Creating Dashboards Using Grafana

ABSTRACT:

- Generally in monitoring the data visualization plays a major role.
- The main goal of this document is to get complete information regarding the exposure of the dashboards from the data that will be extracted.
- The problem in this is finding the errors that occurred during the process and finding it is so difficult but that is for a minor reason due to the complexity of handling many servers at a time.
- So, this helps to solve those major errors which are for minor reasons.

AGENDA

INTRODUCTION

- **Grafana** is a visualization tool that is used to make some good visual representations through which the data is collected from different data sources.
- In the present documentation we used the **InfluxDB** data source which is a database that stores and shares the data in a time series.
- The data that is generated is industrial-based and is stored according to the protocols and the data is mostly generated and stored in the data logs.
- So, the data gonna extracted from the platform which is stored in a log is the **KEPserverEX** based upon the different protocols and it is a Connectivity platform that just connects to the servers to extract the data.
- In this project, data follows the MQTT protocol and based upon that, data transfer takes place to the InfluxDB.
- To share the data InfluxDB has its in-built agent i.e., **Telegraf** which is a server between two clients.

STEPS FROM EXTRACTION TO VISUALIZATION

There are various steps in the process of creating dashboards from the extraction of the data but the major things that are included in the process are:

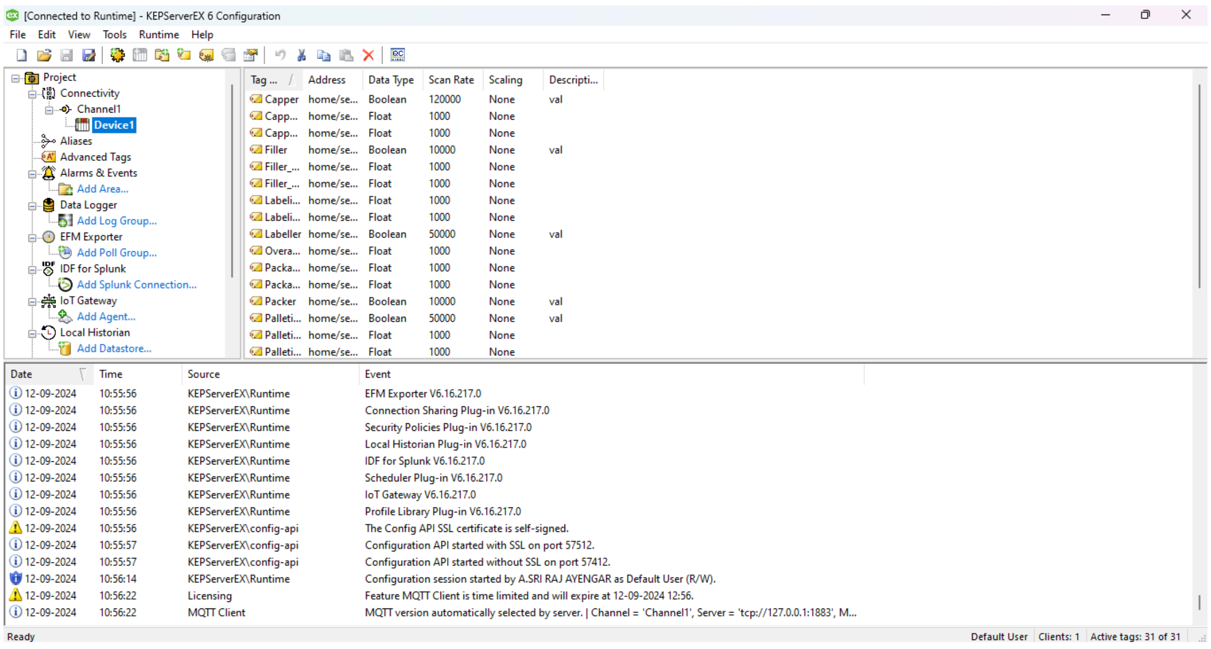
- Data Extraction
- Data Configuration through Telegraf
- Setting up the data resources in Grafana
- Steps to create Visualization

EXTRACTION OF DATA

- Before going to the extraction of data install the KepserverEX using this link.
- The data stored in the KEPserverEX follow different protocols based upon which the project files or the data logs follow.
- Extraction of data conveys how the data gets extracted i.e., extraction of the protocol that the client of Data follows.
- To know on which protocol the data is stored, make sure to check the properties of the logs or devices which gives the type of protocol that it follows.
- In this project, a Python JSON payload "Json_Payload.py" is used for the generation of tags through MQTT protocol, so that extraction of data can be done for further steps.

- **Steps to check whether the data is generated in KepserverEX through the Quick Client:**

1. Before executing the Python code install the JRE server through the Link and start the server.
2. Open the KeprserverEX file which is in the format of ".opf" i.e., "Cococola.opf" file, and navigate to the tags present in the devices.



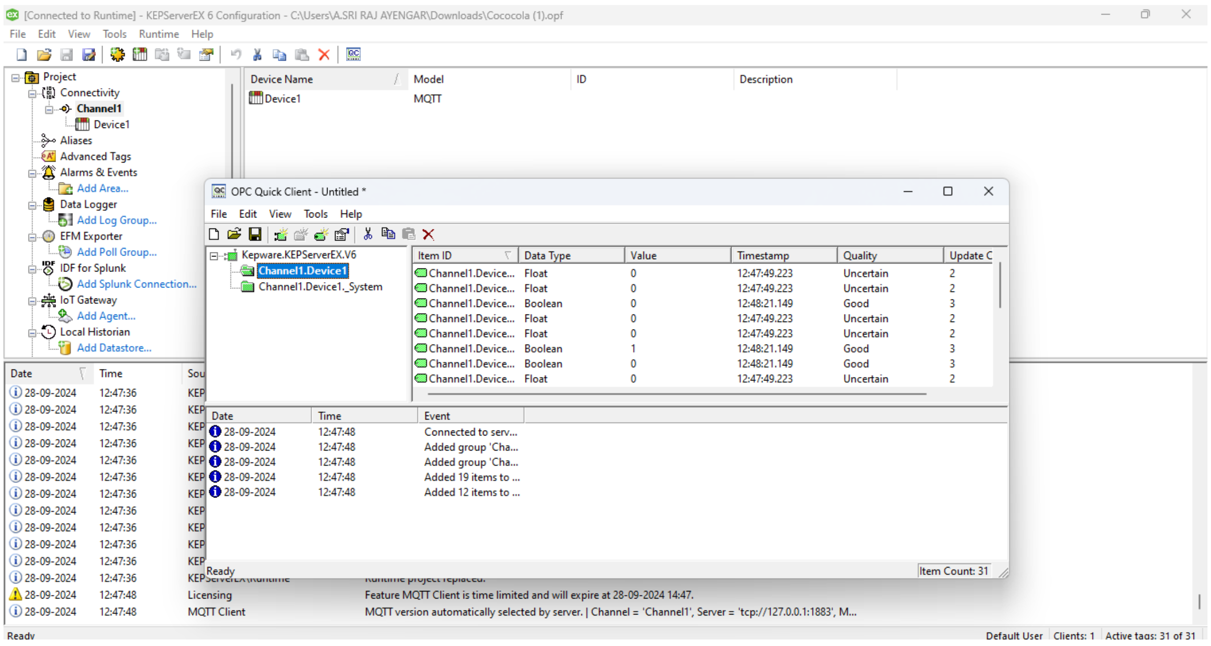
Tag ...	Address	Data Type	Scan Rate	Scaling	Descripti...
Capper	home/se...	Boolean	120000	None	val
Capp...	home/se...	Float	1000	None	
Capp...	home/se...	Float	1000	None	
Filler	home/se...	Boolean	10000	None	val
Filler_...	home/se...	Float	1000	None	
Filler_...	home/se...	Float	1000	None	
Labeli...	home/se...	Float	1000	None	
Labeli...	home/se...	Float	1000	None	
Labeller	home/se...	Boolean	50000	None	val
Overa...	home/se...	Float	1000	None	
Packa...	home/se...	Float	1000	None	
Packa...	home/se...	Float	1000	None	
Packer	home/se...	Boolean	10000	None	val
Palleti...	home/se...	Boolean	50000	None	val
Palleti...	home/se...	Float	1000	None	
Palleti...	home/se...	Float	1000	None	

```
Date        Time      Source                    Event
12-09-2024  10:55:56  KEPServerEX\Runtime       EFM Exporter V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Connection Sharing Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Security Policies Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Local Historian Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       IDF for Splunk V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Scheduler Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       IoT Gateway V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Profile Library Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\config-api    The Config API SSL certificate is self-signed.
12-09-2024  10:55:57  KEPServerEX\config-api    Configuration API started with SSL on port 57512.
12-09-2024  10:55:57  KEPServerEX\config-api    Configuration API started without SSL on port 57412.
12-09-2024  10:56:14  KEPServerEX\Runtime       Configuration session started by A.SRI RAJ AYENGAR as Default User (R/W).
12-09-2024  10:56:22  Licensing                 Feature MQTT Client is time limited and will expire at 12-09-2024 12:56.
12-09-2024  10:56:22  MQTT Client               MQTT version automatically selected by server. | Channel = 'Channel1', Server = 'tcp://127.0.0.1:1883', M...
```

Ready   Default User | Clients: 1 | Active tags: 31 of 31

Note: Install MQTT using this link. For the unlicensed KEPserverEX, the license of it for MQTT is 2 hours. So, to get 2 hours of license modify or reinstall the KEPserverEX again and again.

3. Right-click on the device i.e., “Device1” and click on the “Launch Quick Client Here” option, a new folder is opened displaying the tag values.
4. Now open the Python file or data generating file i.e., “Json_Payload.py” and run the code in any Python code builder open the previous OPC Quick Client folder, and check whether the values are updated from previous values.



Device Name	Model	ID	Description
Device1	MQTT		

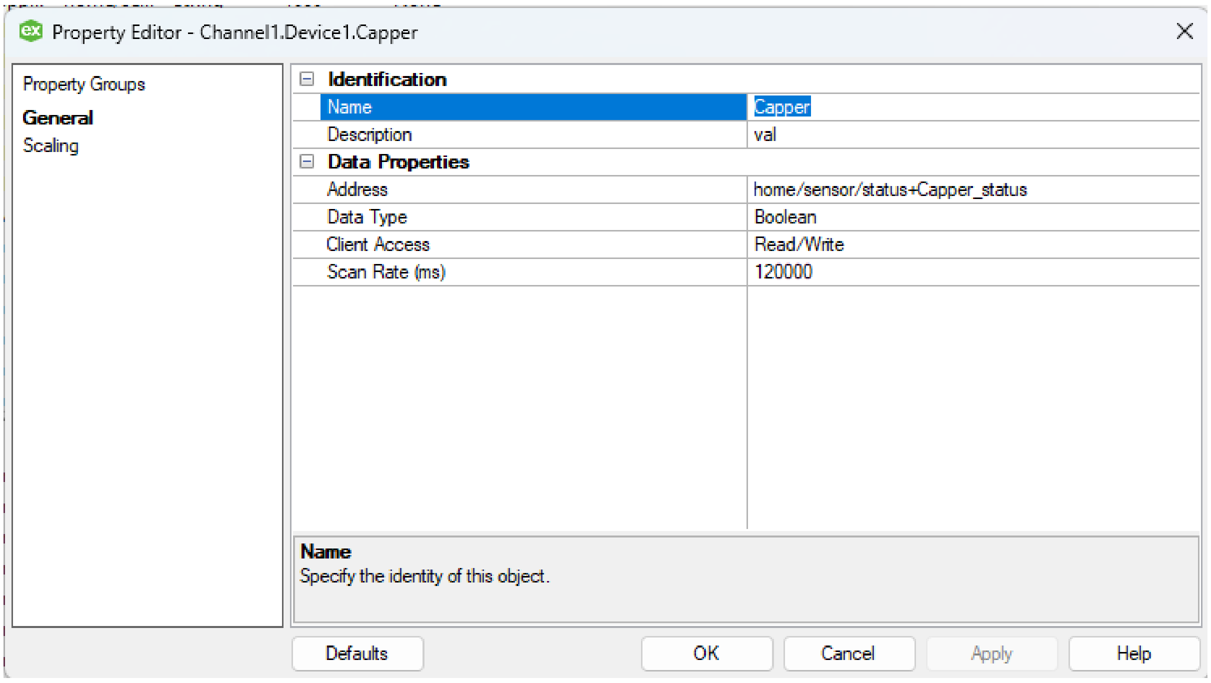
Item ID	Data Type	Value	Timestamp	Quality	Update C
Channel1.Device...	Float	0	12:47:49.223	Uncertain	2
Channel1.Device...	Float	0	12:47:49.223	Uncertain	2
Channel1.Device...	Boolean	0	12:48:21.149	Good	3
Channel1.Device...	Float	0	12:47:49.223	Uncertain	2
Channel1.Device...	Float	0	12:47:49.223	Uncertain	2
Channel1.Device...	Boolean	1	12:48:21.149	Good	3
Channel1.Device...	Boolean	0	12:48:21.149	Good	3
Channel1.Device...	Float	0	12:47:49.223	Uncertain	2

Date	Time	Event
28-09-2024	12:47:48	Connected to serv...
28-09-2024	12:47:48	Added group 'Cha...
28-09-2024	12:47:48	Added group 'Cha...
28-09-2024	12:47:48	Added 19 items to ...
28-09-2024	12:47:48	Added 12 items to ...

```
Date        Time      Sou
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEP
28-09-2024  12:47:36  KEPServerEX\Runtime   Runtime project replaced.
28-09-2024  12:47:48  Licensing             Feature MQTT Client is time limited and will expire at 28-09-2024 14:47.
28-09-2024  12:47:48  MQTT Client           MQTT version automatically selected by server. | Channel = 'Channel1', Server = 'tcp://127.0.0.1:1883', M...
```

Note: After Checking in the OPC Quick Client folder, close the OPC file stop the Python code that is running in the Python code builder, and close the code builder application.

- Each tag will store a single information which changes dynamically according to time.
- The topic is generally found in the address of the tag in the format of “topic+tag_name” and here “tag_name” gives which information that the tag generates for the MQTT protocol.


Property Editor - Channel1.Device1.Capper

Property Groups		
General	**Identification**	
Scaling	Name	Capper
	Description	val
	Data Properties	
	Address	home/sensor/status+Capper_status
	Data Type	Boolean
	Client Access	Read/Write
	Scan Rate (ms)	120000

Name
Specify the identity of this object.

Note: Whatever the data that reads or writes is according to the scan rate of the ordinal generated data and that is stored in properties of the tag of the KepserverEX in the projects. It is necessary to note the topic which is present in the address of the tag.

SET UP CONFIGURATION

- To set up configuration with the InfluxDB Database, there is an inbuilt agent named Telegraf.
- Install Telegraf through this link for the 2. x version and InfluxDB through this link for version 2. x.
- To do a general configuration with InfluxDB basic prerequisites of input and output are needed to be given in the Telegraf configuration file.

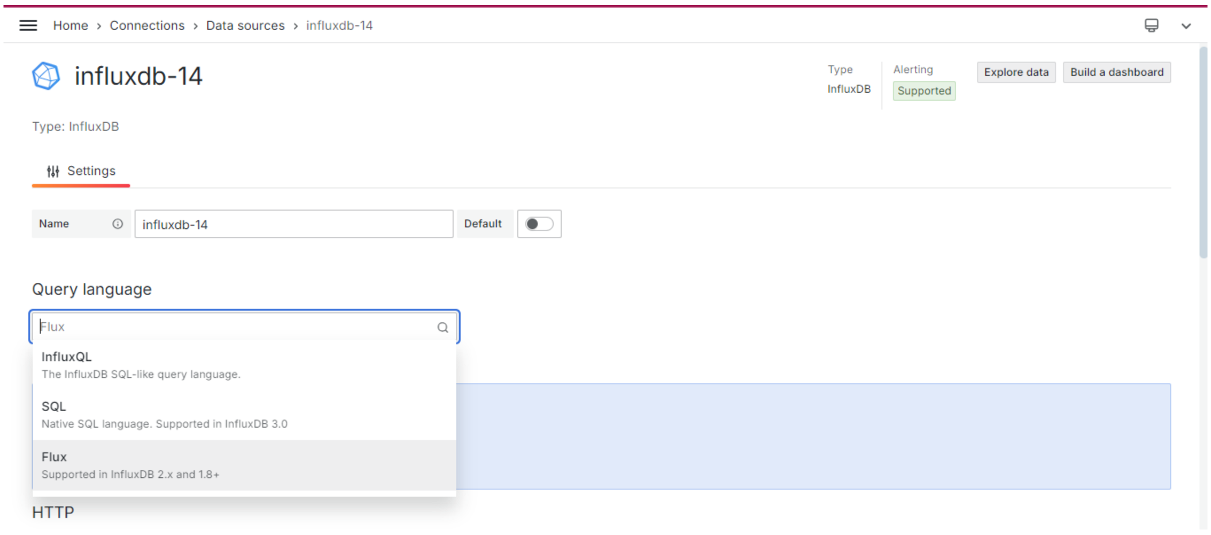
- **Steps to Configure InfluxDB through Telegraf:**
- Before going to Telegraf, open the command prompt, navigate to the influx execution file, execute it through the command "influxd" and don't close or stop the command window.
- Open the official website of InfluxDB through the URL "http://localhost:8086" and create a bucket e.g., "Data" and create a token under the option "API Tokens" copy the token, and paste it somewhere.
- Open the Notepad as Administrator and navigate to the telegraf.conf file.
- Copy the below text and paste it under the "OUTPUT PLUGINS".
- Now replace the bucket value with the before created value e.g., "Data" and replace the token with before copied token.
- Copy and paste the organization details from influxdb official URL.
- Paste the topic that is copied from the tag properties of KepserverEX to the topics of "[[inputs.mqtt_consumer]]".
- Just make sure to check whether any values are String/Boolean through the tag properties or Quick OPC Client of that device so that paste their tag names which are provided in the address of the tags in the "json_string_fields" of "[[inputs.mqtt_consumer]]" in telegraf.conf file.

```
# Output Plugin for InfluxDB
[[outputs.influxdb_v2]]
  urls = ["http://localhost:8086"]
  token = "DVtiVRI84dxTawjXsufQPv2ngiwTbO6BHpIy7FTcBpOHJ-
Fhf7VBJP3NZ2Y2wtAFVHqflJ0dDOEWGcNdlXvUIQ=="  # Replace with your actual token
  organization = "tech"  # Replace with your actual organization
  bucket = "try9"  # Replace with your actual bucket

# Read metrics from MQTT topic(s)
[[inputs.mqtt_consumer]]
  ## Broker URLs for the MQTT server or cluster.  To connect to multiple
  ## clusters or standalone servers, use a separate plugin instance.
  ##   example: servers = ["tcp://localhost:1883"]
  ##            servers = ["ssl://localhost:1883"]
  ##            servers = ["ws://localhost:1883"]
  servers = ["tcp://127.0.0.1:1883"]

  ## Topics that will be subscribed to.
  topics = [
    "home/sensor/status"
  ]
  data_format = "json"

  json_string_fields = [
    "Capper_status",
    "Filler_status",
    "Labeling_status",
    "Packaging_status",
    "Palletizing_status",
    "Wrapping_status"
  ]
```

- Save the telegraf file open the command prompt and navigate to the telegraf.conf file and execute the file through the command "telegraf --config telegraf.conf --test" or for powershell ".\telegraf.exe --config telegraf.conf".
- Check in the previously run command prompt of InfluxDB as it shows some updates.
- Now in the Chrome Browser close the previously opened 8086 containing URL(InfluxDB) and again open the same URL and open the respective bucket in Data Explorer.
- Now open the data generation python file which is used to send the data to MQTT in this project and run the program and minimize the tab(don't close it).
- Again some more updates take place in the command prompt of influx execution.
- Again close and open the tab of 8086 containing URL of Chrome Browser tab and check the same bucket in the Data Explorer, mqtt_consumer named measurement is appeared which means configuration part is completed where tags values are stored in InfluxDB Database.
- Now the data is stored in the InfluxDB database as a time-series.

SET UP WITH DATA SOURCES IN GRAFANA

- Install Grafana using this link and create an account through the provided link.
- Generally data-sources are used to extract the data used after used to visualize it.
- There are different data sources provided in Grafana to collect the data.
- Open Grafana through the link "http://localhost:3000".
- In this project, the InfluxDB data source is added, and to add that data source, it is necessary to add the InfluxDB data source based on the version installed.
- In this project, InfluxDB version 2. x is used, and the type of influx query builder is selected accordingly with the version of InfluxDB. Therefore, in this type, it is necessary to give prerequisites of bucket and organization names along with the token and URL of the InfluxDB that were previously used to store the data of KEPserverEX.


Home › Connections › Data sources › influxdb-14

influxdb-14

Type: InfluxDB

Type: InfluxDB

Alerting: Supported

Explore data | Build a dashboard

Settings

Name: influxdb-14

Default

Query language: Flux

InfluxQL
The InfluxDB SQL-like query language.

SQL
Native SQL language. Supported in InfluxDB 3.0

Flux
Supported in InfluxDB 2.x and 1.8+

HTTP

- In this way the InfluxDB data source is added to the Grafana.

STEPS TO CREATE A VISUALIZATION

- Visualization is a representation of the data in different forms.
- The set of visualizations as panels is known as the dashboard.
- In this project, 2 Dashboards and one is 11 visualizations are used and for the other 5 visualizations are used.
- Coming to the extraction of required visualization firstly it is required to create a dashboard and in that create the panel and add up the data source.
- In this project, for the first Dashboard 11 panels are created and accordingly the 11 panels are from the same data source of InfluxDB, and for the second Dashboard 5 panels respectively.
- Coming to the extraction of the data from the source to the visual representation, it is necessary to give the prerequisites to the query section the query section has three types that are given before while adding the query type in adding the data source and the flux query is used in this project.
- In this project for the Cococola Dashboard, 11 visualizations are used and the flux codes of them are:

Filler Machine Status code:

```
from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "Filler_status")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
  |> yield(name: "last")
```

Capper Machine Status code:

```
from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "Capper_status")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
  |> yield(name: "last")
```

Labelling Machine Status code:

```
from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "Labeling_status")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
  |> yield(name: "last")
```

Palletizer Machine Status code:

```
from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "Palletizing_status")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
  |> yield(name: "last")
```

Packaging Machine Status code:

```
from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "Packaging_status")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
  |> yield(name: "last")
```

Wrapper Machine Status code:

```
from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "Wrapping_status")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
  |> yield(name: "last")
```

EXPLANATION:

- "from(bucket: "hst1")" specifies the data source (InfluxDB bucket) from which pulling the data. In this case, the bucket is called "hst1", which contains the MQTT data that has been sent through KepServerEX and Telegraf into InfluxDB.
- "|> range(start: v.timeRangeStart, stop: v.timeRangeStop)" defines the time range for the data query. In Grafana, v.timeRangeStart and v.timeRangeStop are dynamic variables that automatically adjust to the time range selected in the dashboard. For example, if you choose the last 24 hours in Grafana, this range will adjust to only fetch data within that time window.
- "|> filter(fn: (r) => r._measurement == "mqtt_consumer")" ensures that only pulling records from the measurement named "mqtt_consumer". Measurements in InfluxDB are like tables in traditional databases, so this narrows the data down to only the relevant MQTT data related to machine statuses.
- "|> filter(fn: (r) => r._field == "Filler_status") (or other machine-specific fields like Capper_status, Labeling_status, etc.)" specifies which machine's status want to extract. The _field corresponds to the status field (e.g., Filler_status, Capper_status, etc.) in your data. By using different fields, you can extract status information for specific machines.
- "|> map(fn: (r) => ({ r with _value: if r._value == true then "On" else "Off" }))" function allows to transform the data. In this case, it checks the value of the _value field, which is either true or false. This field represents the machine's status:
- If _value == true, the machine is "On".
- If _value == false, the machine is "Off".
- The map() function changes the boolean values (true or false) into human-readable strings ("On" or "Off") for easier interpretation in Grafana.
- "|> yield(name: "filler_status") (or other machine-specific yield names)" function outputs the result of the query. The optional name parameter allows to label of the output. For example, "filler_status" labels the query result as the status of the Filler machine.
- Each query will have a different yield name (capper_status, labeling_status, etc.), so it can easily identify which status belongs to which machine when setting up Grafana panels.

OEE:

Simplified OEE calculation based on failures and total operating time.

FLUX CODE:

```
// Aggregate total operating time
total_operating_time = from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] =~ /(.+)_operating_time/)
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "total_operating_time"})

// Aggregate failures
failures = from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] =~ /(.*)_failure_count/)
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "failures"})

// Aggregate overall scrap count
scrap_count = from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "Overall_scrap_count")
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "overall_scrap_count"})

// Combine aggregated results into a single table
combined = union(tables: [total_operating_time, failures, scrap_count])
  |> group(columns: [])
  |> reduce(
    fn: (r, accumulator) => ({
      total_operating_time: accumulator.total_operating_time + (if exists
r.total_operating_time then float(v: r.total_operating_time) else 0.0),
      failures: accumulator.failures + (if exists r.failures then float(v:
r.failures) else 0.0),
      overall_scrap_count: accumulator.overall_scrap_count + (if exists
r.overall_scrap_count then float(v: r.overall_scrap_count) else 0.0)
    }),
    identity: {total_operating_time: 0.0, failures: 0.0,
overall_scrap_count: 0.0}
```

```
  )

// Calculate OEE
oee = combined
  |> map(fn: (r) => ({
      _time: now(),
      _value: if r.total_operating_time == 0.0 then 0.0 else (1.0 -
(r.failures / r.total_operating_time)) * 100.0  // Simplified OEE formula
  }))
  |> rename(columns: {_value: "OEE"})
  |> yield(name: "OEE")
```

EXPLANATION:

- "filter(fn: (r) => r["_field"] =~ /(.+)_operating_time/)" filters fields that end with _operating_time, which represent the operating times of different machines (e.g., Capper_operating_time, Filler_operating_time).
- "sum(column: "_value")" sums up all the operating times across the different machines.
- "group(columns: [])" removes any grouping, meaning the aggregation will occur across all machines.
- "rename(columns: {_value: "total_operating_time"})" renames the result to make it more readable as total_operating_time.

MTBF:

The mean time between failures calculation.

FLUX CODE:

```
// Aggregate total operating time
total_operating_time = from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] =~ /(.+)_operating_time/)
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "total_operating_time"})

// Aggregate failures
failures = from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] =~ /(.*)_failure_count/)
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "failures"})

// Combine aggregated results into a single table
combined = union(tables: [total_operating_time, failures])
  |> group(columns: [])
  |> reduce(
    fn: (r, accumulator) => ({  // Changed the order of arguments
      total_operating_time: accumulator.total_operating_time + (if exists
r.total_operating_time then float(v: r.total_operating_time) else 0.0),
      failures: accumulator.failures + (if exists r.failures then float(v:
r.failures) else 0.0)
    }),
    identity: {total_operating_time: 0.0, failures: 0.0}
  )

// Calculate MTBF from aggregated values
mtbf = combined
  |> map(fn: (r) => ({
      _time: now(),
      _value: if r.failures == 0.0 then 0.0 else r.total_operating_time /
r.failures
  }))
  |> yield(name: "MTBF")
```

EXPLANATION:

- "filter(fn: (r) => r["_field"] =~ /(.*)_failure_count/)" filters for fields ending with _failure_count to gather the failure counts for each machine.
- "sum(column: "_value")" sums all failure counts across all machines.
- "group(columns: [])" aggregates the results across all machines.
- "rename(columns: {_value: "failures"})" renames the sum to failures.

Scrap Count:

The sum of the overall scrap count.

FLUX CODE:

```
from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) =>
    r["_field"] == "Capper_failure_count" or
    r["_field"] == "Filler_failure_count" or
    r["_field"] == "Labeling_failure_count" or
    r["_field"] == "Packaging_failure_count" or
    r["_field"] == "Palletizing_failure_count" or
    r["_field"] == "Wrapping_failure_count")
  |> group()
  |> sum(column: "_value")
  |> rename(columns: {_value: "Total_scrap_count"})  // Renaming for better
readability
```

EXPLANATION:

- "filter(fn: (r) => r["_field"] == "Overall_scrap_count")" filters for the field representing the overall scrap count.
- "sum(column: "_value")" sums up the total scrap count.
- "group(columns: [])" aggregates the scrap count across all machines.
- "rename(columns: {_value: "overall_scrap_count"})" renames the column to make it clear it represents the overall_scrap_count.

Machine Run time:

FLUX CODE:

```
from(bucket: "th")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Dynamically
adjust the time range based on dashboard
  |> filter(fn: (r) => r._measurement == "mqtt_consumer")
  |> filter(fn: (r) => r._field =~
/^(Capper_status|Filler_status|Labeling_status|Packaging_status|Palletizing_
status|Wrapping_status)$/)  // Dynamically filter for the six fields
  |> filter(fn: (r) => r._value == true)  // Only consider true (on) values
  |> group(columns: ["_field"])  // Group by the field name to separate
machines' runtime
  |> count(column: "_value")  // Count occurrences of true values (ticks
representing runtime)
  |> map(fn: (r) => ({ r with _value: r._value * 30 }))  // Convert the
counts to minutes (assuming 30s intervals)
  |> keep(columns: ["_field", "_value"])  // Keep only _field (machine
status) and _value (runtime)
  |> yield(name: "runtime_per_machine")
```

EXPLANATION:

- "filter()" function is used to filter records based on the field names. In this case, it filters for failure counts of specific machines "Capper_failure_count", "Filler_failure_count","Labeling_failure_count", "Packaging_failure_count", "Palletizing_failure_count", "Wrapping_failure_count".
- These fields correspond to the failure counts for each machine. The or operators ensure that only records where the field matches one of these specific machine failure counts are kept.
- "group()" groups the data by the default behavior (i.e., based on the measurement and the tag columns). In this context, it's grouping the results across all machines.
- "sum(column: "_value")" sums the values in the _value column, which represents the failure counts for each machine. The result will give the total number of failures across all machines during the specified time range.
- "rename(columns: {_value: "Total_scrap_count"})" renames the summed failure count result to Total_scrap_count to make it more readable in the output.

Status and Work Order Quality:

FLUX CODE:

```
from(bucket: "try9")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")  // Adjust
measurement name if needed
  |> limit(n: 10)
  |> yield(name: "sample_data")
```

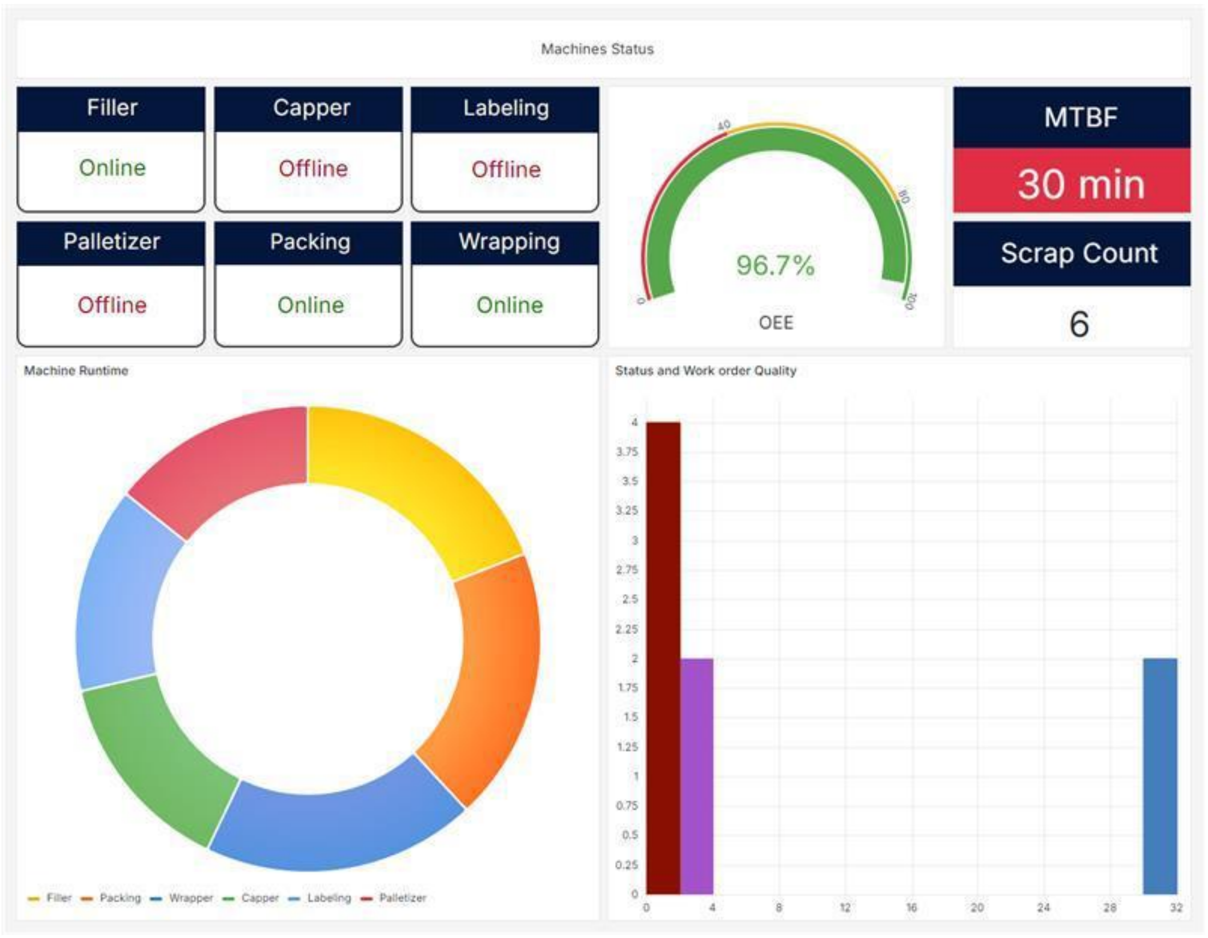
EXPLANATION:

- "limit(n: 10)" function limits the result set to 10 records. It is useful when only want to see a sample or a small subset of data.

Panel settings:

- To name the title for the panel go to panel settings which are present at the right end and create a panel title.
- In the case of giving status in the first panel (Status of the machine) just make the search for the option mapping after scrolling the panel options at the right end and just map the values like in this case value “true” is “Online” and “false” is “Offline”.
- While going to edit the panel just select the visualization on which it is going to be prepared which is present on the right top.
- Now to change the color of the value or to change the properties of their own wish just select the Overrides which are also present in the panel settings and just select the query type of which property is to be changed.
- After all the things it is mandatory to save them as an overall dashboard.

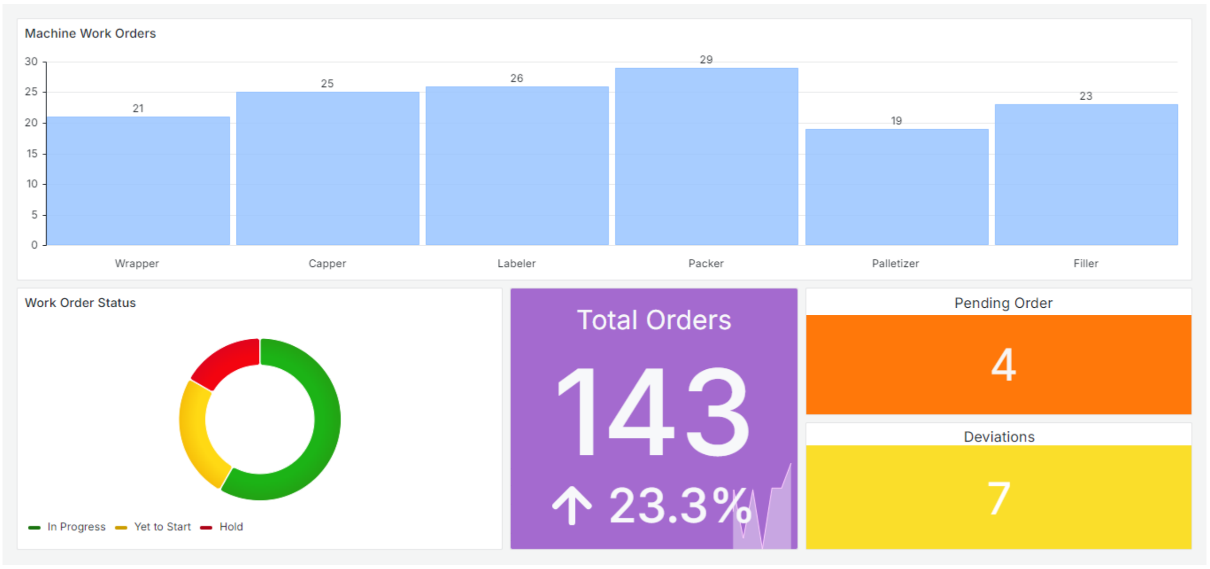
After all the codes are executed the final dashboard will be:



Note: Make sure to check whether the Telegraf server is running or to execute the .exe file of telegraf if the values are to be updated in the Grafana at present. To get past the dashboard and not to update the values then just run influxdb execution file and don't run the telegraf execution file.

- In the above Dashboard OEE and MTBF vary the colors with the threshold values set in that panel settings.

In a similar way another dashboard is created:



Note: Change the buckets in the bucket side heading of flux codes.

Flux Codes :

- Machine Work Orders:

```
union(
  tables: [
    // Filler machine
    from(bucket: "tr1")
      |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
      |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
      |> filter(fn: (r) => r["_field"] == "filler_work_order")
      |> last()
      |> map(fn: (r) => ({ _time: r._time, _value: r._value, machine_name:
"Filler" })),

    // Capper machine
    from(bucket: "tr1")
      |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
      |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
      |> filter(fn: (r) => r["_field"] == "capper_work_order")
      |> last()
      |> map(fn: (r) => ({ _time: r._time, _value: r._value, machine_name:
"Capper" })),

    // Labeler machine
    from(bucket: "tr1")
      |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
      |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
      |> filter(fn: (r) => r["_field"] == "labeler_work_order")
      |> last()
      |> map(fn: (r) => ({ _time: r._time, _value: r._value, machine_name:
"Labeler" })),

    // Palletizer machine
    from(bucket: "tr1")
      |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
      |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
      |> filter(fn: (r) => r["_field"] == "palletizer_work_order")
      |> last()
      |> map(fn: (r) => ({ _time: r._time, _value: r._value, machine_name:
"Palletizer" })),

    // Packer machine
    from(bucket: "tr1")
      |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
      |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
      |> filter(fn: (r) => r["_field"] == "packer_work_order")
      |> last()
      |> map(fn: (r) => ({ _time: r._time, _value: r._value, machine_name:
"Packer" })),

    // Wrapper machine
    from(bucket: "tr1")
      |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
      |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
      |> filter(fn: (r) => r["_field"] == "wrapper_work_order")
      |> last()
      |> map(fn: (r) => ({ _time: r._time, _value: r._value, machine_name:
"Wrapper" }))
  ]
)
|> yield()
```

- Total Orders:

```
from(bucket: "tr1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "total_orders_today")
  |> yield(name: "last")
```

- Pending Order:

```
from(bucket: "tr1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "pending_orders")
  |> yield(name: "last")
```

- Deviations:

```
from(bucket: "tr1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "deviations")
  |> yield(name: "last")
```

- Work Order Status:

```
from(bucket: "tr1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Dynamically
adjust the time range based on dashboard
  |> filter(fn: (r) => r._measurement == "mqtt_consumer")
  |> filter(fn: (r) => r._field =~
/^(InProgress_work_order_count|YetToStart_work_order_count|OnHold_work_order
_count)$/)  // Only consider true (on) values
  |> group(columns: ["_field"])     // Convert the counts to minutes
(assuming 30s intervals)
  |> keep(columns: ["_field", "_value"])  // Keep only _field (machine
status) and _value (runtime)
  |> yield(name: "runtime_per_machine")
```

Note: To create multiple dashboards in Grafana make sure to create different buckets for different dashboards and tokens might be the same in InfluxDB so accordingly update to the telegraf.conf file.

CONCLUSION:

In this documentation, we have successfully demonstrated the process of sending data from KepserverEX to Grafana for visualization.

The key steps involved were:

- **Data Generation and MQTT Integration:** We used Python code to simulate data and publish it to an MQTT server. This data was received by KepserverEX, where we configured tags with specific addresses representing the incoming MQTT data.
- **Telegraf Configuration for InfluxDB:** The tag addresses from KepserverEX were used to configure Telegraf, the agent responsible for collecting data and sending it to InfluxDB. This was achieved by editing the Telegraf .conf file, and adjusting both input (MQTT data) and output (InfluxDB) plugins to match the necessary configuration.
- **Data Visualization in Grafana:** With InfluxDB as the time-series database, the data source was configured in Grafana. The data was then visualized through dashboards, enabling real-time monitoring of the data generated from KepserverEX and transferred through the complete pipeline.

This setup provides a robust and scalable solution for industrial data monitoring and visualization. The combination of MQTT, KepserverEX, Telegraf, InfluxDB, and Grafana ensures smooth data flow from the source (Python-simulated data) to Grafana's dashboards, offering powerful insights into your system’s performance and operational metrics.

RESULT:

Got assumed dashboard successfully after following all the previous notes from extraction of the data to the creating dashboard. This pipeline demonstrates an effective method for real-time data monitoring and visualization, combining various tools for seamless industrial data integration and analysis. The setup is fully functional and capable of handling large-scale data flows from MQTT to Grafana, providing actionable insights through dynamic visualizations.