Project Documentation — Object Detection Microservice

Prepared by **Ullas G H** • Python 3.11 • Flask • YOLO Object Detection

1. Introduction

The objective of this project was to design and implement an **Object Detection Microservice** that processes images, detects objects, and returns both annotated images (with bounding boxes) and JSON metadata containing detection results. This project demonstrates practical understanding of microservice design, REST APIs, model integration, and full-stack AI deployment.

The system is divided into two main components:

- **AI Backend** — Handles image processing, object detection using a pre-trained model (YOLOv5/YOLOv3-tiny), and generates output images and JSON files.
- **UI Backend** — Acts as a client-facing REST API. It accepts images, forwards them to the AI backend, and returns processed responses to users.

2. Problem Statement

The challenge was to create a service where a user can upload an image via an API endpoint and receive the following:

1. An output image annotated with bounding boxes and class labels.
2. A structured JSON file listing all detections, bounding box coordinates, and confidence scores.
3. Ability to test and verify locally without requiring cloud APIs or paid services.

Additionally, the architecture needed to support easy deployment via `Flask` and optionally through `Docker Compose`.

3. Design and Architecture

The architecture was modularized into two microservices communicating over REST:

Component	Description	Port
AI Backend	Loads the YOLO model, performs object detection, saves output images and JSON files.	5001
UI Backend	Frontend API layer — handles uploads, forwards to AI backend, serves processed outputs.	5000

Each backend was designed with its own virtual environment and dependency management via `requirements.txt`.

Core Components

- `Flask` — lightweight Python web framework for REST endpoints.
- `OpenCV` — for image handling, reading, and saving annotated outputs.
- `YOLOv3-tiny` — a compact, fast version of YOLO for CPU inference.
- `requests` — for inter-service HTTP communication.
- `Werkzeug` — for secure file upload handling.

4. Step-by-Step Development Process

Step 1 — Setting up the Repository

Created the base repository structure:

```
object-detection-microservice/
├── ai-backend/
├── ui-backend/
├── test_images/
├── outputs/
├── models/
├── docker-compose.yml
└── README.md
```

Both backend directories were initialized with `Flask` applications and separate `venv` environments to ensure isolation.

Step 2 — Implementing the AI Backend

The AI backend handles inference logic:

- Accepts image upload via `/detect` endpoint.
- Performs inference using YOLO model (via `torch.hub` or pre-downloaded weights).
- Draws bounding boxes using `cv2.rectangle()`.
- Saves annotated image and JSON output to `outputs/`.

Key Code Snippet:

```
@app.route("/detect", methods=["POST"])
def detect():
    image = request.files["image"]
    filename = secure_filename(image.filename)
    image.save(os.path.join(UPLOAD_FOLDER, filename))
    results = model(image_path)
    annotated = results.render()[0]
    cv2.imwrite(f"outputs/result_{filename}", annotated)
    return jsonify(results.pandas().xyxy[0].to_dict(orient="records"))
```

Step 3 — Implementing the UI Backend

The UI backend acts as a proxy layer between the user and AI backend:

- Exposes endpoints: `/detect` and `/health`.
- Uses `requests.post()` to send the image to AI backend.
- Fetches annotated output and re-serves it.

Example health check route:

```
@app.route("/health")
def health():
    try:
        r = requests.get(f"{AI_URL}/health", timeout=10)
        return jsonify({"ui": "ok", "ai": r.json()}), 200
    except Exception as e:
        return jsonify({"ui": "ok", "ai": {"status": "down", "error": str(e)}}), 200
```

Step 4 — Testing the Services

Both backends were launched separately:

```
# AI backend
cd ai-backend
python app.py

# UI backend
cd ui-backend
python app.py
```

Health check verified via:

```
Invoke-WebRequest -Uri "http://127.0.0.1:5000/health" | Select-Object -Expand Content
```

Then test detection using PowerShell-safe `curl` command:

```
curl.exe -X POST -F 'image=@"F:/PROJECTS/object-detection-microservice/test_images/sample.jpg"' http://
```

Step 5 — Fixing Key Issues

- **Problem:** Health check timeouts → Resolved by increasing timeout to 10 seconds.
- **Problem:** Wrong PowerShell escaping in `curl` commands → Switched to single quotes to prevent command parsing issues.
- **Problem:** Missing dependency `requests` → Added to `requirements.txt`.
- **Problem:** IndentationError in `app.py` → Repaired structure and formatted properly.

Step 6 — Adding Output JSON Logging

Modified the AI backend to save both image and JSON files:

```
json_path = os.path.join("outputs", f"result_{filename.split('.')[0]}.json")
with open(json_path, "w") as jf:
    json.dump(result_data, jf, indent=2)
```

---

5. Testing and Validation

Test Environment

- OS: Windows 10 (PowerShell terminal)
- Python: 3.11.9
- Dependencies installed from `requirements.txt`
- Flask development server used for local deployment

Results

- **Input:** `sample.jpg`
- **Output image:** `outputs/result_sample.jpeg` (bounding boxes drawn)
- **Output JSON:** `outputs/result_sample.json`

Sample Output JSON:

```
{
  "detections": [
    {
      "bounding_box": { "height": 207, "width": 193, "x": 4, "y": 20 },
      "class": "dog",
      "confidence": 0.9906
    }
  ],
  "output_filename": "result_sample.jpeg",
  "output_url": "/output/result_sample.jpeg"
}
```

---

6. Docker Integration

To simplify setup, Dockerfiles were created for each backend, and a `docker-compose.yml` file was added:

```
version: '3'
services:
  ai-backend:
    build: ./ai-backend
    ports:
      - "5001:5001"
  ui-backend:
    build: ./ui-backend
    ports:
      - "5000:5000"
    depends_on:
      - ai-backend
```

This allows launching the complete stack via:

```
docker-compose up --build
```

---

7. References and Learning Resources

- Flask Official Documentation
- OpenCV Python Docs
- YOLOv5 GitHub Repository (Ultralytics)
- Python JSON Module Docs
- StackOverflow discussions for *“Flask image upload”* and *“PowerShell curl escaping”*.

---

8. Conclusion

The **Object Detection Microservice** successfully demonstrates a scalable, modular approach to deploying computer vision models via RESTful APIs. Through careful debugging and architectural iteration, the project achieved a working local pipeline with accurate detections, clean JSON outputs, and Docker support.

Key achievements:

- Built dual microservices (AI + UI) using Flask.
- Implemented YOLO-based detection returning bounding boxes and confidence scores.
- Saved both annotated images and structured JSON output files.

- Supported containerized deployment using Docker Compose.
- Created thorough test and documentation scripts for job submission.

Final Note: The project emphasizes clear communication between microservices, error handling, and real-world testing practices — all essential for production-level AI engineering.

© 2025 Ullas G H — Object Detection Microservice Assignment Documentation