



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

SOFTWARE ENGINEERING FOR THE IOT

Smart Luggage Tracking

Author :
Juliette WALTREGNY 293652

Professor : D. DI RUSCIO
Year : 2023-2024

Contents

1	Introduction	2
2	System architecture	2
3	Docker	3
4	Components	4
4.1	Data	4
4.2	Mosquitto	4
4.3	Node-Red	4
4.4	Telegraf	5
4.4.1	Challenges	6
4.5	InfluxDB	7
4.6	Grafana	7
5	Testing	8

1 Introduction

In today's rapidly evolving technological world, smart luggage tracking emerges as an innovative solution for travelers. With the increasing volume of travelers and the growing complexity of transport systems, luggage management has become a significant challenge for airlines, airport operators, and travelers themselves. Smart luggage tracking systems leverage advanced technologies such as geolocation and wireless communication to provide an integrated solution for locating and monitoring in real-time.

This project aims at creating a simplified version of this smart luggage monitoring. It explores the various technologies and architectures involved in smart luggage tracking, examines the benefits and challenges associated with their implementation, and proposes recommendations to enhance the efficiency and reliability of these systems. By investigating the integration of components such as Docker, Mosquitto, Node-RED, Telegraf, InfluxDB, and Grafana, this report aims to provide a comprehensive overview of the system architecture and the processes involved in smart luggage tracking.

2 System architecture

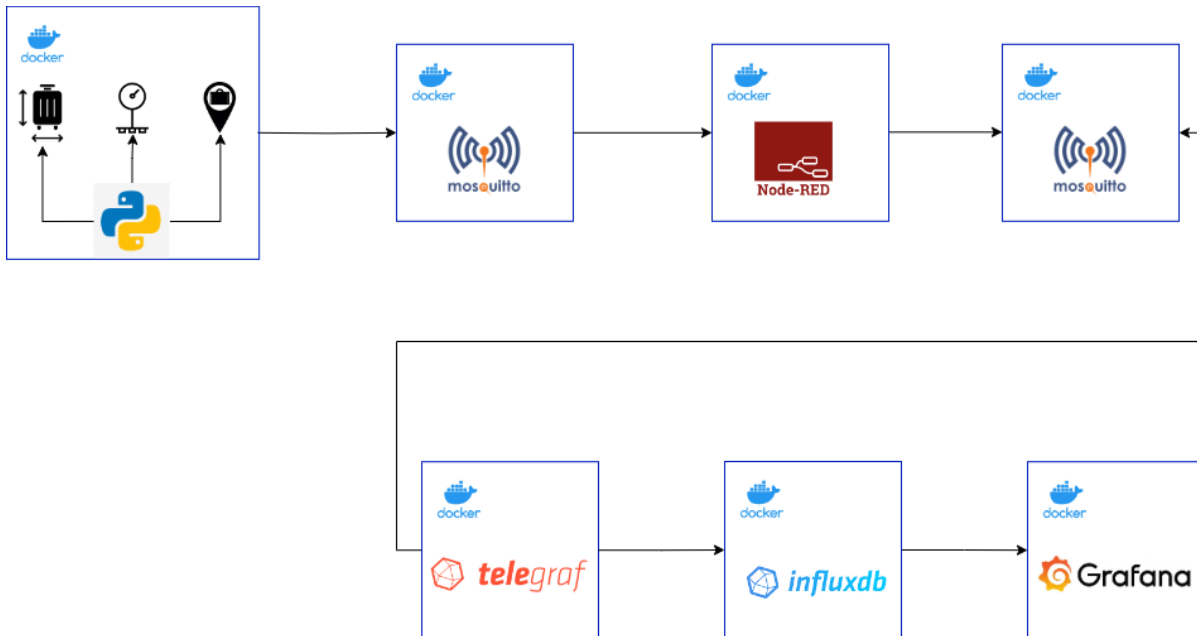


Figure 1: System architecture

The system is composed of seven different components and articulated in four main steps:

1. **Data Collection:** The python script generates data from different sensors attached to the luggage and publishes it to Mosquitto MQTT Broker.
2. **Data Processing:** The first Mosquitto Broker passes the data to Node-Red, which wires together the services and transforms the data into a format suitable for telegraf. Node-Red publishes the transformed data to the second Mosquitto broker.
3. **Data Distribution:** telegraf collects the data from the MQTT topic and sends it to InfluxDB for storage.

4. **Data Visualization:** Grafana retrieves the data from influxDB and provides a user-friendly interface to visualize the data.

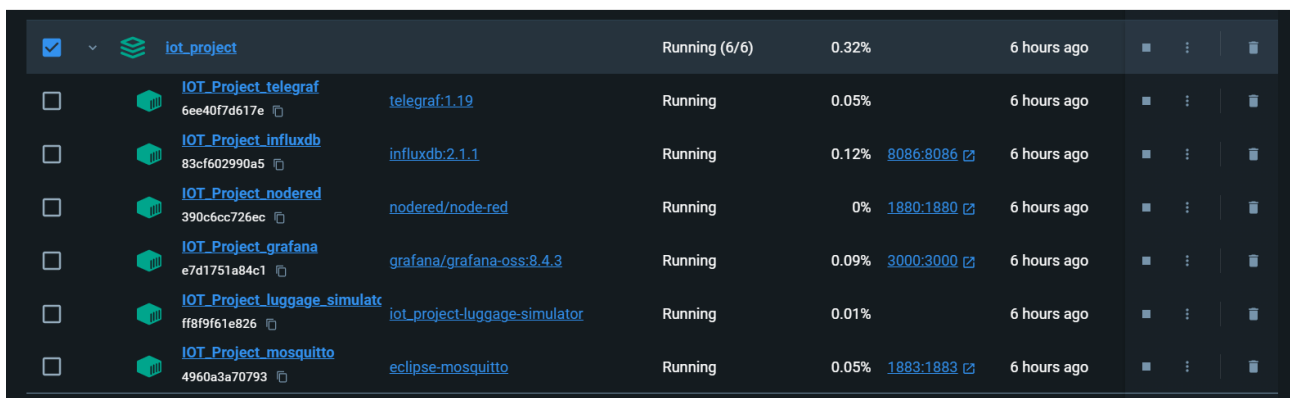
This is a brief overview of the system, each component will now be explained in details.

3 Docker

Every component presented above is containerized in Docker. In particular, Docker Compose was used for several advantages it offers:

- **Simplified Multi-container Deployment and Configuration:** thanks to the docker-compose.yml file, all containers are defined and managed simultaneously, especially the images, services, networks, dependencies and volumes needed. The python script is containerized too. In addition, the environment variables and ports are defined and centralized which makes it more manageable.
- **Dependency Management:** docker compose handles the dependencies automatically with the 'depends_on' directive. This allows node-red and mosquitto to be in synch as well as telegraf and influxdb.
- **Networking:** Docker Compose automatically creates networks and allows services to discover eachother by their service names defined in the 'docker-compose.yml' file. This simplifies the setup of complex networking files.
- **Environment Management:** passing environment variables to containers can lead to mistakes if done manually for multiple containers. Docker compose allows to define environment variables in a separate .env file, simplifying environment management.

Here is an overview of the Docker Compose container.



iot_project		Running (6/6)	0.32%	6 hours ago	
<input type="checkbox"/>	IOT_Project_telegraf 6ee40f7d617e	telegraf:1.19	Running	0.05%	6 hours ago
<input type="checkbox"/>	IOT_Project_influxdb 83cf602990a5	influxdb:2.1.1	Running	0.12% 8086:8086	6 hours ago
<input type="checkbox"/>	IOT_Project_nodered 390c6cc726ec	nodered/node-red	Running	0% 1880:1880	6 hours ago
<input type="checkbox"/>	IOT_Project_grafana e7d1751a84c1	grafana/grafana-oss:8.4.3	Running	0.09% 3000:3000	6 hours ago
<input type="checkbox"/>	IOT_Project_luggage_simulator ff8f9f61e826	iot_project-luggage-simulator	Running	0.01%	6 hours ago
<input type="checkbox"/>	IOT_Project_mosquitto 4960a3a70793	eclipse-mosquitto	Running	0.05% 1883:1883	6 hours ago

Figure 2: Containers overview

4 Components

4.1 Data

The luggages data was simulated from a python script. 1000 luggages were created with different types of sensors generated:

- **Location:**
 - Latitude: between -90 and +90 degrees.
 - Longitude: between -180 and 180 degrees.

Location is updated in a range from -0.5 and +0.5 degrees every 5s for random luggages.

- **Size:**
 - Height: between 30 and 100 cm.
 - Width: between 20 and 40 cm.
 - Length: between 30 and 60 cm.
- **Weight:** between 5 and 30 kg.

The simulation of these sensors was done through the "simulate_luggage.py" Python script to simulate the real-time tracking of 1000 pieces of luggage using MQTT. The script begins by setting up the MQTT connection to a broker, with the default broker address of **localhost** and the topic named "luggage/tracking" for publishing the data. It generates initial data for each piece of luggage, including unique IDs, random locations (latitude and longitude), sizes (length, width, height), weight, and a timestamp. The script then enters a loop where it randomly selects a piece of luggage, updates its location by slightly altering its latitude and longitude, and publishes the updated data to the MQTT broker every 5 seconds. The connection to the MQTT broker is established and maintained using the `paho.mqtt.client` library, with proper handling for successful connection and reconnection if needed. The script also includes a graceful shutdown mechanism to stop the MQTT loop and disconnect from the broker when interrupted, ensuring a clean exit. This approach allows for continuous and realistic simulation of luggage tracking data being sent to an MQTT broker for tracking and monitoring luggages.

4.2 Mosquitto

The Mosquitto MQTT broker facilitates the transfer of data. Initially, data is published to the "luggage/tracking" topic. This data is then processed by Node-RED and then published to another topic, "telegraf/luggage/tracking." Telegraf subscribes to this topic to retrieve the processed data and send it to InfluxDB for storage and analysis.

4.3 Node-Red

Node-Red is responsible for connecting the simulated sensors and Telegraf. Here's how the flow is built:

1. The data is first sent to an "MQTT-in" node.
2. The "format_to_telegraf" function converts the MQTT message into a structure with measurements, fields, tags, and a correctly formatted timestamp, ensuring the data is properly structured for Telegraf.
3. Finally, Telegraf retrieves the correctly formatted data from the "MQTT-out" node, allowing it to send the data to InfluxDB.

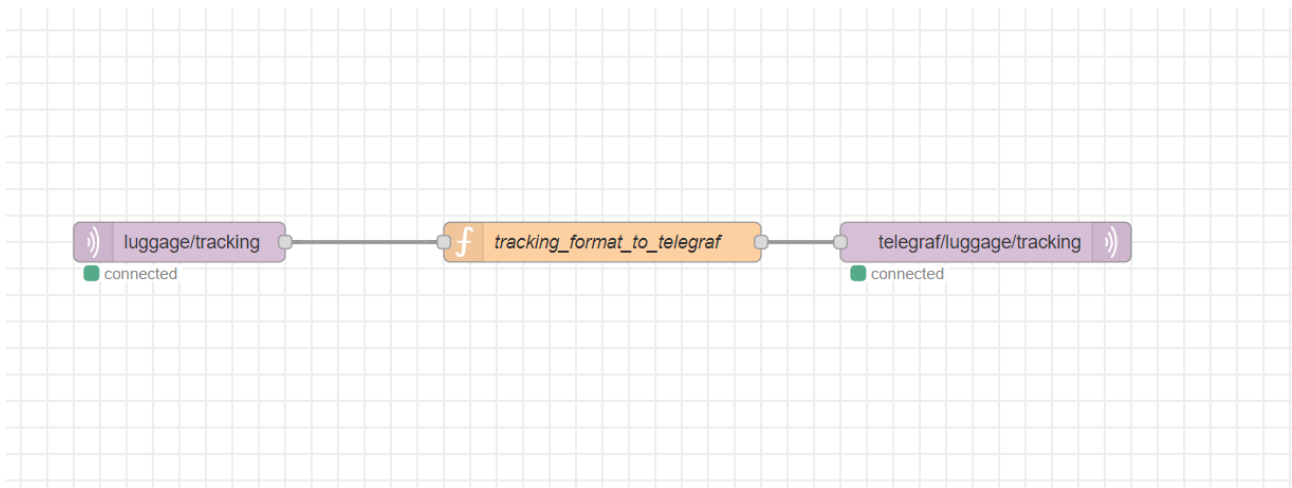


Figure 3: Node-Red data flow

Here is the Node-Red data flow:

And the "tracking_format_to_telegraf" function's body:

```

1  var payload = msg.payload;
2
3  // Log the incoming payload for debugging
4  node.warn("Incoming payload: " + JSON.stringify(payload));
5
6  // Ensure the payload has the required structure
7  if (payload && payload.location && payload.size && payload.id && payload.timestamp && payload.status) {
8    // Create the new payload structure
9    msg.payload = [
10     {
11       measurement: "luggage_tracking",
12       fields: {
13         latitude: payload.location.latitude,
14         longitude: payload.location.longitude,
15         weight: payload.weight,
16         height: payload.size.height,
17         length: payload.size.length,
18         width: payload.size.width,
19         status: payload.status
20       },
21       tags: {
22         id: payload.id
23       },
24       timestamp: payload.timestamp * 1000000000
25     }
26   ];
27
28   // Add a message to the payload
29   msg.message = "Luggage tracking data transformed for Telegraf";
30
31   // Log the transformed payload for debugging
32   node.warn("Transformed payload: " + JSON.stringify(msg.payload));
33 } else {
34   // Log an error if the payload structure is incorrect
35   node.error("Invalid payload structure", msg);
36 }
37
38 // Return the transformed message
39 return msg;
40

```

Figure 4: Format to telegraf nodered function

4.4 Telegraf

Telegraf serves as a bridge between the MQTT broker and InfluxDB. As stated above, it retrieves the data from "telegraf/lugagge/tracking" and publishes it to the "luggage_track" bucket in influxDB. A good telegraf configuration is primordial for the data to be transferred correctly.

Here is the content of the "telegraf.conf file":

The most important fields are found under "inputs.mqtt_consumer", which define where telegraf retrieves the data and "outputs.influxdb.v2", which define where telegraf has to send the data.

More specifically, the server and urls have to match the MQTT broker address and port and influxDB address and port.

```

# Global tags can be specified here in key="value" format.
[global_tags]

# Configuration for telegraf agent
[agent]
  interval = "5s"
  round_interval = true
  metric_batch_size = 1000
  metric_buffer_limit = 10000
  collection_jitter = "0s"
  flush_interval = "5s"
  flush_jitter = "0s"
  precision = ""
  hostname = ""
  omit_hostname = false

[[inputs.mqtt_consumer]]
  servers = ["tcp://IOT_Project_mosquitto:1883"]
  topics = ["telegraf/luggage/tracking"]
  qos = 0
  connection_timeout = "30s"
  data_format = "json"

[[outputs.influxdb_v2]]
  urls = ["http://influxdb:8086"]
  token = "${DOCKER_INFLUXDB_INIT_ADMIN_TOKEN}"
  organization = "${DOCKER_INFLUXDB_INIT_ORG}"
  bucket = "${DOCKER_INFLUXDB_INIT_BUCKET}"
  insecure_skip_verify = false

# Read metrics about cpu usage
[[inputs.cpu]]
  percpu = true
  totalcpu = true
  collect_cpu_time = false
  report_active = false

# Read metrics about disk usage by mount point
[[inputs.disk]]
  ignore_fs = ["tmpfs", "devtmpfs", "devfs", "iso9660", "overlay", "aufs", "squashfs"]

```

Figure 5: Telegraf.conf file

4.4.1 Challenges

We ran into some challenges with Telegraf.

There was an issue with the order of service execution. Telegraf would start before InfluxDB, but it depends on InfluxDB being up to publish data correctly.

We first tried to fix this with a health check in the docker-compose.yml file, ensuring InfluxDB was running before starting Telegraf. However, some issues persisted.

Next, we attempted to use the "wait-for-it" script in the entrypoint to force Telegraf to wait until InfluxDB was fully set up before starting. Unfortunately, this didn't work on Windows because the system recognized "wait-for-it" as a directory and couldn't execute it as a file.

Our third solution was to manually start InfluxDB first, create the required bucket, and then start Telegraf. This method worked, allowing Telegraf to send data to the correct bucket.

4.5 InfluxDB

InfluxDB stores the data in the bucket "luggage_track". Each measure is stored alongside a timestamp, as the location is frequently updated. Here is a snippet of the data stored, with an overview of the fields from one specific luggage.

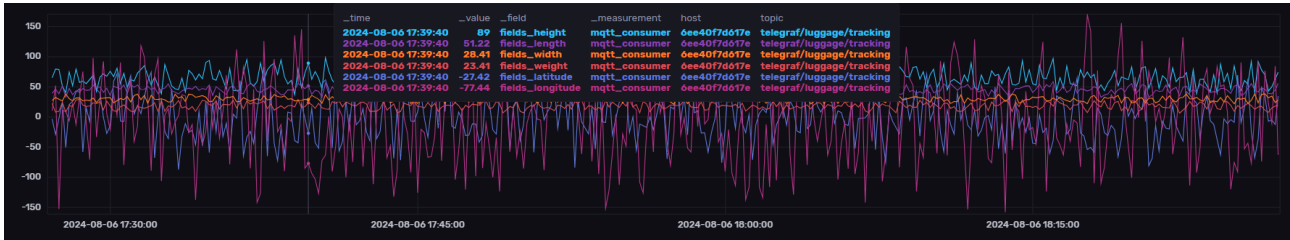


Figure 6: InfluxDb Example Data

4.6 Grafana

The final step in our smart luggage tracking system is data visualization, facilitated by Grafana. Grafana provides a user-friendly platform for creating interactive dashboards and panels. For this project, luggage data is displayed on a geomap panel, as illustrated below:

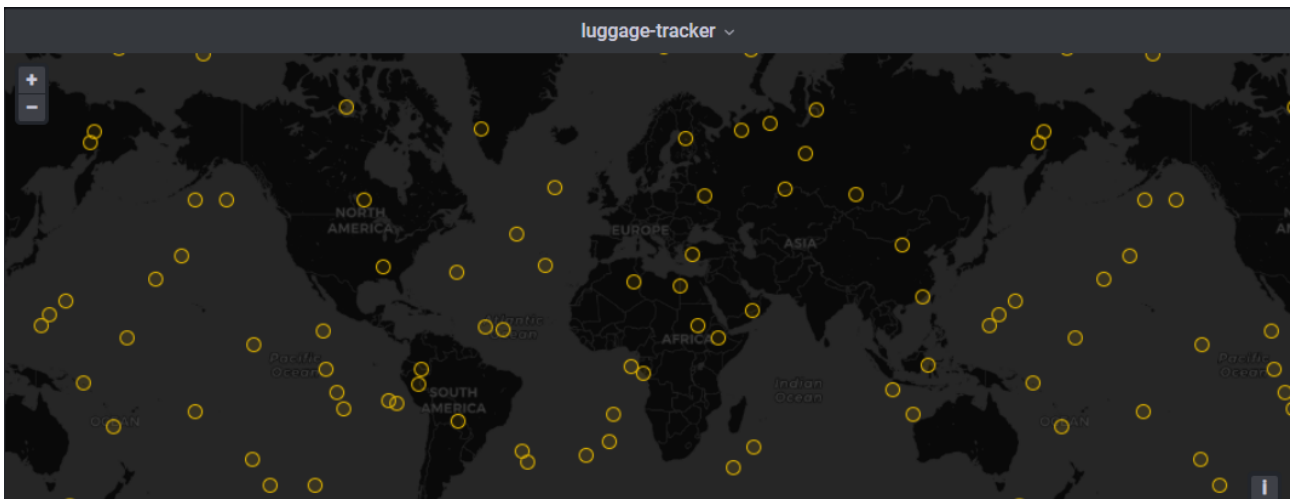


Figure 7: Grafana Dashboard

To enhance visualization, we filtered the data to display only 100 pieces of luggage. Hovering over each dot on the map reveals additional fields related to each piece of luggage, allowing for comprehensive real-time monitoring.



Figure 8: Displaying luggage fields

We utilized a Flux query to filter the desired data, which enabled us to select the latest data entries for 100 pieces of luggage. This approach ensures that the dashboard displays the most current location updates effectively.

Below is the Flux query that enabled the display of data:

```

1 from(bucket: "luggage_track")
2   |> range(start: -1h)
3   |> filter(fn: (r) => r._measurement == "mqtt_consumer")
4   |> pivot(rowKey: ["_time"], columnKey: ["_field"], valueColumn: "_value")
5   |> sort(columns: ["_time"], desc: true)
6   |> sort(columns: ["tags_id"], desc: true)
7   |> unique(column: "tags_id")
8   |> keep(columns: ["fields_latitude", "fields_longitude", "fields_height", "fields_length", "fields_width", "fields_status", "fields_weight", "tags_id"])
9   |> limit(n:100)
10  |> map(fn: (r) => ({ r with latitude: float(v: r.fields_latitude), longitude: float(v: r.fields_longitude) })))

```

Figure 9: Flux query

5 Testing

The setup and credential information to use the system are explained in the project's repository.

Note: It will take several minutes for all the luggages to appear in the dashboard as they are published one by one.