Project Documentation: OCR Django Application

### **Overview of the Project**

This Optical Character Recognition (OCR) project is designed to extract text from PDF documents and store the extracted information into a database. Built using the Django framework, the project leverages Celery for asynchronous task management, RabbitMQ as the message broker, and Docker for containerization. The application ensures efficient and scalable processing of OCR tasks while maintaining a robust architecture for handling user requests and database operations.

### **Workflow and Thought Process**

1. Environment Setup:
   * Establish a virtual environment to manage project-specific dependencies.
   * Install necessary Python packages as specified in requirements.txt.
2. Asynchronous Task Management:
   * Utilize Celery to handle long-running OCR tasks asynchronously, ensuring the main application remains responsive.
   * Configure RabbitMQ as the message broker to enable smooth communication between Django and Celery workers.
3. Database Integration:
   * Use Django ORM to create database models for storing extracted text and related metadata.
   * Apply database migrations to set up the schema.
4. API Development:
   * Develop RESTful APIs to handle file uploads and trigger OCR processing.
   * Implement endpoints for storing and retrieving extracted text from the database.
5. Containerization:
   * Create a Dockerfile to define the application environment.
   * Use docker-compose.yml to orchestrate multi-container deployment for Django, Celery, and RabbitMQ services.
6. Testing and Validation:
   * Test the API endpoints using the provided Postman collection (Assignment.postman\_collection.json) to ensure they function as intended.

### **Technical Documentation**

Architecture:

* Django: Web framework for handling HTTP requests, routing, and ORM-based database interactions.
* Celery: Framework for asynchronous task processing, used for running OCR tasks in the background.
* RabbitMQ: Message broker for communication between Django and Celery.
* Docker: Containerization tool to encapsulate the application and its dependencies.

Dependencies:

* Listed in requirements.txt, including:
  + Django
  + Celery
  + RabbitMQ
  + Other Python libraries required for OCR and API functionality.

Setup Instructions:

1. Clone the repository:

git clone https://github.com/Kmalwade/OCR.git

1. Navigate to the project directory:

cd OCR/

1. Set up a virtual environment:

python -m venv env

source env/bin/activate # On Unix or MacOS

env\Scripts\activate # On Windows

1. Install dependencies:

pip install -r requirements.txt

1. Install RabbitMQ:
   * Download and install RabbitMQ from the official website.
   * Ensure Erlang is installed as it is a prerequisite.
2. Apply database migrations:

python manage.py makemigrations

python manage.py migrate

1. Start services:
   * Django Server:

python manage.py runserver 8000

* + Celery Worker:

celery -A OCR.celery worker --loglevel=info

* + RabbitMQ Server:

rabbitmq-server start

1. Use Docker (Optional):
   * Build and start the containers using Docker Compose:

docker-compose up --build

Testing:

* Use the provided Postman collection to test the following API endpoints:
  + http://localhost:8000/api/v1/OCRExtraction
  + http://localhost:8000/api/v1/StoreIntoDB

### **Key Learnings from the Project**

* Asynchronous Processing: Leveraging Celery with RabbitMQ enables efficient handling of long-running tasks without blocking the application.
* Containerization: Docker ensures consistency in development and production environments, simplifying deployment and scaling.
* API Design: Structuring RESTful APIs provides a scalable and maintainable way to interact with the application.

### **Time Spent on the Project**

* Environment Setup: 2 hours
* Asynchronous Task Integration: 3 hours
* Database Configuration and Migrations: 1.5 hours
* API Development and Testing: 3 hours
* Containerization with Docker: 2 hours
* Documentation and Debugging: 2 hours

### **Explanation of What Works Well**

1. Asynchronous OCR Processing:
   * The use of Celery ensures that OCR tasks do not block the main thread, leading to a responsive application.
2. Scalable Architecture:
   * The combination of Django, RabbitMQ, and Celery offers a robust and scalable architecture for handling large volumes of tasks.
3. Containerization:
   * Docker simplifies the setup process and ensures consistency across development and production environments.
4. API Functionality:
   * The RESTful API endpoints provide clear and intuitive ways to interact with the application.

### **Known Limitations**

1. Resource Intensity:
   * The OCR process can be resource-intensive, requiring adequate hardware for optimal performance.
2. Error Handling:
   * Limited error handling for edge cases in file uploads and database operations.
3. Scalability with RabbitMQ:
   * While RabbitMQ handles current loads effectively, very high traffic may require additional scaling strategies.