IoT Enhanced Smart Greenhouse

- Lorenzo Cavallaro - s346742
- Negar Abdi - s343747

OverView

Objective:

To help farmers and greenhouse operators optimize crop growth and reduce resource waste by providing an intelligent, automated monitoring and control platform.

Manage Operations

Recording and Analysing data from sensors

Remote greenhouse monitoring and control

Functionalities

- **Collect** sensor data (temperature, humidity, light, soil moisture, NPK, pH)
- **Analyze** data and **automate** corrective actions
- **Schedule** parameter values (time, day, periodicity)
- Web app: **manage** greenhouses, sensors, **set** thresholds, **show** plots
- Telegram bot: send **status notifications** through alerts and warnings
- ThingSpeak: **real-time** data **storage**

Communication Protocols

REST Web Service

Uses HTTP methods to access and manage resources identified by URLs.

Request

Devices call GET /get_sensors on the catalog, passing greenhouse_id and device_name as parameters.

Response

Catalog verifies the device and returns the list of relevant sensors for that device.

Request & Response - Examples:

- GET /get_sensors?greenhouse_id=28&device_name=DeviceConnector
- 200 OK - {”device_id”: 76, “sensors”: [<sensor_1>, <sensor_2>, ...]}

Communication Protocols

MQTT

Lightweight and reliable messaging protocol, ideal for IoT.

Publish

Sensors (through the DeviceConnector) publish measurements in SenML format.

Receive

Management services (humidity, light, nutrients) listen for events and act accordingly.

Message topics - Examples:

- greenhouse_33/area_30/sensor_19 → measurement
- greenhouse_33/area_30/action/sensor_19 → action command
- greenhouse_33/area_30/event/sensor_19 → scheduled event

Communication Protocols

REST Web Service

Smart devices expose services via HTTP:

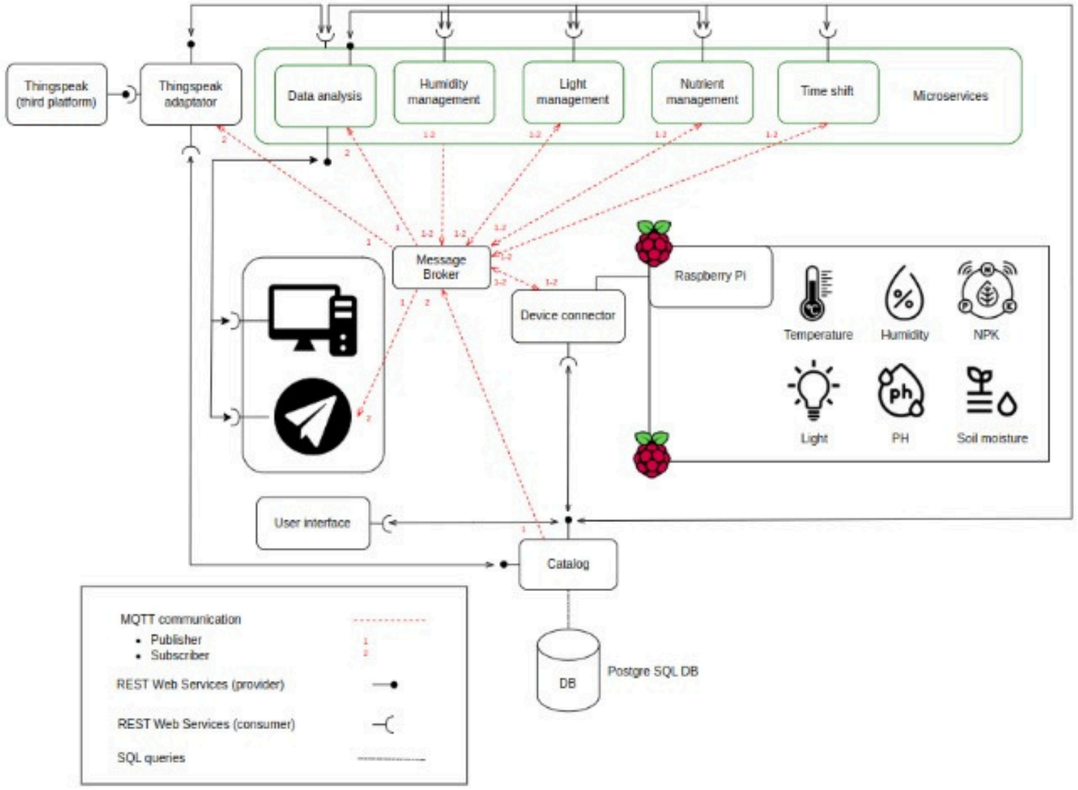
- **Catalog**, maps URLs to database methods, and other devices query it by sending HTTP requests with the appropriate method.
- **Data Analysis**, handles GET requests, providing statistics and predictions to management components.

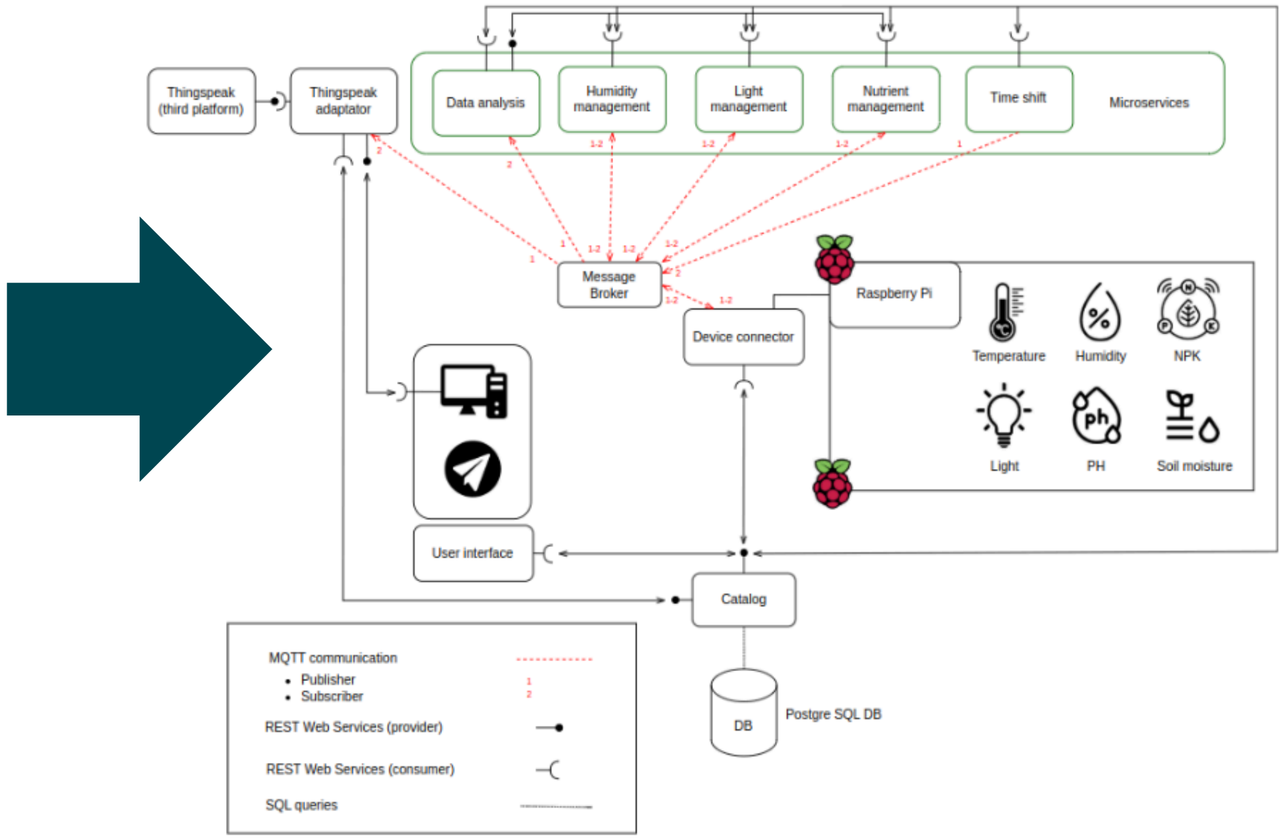
MQTT connection

Used by IoT devices to share asynchronous data:

- **Device Connector**, reads and publishes sensor data.
- **Management**, triggers actions when needed.
- **Time Shift**, publishes scheduled events.
- **Data Analysis**, consumes sensor values to compute statistics.
- **ThingSpeak Adaptor**, stores data in ThingSpeak.

Use Case Diagram





WebApp - Overwiew

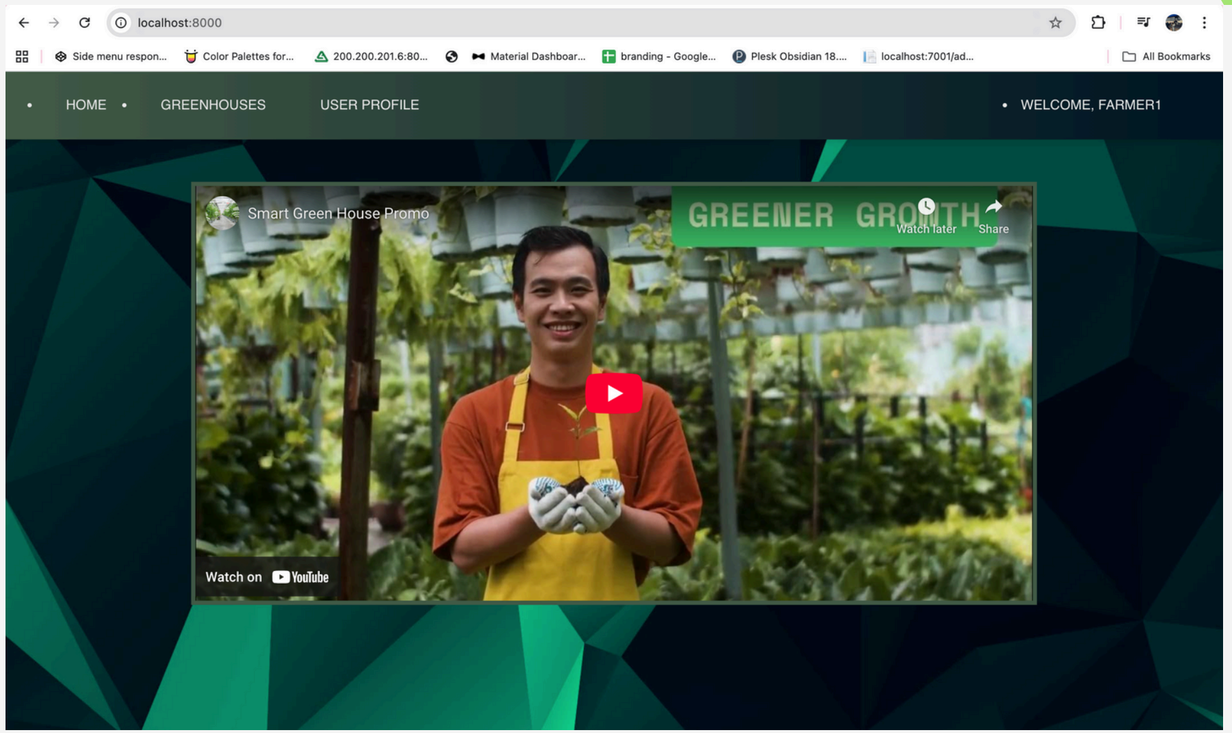
- **Accounts:**
  - Users can register, log in, and manage their sessions securely.

- **Greenhouses:**
  - Create one or more greenhouses and manage each independently.

- **Areas:**
  - Define areas within a greenhouse and configure their settings.

- **Assets:**
  - Add and manage sensors and plants per area (remove, assign)

- **Monitoring:**
  - Track sensor status in real time with live charts that blend in-app Chart.js visuals with ThingSpeak data feeds for historical context.



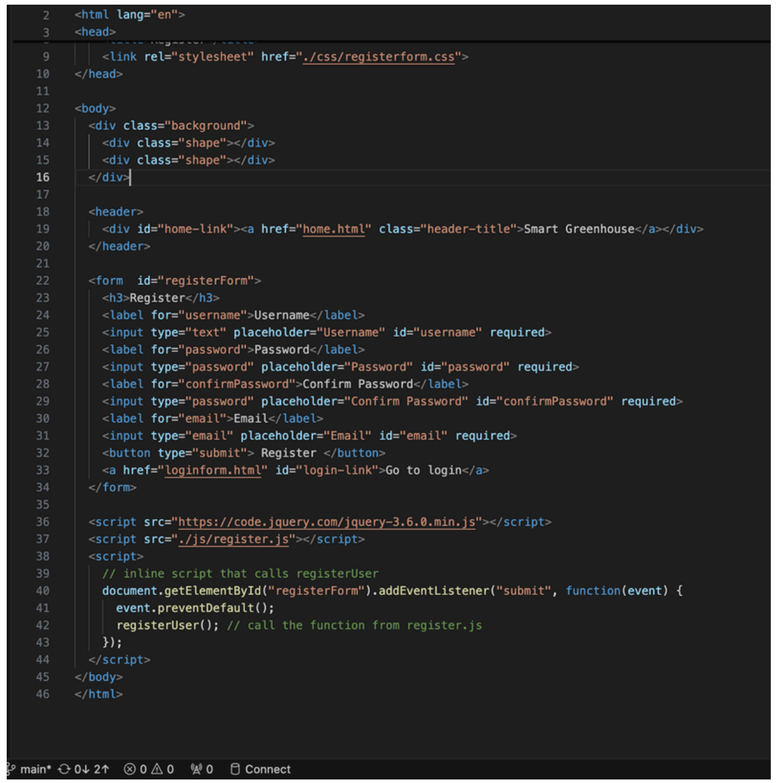
WebApp - Functionality

- The WebApp is built with HTML pages for structure and CSS styling, complemented by Bootstrap to ensure a responsive and consistent design across different devices. Each page (login, dashboard, greenhouse view, area view, sensor management, etc.) is styled to be intuitive and easy to navigate.

- For dynamic interactions, the WebApp relies on AJAX requests that communicate asynchronously with the Catalog API endpoints. The frontend sends JSON-encoded requests to the backend API and receives JSON responses containing data or confirmations. These responses are then processed by JavaScript to update the interface in real time.

WebApp - Implementation

- **User management:**
  - AJAX calls send credentials in JSON format to the Catalog API for registration and login; responses are used to establish and validate sessions.

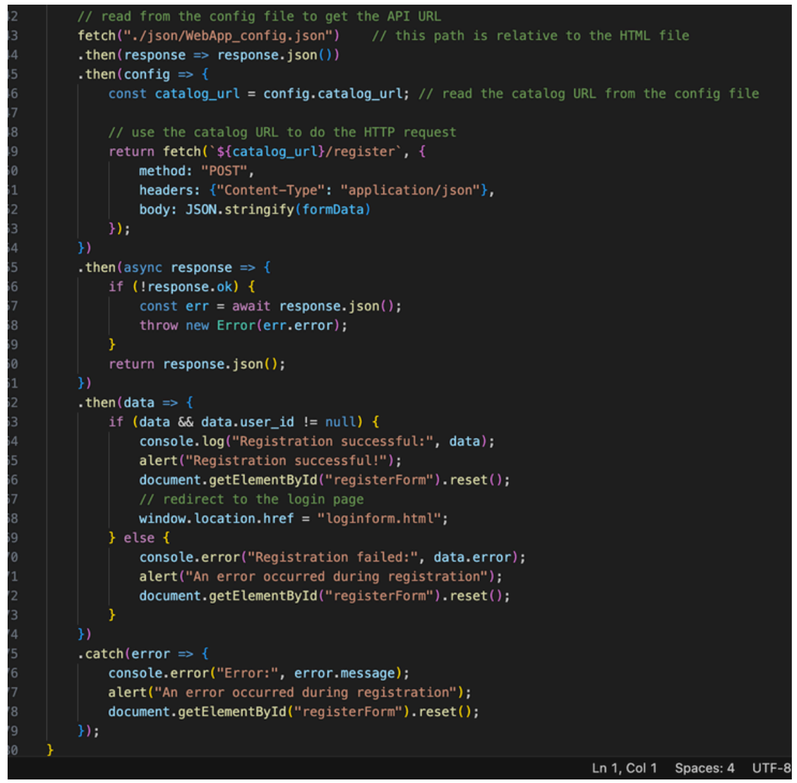


```
<html lang="en">
<head>
    <link rel="stylesheet" href="./css/registerform.css">
</head>

<body>
  <div class="background">
    <div class="shape"></div>
    <div class="shape"></div>
  </div>

  <header>
    <div id="home-link"><a href="home.html" class="header-title">Smart Greenhouse</a></div>
  </header>

  <form  id="registerForm">
    <h3>Register</h3>
    <label for="username">Username</label>
    <input type="text" placeholder="Username" id="username" required>
    <label for="password">Password</label>
    <input type="password" placeholder="Password" id="password" required>
    <label for="confirmPassword">Confirm Password</label>
    <input type="password" placeholder="Confirm Password" id="confirmPassword" required>
    <label for="email">Email</label>
    <input type="email" placeholder="Email" id="email" required>
    <button type="submit"> Register </button>
    <a href="loginform.html" id="login-link">Go to login</a>
  </form>

  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script src="./js/register.js"></script>
  <script>
    // inline script that calls registerUser
    document.getElementById("registerForm").addEventListener("submit", function(event) {
      event.preventDefault();
      registerUser(); // call the function from register.js
    });
  </script>
</body>
</html>
```

main* 0↓ 2↑ 0 0 0 Connect

```
    width: 430px;
    height: 520px;
    position: absolute;
    transform: translate(-50%, -50%);
    left: 50%;
    top: 50%;
}

.background .shape {
    height: 200px;
    width: 200px;
    position: absolute;
    border-radius: 50%;
}

.shape:first-child {
    background: linear-gradient(#0b212a, #435c48);
    left: -80px;
    top: -80px;
}

.shape:last-child {
    background: linear-gradient(to right,#ff512f,#f09819);
    right: -30px;
    bottom: -80px;
}

form {
    height: 670px;
    width: 400px;
    background-color: rgba(255, 255, 255, 0.13);
    position: absolute;
    transform: translate(-50%, -50%);
    top: 50%;
    left: 50%;
    border-radius: 10px;
    backdrop-filter: blur(10px);
    border: 2px solid rgba(255, 255, 255, 0.1);
    box-shadow: 0 0 40px rgba(8, 7, 16, 0.6);
    padding: 50px 35px;
}
```

Ln 16,



```
    // read from the config file to get the API URL
    fetch("./json/WebApp_config.json")    // this path is relative to the HTML file
    .then(response => response.json())
    .then(config => {
        const catalog_url = config.catalog_url; // read the catalog URL from the config file

        // use the catalog URL to do the HTTP request
        return fetch(`${catalog_url}/register`, {
            method: "POST",
            headers: {"Content-Type": "application/json"},
            body: JSON.stringify(formData)
        });
    })
    .then(async response => {
        if (!response.ok) {
            const err = await response.json();
            throw new Error(err.error);
        }
        return response.json();
    })
    .then(data => {
        if (data && data.user_id != null) {
            console.log("Registration successful:", data);
            alert("Registration successful!");
            document.getElementById("registerForm").reset();
            // redirect to the login page
            window.location.href = "loginform.html";
        } else {
            console.error("Registration failed:", data.error);
            alert("An error occurred during registration");
            document.getElementById("registerForm").reset();
        }
    })
    .catch(error => {
        console.error("Error:", error.message);
        alert("An error occurred during registration");
        document.getElementById("registerForm").reset();
    });
}
```

Ln 1, Col 1 Spaces: 4 UTF-8

WebApp - Implementation

- **Greenhouse and area management:**
  - Adding or removing greenhouses and areas is performed by sending structured JSON objects (e.g., greenhouse name, area ID, dimensions) to the API; the WebApp immediately reflects these changes in the UI.

```
// Read from the config file to get the API URL
fetch("./json/WebApp_config.json")    // This path is relative to the HTML file
.then(response => response.json())
.then(config => {
    const catalog_url = config.catalog_url; // Read the catalog URL from the config file

    // Use the catalog URL to do the HTTP request
    return fetch(`${catalog_url}/add_greenhouse`, {
        method: "POST",
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify(formData)
    });
})
.then(async response => {
    if (!response.ok) {
        const err = await response.json();
        throw new Error(err.error);
    }
    return response.json();
})
.then(data => {
    alert(data.message || "Greenhouse added successfully!");
    // redirect to the greenhouses page
    window.location.href = "greenhouses.html"; // Redirect to the greenhouses page
})
.catch(error => {
    console.error("Error:", error.message);
    alert("An error occurred while adding the greenhouse.");
    document.getElementById("addGreenhouseForm").reset(); // Reset the form
});
}

document.getElementById("AddGreenhouseButton")?.addEventListener("click", function () {
    addGreenhouse();
});
```

Ln 57, Col 7 Spaces: 4 UTF-8

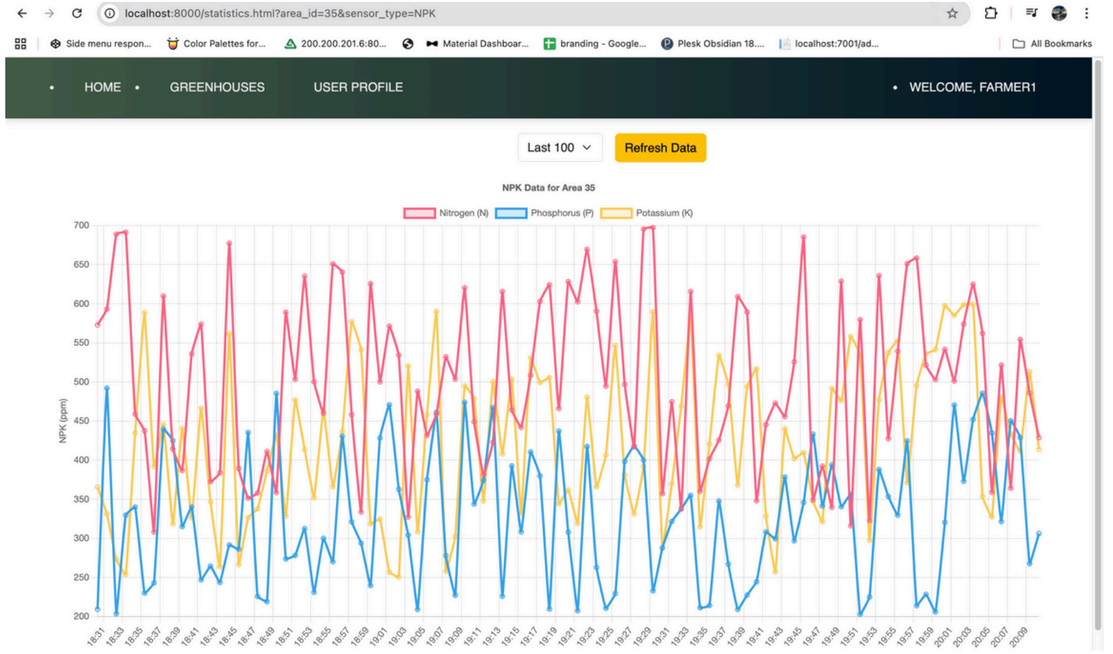
```
        fetch("./json/WebApp_config.json")
            .then((res) => res.json())
            .then((config) => {
            const catalog_url = config.catalog_url;
            return fetch(`${catalog_url}/add_area`, {
                method: "POST",
                headers: {
                "Content-Type": "application/json",
                Authorization: `Bearer ${token}`
                },
                body: JSON.stringify({
                greenhouse_id: parseInt(greenhouseId),
                name: areaName
                })
            });
            })
            .then((res) => res.json())
            .then((result) => {
            alert(result.message || "Area added successfully.");
            location.reload();
            })
    .catch((err) => {
      console.error("Failed to add area:", err);
      alert("Failed to add area.");
      document.getElementById("area-modal").classList.add("d-none");
    });
});
})
      .catch((err) => {
        console.error("Error during initial fetch:", err);
        alert("Failed to load greenhouse data.");
      });
  });
```

Ln 186,

WebApp - Implementation

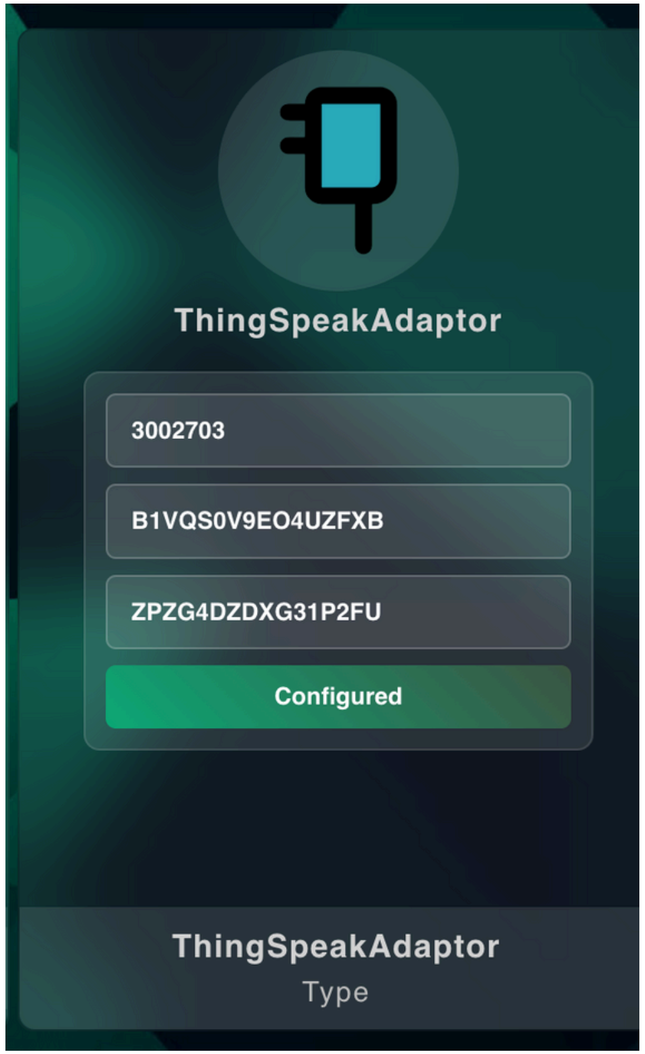
- **Sensor and plant management:**
  - When a sensor or plant is added, edited, or deleted, AJAX requests are made to the relevant API endpoints, and the updated state is re-rendered dynamically.

- **Monitoring:**
  - Fetches ThingSpeak config from Catalog API, then retrieves live data from ThingSpeak API.
  - Parses sensor fields (Temperature, Humidity, SoilMoisture, pH, Light, NPK) → updates interactive charts.
  - Auto-refresh every 30s + user controls for data range and manual refresh.
  - Fallback to mock data if ThingSpeak is unavailable.



ThingSpeak Adaptor

- reads the configuration and all the sensors from the catalog
- for each sensor subscribe to the MQTT topics
- have 60 sec periodic checks of sensors
- receive the measurement messages from the sensors and extract the area_id from them
- for each area of the given greenhouse send the value of the sensors
- each sensor type is already mapped to a field
- push the values to the thingspeak write API



ThingSpeak + chart.js

- by calling catalog we get the thingspeak configuration
- then request to the thingspeak API to read the fresh data
- then map each fields to our sensors in each area
- render the data in chart.js in real time
- refresh every 30 seconds

IoT_AutonomousFarm > thingSpeakAdaptor > {} ThingSpeakAdaptor_config.json > ...

```
{
    "catalog_url": "http://192.168.0.76:8080",
    "greenhouse_id": 28,
    "mqtt_connection": {
        "mqtt_port": 1883,
        "keep_alive": 180,
        "mqtt_broker": "mosquitto"
    },
    "write_url": "https://api.thingspeak.com/update",
    "read_url": "https://api.thingspeak.com/channels"
}
```

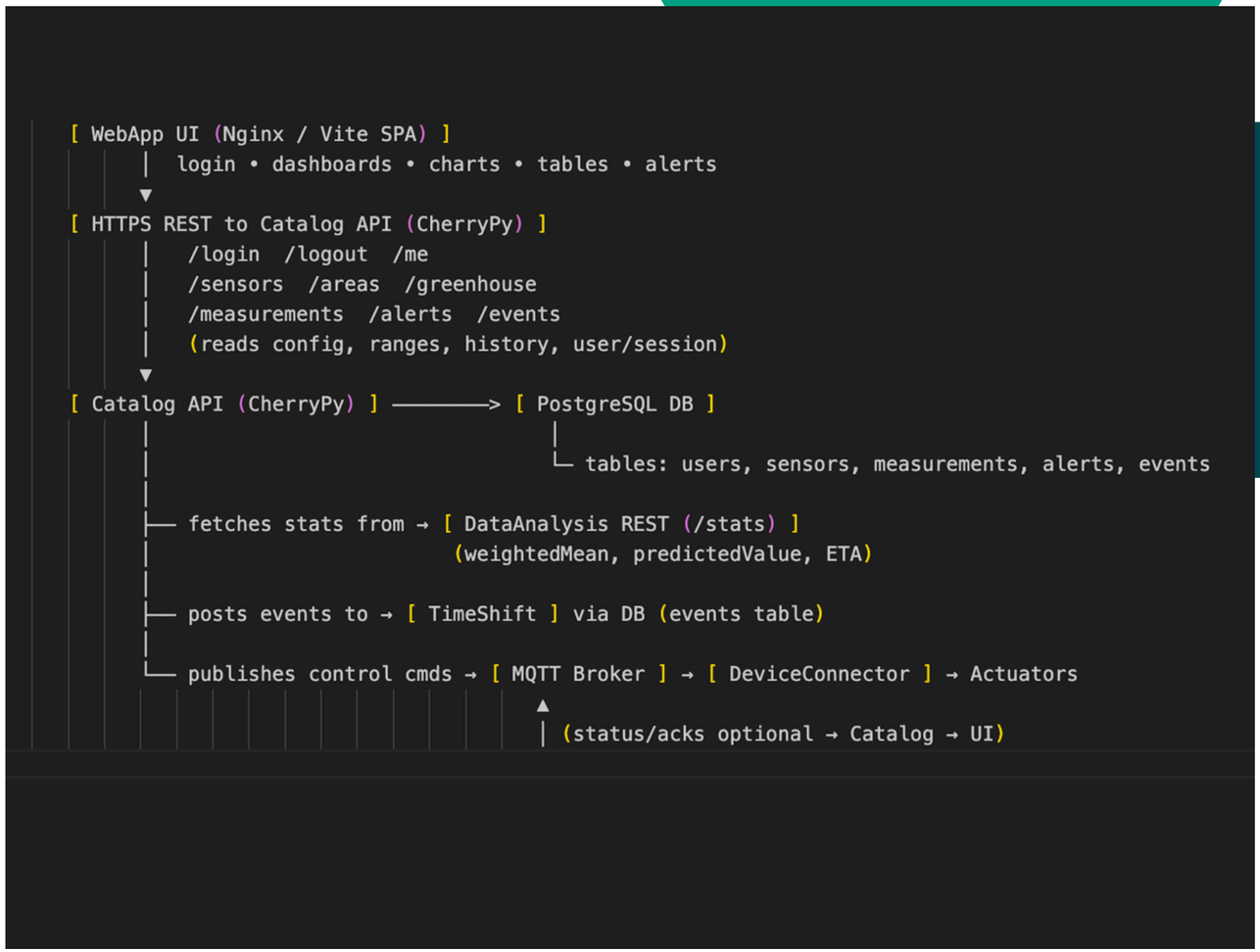
Cooperation of Web App with Other Components

- **Catalog API (CherryPy) :**
  - main backend, handles login, greenhouse/area/sensor/plant management, retrieves measurements, alerts, and schedules.

- **PostgreSQL Database :**
  - (indirect through Catalog API) → stores all persistent data the WebApp works with (users, sensors, measurements, events).

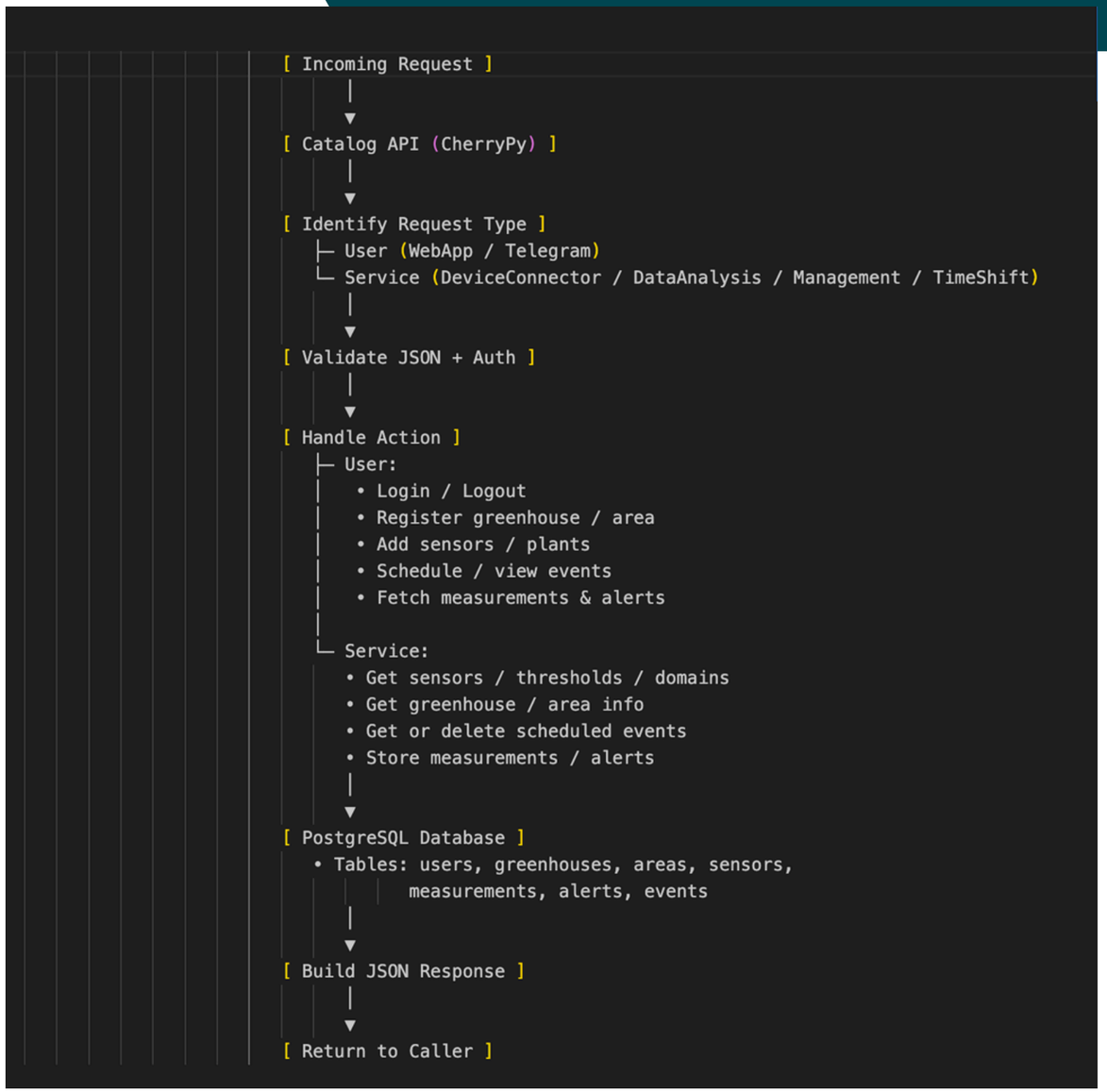
- **TimeShift Service :**
  - handles scheduling; WebApp creates/edits events that TimeShift later executes

- **Thingspeak :**
  - supplies live/historical sensor data, combined with Chart.js for monitoring.



Role of Catalog

- Acts as the central registry and API gateway of the system.
- Provides configuration and metadata for all components:
  - Greenhouses
  - Areas
  - Sensors
  - Users and Telegram links
  - Scheduled events and alerts
- Exposes REST endpoints to WebApp, Telegram, DeviceConnector, Management services, and TimeShift.
- Ensures consistency by storing and validating everything in the PostgreSQL Database.

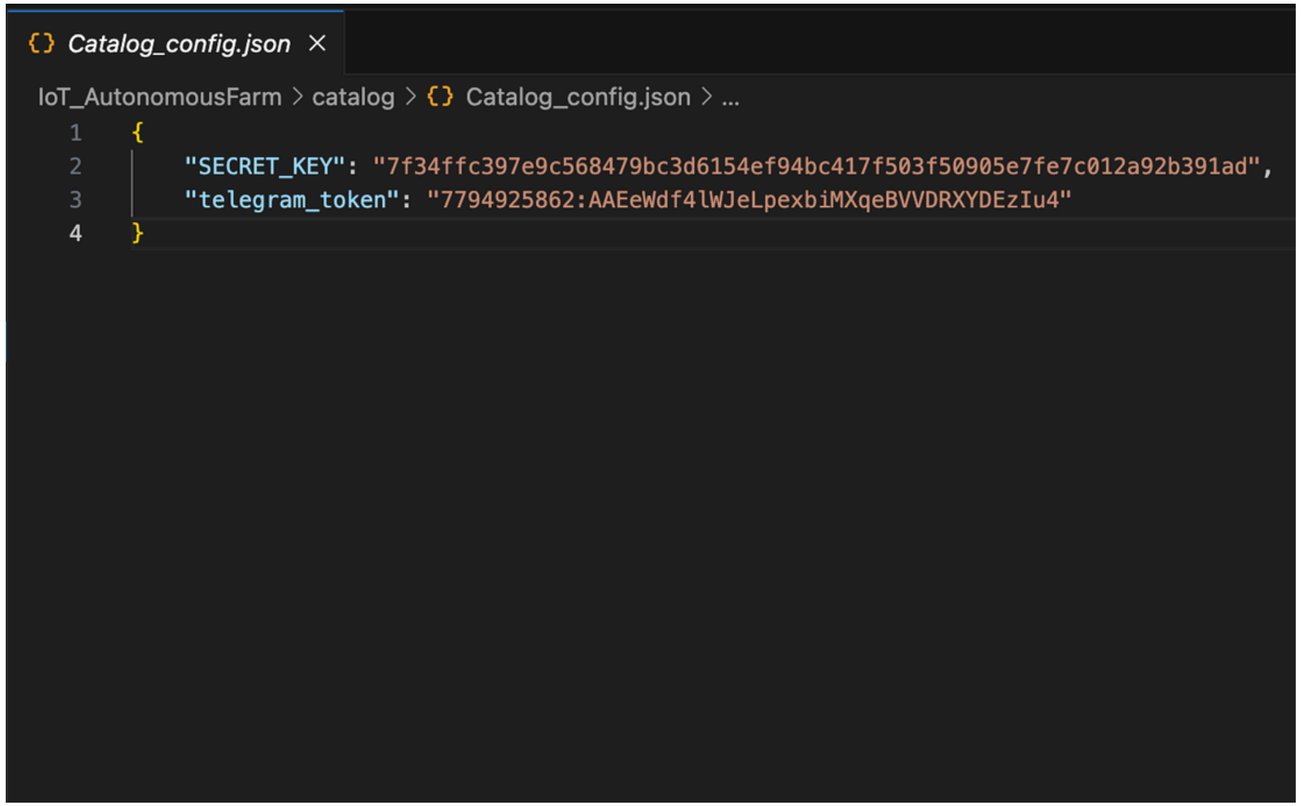


Catalog configuration

It communicates with all the other actors in the platform exploiting REST communication.

Catalog_config.json:

- Stores *secret_key* used to generate an unique connection token for the logged in users.
- Stores *telegram_token* to send chat messages when OTP is confirmed in the web app.



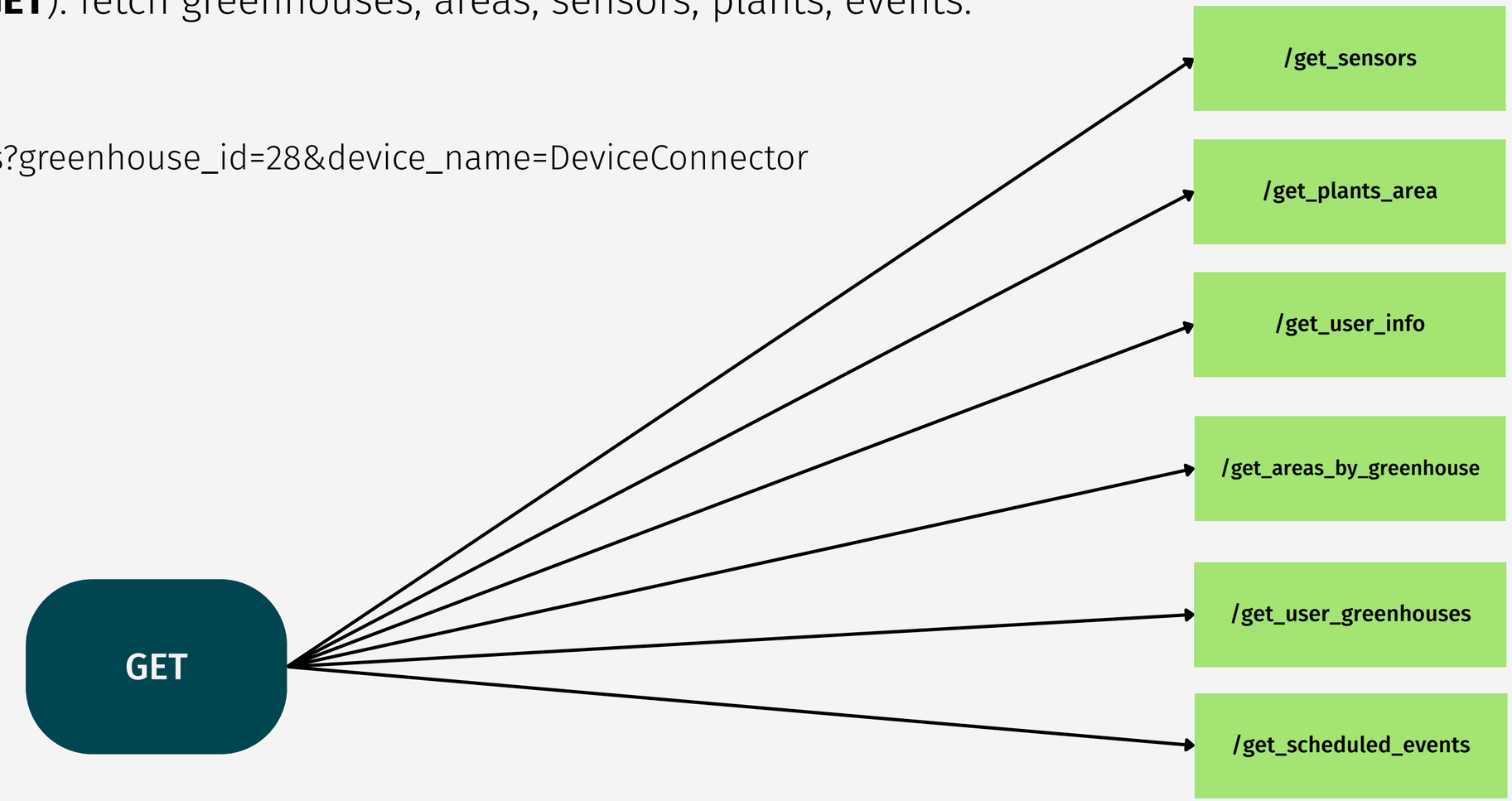
Catalog Core Functionalities

- **User Management:**
  - Register / Login / Logout
  - Authentication with token
  - Telegram OTP binding (/otp, /logout_telegram)
- **Greenhouse & Area Management:**
  - Add / delete greenhouses and areas
  - Retrieve configurations
- **Sensor & device Registry:**
  - Associate sensors with areas
  - Store thresholds, domains, ranges
  - Add devices and provide info for them priodically
- **Events & Scheduling:**
  - Create and store events (Daily, Weekly, Monthly, Once)
  - Used by TimeShift to publish MQTT events

Catalog – REST Functions

- Retrieve (**GET**): fetch greenhouses, areas, sensors, plants, events.

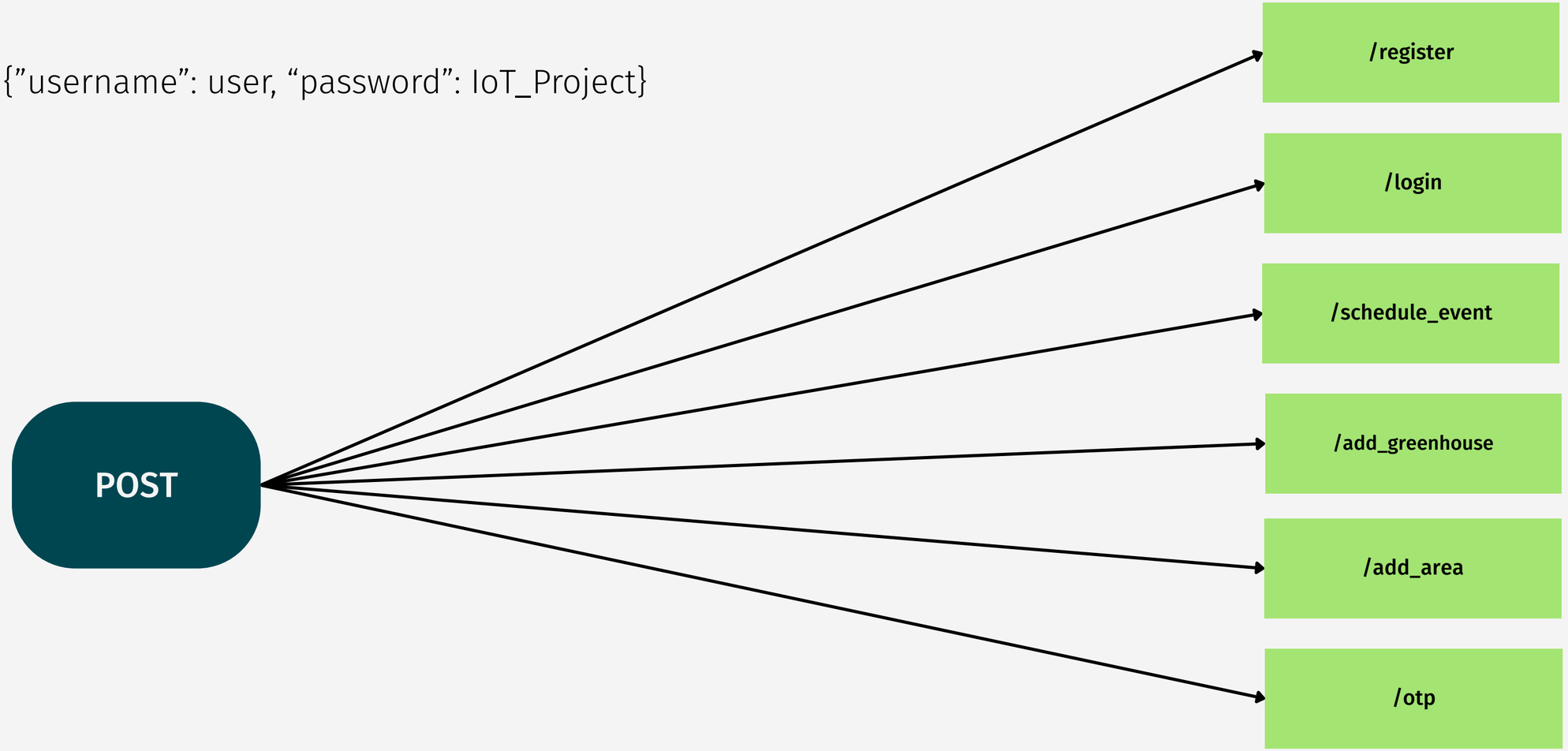
Example:
GET /get_sensors?greenhouse_id=28&device_name=DeviceConnector



Catalog – REST Functions

- Add (**POST**): creates a new entity (user, event, area) or submits data (login) with encrypted body.

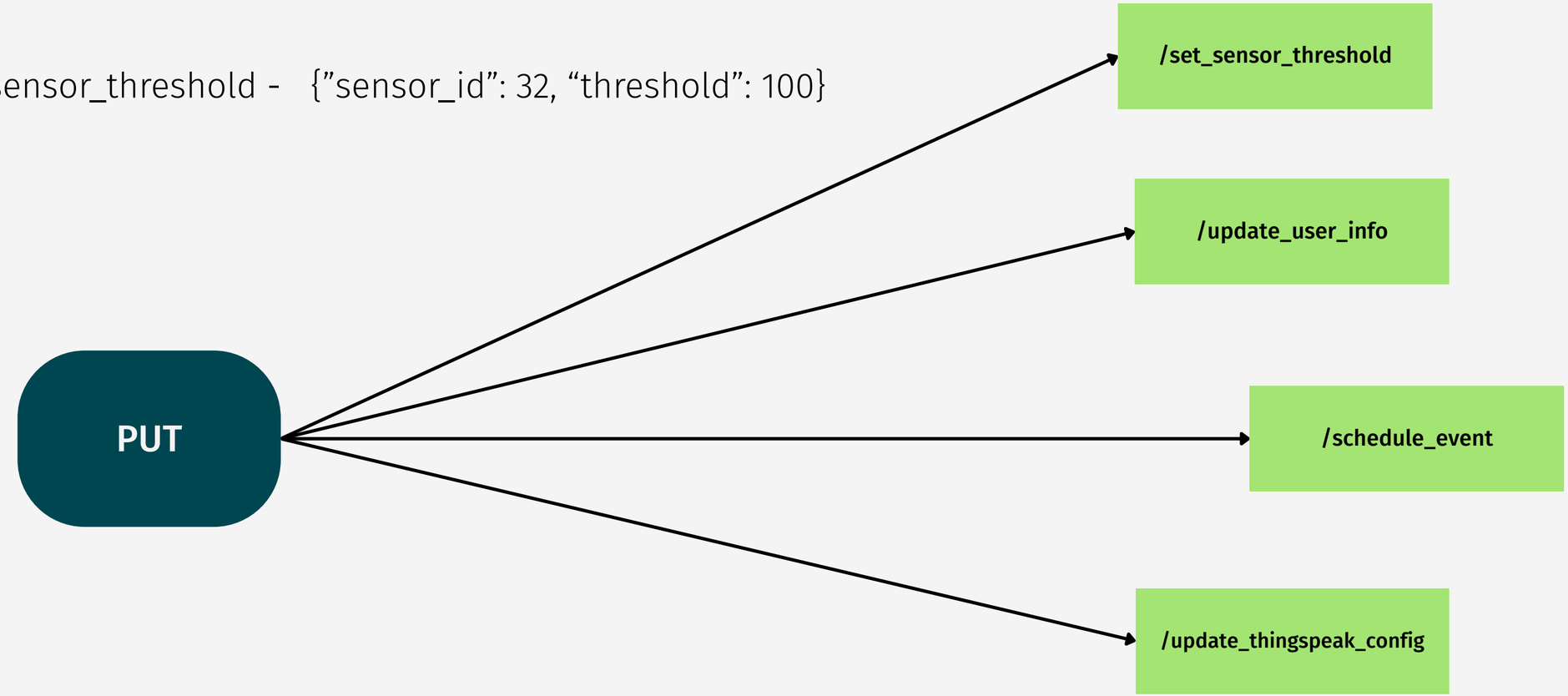
Example:
POST /login - {”username”: user, “password”: IoT_Project}



Catalog – REST Functions

- Update (**PUT**): update an existing entity changes

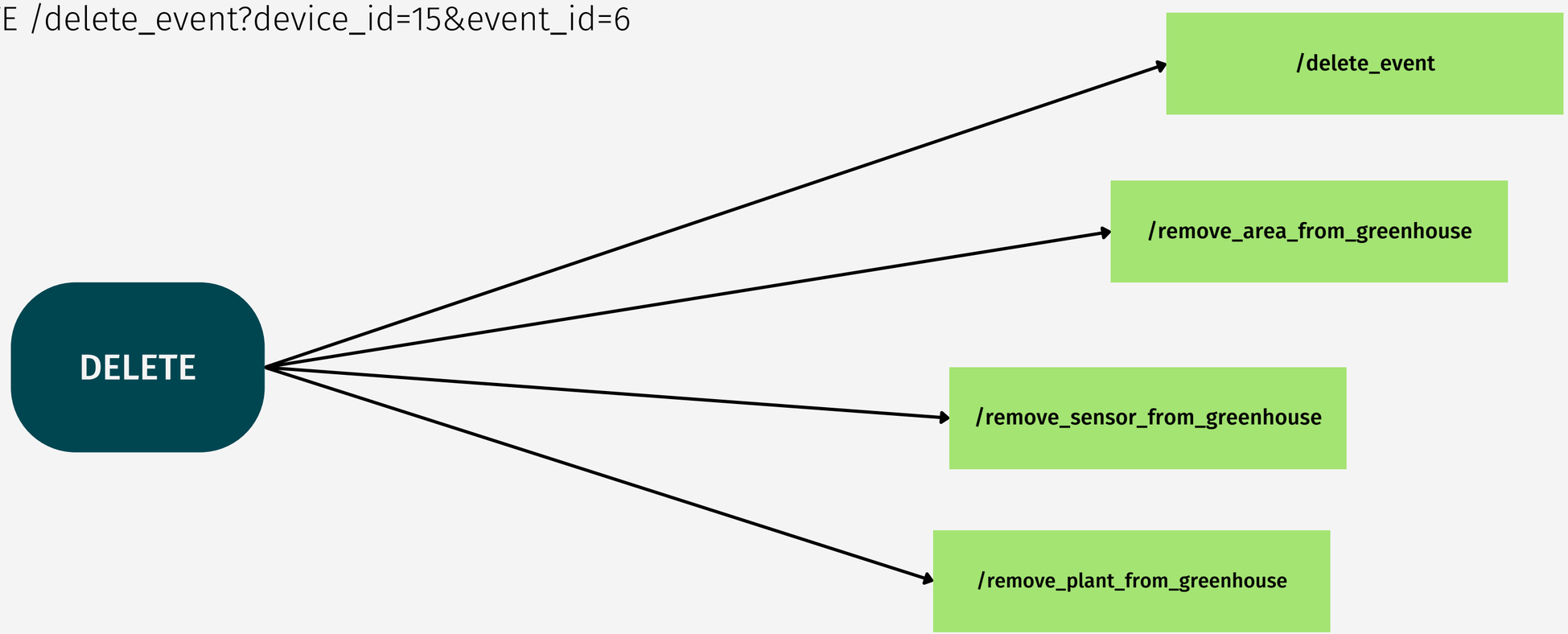
Example:
PUT /set_sensor_threshold - {”sensor_id”: 32, “threshold”: 100}



Catalog – REST Functions

- Remove (**DELETE**): deletes an entity from system.

Example:
DELETE /delete_event?device_id=15&event_id=6



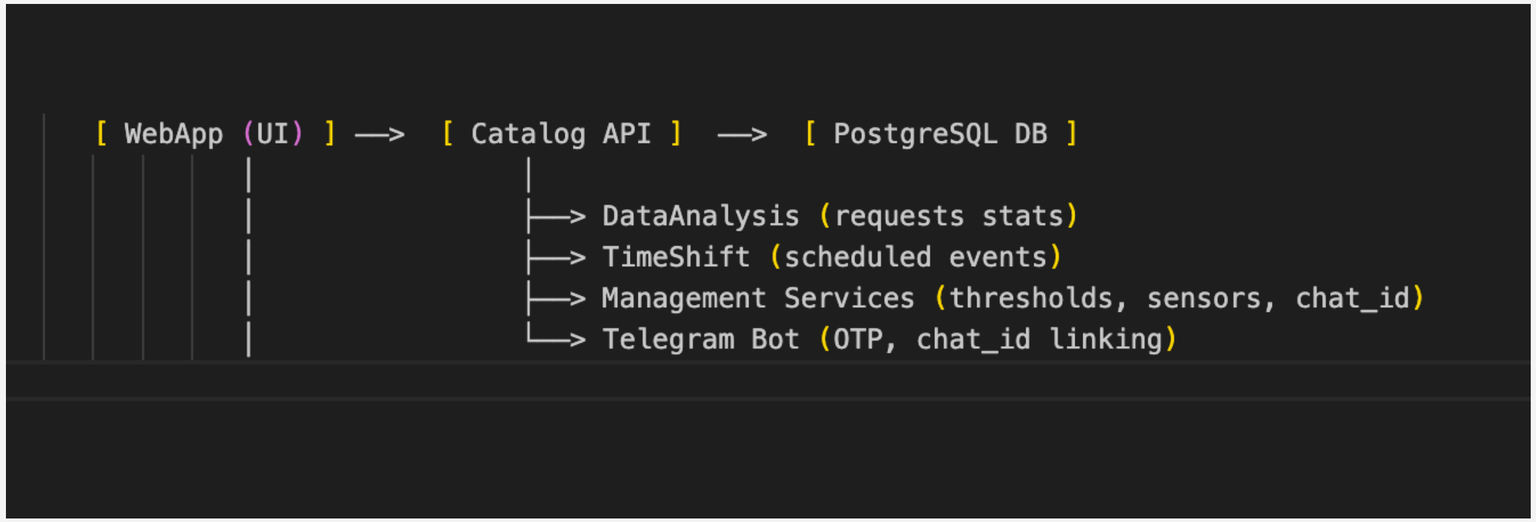
Cooperation of Catalog with PostgreSQL DB

```
smartfarm_db-# \dt
                List of relations
 Schema |        Name        | Type  |   Owner
--------+--------------------+-------+------------
 public | area_plants        | table | iotproject
 public | areas              | table | iotproject
 public | availabledevices   | table | iotproject
 public | availablesensors   | table | iotproject
 public | devices            | table | iotproject
 public | greenhouses        | table | iotproject
 public | otps               | table | iotproject
 public | plants             | table | iotproject
 public | scheduled_events   | table | iotproject
 public | sensors            | table | iotproject
 public | users              | table | iotproject
(11 rows)

smartfarm_db-#
```

```python
# ------------------------------------------------
# 2) Server Utilities (DB, CORS, JWT)
# ------------------------------------------------
def get_db_connection():
    """Retry up to 5x to connect to Postgres."""
    for _ in range(5):
        try:
            write_log("Database connection estabilished")
            return psycopg2.connect(
                dbname="smartfarm_db",
                user="iotproject",
                password="WeWillDieForIoT",
                host="db",
                port="5432"
            )
        except psycopg2.Error as e:
            write_log(f"Error connecting to database: {e}")
            if _ == 4:
                write_log("Failed to connect to the database after 5 attempts")
                raise Exception("Unable to connect to the database")
            time.sleep(60)
```

```python
    cur.execute(
        sql.SQL("""
            INSERT INTO sensors (greenhouse_id, type, name, unit, threshold_range, domain, area_id)
            VALUES (%s, %s, %s, %s, %s, %s, %s)
            RETURNING greenhouse_id
        """),
```

Cooperation of Catalog with Other Components



- **WebApp:**
  - CRUD for users, greenhouses, areas, sensors, events.
- **DeviceConnector:**
  - fetches sensors list & info on startup.
- **DataAnalysis:**
  - pulls sensors list for computing predictions.
- **Management Components :**
  - get the area info, handling events and get sensors for each device.
- **TimeShift:**
  - gets and deletes scheduled events.
- **Telegram Bot :**
  - /otp, /logout_telegram/verify

Security and Reliability

- Authentication with token/session.
- Catalog validates input JSON requests.
- Logs errors in service logs.
- Redundancy possible by scaling CherryPy + DB replicas.

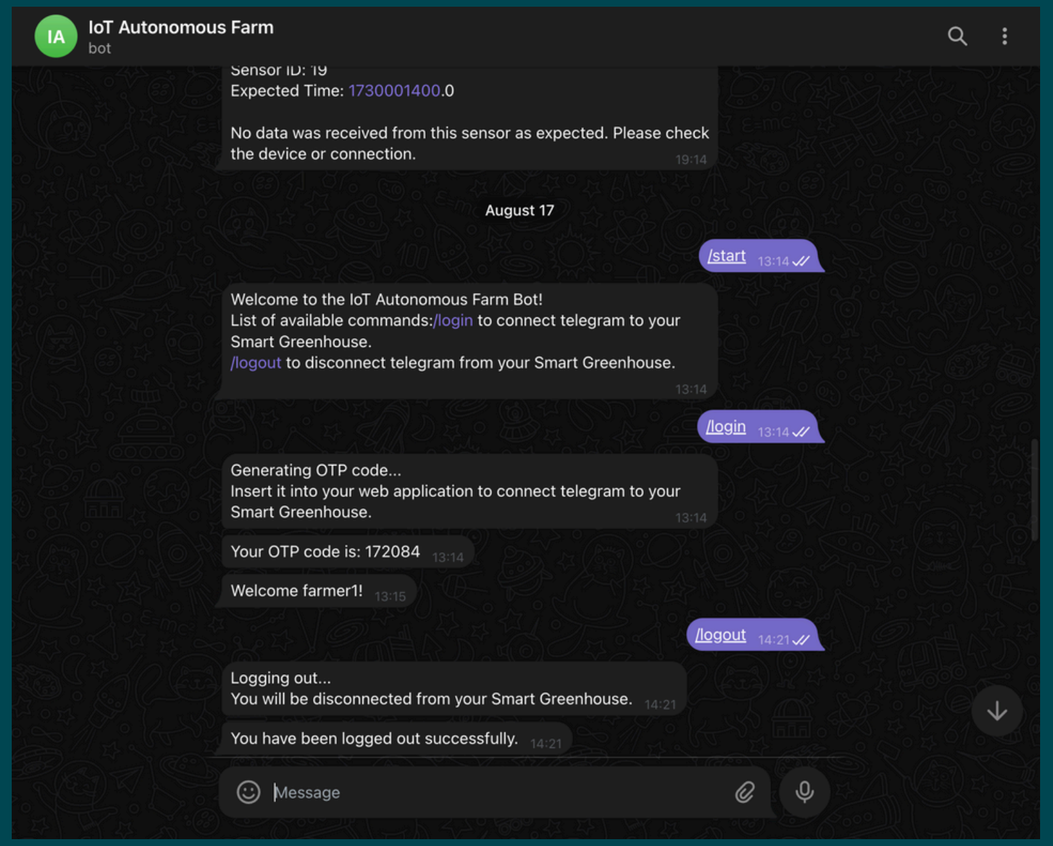
Why Catalog is Essential?

- Single source of truth for all system data.
- Decouples components (they don’t need direct knowledge of each other).
- Provides flexibility:
- Add new management services without redesign.
- Add new sensors/plants with minimal configuration.
- Acts as integration hub for WebApp, Telegram, DeviceConnector, Management, TimeShift, DataAnalysis.

Telegram Bot – Handler

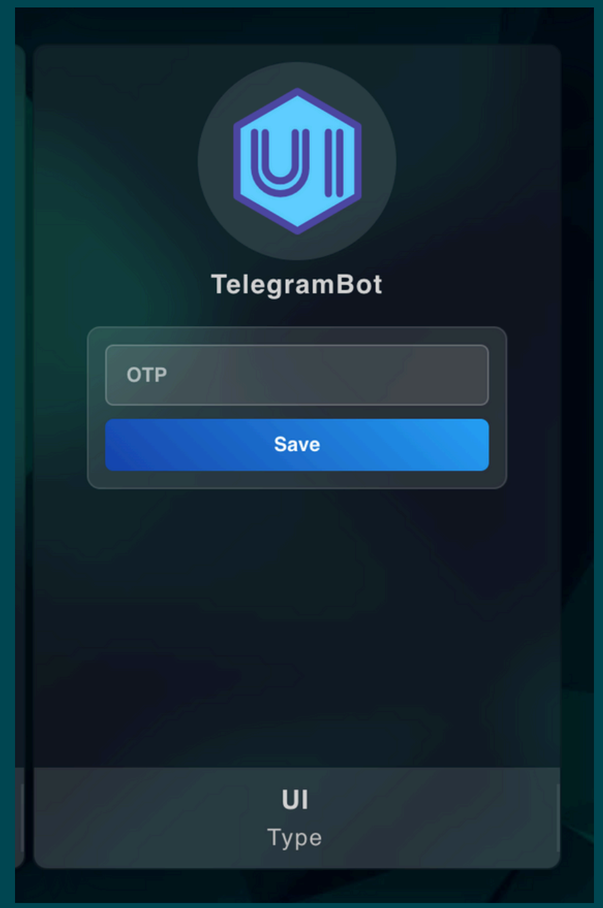
commands:

- /start => welcome message and quick guidance
- /login => login to get an OTP that must be inserted in the Web App.
- /logout => cancel out the connection with the system, telegram chat will no more receives notifications.



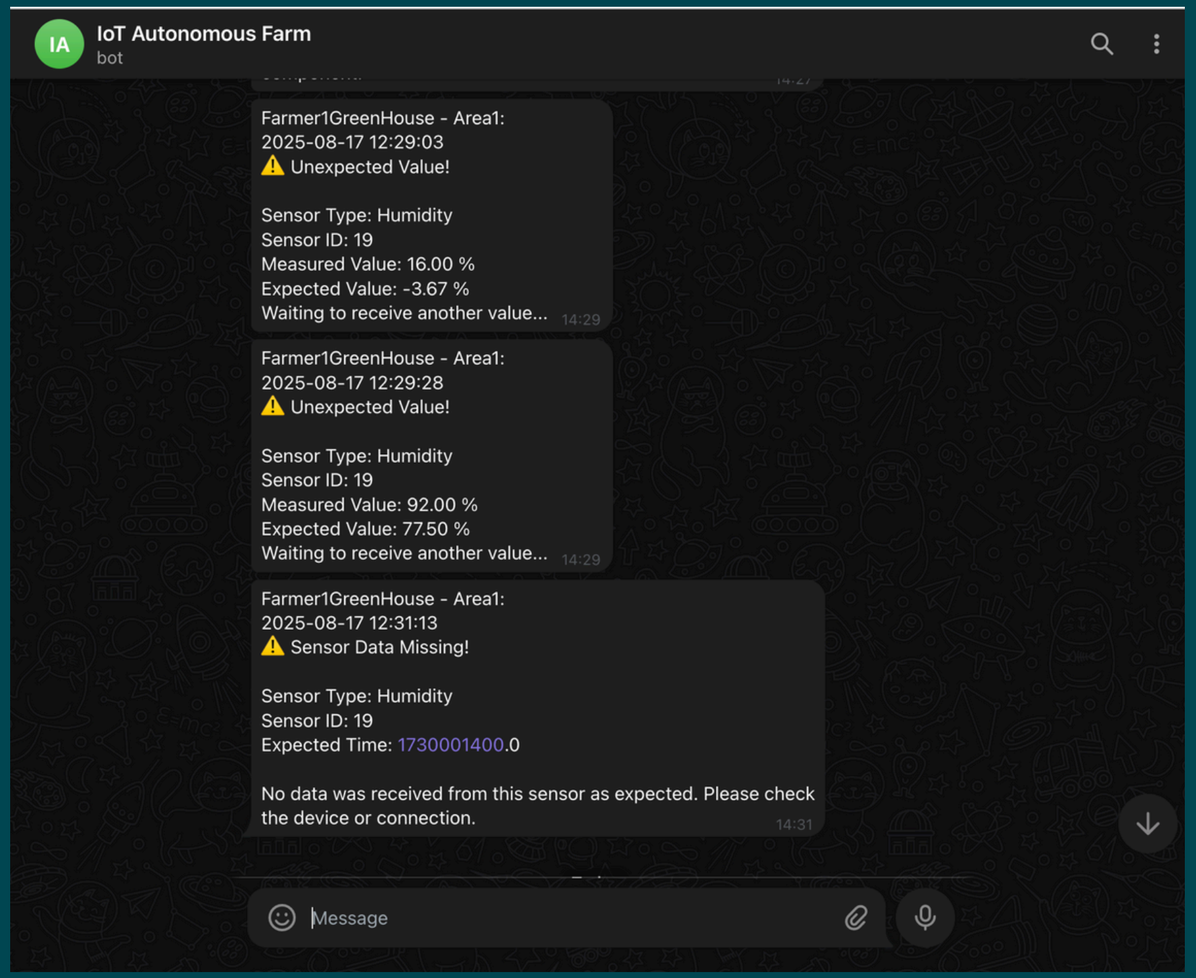
Telegram Bot - functionality

- User opens Telegram → sends /login to the bot.
- Bot replies with a 6-digit OTP and POSTs {user_id, otp_code} to the Catalog.
- User opens your web app (already logged in with their platform account) → enters OTP.
- Web app → Catalog validates OTP and binds
- From now on, our backend can send Telegram notifications to that user
- If the user sends /logout, the Catalog removes that binding.



Telegram Bot - Notifications

User receives periodic alerts and notifications about the status of their greenhouses.



Cooperation of Telegram Bot with Other Components

- **Catalog API + DB:**
  - Accept the OTP from the bot (POST /otp) and store it with the telegram_user_id.
  - Provide a verification endpoint your web app calls (e.g., POST /verify_otp {user_id, otp_code }) that:
  - Checks that OTP exists, not expired, and maps to telegram_user_id.
  - Deletes or invalidates the OTP.
  - Handle /logout_telegram to remove the association.

- **Management services (e.g., HumidityManagement):**
  - On events, they query Catalog for greenhouse name and telegram_chat_id, then send directly to Telegram Bot API

DeviceConnector -Overview

- Acts as the bridge between sensors and the platform.
- On startup it fetches configuration from Catalog, then:
- Publishes sensor readings over MQTT (JSON/SenML).
- Subscribes to action topics to receive commands.
- Keeps the system updated with logs and (optionally) last-update pings.

DeviceConnector - Overview

DeviceConnector — What It Enables:

- Real-time telemetry into the platform.
- Closed-loop control when managers publish actions.
- Clean separation of field devices from backend services.

DeviceConnector - Startup Flow

1. **GET** from Catalog:

- /get_sensors?greenhouse_id=<gh>&device_name=DeviceConnector→ receives device_id and the list of sensors (id, type, unit, area).

2. **GET** from Catalog:

- /get_greenhouse_location?greenhouse_id=<gh> → location may influence simulated values.

3. **Builds** topics, creates sensor classes, and connects to MQTT.

DeviceConnector - MQTT Topics

Measurement topic (publish):

greenhouse_<gh>/area_<area>/sensor_<sensor_id>

Action topic (subscribe):

greenhouse_<gh>/area_<area>/action/sensor_<sensor_id>

DeviceConnector
- MQTT Topics & Payloads

Example SenML-style payload (publish):

```
Published to greenhouse_41/area_44/sensor_32: {"bn": "greenhouse_41/area_44/sensor_32", "e": {"n": "Temperature", "v": 20.55, "u": "\u00b0C", "t": 0}}
Published to greenhouse_41/area_44/sensor_31: {"bn": "greenhouse_41/area_44/sensor_31", "e": {"n": "NPK", "v": {"N": 142.07605293430728, "P": 205.7976167587332, "K":
Published to greenhouse_41/area_44/sensor_33: {"bn": "greenhouse_41/area_44/sensor_33", "e": {"n": "Humidity", "v": 77.93, "u": "%", "t": 0}}
Published to greenhouse_41/area_43/sensor_34: {"bn": "greenhouse_41/area_43/sensor_34", "e": {"n": "Temperature", "v": 20.55, "u": "\u00b0C", "t": 0}}

Published to greenhouse_41/area_44/sensor_32: {"bn": "greenhouse_41/area_44/sensor_32", "e": {"n": "Temperature", "v": 20.68, "u": "\u00b0C", "t": 60}}
Published to greenhouse_41/area_44/sensor_31: {"bn": "greenhouse_41/area_44/sensor_31", "e": {"n": "NPK", "v": {"N": 140.73, "P": 205.58, "K": 194.15}, "u": "mg/kg",
Published to greenhouse_41/area_44/sensor_33: {"bn": "greenhouse_41/area_44/sensor_33", "e": {"n": "Humidity", "v": 78.15, "u": "%", "t": 60}}
Published to greenhouse_41/area_43/sensor_34: {"bn": "greenhouse_41/area_43/sensor_34", "e": {"n": "Temperature", "v": 20.07, "u": "\u00b0C", "t": 60}}

Published to greenhouse_41/area_44/sensor_32: {"bn": "greenhouse_41/area_44/sensor_32", "e": {"n": "Temperature", "v": 20.18, "u": "\u00b0C", "t": 120}}
Published to greenhouse_41/area_44/sensor_31: {"bn": "greenhouse_41/area_44/sensor_31", "e": {"n": "NPK", "v": {"N": 140.33, "P": 204.4, "K": 193.88}, "u": "mg/kg",
Published to greenhouse_41/area_44/sensor_33: {"bn": "greenhouse_41/area_44/sensor_33", "e": {"n": "Humidity", "v": 78.04, "u": "%", "t": 120}}
Published to greenhouse_41/area_43/sensor_34: {"bn": "greenhouse_41/area_43/sensor_34", "e": {"n": "Temperature", "v": 20.54, "u": "\u00b0C", "t": 120}}
```

Example action payload (subscribe):

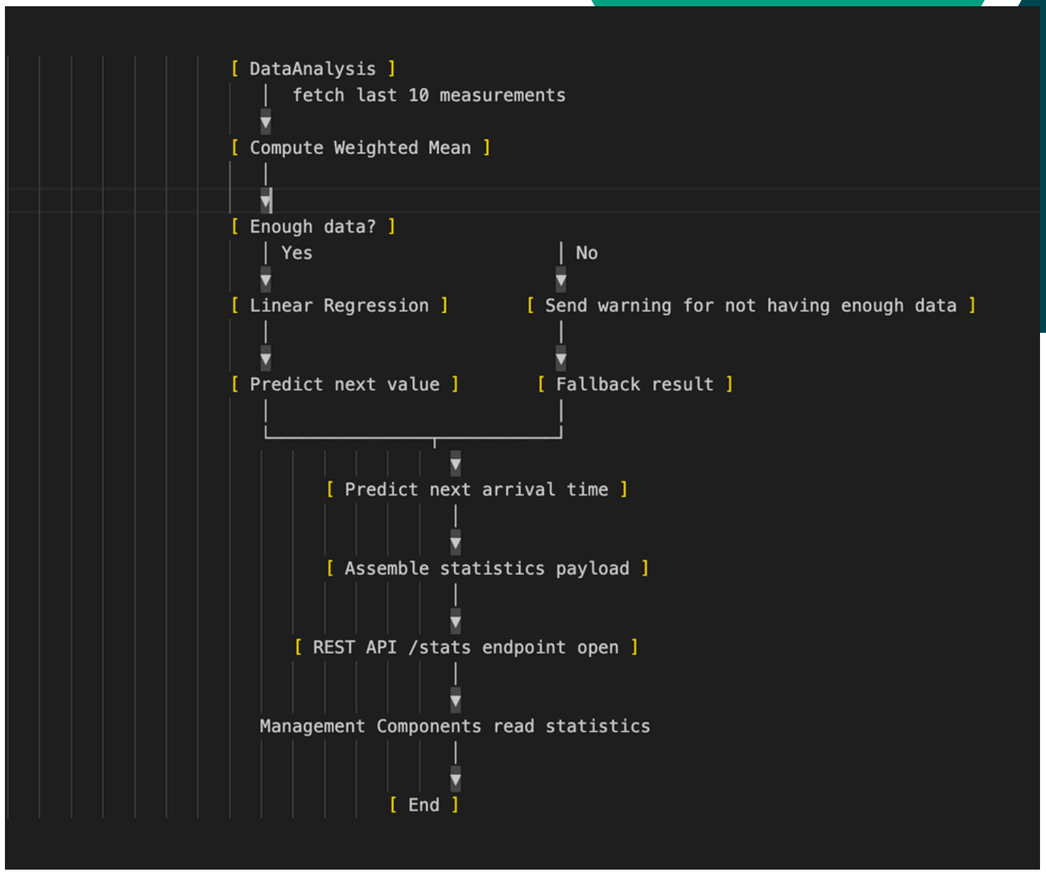
```
{'message': {'action': 'decrease', 'val': 75.63, 'min_treshold': 30, 'max_treshold': 75}, 'parameter': 'Humidity', 'timestamp': 300}
```

DeviceConnector - Behavior

- Measurement loop:
  - Periodically reads/creates a value per sensor (e.g., every ~60s).
  - Publishes to the measurement topic.
- Action handling:
  - Listens on action topics; parses increase/decrease/target style commands.
  - Updates internal sensor state/flags; may influence subsequent readings.
- Logging & resilience:
  - Logs every publish/receive.
  - Reconnects MQTT client on network errors.
  - Waits and repeats the HTTP request in case of errors.

DataAnalysis - Overview

- On each run, fetch the last 10 measurements which are buffered.
- Compute the weighted mean, giving higher importance to more recent values.
- Apply linear regression on the measurements to estimate the next expected value.
- Predict the next arrival time (ETA) based on recent inter-arrival intervals.
- Assemble the statistics payload with weighted mean, predicted value, and ETA.
- Expose a REST web service (read-only) that allows management components to retrieve these statistics.



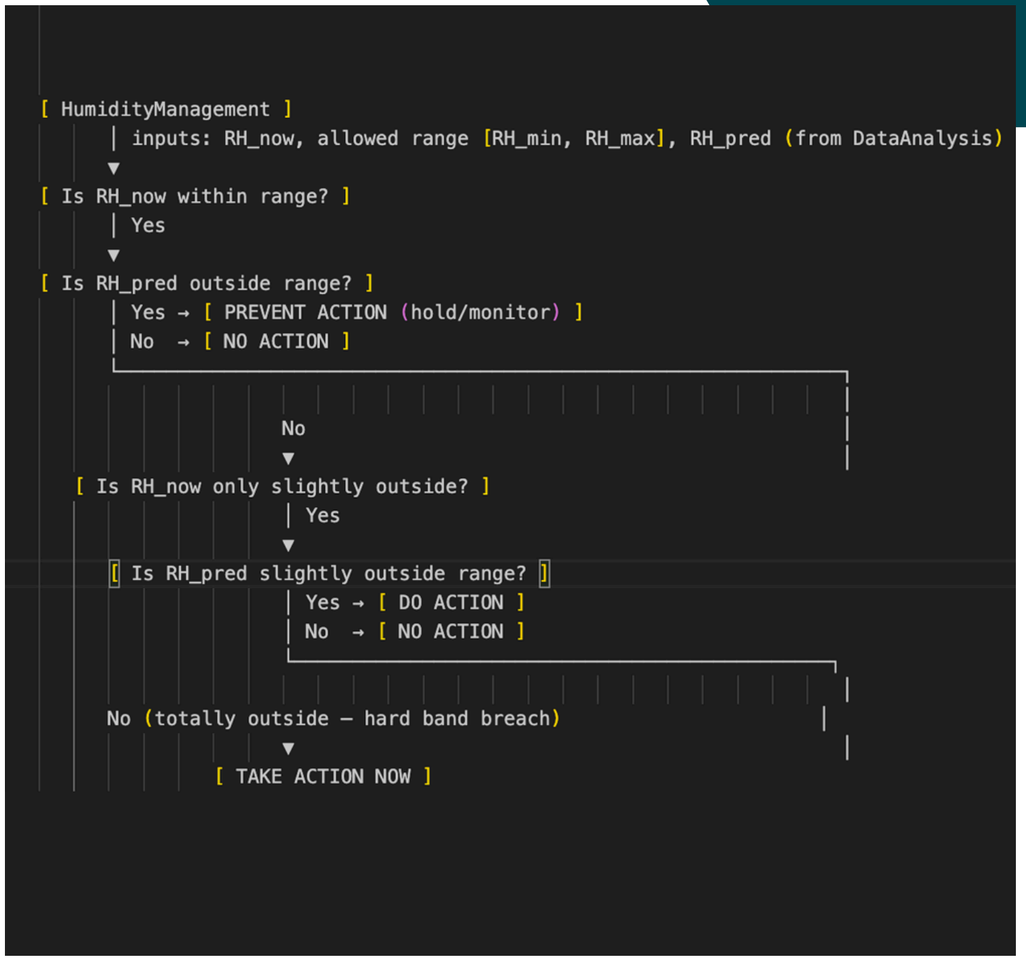
Cooperation of Data Analysis with Other Components

- **Message Broker (MQTT):**
  - Subscribes to sensor measurement topics (via broker).

- **Management Microservices:**
  These consume the statistics/forecasts provided by Data Analysis to decide and execute control actions.
  - Humidity Management
  - Light Management
  - Nutrient Management

Humidity Management - Overview

- Ensures that greenhouse humidity and soil moisture levels remain within safe thresholds.
- Monitors anomalies by checking for unexpected readings.
- Detects missing values using a timer set to the expected arrival time.
- Works as a control service subscribing to sensor topics and publishing corrective actions.
- Receives scheduled events on the sensors under its control, and publish the required actions.
- Prevents stress on plants by reacting to both low and high humidity values.



Other Management Components

Light Management

Based on the same logic, has the aim of ensuring that greenhouse temperature and light intensity values remain within the desired interval.

Nutrient Management

Following the same logic, aims to ensure that Nitrogen (N), Phosphorus (P), Potassium (K), and pH remain within the optimal range for plant growth.

Humidity Management - Functionality

- **Input sources:**
  - Current humidity value from sensors.
  - Allowed humidity range
  - Predicted next humidity value from Data Analysis.

- **Decision logic:**
  - Normal case (within range):
  - If the current humidity is inside the range, the system checks the prediction.
  - If the predicted next value is also inside → no action needed.
  - If the predicted next value is outside → prevent immediate action, issue a warning, and keep monitoring.
  - Slow drift (slightly outside, soft-band breach):
  - If the humidity is only slightly outside the range:
  - If the prediction shows it will remain outside → perform corrective action (e.g., adjust humidifier/fan).
  - Otherwise → no action needed.
  - Hard breach (totally outside range):
  - If the humidity is far outside the limits, the module immediately triggers corrective action.

Humidity Management - Example

Incoming measurements and performed actions:

```
Received: {'bn': 'greenhouse_41/area_44/sensor_33', 'e': {'n': 'Humidity', 'v': 78.04, 'u': '%', 't': 120}}
The next value of Humidity (sensor_33) is expected to be received at: 180.0
The next value of Humidity (sensor_33) is expected to be: 77.92999999999999 %
Wait to see another value to start the decision making for sensor_33 (Humidity)
Periodic sensor check started
Periodic sensor check completed

Received: {'bn': 'greenhouse_41/area_44/sensor_33', 'e': {'n': 'Humidity', 'v': 78.59, 'u': '%', 't': 180}}
The next value of Humidity (sensor_33) is expected to be received at: 240.0
The next value of Humidity (sensor_33) is expected to be: 78.7 %
WARNING: The measured value 78.59 % of Humidity (sensor_33) went outside the range [30, 75]
WARNING: The measured value 78.59 % and the next expected one 78.7 % of sensor_33 (Humidity) are outside the range
Action needed to decrease the value
Periodic sensor check started
Periodic sensor check completed

Published to greenhouse_41/area_44/action/sensor_33: {"message": {"action": "decrease", "val": 78.59, "min_treshold
```

Humidity Management - Scheduled Events

1. It receives scheduled events published by the TimeShift microservice:
   - greenhouse_<id>/area_<id>/event/sensor_<id>

2. Delete the received event from the DB and it schedules the next one following the periodicity.

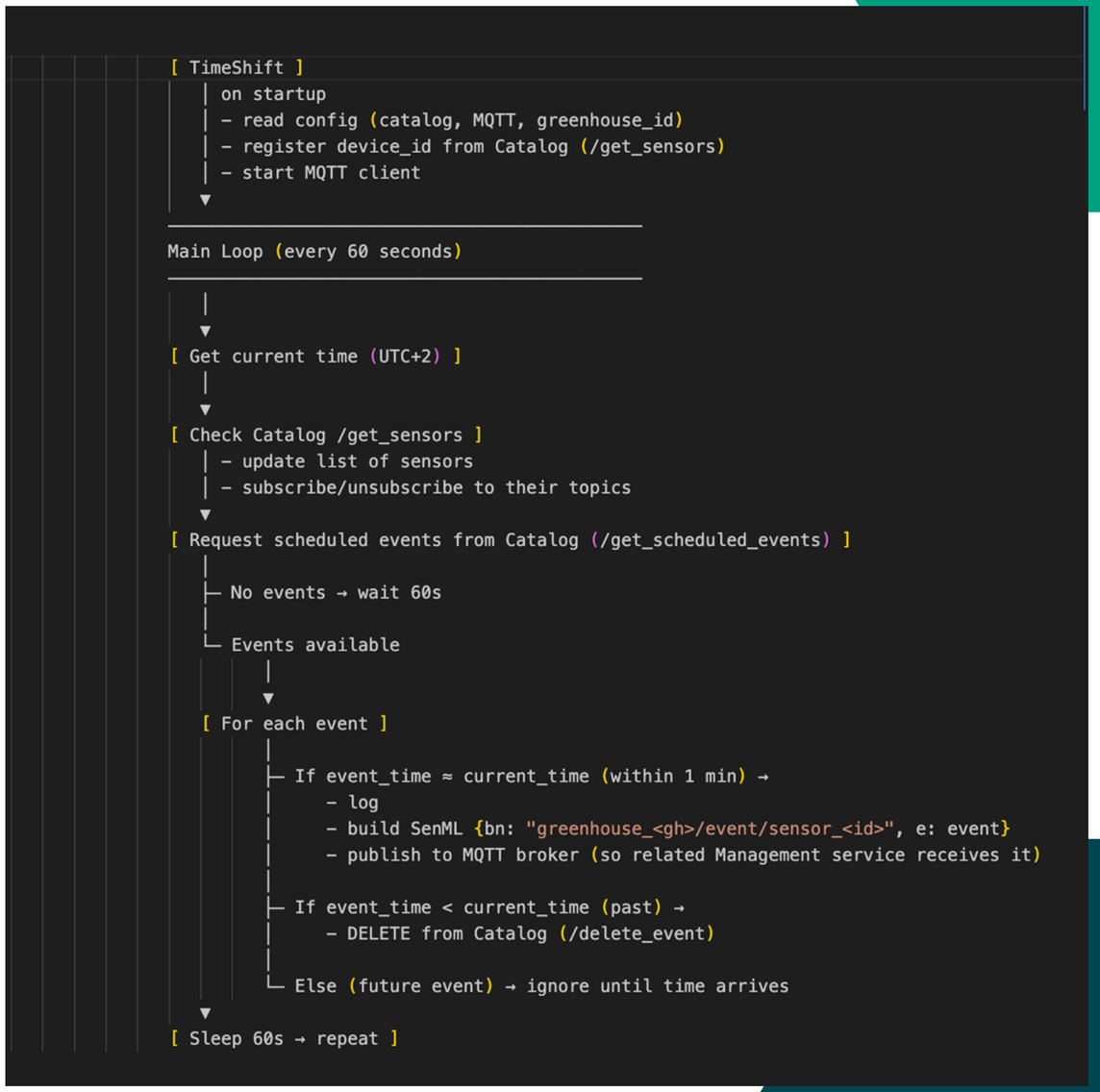
3. Publish the relative action to reach the desired value.
   - greenhouse_<id>/area_<id>/action/sensor_<id>

Cooperation of Management Services with Other Components

- **With Data Analysis:**
  - get the next expected time of arrival (time stamp) and the next expected value.
- **With Catalog:**
  - check for updates in the list of sensors in the Catalog then remove or add topics based on the new list of the sensors
  - fetch the telegram chat_id to send notifications about the status of the greenhouse.
  - delete events by requesting the related end point, and if the event has daily/weekly/monthly frequency it schedules the next one.
- **Telegram Bot API:**
  - send messages to users in the telegram chat associated to the greenhouse.

TimeShift

- Periodically reads scheduled events stored in the Data Base.
- Selects events that need to be executed at the current day and time.
- Publishes selected events to be received by the related management components.
- Deletes events that are no longer relevant or have already been executed.



Future Work

- **Hardware Implementation**
  - Raspberry Pi integrated with sensors and actuators for real-time monitoring and control.
- **Enhanced Trend Detection**
  - Machine Learning models to detect irregular sensor data or equipment failures before they affect crops.
- **Improved Telegram Functionality**
  - Retrieve live sensor readings.
  - Access history of corrective actions.
  - Receive automated plant care suggestions and recommendations.