**1. app/\_\_init\_\_.py**

**Purpose:** This is the main initialization module for the Flask application, where core components are set up.

**Key Components:**

* **Environment Variables:**
  + The load\_dotenv() function loads environment variables from a .env file, helping manage configuration settings securely.
* **Flask App Initialization:**
  + app = Flask(\_\_name\_\_) initializes the Flask application.
  + Configuration settings for SQLAlchemy (database) are loaded from environment variables.
* **Database Setup:**
* db = SQLAlchemy() creates an instance of SQLAlchemy, used for ORM (Object-Relational Mapping) to interact with the database.
* db.init\_app(app) associates the SQLAlchemy instance with the Flask app.
* **Cloudinary Configuration:**
  + cloudinary.config(...) sets up Cloudinary for image and file management, using credentials stored in environment variables.
* **Celery Setup:**
  + celery\_init\_app(app) initializes Celery for background task processing.
  + The custom FlaskTask class ensures that tasks run within the Flask application context, allowing access to app resources.

**2. run.py**

**Purpose:** This script is the entry point for running the Flask application.

**Key Components:**

* **Database Initialization:**
  + db.create\_all() creates all the database tables defined in the models if they don't exist.
* **Blueprint Registration:**
  + The main blueprint (defined in app/routes.py) is registered to handle specific routes and views in the application.
* **Application Running:**
* app.run(debug=True) starts the Flask development server with debugging enabled, allowing for live reloading and error tracking.

**3. app/models.py**

**Purpose:** This module defines the database models used by SQLAlchemy.

**Key Components:**

* **RequestStatus Model:** This class represents a database table for tracking request statuses.
  + id: Primary key for the table.
  + request\_id: A unique identifier for each request.
  + status: A string representing the current status of the request (e.g., "pending," "completed").
  + webhook\_url: URL for webhook callbacks related to the request.
  + input\_csv\_url: URL of the uploaded input CSV file.
  + output\_csv\_url: URL of the generated output CSV file.

**4. app/dto.py**

**Purpose:** This module defines Data Transfer Objects (DTOs) using Pydantic, which are used for data validation and serialization/deserialization of request and response data.

**Key Components:**

* **UploadCSVRequestDTO:** Represents the request data for uploading a CSV file. It contains:
  + webhook\_url: An optional field that must be a valid HTTP URL.
* **UploadCSVResponseDTO:** Represents the response data for a successful CSV upload. It contains:
  + message: A message indicating the result of the upload.
  + request\_id: The unique ID of the request associated with the uploaded CSV.
* **CheckStatusRequestDTO:** Represents the request data for checking the status of a request. It contains:
* request\_id: The unique ID of the request whose status is being checked.
* **CheckStatusResponseDTO:** Represents the response data for the status check. It contains:
  + request\_id: The ID of the request.
  + result: A dictionary containing the details of the status result.

**5. app/routes.py**

**Purpose:** This module defines the API endpoints for handling CSV uploads and checking the status of requests. It processes incoming requests, manages data validation, and interacts with the database.

**Key Components:**

1. **Blueprint Initialization:**
   * main = Blueprint("main", \_\_name\_\_): Initializes a blueprint named main to organize routes and views modularly.
2. **Logging Setup:**
   * Configured for logging messages at the INFO level to monitor and debug the application.
3. **upload\_csv() Function:**

* **Route:** @main.route("/upload\_csv", methods=["POST"]): Defines an endpoint for uploading CSV files.
* **Request Handling:**
  + Checks if a file is present and validates its format.
  + Uses pandas to read the uploaded CSV and checks for the expected columns.
  + Logs warnings and errors (e.g., missing file, empty CSV).
* **File Upload to Cloudinary:**
  + Uploads the CSV file to Cloudinary and retrieves the secure URL.
* **UUID Generation:**
  + Generates a unique request ID for tracking the request status.
* **Data Transfer Object (DTO):**
  + Validates incoming data using UploadCSVRequestDTO.
* **Database Interaction:**
  + Creates an entry in the RequestStatus model, indicating the request status.
  + Commits the entry to the database.
* **Task Processing:**
  + Calls process\_images\_task.delay(request\_id) to process images asynchronously using Celery.
* **Response:**
  + Returns a successful response with the request ID if the upload succeeds.

1. **check\_status() Function:**
   * **Route:** @main.route("/status/<request\_id>", methods=["GET"]): Defines an endpoint to check the status of a request based on its ID.
   * **Request Handling:**
     + Validates the request ID format using CheckStatusRequestDTO.
     + Queries the RequestStatus model to retrieve the request status.
     + Logs warnings for invalid request IDs or non-existent requests.
   * **Response:**
     + Returns the request status with input and output CSV URLs.

**6. app/tasks.py**

**Purpose:** This module defines background tasks to handle image processing asynchronously, utilizing Celery for task management and offloading time-consuming operations.

**Key Components:**

1. **Logging Setup:**
   * Similar to routes.py, logging is configured to monitor task execution and errors.
2. **Celery Task Definition:**
   * @celery.task: Decorates process\_images\_task as a Celery task for background execution.
3. **Task Execution:**

* process\_images\_task(request\_id): Calls asyncio.run(process\_images\_async(request\_id)) to run the asynchronous image processing function.

1. **process\_images\_async() Function:**
   * **Request Status Retrieval:** Checks for an existing request in the database with a "PENDING" status.
   * **CSV Retrieval:**
     + Uses aiohttp to asynchronously download the input CSV from Cloudinary and read it into a Pandas DataFrame.
     + Validates CSV format, updating the status to "FAILED" if checks fail.
   * **Processing Loop:**
     + Iterates through the DataFrame rows, extracting image URLs and processing them asynchronously.
     + Calls process\_image concurrently using asyncio.gather to process each image URL.
   * **Output Generation:**
     + Collects processed image URLs, constructs an output DataFrame, and uploads it to Cloudinary.
   * **Status Update:**
     + Updates the RequestStatus entry in the database with the final status and output CSV URL.
2. **process\_image() Function:**
   * **Asynchronous Image Processing:**
     + Downloads each image using aiohttp, converts it to JPEG format, and uploads it to Cloudinary.
     + Logs errors for network issues, image identification failures, or unexpected errors.
   * **Output Collection:**
     + Appends the secure URLs of processed images to the output\_image\_urls list for generating the output CSV.