The API should not wait for the conversion to complete. It must save the uploaded files to a temporary location, enqueue the job for processing, and immediately return a unique job_id to the client.
2. **API for Job Status:**
    - Create an API endpoint to check the status of a previously submitted job using its job_id.
    - The status should indicate the overall progress of the job (e.g., PENDING, IN_PROGRESS, COMPLETED, FAILED).
    - The response should also include a breakdown of the status of each individual file within the job (identified by its original filename).

    - When a job's status is COMPLETED, the response must include a **download URL** for the generated zip archive.
3. **Asynchronous Processing:**
    - The conversion process must happen in the background. A simple for loop in the API request handler is not an acceptable solution.
    - You must use a message queue (like RabbitMQ, Redis Streams, or a robust in-memory queue) to decouple the API from the conversion workers.
4. **File Handling and Archiving:**
    - **Input:** The API service must handle the multipart/form-data upload, saving the files to a shared temporary storage volume accessible by the workers.
    - **Conversion:** A background worker process should pick up a task, read a DOCX file from the temporary storage, and convert it to PDF. You are free to use any library or tool for this.
    - **Output:** The worker should save the resulting PDF to a job-specific output directory on a shared volume.
    - **Archiving:** Once all files in a job are converted, a final task should be triggered to create a single zip archive containing all the generated PDFs for that job.
5. **API for Downloading Results:**
    - Create an API endpoint (e.g., GET /api/v1/jobs/{job_id}/download) that allows the user to download the final zip archive.
6. **Error Handling:**
    - The system must be resilient to errors. Plan for and handle cases such as:
        - A file is not a valid DOCX file (corrupted or wrong format).
        - The conversion process for a single file fails. A single file failure should **not** stop the entire job.
        - Issues with file system permissions or disk space (conceptual handling is fine).

Non-Functional Requirements

1. **Scalability:** While you don't need to deploy a multi-node cluster, your architecture should conceptually support it. The use of a message queue, stateless workers, and shared volumes is key.
2. **Containerization:** The entire application (API service, workers, etc.) must be containerized using Docker and easily runnable with docker-compose.
3. **Documentation:** Provide a README.md file that includes:
    - A brief explanation of your architectural choices (especially how you manage file storage between containers).
    - Clear instructions on how to build and run the project locally.
    - API documentation (e.g., using Swagger/OpenAPI or just a clear markdown description of the endpoints).

Technical Stack

- **Language/Framework:** You are free to use any framework of python you are comfortable with (e.g., Python/FastAPI/Celery).
- **Data Storage:** A database (e.g. PostgreSQL in Docker) is highly recommended to store job and file status.
- **Message Queue:** Using a real queue system like RabbitMQ or Redis (both available as Docker images) is highly recommended.
- **File Storage:** Use Docker volumes to share files between your containers.

API Specification (Example)

1. Submit a New Conversion Job

- POST /api/v1/jobs

Success Response (202 Accepted):
Generated json
```
    {

  "job_id": "a1b2c3d4-e5f6-7890-1234-567890abcdef",

  "file_count": 2

}
```

- IGNORE_WHEN_COPYING_START
  content_copy download
  Use code with caution. Json
  IGNORE_WHEN_COPYING_END

2. Get Job Status

- GET /api/v1/jobs/{job_id}

Success Response (200 OK):
Generated json
```
    {

  "job_id": "a1b2c3d4-e5f6-7890-1234-567890abcdef",

  "status": "COMPLETED", // PENDING, IN_PROGRESS, COMPLETED, FAILED

  "created_at": "2023-10-27T10:00:00Z",

  "download_url": "http://localhost:8000/api/v1/jobs/a1b2c3d4.../download", // Only present if status is COMPLETED
```

```
  "files": [
    {
      "filename": "report.docx",
      "status": "COMPLETED"
    },
    {
      "filename": "proposal.docx",
      "status": "COMPLETED"
    },
    {
      "filename": "bad-file.docx",
      "status": "FAILED",
      "error_message": "Invalid file format or corrupted DOCX."
    }
  ]
}
```


3. Download Results

- GET /api/v1/jobs/{job_id}/download
- **Success Response:** A 200 OK response with the Content-Type: application/zip header and the zip file as the body.

What to Submit

1. A link to a Git repository (e.g., GitHub, GitLab) containing your full source code.
2. The README.md file as described in the non-functional requirements.
3. A docker-compose.yml file that sets up and runs the entire application stack with a single command.
4. Deployed link of the application.

Evaluation Criteria

We will be looking at:

- **Correctness:** Does the service fulfill all the functional requirements?
- **Architecture:** Is the system designed in a scalable and resilient way? (i.e., proper use of asynchronous processing and shared state/storage).
- **Code Quality:** Is the code clean, well-structured, readable, and maintainable?
- **API Design:** Is the API intuitive, RESTful, and well-documented?
- **File Handling:** Is the management of file uploads, temporary storage, and final artifact creation robust and efficient?
- **Production Readiness:** How easy is it to run the application? Is the containerization setup clean?
- **Testing:** (Bonus points) How well is the code tested?

Good luck! We are excited to see your solution.