Pictoora API

A FastAPI-based image processing API that handles image manipulation using OpenAI's DALL-E model. This project includes features like file upload, background processing, TTL-based caching, and structured logging.

Table of Contents

- Validation Rules
- Cache System

Features

✨ Core Features

- Health check system
- Secure file upload with validation
- Image processing with OpenAI GPT-Image-1 model
- TTL-based caching with thread-safe operations
- Structured logging with rotation
- API key authentication middleware
- Background task processing
- Standardized response format
- Custom status codes
- CORS support
- Static file serving
- Process status tracking
- Cache monitoring

Project Structure

```
pictoora/
├── app/
│   ├── api/
│   │   └── endpoints/          # API endpoint handlers
│   │       ├── health.py      # Health check endpoint
│   │       ├── upload.py      # File upload endpoint
│   │       ├── process.py     # Image processing endpoints
│   │       └── cache.py       # Cache monitoring endpoint
│   ├── core/                   # Core functionality
│   │   ├── config.py         # Configuration settings
│   │   ├── logger.py         # Logging setup
│   │   └── cache.py         # TTL Cache implementation
│   ├── middleware/           # Middleware components
│   │   └── api_key.py       # API key authentication
│   ├── schemas/             # Data models
│   │   └── responses.py     # Response schemas
│   └── main.py             # Main application entry
├── storage/                # Storage directory
│   └── uploads/           # Uploaded files location
```

```
├── logs/                  # Application logs
├── .env                   # Environment variables
├── requirements.txt       # Project dependencies
└── README.md             # Project documentation
```

Setup Instructions

Prerequisites

- Python 3.8 or higher
- OpenAI API key
- Git

Windows Setup

1. Clone the repository:

```
https://github.com/euitsol/pictoora-ai-engine.git
cd pictoora-ai-engine
```

2. Create and activate virtual environment:

```
python -m venv venv
.\venv\Scripts\activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

Mac/Linux Setup

1. Clone the repository:

```
https://github.com/euitsol/pictoora-ai-engine.git
cd pictoora-ai-engine
```

2. Create and activate virtual environment:

```
python3 -m venv venv
source venv/bin/activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

Configuration

1. Create a `.env` file in the root directory:

```
# App Configuration
APP_NAME=Pictoora
APP_URL=http://localhost:8000

# API Authentication
API_KEY=your_api_key_here

# OpenAI Configuration
OPENAI_API_KEY=your_openai_api_key
```

Running the Project

1. Activate virtual environment (if not already activated):

- Windows: `.\venv\Scripts\activate`
- Mac/Linux: `source venv/bin/activate`

2. Start the FastAPI server:

```
uvicorn app.main:app --reload
```

3. Access the API:

- API: http://localhost:8000
- Swagger Documentation: http://localhost:8000/docs
- ReDoc Documentation: http://localhost:8000/redoc

How It Works

1. **File Upload Process**:
    - Client uploads image files through `/upload` endpoint
    - System validates file type (png, jpg, jpeg)
    - Files are stored in `storage/uploads` with unique names
    - Returns file ID for future reference
2. **Image Processing Workflow**:

```
[Client]
   |
   ▼
[1. Initiate Process] (/initiate-process)
   |
   ▼
[2. Upload Files] (/upload)
   |
   ▼
[3. Process Book] (/process/book)
   |
   ▼
[4. Background Processing]
   |
   ▼
[5. Check Status] (/process/status)
```

API Endpoints

1. **Root**

- Path: `GET /api/v1/`
- Purpose: Welcome message
- Auth: Not required

2. **Health Check**

   - Path: `GET /api/v1/health`
   - Purpose: System health monitoring
   - Auth: Not required

3. **File Upload**

   - Path: `POST /api/v1/upload`
   - Purpose: Upload image files
   - Auth: Required (X-API-Key header)
   - Returns: File path and URL

4. **Process Initiation**

   - Path: `POST /api/v1/initiate-process`
   - Purpose: Start new processing session
   - Auth: Required
   - Returns: Unique init_id

5. **Book Processing**

   - Path: `POST /api/v1/process/book`
   - Purpose: Process images with DALL-E
   - Auth: Required
   - Body: init_id, source_url, target_url, prompt
   - Returns: Process status

6. **Process Status**

   - Path: `POST /api/v1/process/status`
   - Purpose: Check processing status
   - Auth: Required
   - Returns: Current status and result URL

7. **Cache Status**

   - Path: `GET /api/v1/cache/status`

- Purpose: Monitor cache system
- Auth: Required
- Returns: Cache statistics and entries

Status Codes

Code	Endpoint	Description
1000	/	Welcome message
1001	/	System error
1000	/health	System healthy
1001	/health	System unhealthy
2000	/upload	Upload successful
2001	/upload	Upload failed
3000	/initiate-process	Process initiated
3001	/initiate-process	Initiation failed
4000	/process/book	Processing started
4001	/process/book	Processing failed
5000	/process/status	Status retrieved
5001	/process/status	Status retrieval failed
6000	/cache/status	Cache status retrieved
6001	/cache/status	Cache status failed

Validation Rules

1. **File Upload**:
   - Allowed extensions: png, jpg, jpeg
   - Automatic unique filename generation
   - File size: Limited by FastAPI default
   - Proper file path handling with Path

2. **API Authentication**:
    - Header: X-API-Key
    - Required for all endpoints except / and /health
    - Configurable through .env file
    - Middleware-based validation
3. **Process Validation**:
    - Valid init_id required for processing
    - Source and target files must exist
    - Valid OpenAI API key required
    - Background task processing
    - Status tracking with TTL cache

Cache System

1. **TTL (Time-To-Live) Cache**:
    - Default TTL: 1 hour (3600 seconds)
    - Maximum cache size: 1000 items
    - Thread-safe implementation
    - Custom TTL support per item
    - Automatic cleanup of expired items
    - Memory usage monitoring
    - Cache statistics endpoint
2. **Cache Operations**:
    - get/set operations with O(1) complexity
    - Custom expiration time support
    - Error handling and logging
    - Status monitoring
    - Memory-efficient storage
    - No external service dependencies