Retail-Pulse-Service

(Backend Intern Assignment)

1. Description

The **Retail Pulse Service** is a Go-based application that processes retail data, handles job assignments, and supports downloading resources like images. It is designed for scalability, with containerized deployment options using Docker. The project includes modular code for handling APIs, processing jobs, logging, and managing resources.

The **Retail Pulse Service** is a backend system designed to process images collected from retail stores. The system receives jobs containing image URLs and store IDs, processes these images to calculate their perimeters, and stores the results for reporting and analysis.

The main objectives of the service are:

1. **Job Submission**: Accept jobs containing multiple store visits, each with associated image URLs, and store visit metadata.
2. **Image Processing**:
    - Download each image from the provided URL.
    - Calculate the **perimeter** of each image as `2 * (Height + Width)`.
    - Simulate GPU processing with a random sleep time between 0.1 and 0.4 seconds.
3. **Store Master Integration**:
    - Validate `store_id` using the Store Master database, which contains metadata like `store_name` and `area_code`.
4. **Job Status Monitoring**: Allow users to query the status of submitted jobs to track progress or identify errors.
    - **Completed**: All images successfully processed.
    - **Ongoing**: The job is still being processed.
    - **Failed**: Errors occurred, such as invalid `store_id` or failed image downloads.

This service is designed to handle high concurrency, allowing multiple jobs with thousands of images to be processed simultaneously. It ensures reliability by storing job results and providing detailed error reports for failed operations.

2. Assumptions

The following assumptions were made while developing the service:

- **Data:** The application assumes valid input data is provided in a structured format (e.g., `StoreMasterAssignment.csv`). The Store Master database is accurate and up-to-date.
- **Input Validation**: Jobs are submitted with correctly formatted JSON payloads, including valid store_id values and accessible image URLs.
- **Concurrency**: The service is expected to handle multiple jobs simultaneously and process images efficiently.
- **Error Scenarios**: Image download failures or invalid store_id values will result in partial or complete job failures.
- **Sleep Times**: Random sleep intervals (0.1–0.4 seconds) realistically mimic GPU processing time.

3. Installation and Testing Instructions

Option 1: Using Docker

1. **Prerequisites**:
    - Install Docker and Docker Compose.
    - Ensure ports specified in `docker-compose.yml` are available.
2. **Setup**:
    - Navigate to the project directory.
    - Build and start the containerized application:

      ```
      docker-compose build
      docker-compose up
      ```

    - Access the service using the endpoint specified in `docker-compose.yml`.
3. **Logs**:
    - Logs are written to `retail-pulse.log`. Monitor logs using:

      ```
      tail -f retail-pulse.log
      ```

4. **Testing**:
    - Use sample data (StoreMasterAssignment.csv) to test functionality.
    - APIs can be tested using tools like Postman or curl.

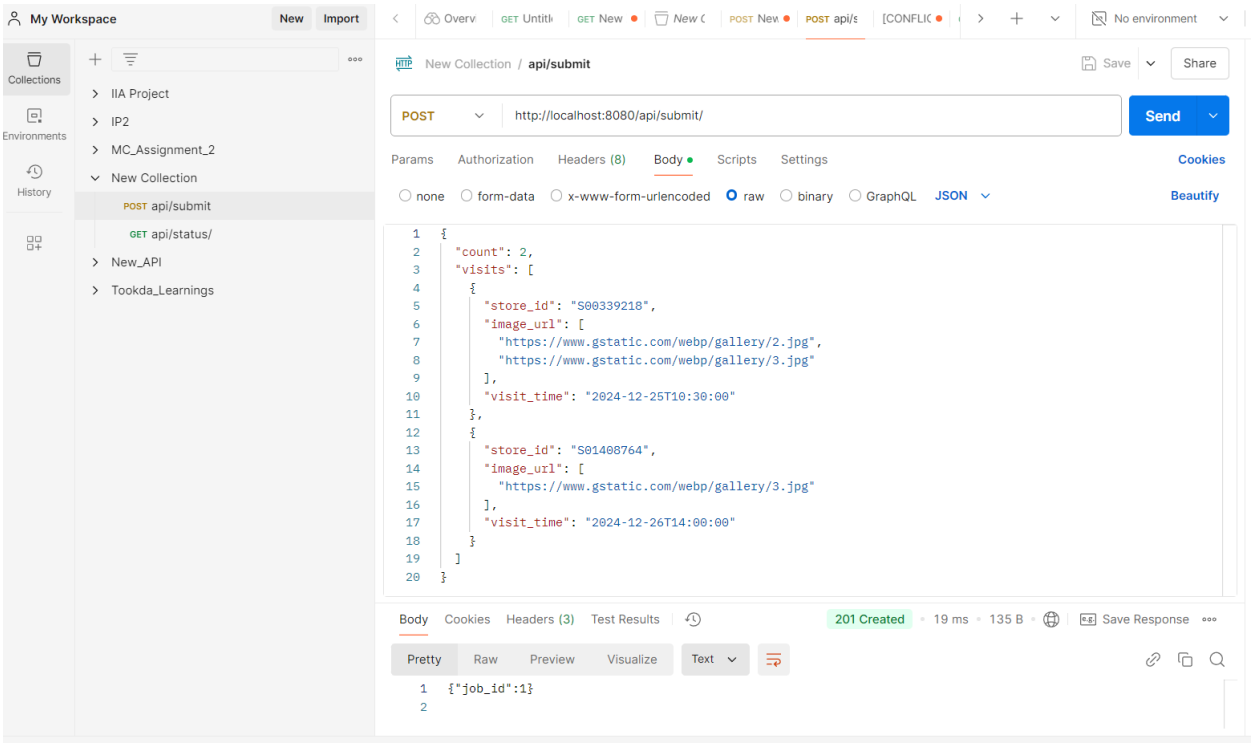
Option 2: Local Setup Without Docker

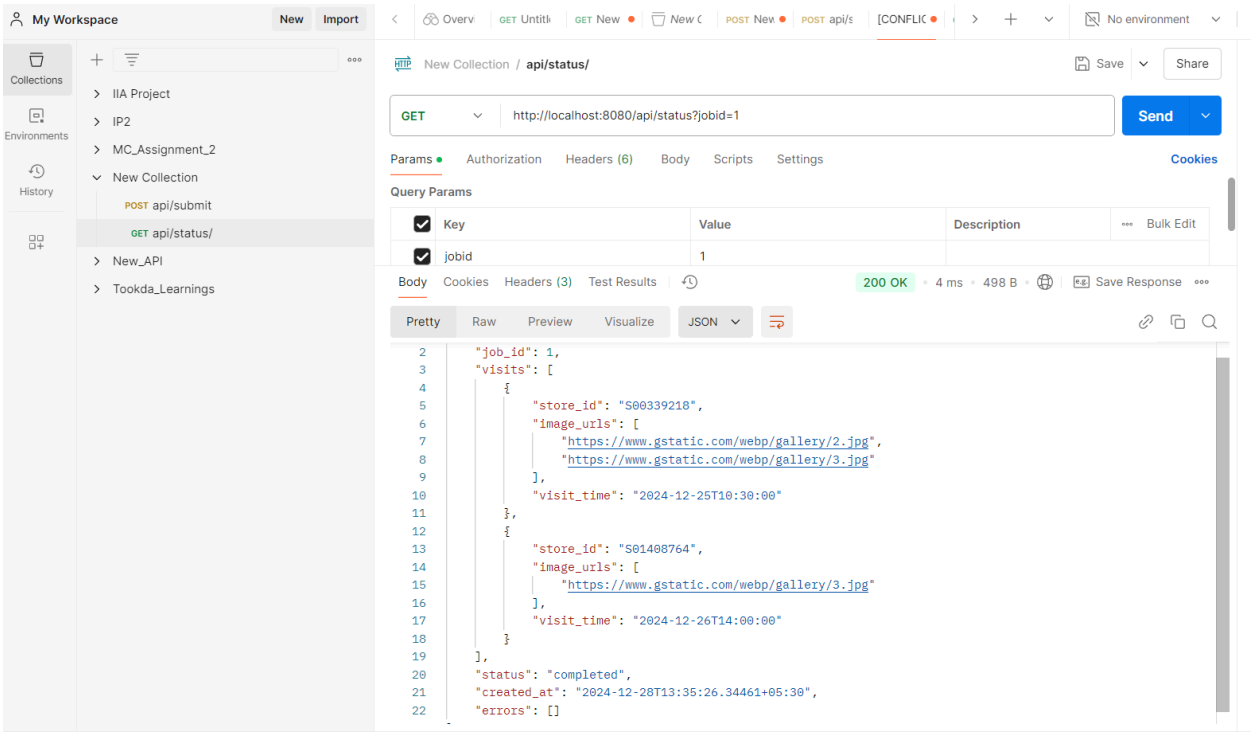
1. **Install Dependencies**:
   - Ensure Go is installed (version specified in `go.mod`). If it is not there then can use command `go mod init retail-pulse-service`.
   - Install dependencies:
     ```
     go mod tidy
     ```

2. **Run the Application**:
   - Start the application:
     ```
     go run main.go
     ```

3. **Testing**:
   - Test API endpoints locally at `http://localhost:<8080>`.
   - APIs can be tested using tools like Postman or curl.

4. **Screenshots**:





4. Work Environment

This project was developed in the following environment:

- **Programming Language**: Go (Golang)
- **Dependency Management**: Go Modules (go.mod and go.sum)
- **Containerization**: Docker and Docker Compose
- **Editor/IDE**: Visual Studio Code with Go extensions
- **APIs**:
    - /api/submit/: Submit a new job.
    - /api/status: Monitor job status.

5. Future Improvements

1. **Scalability**:
    - Integrate GPU processing to handle large image datasets more efficiently.
    - Add support for Kubernetes to allow dynamic scaling based on workload.
2. **Improved Error Handling**:
    - Include detailed error messages for image download failures (e.g., HTTP errors).
    - Retry failed downloads with exponential backoff.
3. **Logging and Monitoring**:

    - Use a centralized logging system (e.g., ELK stack or Fluentd) for real-time monitoring.
    - Integrate metrics collection tools like Prometheus and Grafana.
4. **API Enhancements**:
    - Add pagination to /api/status for jobs with thousands of images.
    - Provide a summary of processed and failed images for completed jobs.
5. **Data Validation**: (Important)
    - Implement stricter validation for store_id and image_url fields.
    - Add a schema validation service for incoming payloads.
6. **Frontend Dashboard**:
    - Develop a user-friendly dashboard for monitoring job statuses and results.
    - Allow users to upload jobs and view processed results graphically.
7. **Improved Resource Management**:
    - Automatically clean up old jobs and downloaded images to free up storage.
    - Move image storage to a cloud-based system like AWS S3 or Google Cloud Storage.
8. **Security Enhancements**:
    - Add authentication to API endpoints using JWT.
    - Enforce HTTPS to secure data in transit.

6. Project Structure

Configuration and Setup:

- docker-compose.yml: For Dockerized multi-service deployment.
- Dockerfile: To containerize the service.
- go.mod and go.sum: Dependency management for the Go language.

Main Application:

- main.go: Entry point for the Go application.

Logs and Data:

- retail-pulse.log: Log file capturing application activity.
- StoreMasterAssignment.csv: A CSV file, likely containing test or sample data.

Code Organization:

- api/job_handler.go: API handlers.
- models/: Contains models for job.go and store.go.
- services/job_processor.go: Service logic for job processing.
- utils/downloader.go: Likely for handling downloads.
- utils/logger.go: Custom logging utilities.

Resources:

- downloaded_images/: Directory containing images.