Integration Guide

ML Backend Services - Face Detection & Recognition API

Service Overview

The Python FastAPI service provides ML capabilities:

- **Face Detection** (YuNet model)
- **Face Recognition** (Sface model)
- **Feature Extraction** (for caching/optimization)

Your TypeScript backend calls these services via HTTP/WebSocket.

Assumptions

- TypeScript backend handles visitor management
- database stores visitor images (base64) with `visitor_id`
- No pre-computed face features - ML service extracts features on-the-fly
- PostgreSQL database accessible to both services

1. Database Schema (Expected by ML Service)

The ML service expects your database to have visitor images accessible. TypeScript backend manages the schema, but the ML service queries:

```
-- ML service queries these tables for recognition
SELECT visitor_id, base64Image FROM visitors WHERE base64Image IS NOT NULL
-- OR
SELECT visitor_id, base64Image FROM visitor_images WHERE base64Image IS NO
```

> **Note:** The ML service extracts face features on-the-fly from stored images. No separate `face_features` table needed.

2. Registering Visitors (TypeScript Backend)

TypeScript backend handles registration:

1. **Capture face images** (1-10 images recommended, more = better accuracy)
2. **Store images in your database** via your TypeScript backend
3. **No ML service call needed** - ML service extracts features on-the-fly during recognition

The ML service is **not involved** in registration - it only handles detection and recognition.

3. Calling ML Services from TypeScript Backend

Option A: REST API (Recommended)

```
// TypeScript backend calling ML service
const recognizeVisitor = async (imageBase64: string, threshold: number = 0
  const response = await fetch('http://face-recog-api:8000/api/v1/recogniz
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      image: imageBase64,
      threshold: threshold
    })
  });

  const result = await response.json();
  // result: { visitor_id: string | null, confidence: number | null, match

  if (result.matched && result.visitor_id) {
    // Fetch visitor info from your database
    const visitor = await getVisitorFromDB(result.visitor_id);
    return { visitor, confidence: result.confidence };
  }

  return null;
};
```

Option B: WebSocket (Real-time Camera)

```
// TypeScript backend WebSocket client
const ws = new WebSocket('ws://face-recog-api:8000/ws/realtime');

ws.onopen = () => {
  // Send frame from camera
  ws.send(JSON.stringify({
    type: 'frame',
    image: base64Frame
  }));
};

ws.onmessage = (event) => {
  const result = JSON.parse(event.data);
  // result: { type: 'results', faces: [...], count: number }

  result.faces.forEach(face => {
    if (face.matched && face.visitor_id) {
      // Handle recognized visitor
      handleRecognizedVisitor(face.visitor_id, face.confidence);
    }
  });
};
```

4. TypeScript Integration Examples

Face Detection Only

```
const detectFaces = async (imageBase64: string) => {
  const response = await fetch('http://face-recog-api:8000/api/v1/detect'
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ image: imageBase64 })
  });

  const result = await response.json();
  // result: { faces: [{ bbox: [x, y, w, h], confidence: number }], count
  return result.faces;
};
```

Face Recognition

```
const recognizeVisitor = async (imageBase64: string, threshold: number = 
  const response = await fetch('http://face-recog-api:8000/api/v1/recogni
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ image: imageBase64, threshold })
  });

  const result = await response.json();
  // result: { visitor_id: string | null, confidence: number | null, match

  if (result.matched && result.visitor_id) {
    // Your TypeScript backend fetches visitor info
    const visitor = await db.query(
      'SELECT * FROM visitors WHERE visitor_id = $1',
      [result.visitor_id]
    );
    return { visitor, confidence: result.confidence };
  }
  return null;
};
```

Optional: Extract Features for Caching

```
// Pre-extract features to cache for better performance
const extractFeatures = async (imageBase64: string) => {
  const response = await fetch('http://face-recog-api:8000/api/v1/extract
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ image: imageBase64 })
  });

  const result = await response.json();
  // result: { feature_vector: number[] | null, face_detected: boolean }

  if (result.face_detected && result.feature_vector) {
    // Cache in your database for faster recognition
    await cacheFeatureVector(visitorId, result.feature_vector);
  }
};
```

Deployment

ML Service (Python FastAPI)

```
# Start ML service
docker-compose up -d face-recog-api postgres

# ML service runs on: http://face-recog-api:8000
# Database: postgres:5432 (shared with TypeScript backend)
```

TypeScript Backend Integration

- Connect to ML service at: `http://face-recog-api:8000` (or environment variable)
- Use same PostgreSQL database for visitor data
- ML service queries visitor images from shared database

ML Service API Endpoints

ENDPOINT	METHOD	PURPOSE	TYPESCRIPT USAGE
`/api/v1/detect`	POST	Detect faces in image	Call from TypeScript backend when you need face detection only
`/api/v1/extract-features`	POST	Extract face features	Optional: Pre-extract and cache features for performance
`/api/v1/recognize`	POST	Recognize face	**Main endpoint** - Call from TypeScript backend for visitor recognition
`/ws/realtime`	WebSocket	Real-time camera processing	Use for live camera feed processing
`/health`	GET	Health check	Monitor service status

Performance Considerations

On-the-fly feature extraction means the API extracts features from stored images during recognition. This is convenient but can be slower with many visitors.

For better performance:

1. **Limit number of visitors** processed per recognition (e.g., only active visitors)
2. **Use caching:** Optionally extract and cache features in a separate table
3. **Optimize images:** Store smaller images (e.g., 320x320) to speed up processing
4. **Batch processing:** Process multiple stored images in parallel (future enhancement)

Typical performance:

- Small database (< 100 visitors): ~200-500ms per recognition
- Medium database (100-1000 visitors): ~500-2000ms per recognition
- Large database (> 1000 visitors): Consider caching features

Understanding Similarity Scores

Cosine Similarity Basics

The API uses **cosine similarity** to compare face features. The score indicates how similar two faces are:

- **Score Range:** 0.0 (different people) to 1.0 (identical)
- **Default Threshold:** 0.363 (Sface model default)
- **Higher score = More similar faces**

Score Interpretation

SCORE	MEANING	MATCH? (THRESHOLD=0.363)
0.7 - 1.0	Very similar	✅ **MATCH** (high confidence)
0.5 - 0.7	Similar	✅ **MATCH** (good confidence)
0.363 - 0.5	Somewhat similar	✅ **MATCH** (above threshold)
0.3 - 0.363	Borderline	❌ **NO MATCH** (below threshold)
0.0 - 0.3	Different	❌ **NO MATCH**

Adjusting Threshold

```
# Strict (high accuracy, fewer false positives)
threshold = 0.5

# Default (balanced)
threshold = 0.363

# Lenient (catches more matches, may have false positives)
threshold = 0.3
```

See `COSINE_SIMILARITY_GUIDE.md` for detailed explanation.

Architecture Overview

```
┌─────────────────────┐
│  TypeScript Backend │   (Your main application)
│  - Visitor Management│
│  - Business Logic    │
│  - Database Access  │
└──────────┬──────────┘
```

```
                  │ HTTP/WebSocket
                  │
        ┌─────────▼─────────┐
        │  Python ML Service │    (Face Detection & Recognition)
        │  - YuNet (Detection) │
        │  - Sface (Recognition)│
        │  - Feature Extraction│
        └─────────┬─────────┘
                  │
        ┌─────────▼─────────┐
        │   PostgreSQL DB    │    (Shared database)
        │  - Visitor data    │
        │  - Face images     │
        └───────────────────┘
```

TypeScript Type Definitions

```
// Add these types to your TypeScript project

interface DetectRequest {
  image: string; // base64 encoded image
}

interface DetectResponse {
  faces: Array<{
    bbox: [number, number, number, number]; // [x, y, w, h]
    confidence: number;
  }>;
  count: number;
}

interface RecognizeRequest {
  image: string; // base64 encoded image
  threshold?: number; // default: 0.363
}
```

```
interface RecognizeResponse {
  visitor_id: string | null;
  confidence: number | null;
  matched: boolean;
}

interface ExtractFeaturesRequest {
  image: string; // base64 encoded image
}

interface ExtractFeaturesResponse {
  feature_vector: number[] | null; // 512-dim array
  face_detected: boolean;
}
```