Real-Time IOT Dashboard With MQTT, InfluxDB & Grafana..

TABLE OF CONTENTS

Contents

❖ INTRODUCTION

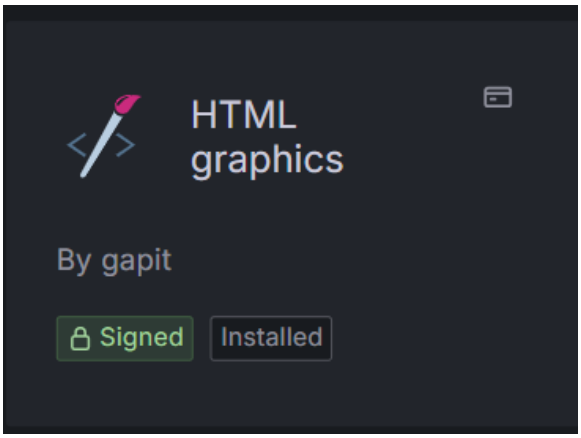
This system is designed to create a comprehensive real-time monitoring and control infrastructure for IoT applications by integrating several key technologies. It begins with collecting sensor data from an ESP32, which reads sensor data and transmits the data wirelessly using the lightweight MQTT (Message Queuing Telemetry Transport) protocol. These MQTT messages are published to a broker, where they are either subscribed to directly for visualization or processed by an intermediate script that parses the data and stores it in InfluxDB, a high-performance time-series database optimized for handling chronological sensor data. Grafana is then used as the front-end visualization tool, reading from InfluxDB and rendering the data in interactive dashboards that update in real-time.

Additionally, the system includes a real-time monitoring interface using Grafana's HTML panel plugin, where sensor values and device status are dynamically updated using JavaScript and mqtt.js over WebSocket MQTT communication. This panel automatically listens to live MQTT topics (factory/all for sensor data and factory/status for device status), parses incoming JSON messages, and updates the displayed values and connection status visually. It also caches the last received data in the browser's local storage, allowing the panel to show the most recent sensor readings and device status even after page refreshes or network interruptions. This design ensures a seamless, automatic, and user-friendly way to visualize real-time sensor updates and device connectivity health, enhancing system reliability and user experience without requiring active manual control actions.

❖ SYSTEM COMPONENTS AND THEIR ROLES

Component	Description
ESP32 Board	The ESP32 collects physical sensor data (such as temperature, current, etc.) from its connected sensors. It packages the data into JSON format and publishes it wirelessly to specific MQTT topics (e.g., factory/all, factory/status) via a Wi-Fi connection.
MQTT broker	MQTT broker acts as the central message router between publishers (like the ESP32) and subscribers (like the InfluxDB logger or the Grafana HTML panel). It ensures reliable delivery of MQTT messages over TCP port and WebSocket port connections.
InfluxDB	InfluxDB receives sensor data from the Python data ingestor or another service subscribed to MQTT. It stores the sensor measurements along a timestamp, device ID, and field values, enabling efficient chronological querying and historical analysis.
Grafana	Grafana reads stored sensor data from InfluxDB and renders it in real-time on highly customizable dashboards. It also integrates a HTML panel (using mqtt.js) to display live data updates and device statuses via MQTT WebSocket connections.
Docker-compose file	A docker-compose.yml file defines and runs all the necessary services (InfluxDB, Grafana, Python scripts) as isolated containers, ensuring they start automatically, communicate within a secure network, and recover easily from failures.

❖ HTML GRAPHICS PANEL



The HTML graphics Panel plugin in Grafana allows embedding custom HTML, CSS, and JavaScript directly inside a Grafana dashboard. This flexibility enables you to create highly customized and interactive visual elements beyond standard Grafana charts. Such as live MQTT-connected sensor cards, status indicators or control interfaces (like toggle button). In this system, the HTML panel uses mqtt.js to establish a real-time WebSocket connection to the MQTT broker, subscribe to topics, and dynamically update the dashboard without refreshing the page. It also uses browser localStorage to cache sensor data and device statuses for persistence across reloads.

Installation

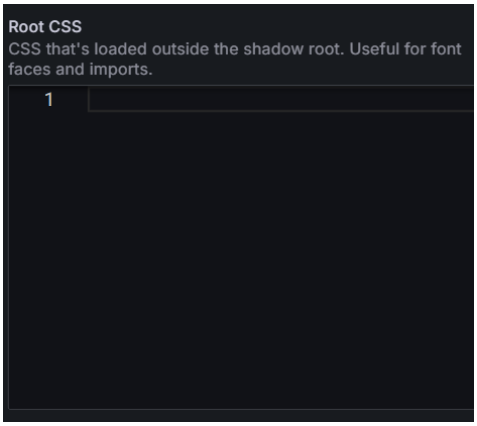
Docker-compose (recommended)

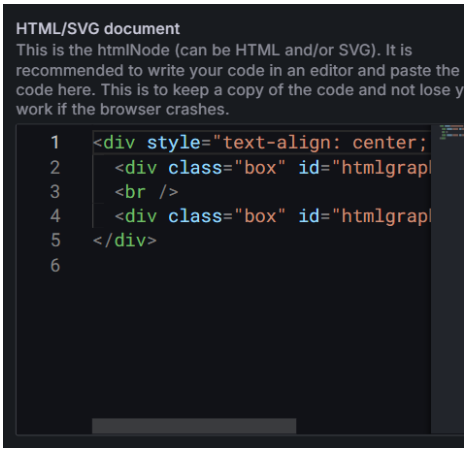
```yaml
version: '3'
services:
  grafana:
    image: grafana/grafana
    container_name: grafana
    restart: always
    networks:
      - grafana
    ports:
      - 3000:3000
    environment:
      - GF_INSTALL_PLUGINS=gapit-htmlgraphics-panel

networks:
  grafana:
    name: grafana
```

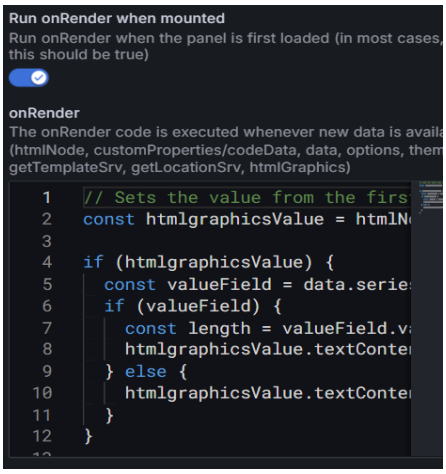
For more details : https://gapit-htmlgraphics-panel.gapit.io/docs/installation/

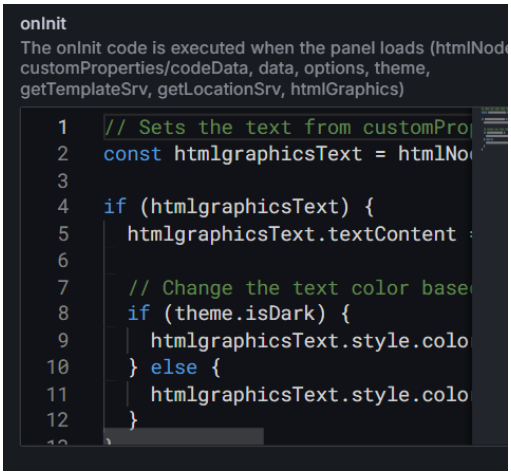
Tabs in HTML graphics panel

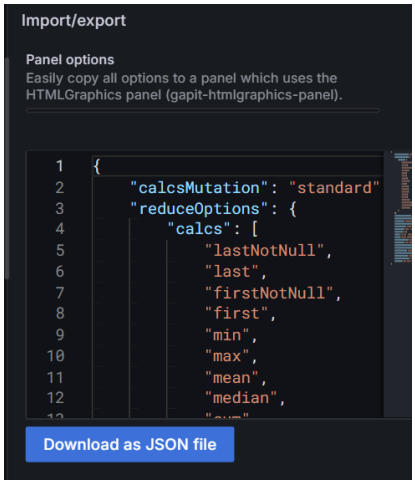
When you configure an HTML Graphics Panel in Grafana using the gapit-htmlgraphics-panel plugin, several important tabs (sections) are available to customize the behavior and appearance of the panel.











Limitations faced when using HTML graphics panel's tab

1. The HTML tab only allows static HTML code. You cannot run server-side scripts (like PHP, Node.js) or fetch secure data dynamically without external APIs.

2. CSS written in CSS tab only applies inside the panel itself. You cannot easily change Grafana's outside frame, header, sidebar, or global dashboard styles from this CSS. Also, sometimes Grafana's own styles override custom CSS if not carefully managed.

3. The JavaScript in the onrender tab runs inside a "sandbox" environment. Only basic web operations (WebSocket, fetch, etc.) are allowed. Complex libraries must be loaded manually (like mqtt.js from a CDN). Loading mqtt.js manually inside JavaScript because Grafana panel has no direct MQTT support.

4. Since you are writing code inside a Grafana panel (not a full IDE like VSCode), it's harder to debug syntax errors, see console outputs properly, or use breakpoints easily. Mistakes in JS can cause the whole panel to fail silently without clear error messages.

5. Handling localStorage to cache data manually, since the HTML panel doesn't remember anything after a browser refresh automatically.

6. Handling WebSocket reconnections manually. Because if MQTT connection drops, no auto-recovery is built into the panel.

7. If too much JavaScript is written, the panel might slow down or freeze (because it runs on the client browser directly).

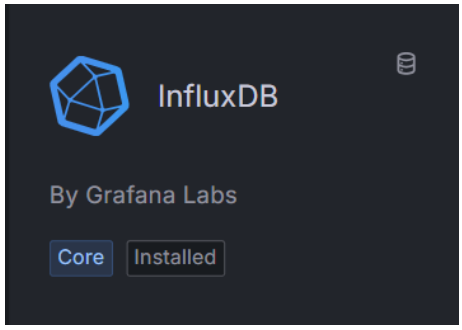
- Because of those limitations in HTML/SVG tab and CSS tab, I only use onrender tab to create my codes.

- In Grafana, HTML graphic panel is just a small piece inside a bigger Grafana page. You can't write full HTML with <html>, <body>, etc. only a fragment is allowed inside the panel's box. Using htmlNode.innerHTML = '...' allows to inject dynamic HTML into this panel. Using innerHTML allows to dynamically create the device cards (device1, device2, device3) during panel load, instead of writing static HTML manually for each device.

❖ INFLUXDB DATASOURCE

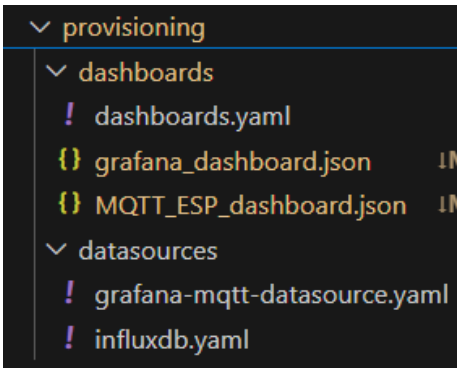
InfluxDB serves as the primary time-series database to store and retrieve sensor data coming from the ESP32 devices via MQTT. The data source configuration in Grafana connects Grafana panels directly to InfluxDB to visualize live and historical sensor values.

Datasource Setup:

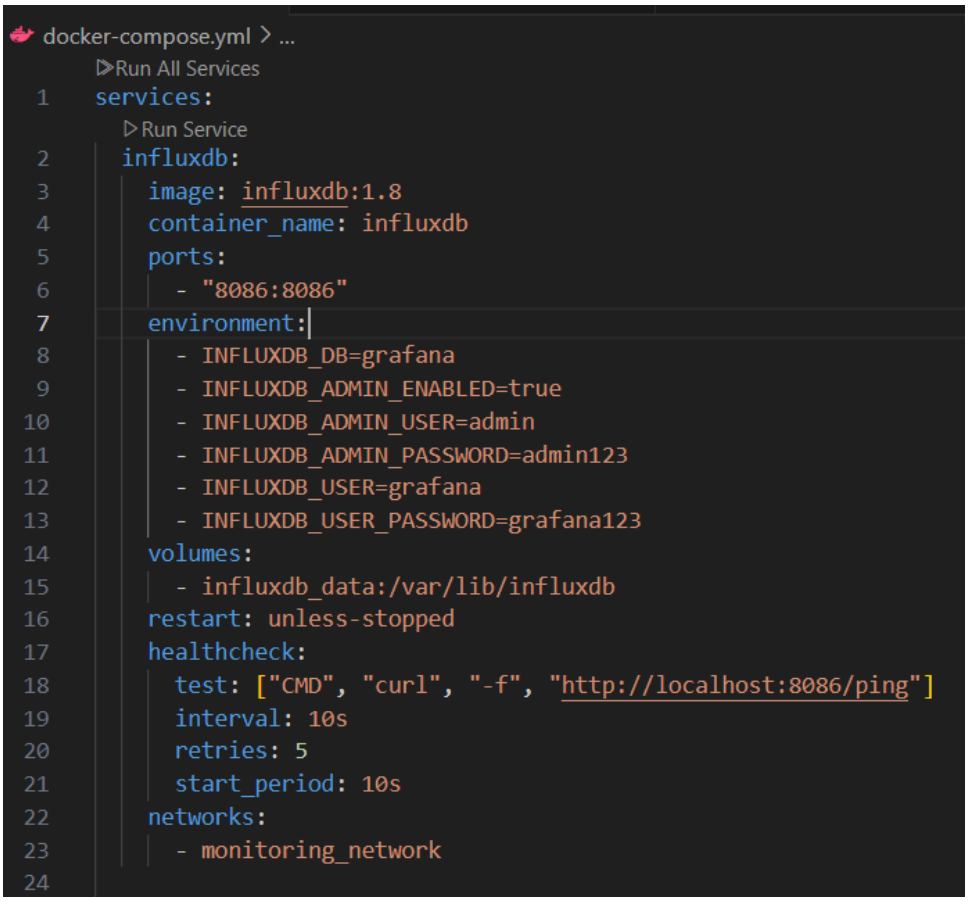


- Database Name: grafana
- Measurements: sensor_data
  ESP_data
- Database Type: InfluxDB 1.8
- Datasource Name in Grafana: InfluxDB
- URL: http://influxdb:8086
- Access: Proxy (Grafana server will directly query InfluxDB)
- User Credentials:
  - User: grafana
  - Password: grafana123
- Query Language: InfluxQL (a SQL-like query language designed for time-series data)

This datasource is provisioned automatically using the influxdb.yaml file inside the Grafana provisioning folder, so the dashboard setup becomes fully automated without manual configuration inside Grafana's UI.



Docker container



- Why Can't You Open InfluxDB 1.8 via http://localhost:8086 in a Web Browser?

Because InfluxDB 1.8 does not provide a web-based user interface (UI) on that port.

Port 8086 is used by InfluxDB to expose its HTTP API.

- Writing data (POST /write)
- Querying data (GET /query)
- Authentication and database management (via API calls)

It is not designed to serve a graphical UI (like MySQL's phpMyAdmin or MongoDB Compass)

If you need to access influxdb, you can use three methods.

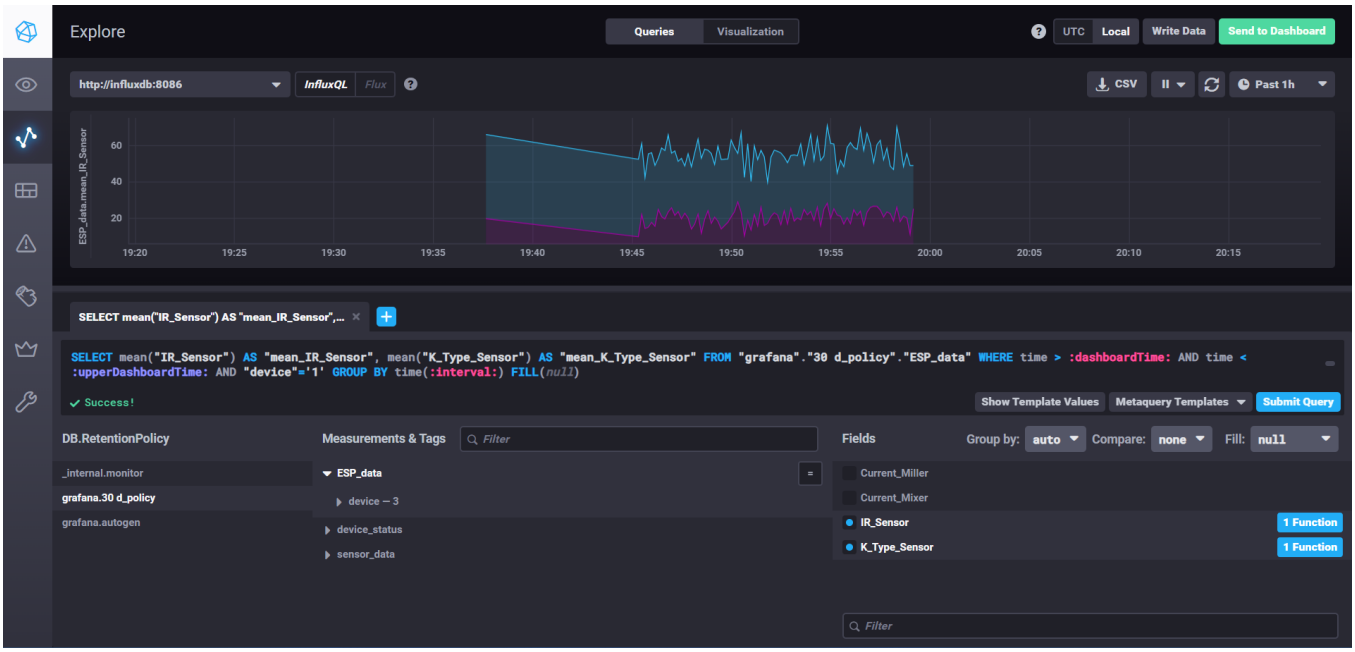
i. InfluxDB CLI
ii. Grafana
iii. Chronograf

Accesiing influxDb using chronograf

InfluxDB 1.8 itself does not provide a UI on a web port.But we can use chronograf as web access option.
First we need to add this chronograf into our docker-compose.yml file:

```
▷ Run Service
chronograf:
  image: chronograf:1.10
  container_name: chronograf
  ports:
    - "8888:8888"
  environment:
    - INFLUXDB_URL=http://influxdb:8086
  depends_on:
    - influxdb
  networks:
    - internal
```

Then you can access it using port : http://localhost:8888



Accessing InfluxDB Using CLI (Command Line Interface)

To manually inspect the sensor data stored in InfluxDB, you can use the built-in InfluxDB CLI tool.

```
PS C:\Users\Hp\Desktop\Grafana-setup> docker exec -it influxdb influx
Connected to http://localhost:8086 version 1.8.10
InfluxDB shell version: 1.8.10
> USE grafana
Using database grafana
> SHOW MEASUREMENTS
name: measurements
name
----
ESP_data
sensor_data
> SELECT * FROM ESP_data LIMIT 5;
name: ESP_data
time                Current_Miller Current_Mixer IR_Sensor K_Type_Sensor device
----                -------------- ------------- --------- ------------- ------
1745476107382039259 32.95          26.28         41.42     28.82         1
1745476107382039259 56.85          38.48         20.8      18.85         3
1745476107382039259 31.38          38.62         34.86     26.65         2
1745476117723146691 46.56          15.29         24.49     11.57         3
1745476117723146691 56.58          36.64         34.09     22.13         2
>
>
```

Retention policy

Also you can change RETENTION POLICIES using this CLI.

- What is a Retention Policy?

A retention policy (RP) in InfluxDB determines how long data is stored in a particular database before it is automatically deleted. It is crucial for managing storage usage, especially in systems that collect data frequently, like sensor-based IoT setups.

By default, InfluxDB stores all data forever unless a retention policy is applied.

```
> SHOW RETENTION POLICIES
name        duration shardGroupDuration replicaN default
----        -------- ------------------ -------- -------
autogen     0s       168h0m0s           1        false
30_d_policy 720h0m0s 24h0m0s            1        true
>
```

- Database: grafana
- Default retention policy is named autogen, which means:
    - No automatic deletion of old data.
    - All sensor readings are kept indefinitely unless manually purged.

You can create and apply a custom retention policy using the InfluxDB CLI.

Example:

```
> CREATE RETENTION POLICY "30d_policy" ON "grafana" DURATION 30d REPLICATION 1 DEFAULT
```

- ✓ "30d_policy": Name of the new retention policy.
- ✓ "grafana": Name of your database.
- ✓ DURATION 30d: Keep data for 30 days.
- ✓ REPLICATION 1: Single replica (default).
- ✓ DEFAULT: Makes this the new default policy

After this, new data written to the database will be automatically deleted after 30 days.

InfluxDB 2.7

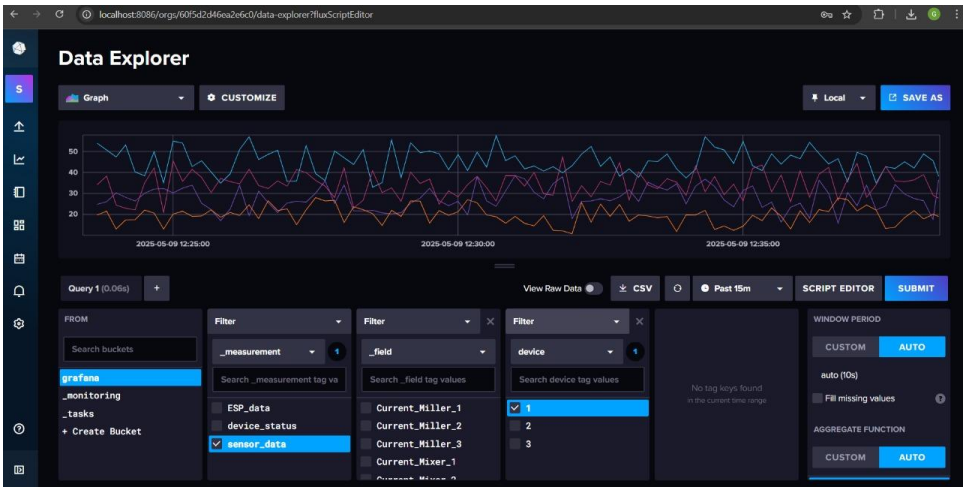
Also we can use influxDB 2.7 ,which is more beneficial,

- InfluxDB 2.x supports Flux, a much more powerful and flexible language than InfluxQL.
- It has token-based authentication.
- Simplified structure using buckets (which include retention policies). Easier to manage expiration and separation of data by use-case.
- Grafana now works directly with Flux queries.
- Reduced memory usage, faster writes and queries, and better stability.

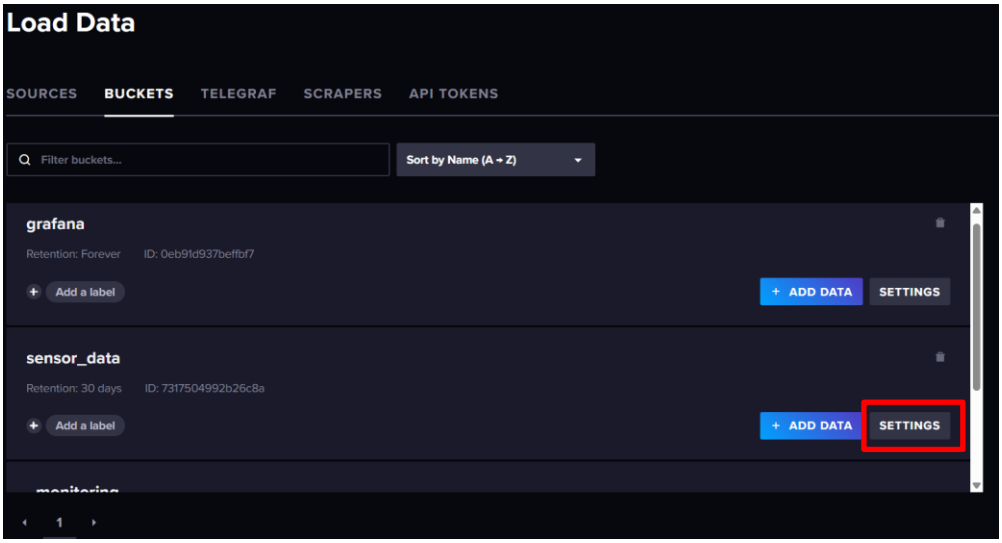
```
▷Run All Services
services:
  ▷Run Service
  influxdb:
    image: influxdb:2.7
    container_name: influxdb
    ports:
      - "8086:8086"
    volumes:
      - influxdb_data:/var/lib/influxdb2
    environment:
      - DOCKER_INFLUXDB_INIT_MODE=setup
      - DOCKER_INFLUXDB_INIT_USERNAME=admin
      - DOCKER_INFLUXDB_INIT_PASSWORD=admin123
      - DOCKER_INFLUXDB_INIT_ORG=SLT
      - DOCKER_INFLUXDB_INIT_BUCKET=grafana
      - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=my-super-token
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8086/health"]
      interval: 10s
      timeout: 5s
      retries: 5
    networks:
      - internal
```

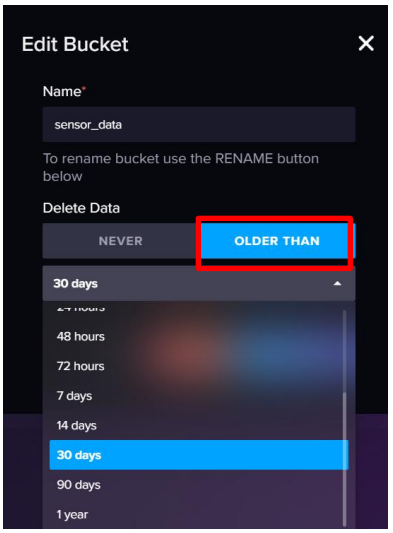
```
provisioning > datasources > ! influxdb.yaml
1  apiVersion: 1
2
3  datasources:
4    - name: InfluxDB
5      type: influxdb
6      access: proxy
7      url: http://influxdb:8086
8      isDefault: true
9      jsonData:
10       version: "Flux"
11       organization: SLT
12       defaultBucket: grafana
13       httpMode: GET
14       timeInterval: "5s"
15     secureJsonData:
16       token: my-super-token
17
```

The InfluxDB 2.7 UI is a web-based interface that provides centralized access to all the key features of the database. so no CLI or third-party tools like chronograf required.
You can access it on port http://localhost:8086 in a Web Browser.



To set retention policy:



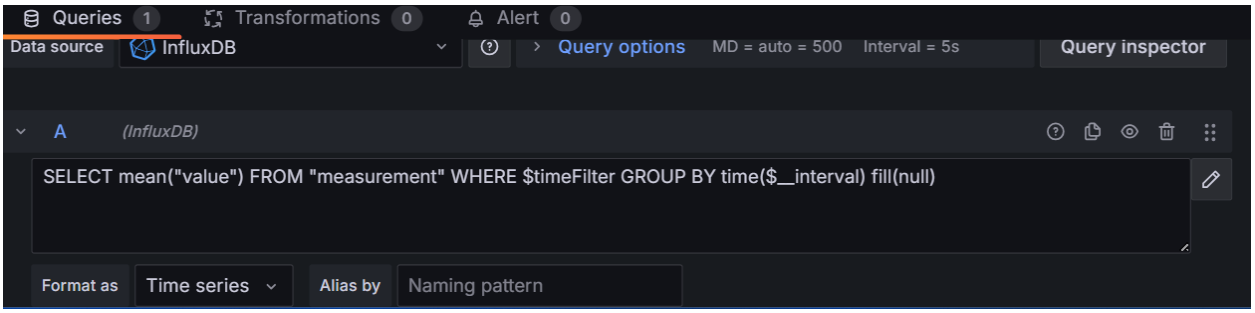


- From the left sidebar, you can go to the buckets.
- In bucket settings, you can set retention policy easily.

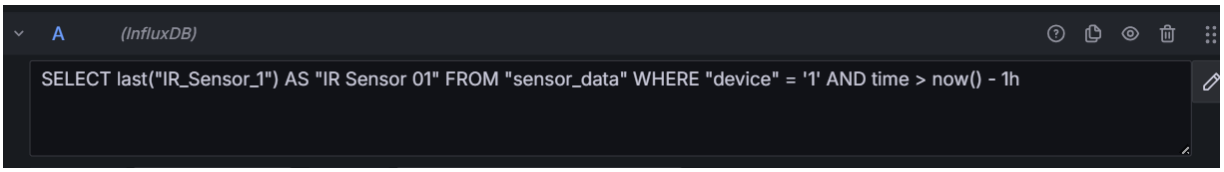
Writing queries in Grafana

- InfluxDB 1.8 - InfluxQL

InfluxQL is used as the query language to retrieve and visualize sensor data from InfluxDB 1.8 in Grafana. InfluxQL is a SQL-like language designed for working with time-series data.



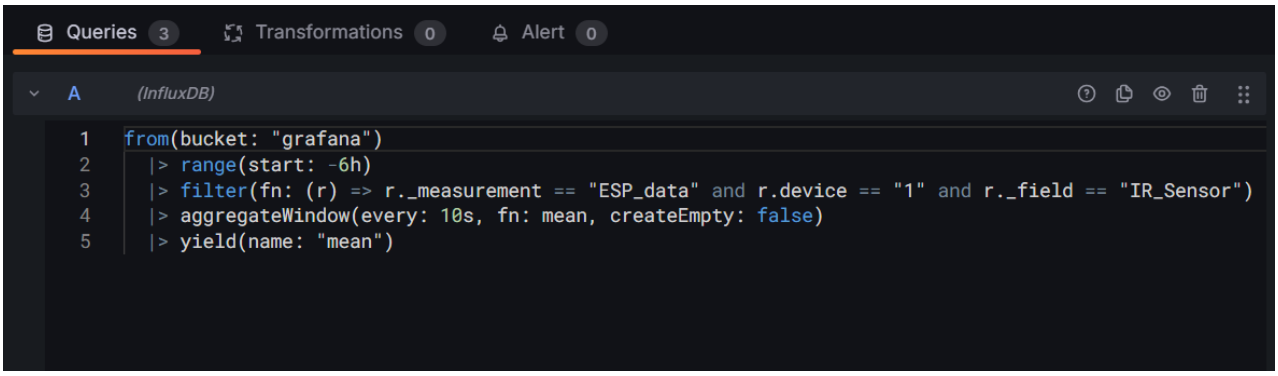
Example:



- InfluxDB 2.7 - Flux

Flux is the powerful, functional data scripting language used in InfluxDB 2.x for querying and processing time series data.

Example:



Flux language syntax: https://docs.influxdata.com/influxdb/v2/query-data/get-started/

Why We Need a Python Script to Write MQTT Data into InfluxDB?

- In this system, a Python script acts as a bridge between the MQTT broker and the InfluxDB database.
  It listens to incoming MQTT messages, parses the JSON data, reformats it into InfluxDB's structure, and writes it properly into the database.

- InfluxDB uses HTTP API (or line protocol) for writing data, but ESP32 (and most microcontrollers) are optimized for lightweight MQTT communication, not HTTP POST requests with InfluxDB's strict format.

- Using python script:

  I. Connects to the MQTT broker and subscribes to the factory/all topic.
  II. Receives MQTT messages containing JSON-encoded sensor data.
  III. Parses the JSON payload to extract device IDs and sensor values.
  IV. Formats the data according to InfluxDB's expected structure (measurement, tags, fields, timestamp).
  V. Writes the data into the grafana database inside InfluxDB using the Python influxdb client.
  VI. Publishes the device "online" status on a separate MQTT topic (factory/status) when the connection is established.

❖ PROCESS OF WEB SOCKET COMMUNICATION

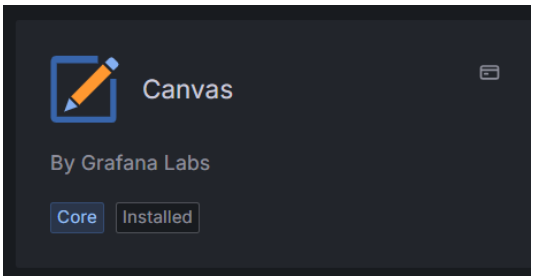
- In this system, WebSockets are used to achieve real-time communication between the MQTT broker and the Grafana HTML panel.
- Specifically, mqtt.js connects via MQTT over WebSockets to subscribe to sensor data topics and update the dashboard instantly when new data arrives.

- WebSocket provides a **continuous two-way communication channel** between the MQTT broker and the Grafana HTML panel.
- When the HTML panel loads inside Grafana, JavaScript (mqtt.js) **creates a WebSocket connection** to the MQTT broker.

- After successful connection, the client **subscribes** to the necessary MQTT topics.
- When a new sensor reading (or device status) is published to factory/all or factory/status, the MQTT broker immediately pushes that message through the WebSocket connection to the browser.

- The browser does not have to request anything, it just receives events as they happen.
- The WebSocket connection **stays open** as long as the dashboard is open.

CANVAS PANEL

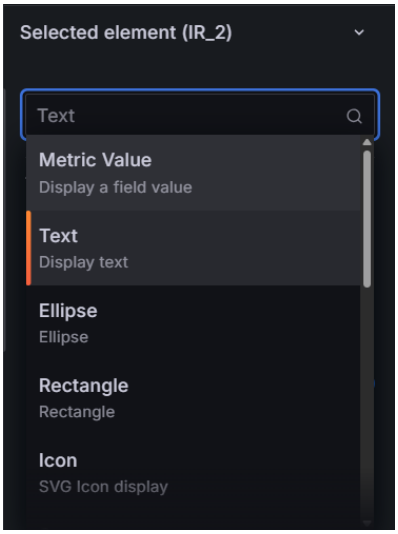


Canvases combine the power of Grafana with the flexibility of custom elements. They are extensible visualizations that allow you to add and arrange elements wherever you want within unstructured static and dynamic layouts. This lets you design custom visualizations and overlay data in ways that aren't possible with standard Grafana visualizations, all within the Grafana UI.

- Elements,

Elements are the basic building blocks of a canvas and they help you visualize data with different shapes and options. There are several types of elements available in canvas panel.

- Text
- Metric value
- Button
- Cloud
- Ellipse
- Rectangle
- Icon
- Server
- Triangle
- Wind turbine
- Drone top
- Drone front
- Drone side
- Parallelogram



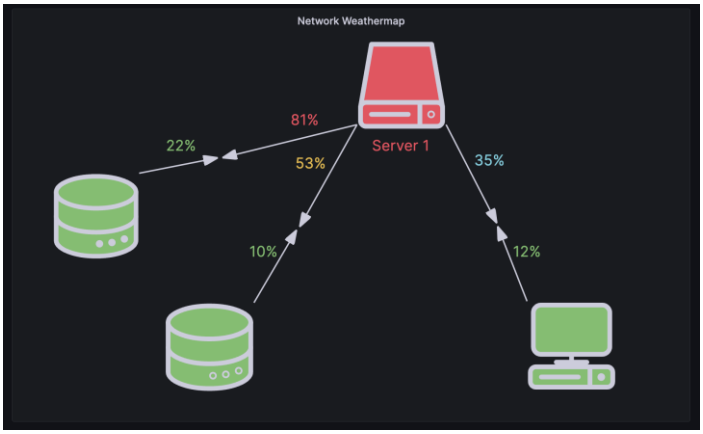
- Basic shape elements like Cloud, Ellipse, Parallelogram, Rectangle and triangle can display text and its background color can be changed based on data thresholds.
- Text element lets you add text to the canvas.
- The metric value element lets you select the data you want to display on a canvas. This element has a unique "edit" mode that can be triggered either through the context menu "Edit" option or by double clicking.
- The icon element lets you add a supported icon to the canvas. Icons can have their color set based on thresholds or value mappings. You can add custom icon by referencing an SVG file.

  Icon > SVG path

  File source > fixed

  URL tab > enter the URL > SELECT

- Server element lets you easily represent a single server, a stack of servers, a database, or a terminal. Server elements support status color, bulb color, and a bulb blink rate all configurable by fixed or field values.



- The button element lets you add a basic button to the canvas. Button elements support triggering basic, unauthenticated API calls. API settings are found in the button element editor. You can also pass template variables in the API editor.

  A button click will only trigger an API call when inline editing is disabled.

  API options:

Option	Description
Endpoint	Enter the endpoint URL.
Method	Choose from **GET**, **POST**, and **PUT**.
Content-Type	Select an option in the drop-down list. Choose from: JSON, Text, JavaScript, HTML, XML, and x-www-form-urlencoded.
Query parameters	Enter as many **Key**, **Value** pairs as you need.
Header parameters	Enter as many **Key**, **Value** pairs as you need.
Payload	Enter the body of the API call.

❖ ESP32 FIRMWARE CODE:

```
#include <WiFi.h>
#include <AsyncMqttClient.h>
#include <ArduinoJson.h>

// WiFi & MQTT Config
#define WIFI_SSID "Gayumi"
#define WIFI_PASSWORD "Gayu1234"
#define MQTT_HOST "broker.emqx.io"
#define MQTT_PORT 1883  // TCP

AsyncMqttClient mqttClient;
TimerHandle_t mqttReconnectTimer; // to manage reconnect logic
TimerHandle_t wifiReconnectTimer;

unsigned long lastPublish = 0; //timestamp of last publish

void connectToWiFi() {
  Serial.print(" Connecting to WiFi...");
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}

void connectToMqtt() {
  Serial.println(" Connecting to MQTT...");
  mqttClient.connect();
}

void onMqttConnect(bool sessionPresent) {  //called when mqtt connects
  Serial.println(" Connected to MQTT broker");

  //  Birth message
  mqttClient.publish("factory/status", 2, true, "online");
}

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) { // call when ESP32
disconnect from mqtt broker
  Serial.print(" Disconnected from MQTT. Reason: ");
  Serial.println(static_cast<int>(reason));
```

```
  xTimerStart(mqttReconnectTimer, 0); // 0 means the timer starts immediately
}

void onWiFiEvent(WiFiEvent_t event) {
  if (event == SYSTEM_EVENT_STA_GOT_IP) {
    Serial.println(" WiFi connected");
    connectToMqtt();
  } else if (event == SYSTEM_EVENT_STA_DISCONNECTED) {
    Serial.println(" WiFi disconnected");
    xTimerStop(mqttReconnectTimer, 0);
    xTimerStart(wifiReconnectTimer, 0);
  }
}

void publishAllDeviceData() {
  StaticJsonDocument<768> doc;

  for (int i = 1; i <= 3; i++) {
    JsonObject dev = doc.createNestedObject("device" + String(i));
    dev["IR_Sensor"] = random(2000, 5000) / 100.0;
    dev["K_Type_Sensor"] = random(1000, 3000) / 100.0;
    dev["Current_Miller"] = random(3000, 6000) / 100.0;
    dev["Current_Mixer"] = random(1500, 4000) / 100.0;
  }

  String payload;
  serializeJson(doc, payload);
  mqttClient.publish("factory/all", 2, true, payload.c_str());

  Serial.println(" Published:");
  Serial.println(payload);
  delay(100);
}

void setup() {
  Serial.begin(115200);
  delay(1000);

  mqttReconnectTimer = xTimerCreate("mqttTimer", pdMS_TO_TICKS(2000), pdFALSE,
NULL,               //Creates a FreeRTOS one-shot timer called "mqttTimer"
```

```
                                    reinterpret_cast<TimerCallbackFunction_t>(connect
ToMqtt));     //Used to automatically retry MQTT connection if it fails by calling
connectToMqtt()
  wifiReconnectTimer = xTimerCreate("wifiTimer", pdMS_TO_TICKS(2000), pdFALSE,
NULL,               //this one is same ,but for wifi
                                    reinterpret_cast<TimerCallbackFunction_t>(connect
ToWiFi));

  WiFi.onEvent(onWiFiEvent); // listen wifi events
  mqttClient.onConnect(onMqttConnect);
  mqttClient.onDisconnect(onMqttDisconnect);

  mqttClient.setServer(MQTT_HOST, MQTT_PORT);
  mqttClient.setClientId("ESP32-MQTT-ALL"); //unique client ID for your ESP32 to
identify itself to the broker.
  mqttClient.setKeepAlive(10);  //If the ESP32 doesn't communicate with the broker in
this time(5), it is considered disconnected.
  mqttClient.setCleanSession(true);  //Tells the broker to not store session data, if
ESP32 disconnects.

  //  Set LWT (broker sends this if ESP32 disconnects ungracefully)
  mqttClient.setWill("factory/status", 2, true, "offline");

  connectToWiFi();
}

void loop() {
  if (WiFi.isConnected() && mqttClient.connected() && millis() - lastPublish > 5000)
{
    lastPublish = millis(); //pdates the lastPublish timestamp to now.
    publishAllDeviceData();
  }
}
```

- Reading sensor values (simulated with random() in this version),
- Formatting the data into JSON,
- Connecting to Wi-Fi and an MQTT broker, and
- Publishing the sensor data periodically to a defined MQTT topic (factory/all), along with device status (factory/status).

❖ ONRENDER TAB CODE:

```
// Load mqtt.js from CDN dynamically
const loadMQTT = () => {
  return new Promise((resolve, reject) => {
    if (window.mqtt) return resolve(window.mqtt);
    const script = document.createElement('script');
    script.src = 'https://unpkg.com/mqtt/dist/mqtt.min.js';
    script.onload = () => resolve(window.mqtt);
    script.onerror = reject;
    document.head.appendChild(script);
  });
};

// Inject HTML content into the Grafana HTML panel
htmlNode.innerHTML = `
  <style>
    .device-container {
      display: flex;
      gap: 50px;
      padding: 30px;
      font-family: sans-serif;
      color: white;
    }

    .device-box {
      background-color: #000000;
      padding: 20px;
      border-radius: 12px;
      min-width: 240px;
      box-shadow: 0 0 12px rgba(0, 255, 255, 0.1);
    }

    .device-box strong {
      color: #009688;
      font-size: 20px;
    }

    .sensor-label {
      font-size: 16px;
      margin: 4px 0;
```

```
    }

    .sensor-value {
      font-weight: bold;
      color: #4fc3f7;
    }

    .status-box {
      font-weight: bold;
      font-size: 18px;
      color: #ffffff;
      margin-bottom: 20px;
    }

    .status-online {
      color: #4caf50;
    }

    .status-offline {
      color: #f44336;
    }
  </style>

  <div class="device-container">
    ${[1, 2, 3].map(i => `
      <div class="device-box" id="device${i}">
        <strong>Device ${i}</strong><br>
        <div class="sensor-label">IR Sensor: <span id="ir${i}" class="sensor-value">-
-</span> °C</div>
        <div class="sensor-label">K Type Sensor: <span id="k${i}" class="sensor-
value">--</span> °C</div>
        <div class="sensor-label">Current Miller: <span id="miller${i}"
class="sensor-value">--</span> °C</div>
        <div class="sensor-label">Current Mixer: <span id="mixer${i}" class="sensor-
value">--</span> °C</div>
        </br><div class="status-box">Status: <span id="status" class="status-
offline">--</span></div>
      </div>
    `).join("")}
  </div>
`;

// Restore previous sensor data
```

```
const cachedSensors = localStorage.getItem('cachedDeviceData');
if (cachedSensors) {
  try {
    const data = JSON.parse(cachedSensors);
    for (let i = 1; i <= 3; i++) {
      const d = data["device" + i];
      if (d) {
        htmlNode.querySelector(`#ir${i}`).textContent = d.IR_Sensor;
        htmlNode.querySelector(`#k${i}`).textContent = d.K_Type_Sensor;
        htmlNode.querySelector(`#miller${i}`).textContent = d.Current_Miller;
        htmlNode.querySelector(`#mixer${i}`).textContent = d.Current_Mixer;
      }
    }
  } catch (e) {
    console.warn('Failed to load cached sensor data:', e);
  }
}

// Restore previous status
const cachedStatus = localStorage.getItem('cachedDeviceStatus');
if (cachedStatus) {
  const statusEl = htmlNode.querySelector('#status');
  const status = cachedStatus.toLowerCase();
  statusEl.textContent = status;
  statusEl.className = status === "online" ? "status-online" : "status-offline";
}

loadMQTT().then(() => {
  const client = mqtt.connect('ws://broker.emqx.io:8083/mqtt', {
    clientId: 'grafana-sync-' + Math.random().toString(16).substr(2, 8),
    clean: true
  });

  client.on('connect', () => {
    console.log('MQTT Connected');
    client.subscribe('factory/all');
    client.subscribe('factory/status');
  });

  client.on('message', (topic, message) => {
    if (topic === 'factory/all') {
      const data = JSON.parse(message.toString());
```

```
      // Cache sensor data
      localStorage.setItem('cachedDeviceData', JSON.stringify(data));

      for (let i = 1; i <= 3; i++) {
        const d = data["device" + i];
        if (d) {
          htmlNode.querySelector(`#ir${i}`).textContent = d.IR_Sensor;
          htmlNode.querySelector(`#k${i}`).textContent = d.K_Type_Sensor;
          htmlNode.querySelector(`#miller${i}`).textContent = d.Current_Miller;
          htmlNode.querySelector(`#mixer${i}`).textContent = d.Current_Mixer;
        }
      }
    }

    if (topic === 'factory/status') {
      const status = message.toString().toLowerCase();

      // Cache status
      localStorage.setItem('cachedDeviceStatus', status);

      const statusEl = htmlNode.querySelector('#status');
      statusEl.textContent = status;
      statusEl.className = status === "online" ? "status-online" : "status-offline";
    }
  });

  client.on('error', err => {
    console.error('MQTT Error:', err);
  });
});
```

- Loads mqtt.js from CDN to enable MQTT in the browser.
- Establishes a WebSocket connection to the MQTT broker (ws://broker.emqx.io:8083/mqtt).
- Subscribes to topics (factory/all, factory/status) to receive real-time sensor and status data.
- Dynamically updates device cards (e.g., IR Sensor, K-Type Sensor) in the dashboard based on live incoming data.
- Caches data in localStorage so it persists across page refreshes

❖ PYTHON SCRIPT FOR WRITE DATA INTO INFLUXDB

```
import json
import paho.mqtt.client as mqtt
from influxdb import InfluxDBClient   #Used to write data to InfluxDB from Python.

# InfluxDB setup
influx = InfluxDBClient(host="influxdb", port=8086)  #use docker container 'influxdb'
influx.switch_database("grafana")   # database name is grafana

# MQTT setup
MQTT_BROKER = "broker.emqx.io"
MQTT_PORT = 1883
MQTT_TOPIC = "factory/all"
STATUS_TOPIC = "factory/status" #online/offline

def on_connect(client, userdata, flags, rc):  #triggered when client connect to mqtt broker
    if rc == 0:  # rc = 0 means connection is successful
        print("Connected to MQTT broker")
        client.publish(STATUS_TOPIC, payload="online", qos=2, retain=True)
        client.subscribe(MQTT_TOPIC, qos=2)
    else:
        print("Failed to connect, return code:", rc)

def on_message(client, userdata, msg): #This function runs every time a message is received
on a subscribed topic
    try:
        payload = json.loads(msg.payload.decode())  #Decodes the message payload (from bytes
to string) and parses the JSON.
        influx_data = []  #empty list to store InfluxDB records.

        for device_key, values in payload.items():
            device_id = device_key.replace("device", "")
            influx_data.append({
                "measurement": "ESP_data",
                "tags": {"device": device_id},
                "fields": {
                    "IR_Sensor": float(values["IR_Sensor"]),
                    "K_Type_Sensor": float(values["K_Type_Sensor"]),
                    "Current_Miller": float(values["Current_Miller"]),
                    "Current_Mixer": float(values["Current_Mixer"])
                }
```

```
            })

        influx.write_points(influx_data) #Sends the list of records to InfluxDB
        print(f"Stored data for devices: {list(payload.keys())}")

    except Exception as e:
        print("Error processing MQTT message:", e)

def main():
    client = mqtt.Client()
    client.on_connect = on_connect
    client.on_message = on_message

    client.connect(MQTT_BROKER, MQTT_PORT, keepalive=60)
    client.loop_forever()

if __name__ == "__main__": #Runs main() only if this file is executed directly (not imported
as a module).
    main()
```

- Connects to the MQTT broker and subscribes to factory/all.
- On receiving a message, it:
    - Parses the JSON payload (e.g., device1: {...}).
    - Formats the data into InfluxDB's required structure.
    - Writes it into the ESP_data measurement in the grafana database using the influxdb client.
- Publishes an online status message to factory/status after successful connection.