1. Overview

The objective of this project is to build an asynchronous system that:

- Receives and validates a CSV file containing product image data.
- Processes images asynchronously (compressing them to 50% quality).
- Stores the processed output and product data in a database.
- Provides immediate feedback with a unique request ID and a status-check API.
- Optionally triggers a webhook callback upon processing completion.

2. System Architecture

The system is composed of several key components working together:

- **API Layer:**

Provides endpoints to upload the CSV file and check processing status. It validates the input and immediately returns a unique request ID to the client.

- **Asynchronous Worker:**

Runs in the background (using a task queue such as Celery in Python or Bull in Node.js) to handle the heavy image processing tasks. It parses the CSV, downloads image, compresses them, and updates the status in the database.

- **Image Processing Service:**

A dedicated module (using libraries like Pillow in Python or Sharp in Node.js) that performs image compression, reducing the image quality by 50%.

- **Database:**

Stores the processing requests, product details, input and output image URLs, and the status of each request. A SQL (e.g., PostgreSQL) or NoSQL (e.g., MongoDB) database can be used.

- **Webhook Handler (Bonus):**

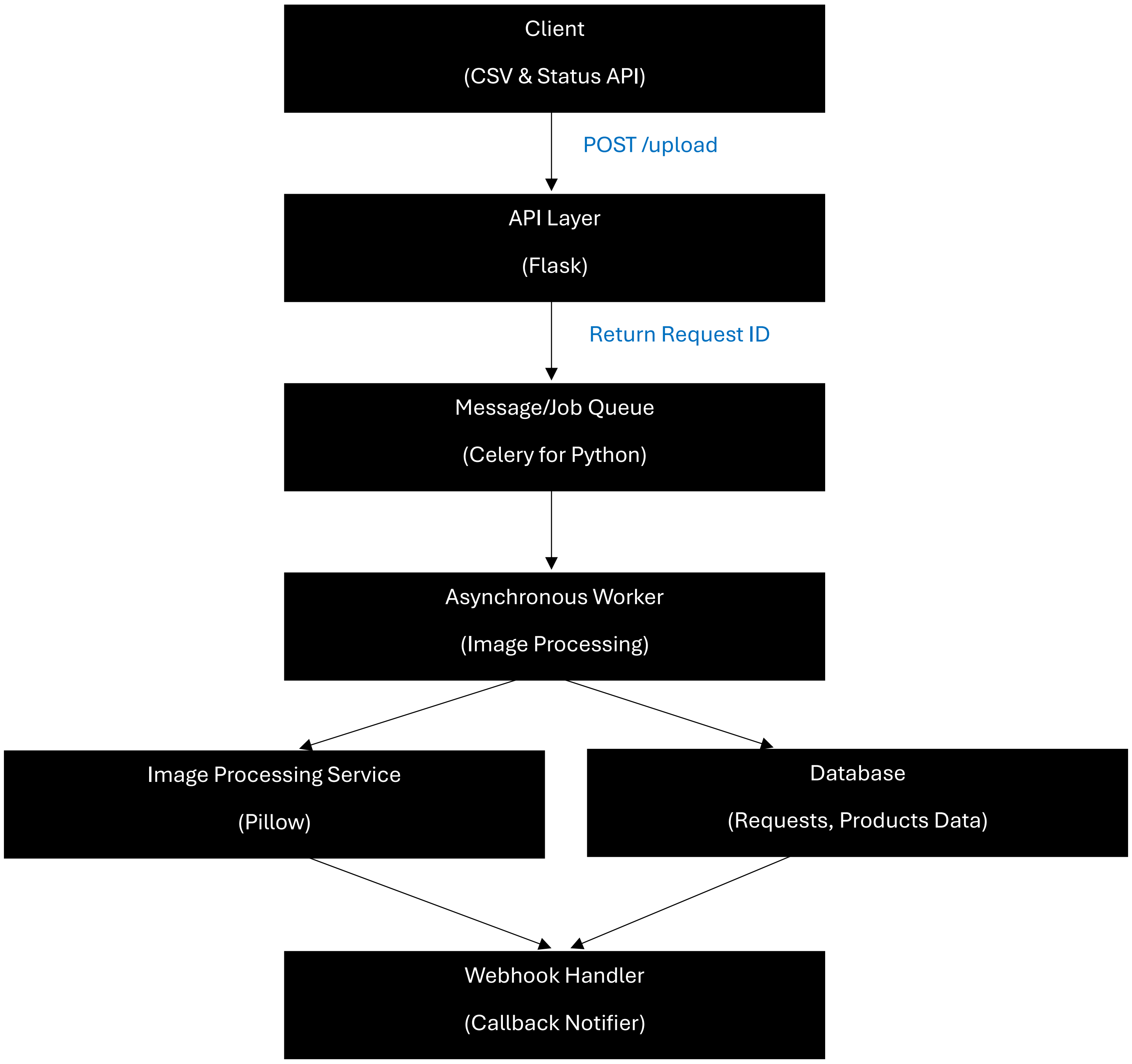
A mechanism to notify a client-specified endpoint after all image processing has been completed. This ensures that external systems can react to processing completion.

- **External File Storage (Optional):**

After processing, images (or CSV files) might be stored in cloud storage, and their URLs saved in the database.

3. Visual System Diagram

Below is a simplified visual diagram that maps out the system components and their interactions:



4. Component Descriptions

- **API Layer**
  - **Role**: Exposes endpoints for CSV file uploads and status queries.
  - **Function**:
  • Receives a CSV file via the POST /upload endpoint.
  • Validates the CSV format (ensuring required columns are present).
  • Generates a unique request ID and stores initial request data in the database.
  • Enqueues a job for asynchronous processing.
  • Provides a GET /status/<request_id> endpoint for checking processing status.

- **Asynchronous Worker**
  - **Role**: Handles background tasks to avoid blocking the API layer.
  - **Function**:
  • Retrieves jobs from the message queue.
  • Processes the CSV: reads product data and image URLs.
  • For each image URL, calls the Image Processing Service to compress the image.
  • Updates the product records with output image URLs.
  • Changes the processing status (e.g., Queued → In Progress → Completed) in the database.
  • If configured, triggers a webhook callback upon completion.

- **Image Processing Service**
  - **Role**: Responsible for downloading and compressing images.
  - **Function**:
  • Downloads each image from its URL.
  • Uses an image library to compress the image to 50% of its original quality.
  • Optionally stores the compressed image in a file system or cloud storage and returns a new URL.

- **Database**
  - **Role**: Serves as persistent storage for processing requests and product information.
  - **Function**:
  • Requests Table: Tracks each processing job by request ID, status, timestamps, and (if applicable) the output CSV URL.
  • Products Table: Stores product data, input image URLs, and corresponding output image URLs.
  • Facilitates querying of processing status by the API.

- **Webhook Handler**
  - **Role**: Provides a mechanism for external notification.
  - **Function**:
  • Once the worker completes processing, it sends a callback (via POST) to a pre-configured webhook URL.
  • The payload includes details such as the request ID and processing status.
  • Enables external systems to receive real-time notifications of job completion.

5. API Documentation

1. Upload API

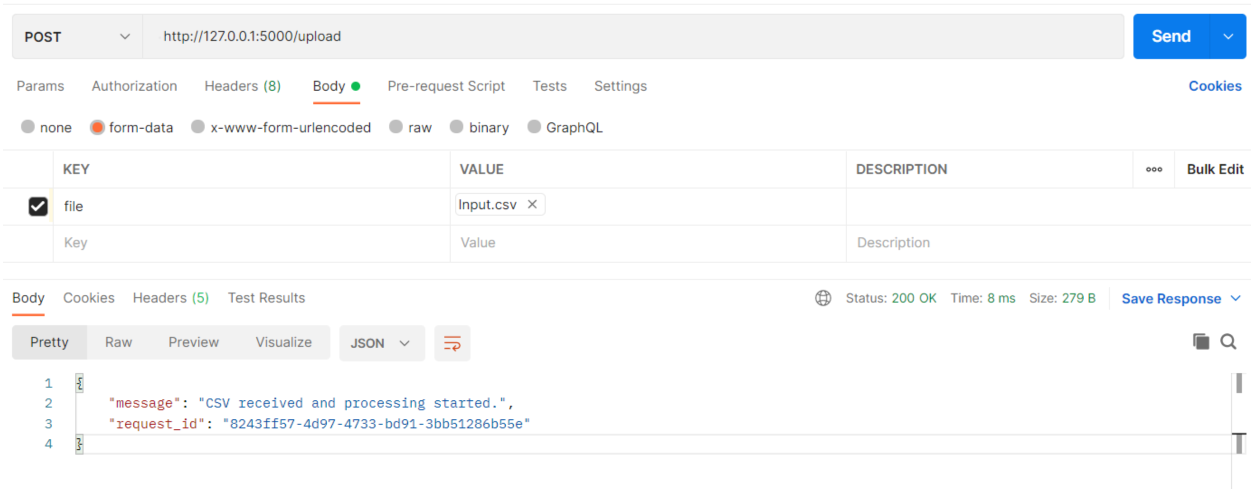
Endpoint: POST /upload

Purpose: Accept a CSV file containing product and image URL data, validate its format, and initiate asynchronous processing. Returns a unique request ID immediately.

Request Format:
- Headers:
  - Content-Type: multipart/form-data
- Body (Form-Data):
  - Key: file
  - Type: File (CSV file)
  - Expected CSV Format: The CSV should have the following columns:
    - Serial Number: Unique serial number for each product.
    - Product Name: Name of the product.
    - Input Image Urls: Comma-separated list of image URLs.

Successful Response:
- HTTP Status Code: 200 OK
- Response Body (JSON):

```
{
  "message": "CSV received and processing started.",
  "request_id": "8243ff57-4d97-4733-bd91-3bb51286b55e"
}
```



http://127.0.0.1:5000/upload

	KEY	VALUE	DESCRIPTION
✓	file	Input.csv	
	Key	Value	Description

Status: 200 OK Time: 8 ms Size: 279 B

```
{
  "message": "CSV received and processing started.",
  "request_id": "8243ff57-4d97-4733-bd91-3bb51286b55e"
}
```

2. Status API

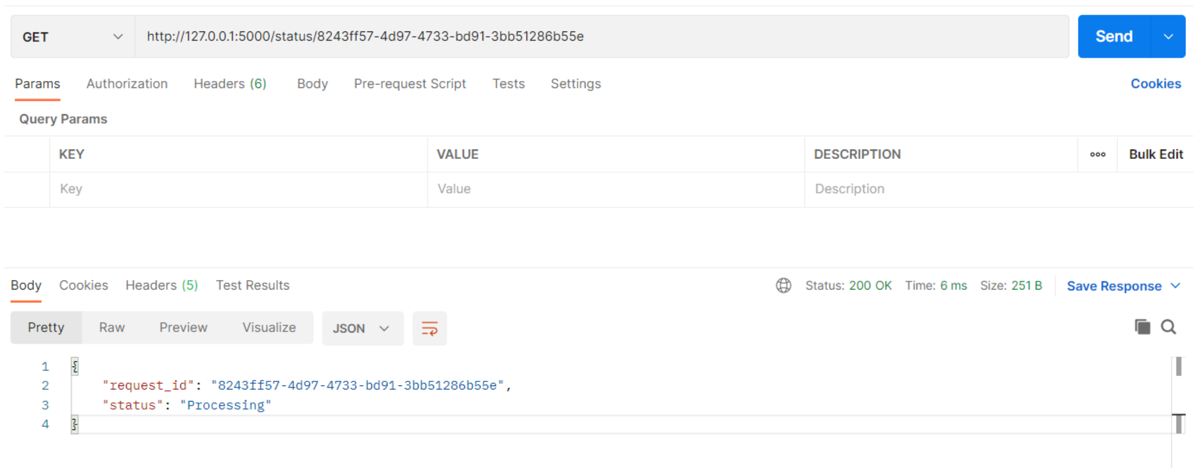
Endpoint: GET /status/<request_id>

Purpose: Allow the client to query the processing status of a submitted CSV using the unique request ID. Provides real-time status and, once complete, a link or reference to the output CSV.

Request Format:
• URL Parameter:
  - <request_id>: The unique identifier returned from the Upload API.

Successful Response (Processing Ongoing):
• HTTP Status Code: 200 OK
• Response Body (JSON):

```
{
  "request_id": "8243ff57-4d97-4733-bd91-3bb51286b55e",
  "status": "Processing"
}
```



http://127.0.0.1:5000/status/8243ff57-4d97-4733-bd91-3bb51286b55e

Status: 200 OK Time: 6 ms Size: 251 B

```
{
    "request_id": "8243ff57-4d97-4733-bd91-3bb51286b55e",
    "status": "Processing"
}
```

Successful Response (Processing Completed):
• HTTP Status Code: 200 OK
• Response Body (JSON):

```
{
  "request_id": "8243ff57-4d97-4733-bd91-3bb51286b55e ",
  "status": "Completed",
  "output_csv_url": "https://storage.example.com/output_8243ff57 .csv"
}
```

6. Database Schema Design

The schema consists of two primary tables: Requests and Products.

1. Requests Table

Purpose: Tracks each CSV upload request, its processing status, and a reference to the generated output CSV (if available).

Fields:

• id (Primary Key):
  - Type: UUID or String
  - Description: A unique identifier generated upon CSV upload, used to track the request.

• status:
  - Type: ENUM or String
  - Possible Values: “Queued”, “Processing”, “Completed”, “Failed”
  - Description: Indicates the current processing state of the request.

• created_at:
  - Type: Timestamp
  - Description: Records when the request was created.

• updated_at:
  - Type: Timestamp
  - Description: Records when the request status was last updated.

• output_csv_url:
  - Type: String (URL)
  - Description: The URL or file path to the output CSV containing processed image URLs. This field is populated once processing is completed.

Example Record:

id	status	created_at	updated_at	output_csv_url
8243ff57-4d97-4733-bd91-3bb51286b55e	Processing	2025-03-03 12:00:00	2025-03-03 12:05:00	https://storage.example.com/output_8243ff57 .csv

2. Products Table

Purpose: Stores detailed information for each product listed in the CSV, including the input image URLs and the corresponding output (processed) image URLs.

Fields:

• id (Primary Key):

- Type: Auto-increment integer or UUID
- Description: Unique identifier for the product record.• status:

• request_id:

- Type: UUID or String
- Description: Foreign key referencing the Requests table (id). This links each product record to a specific CSV processing request.

• serial_number:

- Type: String
- Description: The serial number from the CSV file.

• product_name:

- Type: String
- Description: Name of the product.

• input_image_urls:

- Type: Text or JSON Array
- Description: Comma-separated string or an array of URLs for the original images provided in the CSV.

• output_image_urls:

- Type: Text or JSON Array
- Description: Comma-separated string or an array of URLs for the processed images (after compression).

Example Record:

id	serial_number	product_name	input_image_urls	output_image_urls
8243ff57-4d97-4733-bd91-3bb51286b55e	1	SKU1	https://wallpaperaccess.com/full/4723250.jpg	https://storage.example.com/2fdc5fbf-3248-45a6-a83a-3dbf13b6c041.jpg

3. Relationships

One-to-Many Relationship:
Each record in the Requests table can be associated with multiple records in the Products table.

- **Implementation**:

  The Products table includes a foreign key (request_id) that references the primary key (id) of the Requests table.

- **Data Flow**:

  1. **Upload**: When a CSV is uploaded, a new record is created in the Requests table with the initial status (e.g., “Queued”).
  2. **Processing**: The CSV is parsed, and each row (representing a product) is stored as a record in the Products table.
  3. **Update**: As image processing completes, the corresponding product records are updated with the output image URLs. Once all products are processed, the Requests record is updated (status set to “Completed” and the output CSV URL saved).

7. Asynchronous Worker Documentation

1. Worker Functions and Asynchronous Processing

Worker Functions

- **Job Initialization**:

When a CSV file is uploaded via the Upload API, a new processing job is created with a unique request ID and enqueued in a task queue (e.g., Celery for Python). The job record is stored in the Requests table with an initial status (e.g., “Queued” or “Processing”).

- **CSV Parsing**:

The asynchronous worker reads the CSV file, validates that each row conforms to the expected format (ensuring that columns such as Serial Number, Product Name, and Input Image Urls are present).
For each row:

- The CSV row is parsed into individual fields.
- The Input Image Urls field is split into a list of URLs.

- **Image Processing**:

For each image URL:

• **Download:**
The worker attempts to download the image using an HTTP request.

• **Compression**:
It then compresses the image by reducing its quality to 50% using an image processing library (such as Pillow in Python).

• **Storage:**
The compressed image is saved (e.g., to cloud storage or a local directory), and its URL is generated.

• **Mapping**:
The worker ensures that the sequence of output image URLs matches the order of input image URLs.

• **Database Update**:
The worker updates the corresponding product record in the Products table with the new output image URLs.

Error Handling

• If an image download fails or returns an error (e.g., invalid URL, timeout, or HTTP error code), the worker logs the error and records a failure status for that image.

• For CSV parsing errors (such as missing columns or malformed data), the worker marks the entire request as “Failed” and logs the error details.

• Any exceptions during image processing (e.g., issues with image compression) are caught and logged; the product record may be flagged with an error message or a placeholder value.

THANK YOU!!!