**Project Documentation: Object Detection Microservice with FastAPI & YOLOv3**

**1. Introduction**

In this project, I implemented an object detection microservice using FastAPI and a lightweight YOLOv3 model.  
I designed the architecture as two separate services:

1. UI Backend Service – which handles image uploads from the user.
2. AI Backend Service – performs object detection using **YOLOv3** and returns results in JSON format.

Both services are containerized using Docker so that they can be easily replicated and deployed on any system.

**2. My Approach**

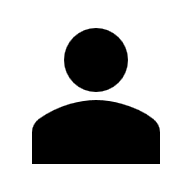
I broke the task into the following steps:

1. **Choosing the Model**: I selected **YOLOv3 (Ultralytics repo)** since it is lightweight, open-source, and works well on CPU when GPU is not available.
2. **Building the AI Backend**: Created a **Fast API** service that loads the YOLOv3 model, preprocesses input images, runs inference, and outputs bounding boxes + class predictions.
3. **Building the UI Backend**: Built another FastAPI service that accepts image uploads from the user and internally communicates with the AI backend.
4. **Containerization**: Created Dockerfiles for both services and used Docker Compose to run them together.
5. **Testing**: Uploaded test images to verify that bounding boxes are correctly generated and JSON responses are returned.
6. **Architecture**

UI backend  
(/upload)

AI backend  
(/detect)

**User**

****

**Yolo model**

**Output images JSON file**

* The **UI Backend** only forwards requests and responses.
* The **AI Backend** is responsible for running the detection model.
* This separation keeps the services modular and easier to scale.

**4. Implementation Details**

**📌 4.1 AI Backend (Detection Service)**

For the AI backend, I used FastAPI. The model is loaded once when the service starts.

Key steps I implemented:

* Read the uploaded image and decode it with OpenCV.
* Preprocess the image (resize with letterbox, normalize).
* Run YOLOv3 inference using PyTorch.
* Apply Non-Maximum Suppression (NMS) to filter predictions.
* Return bounding boxes, confidence, and class IDs as JSON along with output predicted image.

**📌 4.2 UI Backend (User Service)**

The UI backend is a lightweight FastAPI service.

What I implemented here:

* Expose an /upload/ endpoint.
* Accept the image file from the user.
* Forward it to the AI backend using a POST request.
* Return the AI backend’s JSON response back to the user.

This way, the UI backend acts as the “bridge” between the user and the AI service.

**5. Dockerization**

* **AI Backend Dockerfile:** Loads Python dependencies (PyTorch, Ultralytics, OpenCV), exposes port 5000.
* **UI Backend Dockerfile:** Loads Python dependencies (FastAPI, httpx), exposes port 5000.
* **Docker Compose:** Orchestrates both services in a single network (app-network).

**Docker Compose Ports:**

* UI Backend → localhost:5000
* AI Backend → localhost:5001 (internal, not accessed directly by users)

**6. How to Run This Project**

**6.1 Prerequisites**

* Install **Docker** and **Docker Compose** on your system.
* Place your project folder like this:

project\_root/

* ai\_backend/
  + Dockerfile
  + app.py
  + req.txt
* ui\_backend/
  + Dockerfile
  + app.py
  + req.txt
* images/ (optional test images)
* docker-compose.yml

**6.2 Build and Start Services**

From the project root directory:

docker-compose up --build

This will:

1. Build Docker images for both services.
2. Start **AI Backend** and **UI Backend**.
3. Show logs in the terminal.

**6.3 Restart Services After Code Changes**

If you change **Python code only**:

docker-compose restart ui\_backend

If you change **dependencies or Dockerfile**:

docker-compose up --build

**6.4 Test the Service**

Use curl or Postman to test the upload endpoint:

curl -X POST "http://localhost:5000/upload/" -F [file=@C:/path/to/your/test\_image.jpg](mailto:file=@C:/path/to/your/test_image.jpg)

**Expected Response:**

{

"detections": [

{

"bbox": [x1, y1, x2, y2],

"confidence": 0.6,

"class": 0

}

],

"image\_path": "outputs/abc123\_pred.jpg",

"json\_path": "outputs/abc123\_pred.json"

}

* Annotated images are saved in outputs/.
* JSON contains coordinates, confidence, and class IDs.
* To save the output image ( if not saved directly) use below command:  
  ***docker cp project\_root-ai\_backend-1:/app/outputs/1b1bb80d\_pred.jpg .***

**6.5 Notes / Tips**

* **Confidence Threshold:** Can be changed in app.py in the model.predict(..., conf=0.6) argument.
* **Modularity:** You can replace YOLOv3 with YOLOv8 or any custom model.
* **Scaling:** Deploy UI backend and AI backend on separate servers if needed.
* **Logs:** Both services log activity in the console; you can redirect logs to files for production.