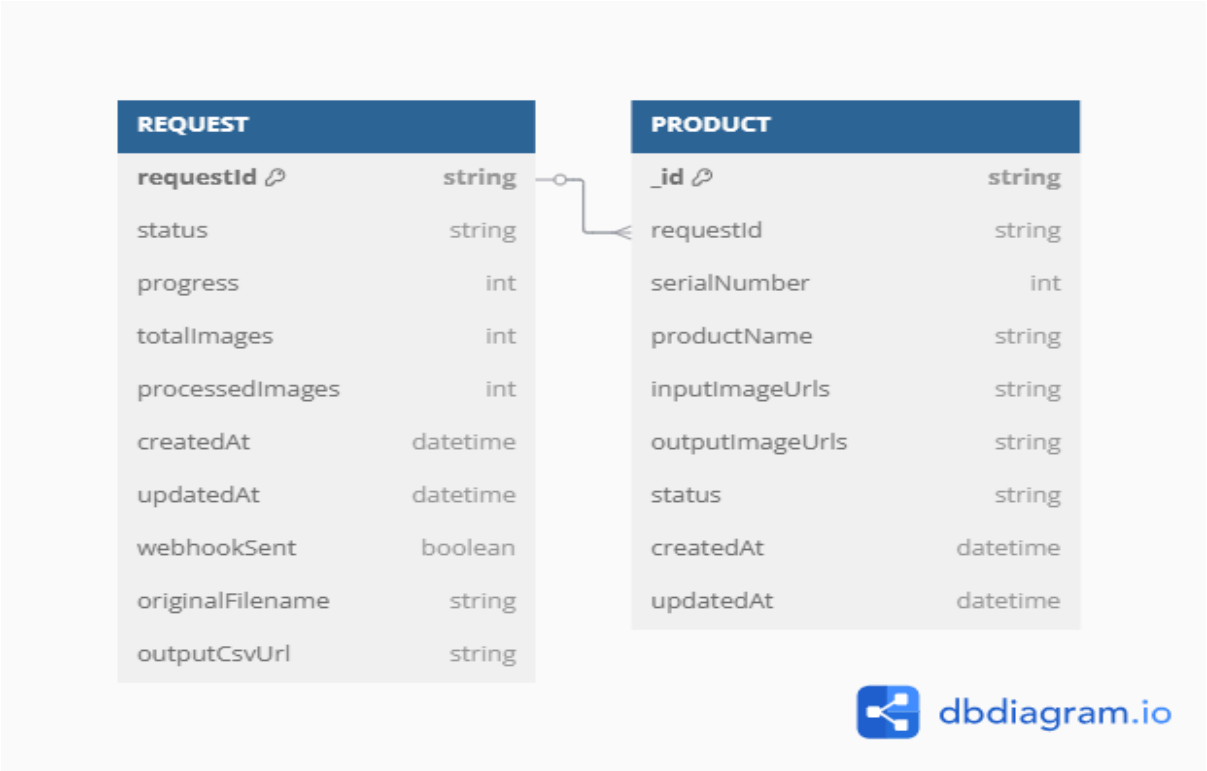
Low-Level Design (LLD) for Image Processing System

A Node.js application that processes images from CSV/TXT files asynchronously.

Overview:
Accepts CSV/TXT files with product information and image URLs
Validates the CSV format
Processes the images asynchronously (compressing them to 50% quality)
Stores the processed data in a database
Provides APIs for uploading and checking status
Implements webhook functionality for completion notifications
Github Link: https://github.com/StoneRaptor5870/image-processing-system
Postman Collection:
https://www.postman.com/lunar-flare-336480/workspace/public/collection/28584614-4598f68d-a281-40ea-8253-fe13011e7a2e?action=share&creator=28584614

Tech Stack:
Node.js with Express for the API server
MongoDB as the NoSQL database
Bull for job queue management (handling asynchronous processing)
Redis as the message broker for Bull
Multer for handling file uploads
CSV-Parser for parsing CSV files
Sharp for image processing

Database Schema



API Documentation

Image Processing System API Documentation

Base URL
```
http://localhost:3000/api
```

Endpoints

1. Upload CSV/TXT File
Upload a CSV/TXT file containing product and image information.

URL: `/upload`
Method: `POST`
Content-Type: `multipart/form-data`

Request Body Parameters:
Parameter	Type	Required	Description
file	File	Yes	CSV / TXT file to be processed

Expected CSV Format:
```
S. No.,Product Name,Input Image Urls
1,SKU1,https://www.public-image-url1.jpg,https://www.public-image-url2.jpg,https://www.public-image-url3.jpg
2,SKU2,https://www.public-image-url1.jpg,https://www.public-image-url2.jpg,https://www.public-image-url3.jpg
```

Success Response:
- **Code**: 200
- **Content**:
```json
{
  "success": true,
  "requestId": "13f5b378-1d34-4601-a8c3-5678bb1234cd",
  "message": "File uploaded successfully. Processing started."
}
```

Error Responses:
- **Bad Request (400)**:
```json
{
  "success": false,
  "message": "No file uploaded"
}
```
OR
```json
{
  "success": false,
  "message": "Only CSV and TXT files are allowed"
}
```
OR
```json
{
  "success": false,
  "message": "Missing required columns: Product Name, Input Image Urls"
}
```

- **Internal Server Error (500)**:
```json
{
  "success": false,
  "message": "An error occurred during file upload",
  "error": "Error message details"
}
```

2. Check Processing Status
Check the status of a previously submitted processing request.

URL: `/status/:requestId`
Method: `GET`
URL Parameters:
Parameter	Type	Required	Description
requestId	String	Yes	ID returned from upload request

Success Response:
- **Code**: 200
- **Content**:
```json
{
  "success": true,
  "data": {
    "requestId": "13f5b378-1d34-4601-a8c3-5678bb1234cd",
    "status": "processing",
    "progress": 45,
    "totalImages": 10,
    "processedImages": 4,
    "createdAt": "2023-10-10T10:15:30.000Z",
    "updatedAt": "2023-10-10T10:16:20.000Z"
  }
}
```

For Completed Requests:
```json
{
  "success": true,
  "data": {
    "requestId": "13f5b378-1d34-4601-a8c3-5678bb1234cd",
    "status": "completed",
    "progress": 100,
    "totalImages": 10,
    "processedImages": 10,
    "createdAt": "2023-10-10T10:15:30.000Z",
    "updatedAt": "2023-10-10T10:20:45.000Z",
    "outputCsvUrl":
"http://localhost:3000/processed/13f5b378-1d34-4601-a8c3-5678bb1234cd-output.csv"
  }
}
```

Error Responses:
- **Not Found (404)**:
```json
{
  "success": false,
  "message": "Request not found"
}
```

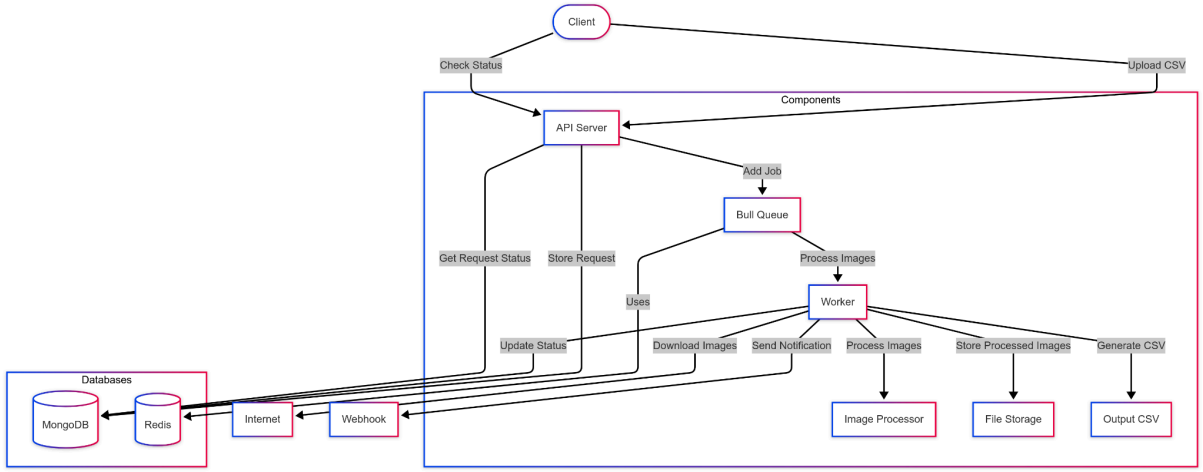
- **Internal Server Error (500)**:
```json
{
  "success": false,
  "message": "An error occurred while checking status",
  "error": "Error message details"
}
```

Status Codes
- `pending`: Request has been received but processing has not yet started
- `processing`: Request is currently being processed
- `completed`: Request has been fully processed successfully
- `failed`: Request processing failed

Webhook Integration
If enabled, a webhook notification will be sent to the configured URL when processing is completed.

Webhook Payload:
```json
{
  "requestId": "13f5b378-1d34-4601-a8c3-5678bb1234cd",
  "status": "completed",
  "totalImages": 10,
  "processedImages": 10,
  "outputCsvUrl":
"http://localhost:3000/processed/13f5b378-1d34-4601-a8c3-5678bb1234cd-output.csv",
  "completedAt": "2023-10-10T10:20:45.000Z"
}
```

Architecture



Component Breakdown

Client Interaction

The client interacts with the system through two primary operations:

- Upload CSV: Submits a CSV file containing product information and image URLs
- Check Status: Queries the status of a previously submitted processing request

API Server Component

Role

The API Server serves as the primary entry point for all client requests. It handles file uploads, validates input, initiates processing jobs, and provides status updates.

Functions

- Validates CSV files to ensure they follow the required format
- Generates unique request IDs for each processing job
- Adds jobs to the processing queue
- Provides status information about ongoing or completed jobs
- Serves static files (processed images and output CSV files)

Implementation Details

- Built using Express.js
- Uses Multer for file upload handling

- Implements validation middleware for CSV verification
- Generates unique IDs via UUID

Bull Queue Component

Role

The Bull Queue manages the asynchronous processing of image data, ensuring the system can handle multiple processing requests without blocking.

Functions

- Manages the queue of image processing jobs
- Distributes work to available workers
- Handles retries and failures
- Tracks job progress and status

Implementation Details

- Uses Bull, a Redis-backed queue for Node.js
- Configures job processing with appropriate concurrency
- Implements event handlers for job completion and failure
- Maintains job state in Redis

Worker Component

Role

The Worker component processes the images referenced in the CSV file, compressing them and tracking progress.

Functions

- Downloads images from provided URLs
- Processes images using Sharp (compression to 50% quality)
- Updates request status and progress
- Generates output CSV with processed image URLs
- Triggers webhook notifications upon completion

Implementation Details

- Uses Sharp for image processing
- Uses Axios for downloading images
- Updates MongoDB with progress information
- Creates processed image directory structure
- Generates output CSV with csv-writer

Webhook Component

Role

The Webhook component notifies external systems when processing is complete.

Functions

- Sends HTTP POST requests to configured webhook URLs
- Includes processing result details in the payload
- Tracks webhook delivery status

Implementation Details

- Uses Axios for HTTP requests
- Configurable via environment variables
- Includes comprehensive job metadata in the payload

Database Components

MongoDB

- Role: Persistent storage for request and product data
- Functions:
    - Stores request metadata and status
    - Tracks processing progress
    - Stores product information and image URLs
    - Enables status queries

Redis

- Role: Handles job queues and temporary storage
- Functions:
    - Backs the Bull queue implementation
    - Stores job state and progress information
    - Enables fast queue operations

File Storage Component

Role

The File Storage component manages the storage of uploaded CSV files, processed images, and output CSV files.

Functions

- Stores uploaded CSV files
- Organizes processed images in a structured directory format
- Provides public URLs for accessing processed files
- Manages file cleanup

Implementation Details

- Uses the local filesystem organized by request ID
- Exposes processed files via static file serving
- Creates required directories on demand

Data Flow

Upload Process Flow

1. Client uploads a CSV/TXT file to the /api/upload endpoint
2. API Server validates the CSV format and content
3. System generates a unique request ID
4. API Server creates a new Request record in MongoDB with 'pending' status
5. API Server adds a job to the Bull Queue
6. API Server returns the request ID to the client
7. Worker pulls the job from the queue
8. Worker updates the Request record to 'processing' status
9. Worker parses the CSV file and creates Product records
10. Worker processes each image (download, compress, save)
11. Worker updates progress in the Request record
12. Worker generates an output CSV file with processed image URLs
13. Worker updates the Request record to 'completed' status
14. Worker sends a webhook notification if configured

Status Check Flow

1. Client queries the /api/status/:requestId endpoint
2. API Server retrieves the Request record from MongoDB
3. API Server returns the current status, progress, and (if completed) output file URL

Asynchronous Worker Processes

Image Processing Worker

Function: src/workers/imageProcessor.js

Description: Processes images from the queue one by one, updating progress as it goes.

Processing Steps:

1. Update request status to 'processing'
2. Parse the CSV file to extract product and image information
3. For each product:
    - Create a Product record
    - Update status to 'processing'
    - Process each image URL:

        - Download the image
        - Compress using Sharp
        - Save to output directory
        - Update progress in Request record
    - Update the Product record with output URLs
4. Generate output CSV file
5. Update request status to 'completed'
6. Send webhook notification if configured

Error Handling:

- Failed image downloads or processing are logged and marked
- Overall request status is set to 'failed' if critical errors occur
- Individual image failures don't fail the entire job

Scalability & Performance Considerations

Queue Configuration

The Bull queue can be configured with concurrency settings to control resource usage:

```
const queue = new Queue('image-processing', {
  concurrency: 5  // Process 5 jobs simultaneously
});
```

Worker Scaling

For increased throughput, multiple worker processes can be deployed to process the queue. This can be managed using:

- Multiple instances of the application
- PM2 cluster mode
- Docker containers with orchestration (Kubernetes)

Database Indexing

The MongoDB collections are indexed on requestId to ensure fast lookups for status checks.

Potential Improvements

- **File Storage**: Integrate with cloud storage (AWS S3, Google Cloud Storage)
- **Authentication**: Add API keys or JWT-based auth for API access
- **Job Management**: Add endpoints to cancel or reprioritize jobs
- **Progress Streaming**: Implement WebSockets for real-time progress updates
- **Failure Recovery**: Add ability to resume failed jobs
- **Multi-tenancy**: Support for multiple users/organizations