**Logic of the FastAPI Application**

The application is designed to allow users to upload receipt images, extract text from those images using Optical Character Recognition (OCR), and save the results (image path, extracted text, total amount) into a database. Here’s how the application flows:

1. **User Interface**: The user accesses a web interface where they can upload a receipt image.
2. **File Upload**: When the user submits the form, the image is sent to the FastAPI backend for processing.
3. **Image Processing**:
   * The application saves the uploaded image to a designated folder.
   * It uses an OCR library (EasyOCR in this case) to read the text from the image.
4. **Extracted Data**:
   * The extracted text is processed to identify the total amount.
   * The application creates a record containing the image path, extracted text, and total amount.
5. **Database Interaction**:
   * The record is saved into a database using ORM to handle database operations more easily.
6. **Response**: After processing, the application returns a response to the user, showing the filename, extracted text, and total amount.

**Understanding ORM**

**Object-Relational Mapping (ORM)** is a programming technique used to convert data between incompatible type systems (i.e., between your object-oriented programming language and a relational database).

**Key Features of ORM:**

1. **Data Abstraction**: ORM provides a high-level abstraction over database interactions. You work with Python objects instead of writing SQL queries directly.
2. **Database Schema Representation**: ORM allows you to define your database schema as Python classes. Each class corresponds to a table in the database.
3. **Automatic SQL Generation**: ORM frameworks can automatically generate SQL statements for you based on the operations you perform on the Python objects.
4. **Database Compatibility**: ORMs can work with different types of databases without needing to change your application code.

**Making a Connection to the Database**

1. **Define the Database URL**: This is a string that contains information about the type of database you’re using, its location, and the database name.

python

Copier le code

DATABASE\_URL = "sqlite:///./receipts.db"

* + Here, we use SQLite, a lightweight database that stores data in a single file.

1. **Create the Database Engine**: The engine is responsible for communicating with the database.

python

Copier le code

engine = create\_engine(DATABASE\_URL)

1. **Session Management**: A session is a temporary connection to the database used for executing commands. Sessions are created using the session factory.

python

Copier le code

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

1. **Define ORM Models**: Using SQLAlchemy's declarative base, you define Python classes that represent tables in the database.

python

Copier le code

Base = declarative\_base()

1. **Create Database Tables**: The create\_all method generates the necessary SQL to create the tables defined by your ORM models.

python

Copier le code

Base.metadata.create\_all(bind=engine)

**Using the EasyOCR Library**

**EasyOCR** is a Python library that provides a simple interface for performing Optical Character Recognition on images. Here’s how it fits into our application:

1. **Initialization**: When you create an EasyOCR reader instance, you specify the languages you want to support. For example:

python

Copier le code

reader = easyocr.Reader(['en'])

This initializes the OCR engine for English text.

1. **Reading Text from Images**: When you call the readtext method on an image, EasyOCR processes the image and returns the detected text.

python

Copier le code

result = reader.readtext(file\_location, detail=0)

* + Setting detail=0 returns only the recognized text, not additional information like bounding boxes.

**Step-by-Step Code Explanation**

Now, let’s break down the code step by step with this understanding in mind. Here's how the different parts of the code fit into the overall logic:

1. **Importing Libraries**: We import necessary libraries for web handling (FastAPI), file handling (PIL), and database interaction (SQLAlchemy).
2. **Creating the FastAPI Instance**:

python

Copier le code

app = FastAPI()

1. **Database Configuration**: Set up the database URL and create the engine and session for interacting with the database.
2. **Defining the ORM Model**: Create a Receipt class that defines how data is structured in the database.
3. **Mounting Static Files**: This allows the application to serve static files (like images) so users can access them later.
4. **Defining Routes**:
   * **Root Route**: The home page where users can upload their receipts.
   * **Upload Route**: The endpoint that processes the uploaded image, extracts text using EasyOCR, and saves the data in the database.
5. **Response Handling**: After processing the image, the application returns a response containing useful information to the user.
6. **Running the Application**: Finally, we run the FastAPI application with Uvicorn.

**What is Uvicorn?**

**Uvicorn** is a lightning-fast ASGI (Asynchronous Server Gateway Interface) server implementation for Python, designed to serve web applications built with frameworks like FastAPI and Starlette. It is built on top of httptools and uvloop, making it efficient and suitable for asynchronous programming.

**Key Features of Uvicorn:**

1. **Asynchronous Support**: Uvicorn supports asynchronous programming, allowing for non-blocking I/O operations, which is beneficial for handling multiple requests simultaneously.
2. **Fast Performance**: It is designed for high performance, leveraging the speed of uvloop (an event loop) and httptools (an HTTP parsing library).
3. **WebSocket Support**: Uvicorn supports WebSockets, enabling real-time communication between clients and servers.
4. **Easy to Use**: Running a FastAPI application with Uvicorn is straightforward, often requiring just a single command.
5. **Hot Reloading**: Uvicorn can be run in development mode, which automatically reloads the server when code changes are detected, making development more efficient.
6. **Multiple Workers**: It can run multiple worker processes to handle more requests simultaneously, making it suitable for production use.

**Running FastAPI with Uvicorn**

To run a FastAPI application with Uvicorn, you typically follow these steps:

1. **Install Uvicorn**: If you haven’t already, you can install Uvicorn using pip:

bash

Copier le code

pip install uvicorn

1. **Create Your FastAPI Application**: For example, you might have a simple FastAPI application like this in a file named main.py:

python

Copier le code

from fastapi import FastAPI

app = FastAPI()

@app.get("/")

def read\_root():

return {"Hello": "World"}

1. **Run the Application**: Use Uvicorn to run your FastAPI application from the command line:

bash

Copier le code

uvicorn main:app --reload

* + Here, main refers to the filename (without .py), and app is the FastAPI instance.
  + The --reload flag enables hot reloading.

**Key Concepts of FastAPI**

**FastAPI** is a modern, high-performance web framework for building APIs with Python based on standard Python type hints. Here’s a comprehensive overview of its features and concepts:

**1. Fast Performance:**

* FastAPI is built on Starlette for the web parts and Pydantic for the data parts, making it one of the fastest web frameworks available.
* It can handle thousands of requests per second, making it ideal for high-performance applications.

**2. Automatic Validation:**

* FastAPI automatically validates request data against the defined types using Pydantic models.
* If a request does not conform to the expected types, FastAPI returns a clear error message.

**3. Type Hints:**

* By using Python type hints, FastAPI generates interactive API documentation and provides editor support (e.g., autocompletion) for request validation and serialization.

**4. Interactive Documentation:**

* FastAPI automatically generates Swagger UI and ReDoc documentation for your API, making it easier for developers to understand and test the API endpoints.
* You can access the Swagger UI at http://localhost:8000/docs and ReDoc at http://localhost:8000/redoc.

**5. Dependency Injection:**

* FastAPI has a powerful dependency injection system that allows you to declare dependencies for your endpoints. This makes it easy to share components like database connections and authentication schemes across your application.

**6. Asynchronous Support:**

* FastAPI is fully compatible with async and await, allowing you to build asynchronous endpoints and handle I/O-bound tasks efficiently.

**7. Path and Query Parameters:**

* FastAPI makes it easy to define path and query parameters in your routes, automatically converting types and validating input.

**8. Middleware Support:**

* You can easily add middleware components to your FastAPI application to handle cross-cutting concerns like authentication, logging, and CORS (Cross-Origin Resource Sharing).

**9. Background Tasks:**

* FastAPI provides support for background tasks, allowing you to run tasks after a request is completed, such as sending emails or processing data.

**Example FastAPI Application with Uvicorn**

Here’s a complete example of a FastAPI application using Uvicorn:

python

Copier le code

# main.py

from fastapi import FastAPI, UploadFile, File

from fastapi.responses import JSONResponse

import easyocr

app = FastAPI()

reader = easyocr.Reader(['en'], gpu=False)

@app.post("/upload/")

async def upload\_receipt(file: UploadFile = File(...)):

contents = await file.read()

# Here, you would process the image with EasyOCR

result = reader.readtext(contents, detail=0) # Extract text from image

extracted\_text = ' '.join(result)

return JSONResponse(content={"extracted\_text": extracted\_text})

# To run the application with Uvicorn

# Command to run in terminal: uvicorn main:app --reload

**Conclusion**

Uvicorn is a key component for serving FastAPI applications, providing performance and scalability through asynchronous support. FastAPI, on the other hand, offers a robust set of features for building APIs quickly and effectively. Together, they create a powerful combination for modern web development in Python. If you have any further questions or need clarification on specific topics, feel free to ask!

Let's break down the FastAPI code step by step, explaining the purpose of each line and component involved, including how to use the ORM (Object-Relational Mapping) for database operations, and the general flow of the application.

**Code Breakdown**

python

Copier le code

from fastapi import FastAPI, UploadFile, File

from fastapi.responses import HTMLResponse

from fastapi.staticfiles import StaticFiles

from sqlalchemy import create\_engine, Column, Integer, String

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

import easyocr

from PIL import Image

import os

1. **Imports**:
   * FastAPI: The main class to create the web application.
   * UploadFile, File: Classes for handling file uploads.
   * HTMLResponse: Allows sending HTML responses.
   * StaticFiles: Used to serve static files (like images).
   * create\_engine, Column, Integer, String: SQLAlchemy components for database interactions.
   * declarative\_base: A factory function that creates a base class for declarative class definitions.
   * sessionmaker: A factory for creating new Session objects for database interaction.
   * easyocr: The OCR library for extracting text from images.
   * Image from PIL: For image processing (though not used in this code).
   * os: Used for operating system functionalities like file path manipulation.

python

Copier le code

app = FastAPI()

1. **Creating the FastAPI Instance**:
   * This line initializes a new FastAPI application instance, allowing you to define routes and manage requests.

**Database Setup**

python

Copier le code

DATABASE\_URL = "sqlite:///./receipts.db"

engine = create\_engine(DATABASE\_URL)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

1. **Database Configuration**:
   * DATABASE\_URL: Defines the database connection URL (in this case, an SQLite database named receipts.db).
   * create\_engine(DATABASE\_URL): Creates a SQLAlchemy engine, which will be used to communicate with the database.
   * sessionmaker(...): Creates a new session factory to interact with the database. Sessions are used to query and save instances.
   * declarative\_base(): Creates a base class for our database models (tables).

python

Copier le code

class Receipt(Base):

\_\_tablename\_\_ = "receipts"

id = Column(Integer, primary\_key=True, index=True)

image\_path = Column(String)

extracted\_text = Column(String)

total\_amount = Column(String)

1. **Defining the Receipt Model**:
   * The Receipt class represents the receipts table in the database.
   * id: An auto-incrementing primary key for each receipt.
   * image\_path: Stores the path of the uploaded image.
   * extracted\_text: Stores the text extracted from the image.
   * total\_amount: Stores the detected total amount from the receipt.

python

Copier le code

Base.metadata.create\_all(bind=engine)

1. **Creating the Database Table**:
   * This line creates the database table(s) defined by the Receipt class if they do not already exist.

**Static Files**

python

Copier le code

app.mount("/static", StaticFiles(directory="static"), name="static")

1. **Serving Static Files**:
   * This line mounts the /static route to serve files from the static directory. Any files uploaded will be saved in this directory and can be accessed via this route.

**Main Application Routes**

**Root Route**

python

Copier le code

@app.get("/", response\_class=HTMLResponse)

async def read\_root():

return """

<html>

<head>

<title>Receipt Scanner</title>

</head>

<body>

<h1>Upload Receipt Image</h1>

<form action="/upload/" enctype="multipart/form-data" method="post">

<input type="file" name="file" accept="image/\*" required>

<input type="submit" value="Upload">

</form>

</body>

</html>

"""

1. **Root Route**:
   * This route serves an HTML form for uploading receipt images.
   * The form submits to the /upload/ endpoint via a POST request with the file data.

**Upload Route**

python

Copier le code

@app.post("/upload/")

async def upload\_image(file: UploadFile = File(...)):

file\_location = f"static/{file.filename}"

with open(file\_location, "wb") as file\_object:

file\_object.write(file.file.read())

1. **Upload Route**:
   * This route handles file uploads from the form.
   * file: UploadFile = File(...): This defines a parameter to receive the uploaded file.
   * The uploaded file is saved to the static directory.

**Text Extraction and Database Interaction**

python

Copier le code

# Process image and extract text

reader = easyocr.Reader(['en'])

result = reader.readtext(file\_location, detail=0)

extracted\_text = ' '.join(result)

1. **Image Processing**:
   * An easyocr.Reader instance is created to handle English text recognition.
   * The image is processed to extract text, which is stored in extracted\_text.

python

Copier le code

# Here, implement logic to find the total amount in the extracted\_text

# For demonstration, we'll just use a placeholder

total\_amount = "Total Amount Detected" # Replace with actual detection logic

1. **Placeholder for Total Amount Detection**:
   * This comment indicates where you would implement logic to detect the total amount from the extracted text. Currently, it's a placeholder.

python

Copier le code

# Save to database

db = SessionLocal()

receipt = Receipt(image\_path=file\_location, extracted\_text=extracted\_text, total\_amount=total\_amount)

db.add(receipt)

db.commit()

db.close()

1. **Database Interaction**:
   * A new database session is created using SessionLocal().
   * A new Receipt instance is created with the extracted data.
   * The instance is added to the session and committed to save it in the database.
   * The session is closed afterward to free resources.

python

Copier le code

return {"filename": file.filename, "extracted\_text": extracted\_text, "total\_amount": total\_amount}

1. **Response**:
   * After processing, the endpoint returns a JSON response containing the filename, extracted text, and detected total amount.

**Running the Application**

python

Copier le code

if \_\_name\_\_ == "\_\_main\_\_":

import uvicorn

uvicorn.run(app, host="0.0.0.0", port=8000)

1. **Running the Server**:
   * This block checks if the script is being run directly and starts the Uvicorn server, making the FastAPI application accessible at http://0.0.0.0:8000.

**Summary**

* This FastAPI application allows users to upload receipt images, processes them to extract text using EasyOCR, and stores the relevant data in a SQLite database.
* Each part of the application is designed to handle specific tasks: defining the API endpoints, processing file uploads, extracting text, and interacting with the database using SQLAlchemy.

If you have further questions about specific parts of the code or would like to explore any feature in more depth, feel free to ask