# **Object Detection Microservice Documentation**

## **Overview**

This project implements a microservice architecture consisting of two main components:

1. **UI Backend Service**: Handles user image uploads and communicates with the AI backend.
2. **AI Backend Service**: Performs object detection on images using YOLOv3 and returns results in a structured JSON format.

The solution is containerized using Docker, and the two services communicate seamlessly to provide a complete object detection solution.

I have tested the FastAPI with the Swagger to test the model.

## **Technologies Used**

* **Docker**: For containerization of both the UI and AI backend services.
* **Python**:
  + **FastAPI**: For building lightweight backend services (UI and AI).
  + **PyTorch**: For loading and running the YOLOv3 model.
* **YOLOv3**: An open-source, real-time object detection model (source: [Ultralytics YOLOv3 GitHub](https://github.com/ultralytics/yolov3)).
* **Docker Compose**: For managing multi-container services (UI backend and AI backend).

## **Architecture**

### **UI Backend Service**

* Built using **FastAPI**.
* Accepts image uploads through a POST request at /upload endpoint.
* Forwards the uploaded image to the **AI Backend Service** for object detection.
* Receives detection results (bounding box coordinates and labels) in JSON format.
* Responds back to the client with the detection results.

### **AI Backend Service**

* Also built using **FastAPI**.
* Loads the YOLOv3 pretrained model using **PyTorch**.
* Accepts an image via a POST request at /detect endpoint.
* Processes the image using YOLOv3 to perform object detection.
* Draws bounding boxes on the image and saves it to the output/ folder.
* Returns a JSON response with object detection details (e.g., class labels, coordinates).

## **Solution Steps**

### **Step 1: Set Up Project Structure**

The project is structured into two main components:

* ui\_backend/ for the UI backend service.
* ai\_backend/ for the AI backend service.
* docker-compose.yml for orchestrating both services.
* swagger for testing api’s.
* output/ to store generated images and JSON results.

### **Step 2: Implement UI Backend**

The UI Backend service is responsible for accepting image uploads and passing them to the AI Backend.

1. Create a FastAPI app in ui\_backend/app.py.
2. Set up an API endpoint /upload to handle image uploads.
3. Use the **requests** library to forward the image to the AI Backend for detection.
4. Return the detection results to the user.

**UI Backend Code**:

from fastapi import FastAPI, File, UploadFile, HTTPException

import httpx

app = FastAPI()

AI\_BACKEND\_URL = "http://127.0.0.1:5001/detect"

@app.post("/upload")

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

if not image:

raise HTTPException(status\_code=400, detail="No image uploaded")

try:

async with httpx.AsyncClient() as client:

response = await client.post(AI\_BACKEND\_URL, files={"image": (image.filename, image.file, image.content\_type)})

return response.json()

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Error connecting to AI backend: {str(e)}")

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

import uvicorn

uvicorn.run(app, host="localhost", port=5000)

### **Step 3: Implement AI Backend**

The AI Backend service performs object detection using YOLOv3.

1. Use **PyTorch** to load the pretrained YOLOv3 model from Ultralytics.
2. Accept an image through a POST request.
3. Perform object detection on the image.
4. Draw bounding boxes around detected objects and save the image.
5. Return a JSON response containing detection information.

**AI Backend Code**:

from fastapi import FastAPI, File, UploadFile

from fastapi.responses import JSONResponse

import torch

from PIL import Image

import os

import json

import shutil

# Initialize FastAPI app

app = FastAPI()

# Load YOLOv3 model (assuming 'yolov5s' is the correct model name)

model = torch.hub.load('ultralytics/yolov3', 'yolov5s', pretrained=True)

# Output directory

OUTPUT\_DIR = "./output"

os.makedirs(OUTPUT\_DIR, exist\_ok=True)

@app.post("/detect")

async def detect(image: UploadFile = File(...)):

try:

# Load the uploaded image

img = Image.open(image.file)

results = model(img)

# Define the output image path in the output folder

output\_image\_path = os.path.join(OUTPUT\_DIR, "output.jpg")

# Save the result image (with bounding boxes) to the output folder

results.save(OUTPUT\_DIR) # This saves the image with bounding boxes to the output folder

detection\_image\_path = os.path.join(OUTPUT\_DIR, "output.jpg") # Set the specific image name for the detection

# If results.save() creates a file with a different name, we copy it to output.jpg

if not os.path.exists(detection\_image\_path): # If the image doesn't exist, we copy the first image

saved\_images = os.listdir(OUTPUT\_DIR)

for img\_file in saved\_images:

if img\_file.endswith(".jpg"): # Look for the saved image with .jpg extension

shutil.copy(os.path.join(OUTPUT\_DIR, img\_file), detection\_image\_path)

# Extract detections

detections = results.pandas().xyxy[0].to\_dict(orient="records")

# Save the detections as a JSON file

output\_json\_path = os.path.join(OUTPUT\_DIR, "detections.json")

with open(output\_json\_path, 'w') as json\_file:

json.dump({"detections": detections}, json\_file, indent=4)

# Return response as JSON with image and JSON file paths

return JSONResponse(content={

"detections": detections,

"image\_path": detection\_image\_path,

"json\_path": output\_json\_path

})

except Exception as e:

return JSONResponse(content={"error": str(e)}, status\_code=500)

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

import uvicorn

uvicorn.run(app, host="localhost", port=5001)

### **Step 4: Dockerize Both Services**

1. **UI Backend Dockerfile**:  
   * Use Python 3.8.
   * Install FastAPI and requests libraries.
   * Expose port 5000 for communication.
2. **AI Backend Dockerfile**:  
   * Use Python 3.8.
   * Install Flask, PyTorch, and other necessary dependencies.
   * Expose port 5001 for communication.

### **Step 5: Docker Compose**

To run both services together:

* **docker-compose.yml** defines the services and their dependencies.

**Docker Compose File**:

services:

ui\_backend:

build: ./ui\_backend

ports:

- "5000:5000"

depends\_on:

- ai\_backend

ai\_backend:

build: ./ai\_backend

ports:

- "5001:5001"

## **Testing the Application**

**Run both the Servers and open the Swagger of the UI Backend API**:  
  
 http://localhost:5000/docs

1. **Send a POST request to upload an image**:  
   * URL: http://localhost:5000/upload
   * Method: POST
   * Body: Form-data with image as the key and the image file as the value.
   * Example using **Postman(also tested)**.
2. **Check the outputs**:  
   * The response will include detection data and the path to the output image with bounding boxes.
   * The image will be saved in the output/ directory and run/ directory.
3. **Output Example**:  
   * JSON response contains the detection results (class labels, confidence scores, bounding box coordinates).
   * An image with bounding boxes will be saved under output/output.jpg and run/ directory.

## **References**

* YOLOv3 GitHub:<https://github.com/ultralytics/yolov3>
* PyTorch Documentation:<https://pytorch.org/docs/stable/index.html>

## **Conclusion**

This solution provides a simple microservice architecture for object detection using YOLOv3. The two components (UI and AI backend) are containerized using Docker and communicate seamlessly to process images and return detection results.

This documentation should cover everything needed to understand and replicate the solution.