Backend Documentation for Smart Water Leakage Detection API

---

1. Project Overview

This project is a **Smart Water Leakage Detection API** built using Python, FastAPI, SQLAlchemy, and a MySQL database. The system monitors water flow through multiple sensors along a pipeline and generates alerts if anomalies, leaks, or low battery levels are detected. The backend exposes RESTful API endpoints for managing sensors, sensor data, and alerts.

The project is structured into the following Python files: - `main.py`: Main FastAPI application with all routes. - `database.py`: Database connection setup using SQLAlchemy. - `models.py`: ORM models representing database tables. - `utils.py`: Utility functions for data processing and alert generation. - `schemas.py`: Pydantic models for input/output validation.

---

2. Setup Instructions

1. Install dependencies:

```
pip install fastapi sqlalchemy pymysql pydantic uvicorn
```

2. Configure database credentials in `database.py`.

3. Run the FastAPI server:

```
uvicorn main:app --reload
```

The server will be available at `http://127.0.0.1:8000`.

---

3. Database (database.py)

Purpose: Set up SQLAlchemy connection to MySQL.

- `DATABASE_URL`: Connection string in the format `mysql+pymysql://user:password@host:port/db`.
- `engine`: SQLAlchemy engine used to connect to the database.
- `SessionLocal`: Factory for creating database sessions.

- `Base` : Declarative base for models.
- `get_db()` : Dependency function to provide DB session to API routes.

Usage: Each API route can depend on `get_db()` to access the database.

---

4. Models (models.py)

Purpose: Define ORM classes corresponding to database tables.

Enum Classes

- `SensorStatus` : `active` , `inactive` , `maintenance`
- `AlertType` : `leak` , `anomaly` , `low_battery`
- `Severity` : `low` , `medium` , `high`
- `AlertStatus` : `active` , `resolved`

Tables

4.1 Sensor

- `sensor_id` : Primary key (string)
- `location` : Location of the sensor
- `pipe_diameter_mm` : Pipe diameter in millimeters
- `install_date` : Installation date
- `status` : SensorStatus enum

4.2 SensorData

- `id` : Primary key (auto-increment)
- `sensor_id` : Foreign key to Sensor
- `timestamp` : Time of reading
- `flow_rate` : Water flow rate (decimal)
- `battery_level` : Sensor battery level (integer)

4.3 ProcessedData

- Stores smoothed sensor readings and flow differences
- Fields: `id` , `sensor_id` , `timestamp` , `smoothed_flow` , `flow_diff`

4.4 Alert

- `alert_id` : Primary key
- `sensor_from` , `sensor_to` : Sensor IDs involved
- `timestamp` : Time of alert
- `alert_type` : AlertType enum

- `severity`: Severity enum
- `probability`: Calculated probability of event
- `status`: AlertStatus enum

---

5. Utilities (utils.py)

Purpose: Contains functions to process sensor data and generate alerts.

5.1 Sigmoid Function

```
def sigmoid(x, x0=0, k=1):
    return 1 / (1 + exp(-k*(x-x0)))
```

- Smooth mapping from any value to 0-1.

5.2 Leak Probability Calculation

```
def compute_leak_probability_sigmoid(flow1, flow2, battery1, battery2):
    ...
```

- Computes leak probability between two sensors. - Inputs: `flow1, flow2` (flow readings), `battery1, battery2` - Uses weighted sigmoid combination to determine probability.

5.3 Process Sensor Data Pairwise

```
def process_sensor_data_pairwise(db, sensors, new_readings):
    ...
```

- Loops over sensors to compute smoothed flow and flow differences. - Checks consecutive sensor pairs for leaks using `compute_leak_probability_sigmoid`. - Generates `Alert` objects for leaks, anomalies, or low battery. - Commits processed data and alerts to the database.

Inputs: - `sensors`: List of Sensor ORM objects in pipeline order - `new_readings`: Dict `{sensor_id: SensorData}`

Output: List of generated Alert objects.

---

6. Main API (main.py)

Purpose: Implements all FastAPI routes for sensors, sensor data, and alerts.

6.1 Dependency

```
def get_db():
    ...
```

Provides a database session for route handlers.

6.2 Sensor Routes

6.2.1 Register a Sensor

`POST /sensors` - Inputs (query params): `sensor_id`, `location`, `pipe_diameter_mm` - Returns success message and sensor ID.

6.2.2 List Sensors

`GET /sensors` - Returns list of all sensors with details.

6.2.3 Update Sensor

`PUT /sensors/{sensor_id}` - Optional fields: `location`, `pipe_diameter_mm`, `status` - Updates only provided fields.

6.2.4 Delete Sensor

`DELETE /sensors/{sensor_id}` - Deletes sensor and all related SensorData and Alerts.

6.3 Sensor Data Routes

6.3.1 Add Sensor Reading

`POST /sensors/{sensor_id}/data` - Inputs: `flow_rate`, `battery_level` - Adds new SensorData, processes pairwise with other sensors, generates alerts. - Returns: `data_id` and generated alerts.

6.3.2 Batch Sensor Readings

`POST /sensors/data` - Inputs: List of `SensorReading` (JSON) - Processes all readings pairwise, returns list of alerts.

6.3.3 Get Recent Sensor Readings

`GET /sensors/{sensor_id}/data` - Query param: `limit` (default 10) - Returns last N readings.

6.3.4 Update Sensor Reading

`PUT /sensors/{sensor_id}/data/{data_id}` - Optional fields: `flow_rate`, `battery_level` - Updates sensor data record.

6.3.5 Delete Sensor Reading

`DELETE /sensors/{sensor_id}/data/{data_id}` - Deletes a specific sensor reading.

6.3.6 Filter Sensor Data by Date

`GET /sensors/{sensor_id}/data/filter` - Query params: `start`, `end` (ISO datetime), `limit` - Returns filtered readings.

6.4 Alert Routes

6.4.1 Get Active Alerts

`GET /alerts` - Returns all active alerts ordered by timestamp.

6.4.2 Resolve Single Alert

`POST /alerts/resolve/{alert_id}` - Marks alert as resolved.

6.4.3 Update Alert

`PUT /alerts/{alert_id}` - Update `severity` and/or `status`.

6.4.4 Delete Alert

`DELETE /alerts/{alert_id}` - Deletes a specific alert.

6.4.5 Filter Alerts

`GET /alerts/filter` - Filter by `sensor_id`, `alert_type`, `severity`, `status`

6.4.6 Bulk Resolve Alerts

`POST /alerts/resolve/bulk` - Input: list of `alert_ids` - Marks all listed alerts as resolved.

6.5 Alert Logic

- `LEAK_FLOW_THRESHOLD = 15.0`
- `LOW_BATTERY_THRESHOLD = 20`
- Alerts are generated when thresholds are exceeded or anomalies detected.
- Pairwise comparisons identify leaks along consecutive sensors.

---

7. Input/Output Examples

Register Sensor

```
POST /sensors?sensor_id=S1&location=Tank&pipe_diameter_mm=50
```

Response:

```
{ "message": "Sensor registered successfully", "sensor": "S1" }
```

Add Sensor Data

```
POST /sensors/S1/data
{ "flow_rate": 20.5, "battery_level": 85 }
```

Response:

```
{
  "message": "Data added successfully",
  "data_id": 101,
  "alerts": [ ... ]
}
```

Get Active Alerts

```
GET /alerts
```

Response:

```
[ { "alert_id": 1, "sensor_from": "S1", "sensor_to": "S2", "alert_type":
"leak", ... } ]
```

8. How It Works

1. Sensors send readings (flow_rate, battery_level) via API.
2. Backend stores readings in `sensor_data` table.
3. Utilities compute smoothed flows and pairwise leak probabilities.
4. Alerts are generated for leaks, anomalies, or low battery.
5. Alerts stored in `alerts` table, accessible via API.
6. Sensor info can be managed through CRUD routes.

End of Backend Documentation