# Building and visualizing a NoSQL time series data store using InfluxDB, Python and Grafana

# Introduction

Big Data, IoT, and analytics, and as a result, new methods of storing this fluid data have emerged. Almost all streaming data, particularly IoT data, but also real-time analytics and server and application monitoring, has a time stamp and thus is timeseries data. This project will go on to demonstrate the differences between traditional databases and NoSQL (time series) databases, why they are used, and how to build a project on top of them. The completed project stores, visualizes, and analyzes data using InfluxDB2, Python, and Grafana using time-stamped CSV data containing bird migration data as a time analytics approach.

# Basics

## NoSQL basics

NoSQL is a different approach to database design that excludes traditional relational database management systems. While relational database systems attempt to abstract the underlying data structure, NoSQL databases are more data-specific, which should make them more efficient for this specific data. They are intended to implement data structures in a manner that is more closely aligned with the target system, whereas traditional relational databases cannot be structured in this sort of way. The motivation for using NoSQL databases is that they do not require the use of a predefined schema, resulting in a simpler design, horizontal scaling, and greater control over data availability. Because NoSQL distributes its data, it must adhere to the CAP theorem, which states that only two of the three following attributes can be achieved: consistency, availability, and partition tolerance. Depending on the application to implement and the needs of the use cases, different databases prioritize consistency over availability, while others prioritize availability over consistency.

CAP-Theorem https://cloudxlab.com/assessment/displayslide/345/nosql-cap-theorem#:~:text=NoSQL%20can%20not%20provide%20consistency,Consistency%2C%20Availability%20and%20Partition%20Tolerance.

Brad Dayley: Sams Teach Yourself NoSQL with MongoDB in 24 Hours, Video Enhanced Edition

Kasun Idrasiri: Design Patterns for Cloud Native Applications

## NoSQL vs relational Databases

NoSQL and relational databases both have the same basic goals: to store and retrieve data, as well as to coordinate changes. The distinction is that NoSQL databases abandon some of the capabilities of relational databases in order to increase scalability. NoSQL databases, in particular, typically have much simpler coordination capabilities than traditional relational systems, sometimes even none at all). NoSQL databases typically remove all or most of the SQL query language, as well as a complex optimizer required for SQL to be useful. The benefit of the tradeoff is that NoSQL databases are often very simple and can handle unstructured data, resulting in higher scalability in the best case. This has the disadvantage of losing overview by storing a large amount of unstructured and unformatted data, and optimization is in the hands of the developr rather than the optimizer when using relational databases. source: Time Series Databases: New Ways to Store and Access Data

## TimeSeries databases

A time series is a collection of values that are organized by time. For example, stock prices may fluctuate throughout the day as trades are executed, or a weather sensor may take atmospheric temperatures every minute. Time-series data are any events that are recorded over time, whether on a regular or sporadic basis. A time-series database is designed to facilitate the retrieval and statistical analysis of time-series data. While time-series data such as orders, shipments, logs, and so on have long been stored in relational databases, the data sizes and volumes were frequently insignificant. New special-purpose databases were required as data grew faster and larger. Time-series databases, among other things, address the needs of growing, high-velocity data volumes from IoT, event and application logs, ad tech, and fintech. These workloads are frequently write-intensive. As a result, memory buffering is frequently used in time-series databases to support fast writes and reads. Looking at the CAP Theorem, InfluxDB mostly focuses on

https://cloudxlab.com/assessment/displayslide/345/nosql-cap-theorem#:~:text=NoSQL%20can%20not%20provide%20consistency,Consistency%2C%20Availability%20and%20Partition%20Tolerance. source: Fundamentals of Data Engineering: Plan and Build Robust Data Systems von Joe Reis (Author), Matt Housley (Author), (June 2022) O'REILLYs

## Why use a time series data

Time series date often needs to focus on fast

Paul Dix, "Why Build a Time Series Data Platform?", 20.07.2017 https://db-engines.com/en/blog_post/71

## InfluxDB basics

According to https://db-engines.com/, InfluxDB is the most popular NoSQL time series data created by InfluxData. Its primary application is handling massive amounts of time-stamped data. Collecting IoT data is a common example because every data set in IoT is time-based. However, it is frequently used for Analytics as well as IoT data, for example, in this project time-stamped data for bird migration will be handled using InfluxDB2. The first version of InfluxDB came out in 2013 and the 1.x version is still commonly used. However, in 2020 InfluxDB was released as a stable version and is the way to go. It features a new query language, which will be explained in the next section, as well as a standalone binary with a graphical web interface to explore data.

Flux Query Language

## InfluxDB vs Timescale vs Prometheus

# Tutorial

## Goals

The purpose of this project is to process a large amount of time series data. The data is stored using a NoSQL approach with InfluxDB. A Grafana Dashboard is constructed to visualize the data, and appropriate visualisation tools are employed. The data will be parsed using Python and processed to InfluxDB using functional programming in combination with the Python client library provided by InfluxDB.

## The data - Movebank: Animal tracking

The data processed in this example is

## Prerequisites

- Python 3.10 (with pip3) https://www.python.org/downloads/
- Docker (20+, Version 20.10.13 is used for this project) https://docs.docker.com/get-docker/
- docker-compose installed (1.20+, Version 1.29.2 is used for this project) https://docs.docker.com/compose/install/

It is advised to use the most recent versions. The following commands can be used to determine whether the requirements have been met:

```
$ python3 --version
# Output: Python 3.10.*
$ docker -version
# Output: Docker version 20.10.13, build a224086
$ docker-comopse --version
# Output: docker-compose version 1.29.2, build 5becea4c
```

## Spinning up the composition

### docker-compose

There is no need to install InfluxDB or Grafana locally to quickly spin up the environment. Although it is possible to follow along using locally installed instances, docker-compose makes it easier to spin up the services. If no project root has yet been created, create a new folder for the project and add the following `docker-compose.yml` file:

```
version: "3.9"

services:
  influxdb:
    image: influxdb:2.2.0-alpine
    container_name: influxdb
    ports:
      - "8083:8083"
      - "8086:8086"
      - "8090:8090"
      - "2003:2003"
    volumes:
      - ./data/influxdb:/var/lib/influxdb2
    networks:
      - network1

  grafana:
    image: grafana/grafana:7.5.16
    container_name: grafana
```

```
    ports:
      - "3000:3000"
    user: "0"
    links:
      - influxdb
    volumes:
      - ./data/grafana:/var/lib/grafana
    networks:
      - network1

networks:
  network1:
```

Volumes are used by both InfluxDB and Grafana. The next step is to make sure that a directory called `data/` is created in the project root directory, which contains two empty subdirectories called `grafana/` and `influxdb/`. These directories will be filled to persist data when using these applications. Later in the tutorial, they will be filled in. This is how the project should be structured now:

```
- <project-root>
  - data/
    - grafana/
    - influxdb/
  docker-compose.yml
```

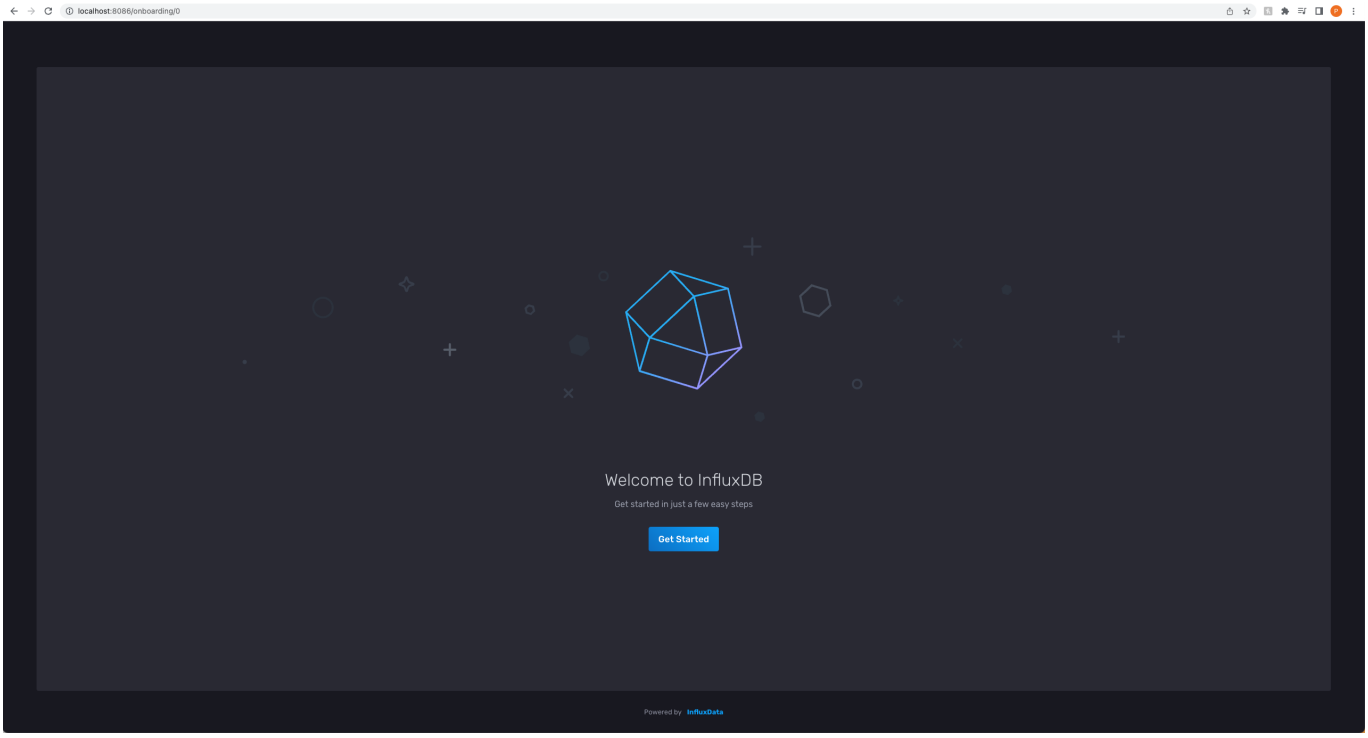
Also a network is added in the `docker-compose.yml` file so our Python application is able to communicate with the services on the same network later on when using Docker. Docker-compose is used to spin up a local instance of InfluxDB and Grafana. To start both services, enter

```
docker-compose up
```

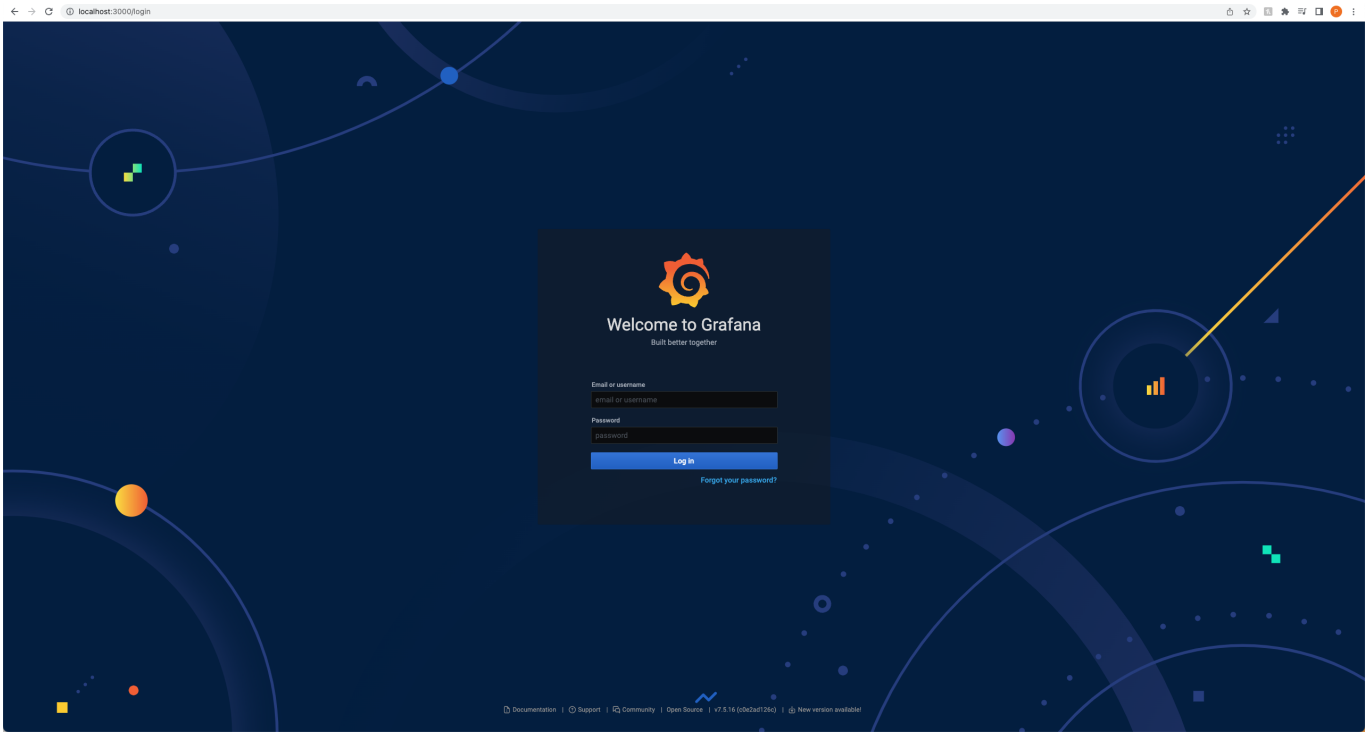
> Note for Unix-Users: If the Grafana-Container fails to start due to permission errors, the permission of the ./data/grafana Folder should be changed to 472 and made sure it is owned by the account using it.

The InfluxDB exposed the Port 8086 for the web interface and should now be reachable locally on `http://localhost:8086/`:

InfluxDB on localhost:8086




While Grafana uses Port 3000 for its web interface and should be reachable by typint `http://localhost:3000` in a browser:

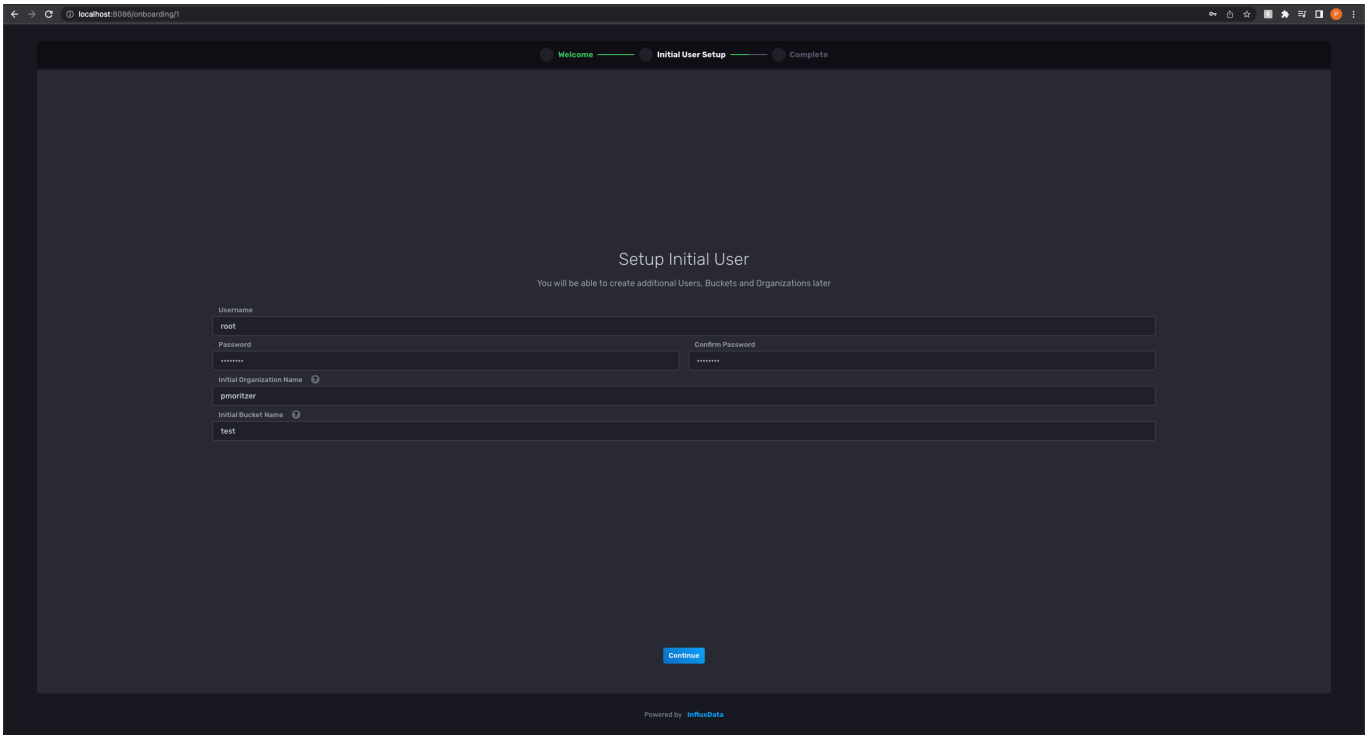



*Grafana on localhost:3000*

## Setting up InfluxDB

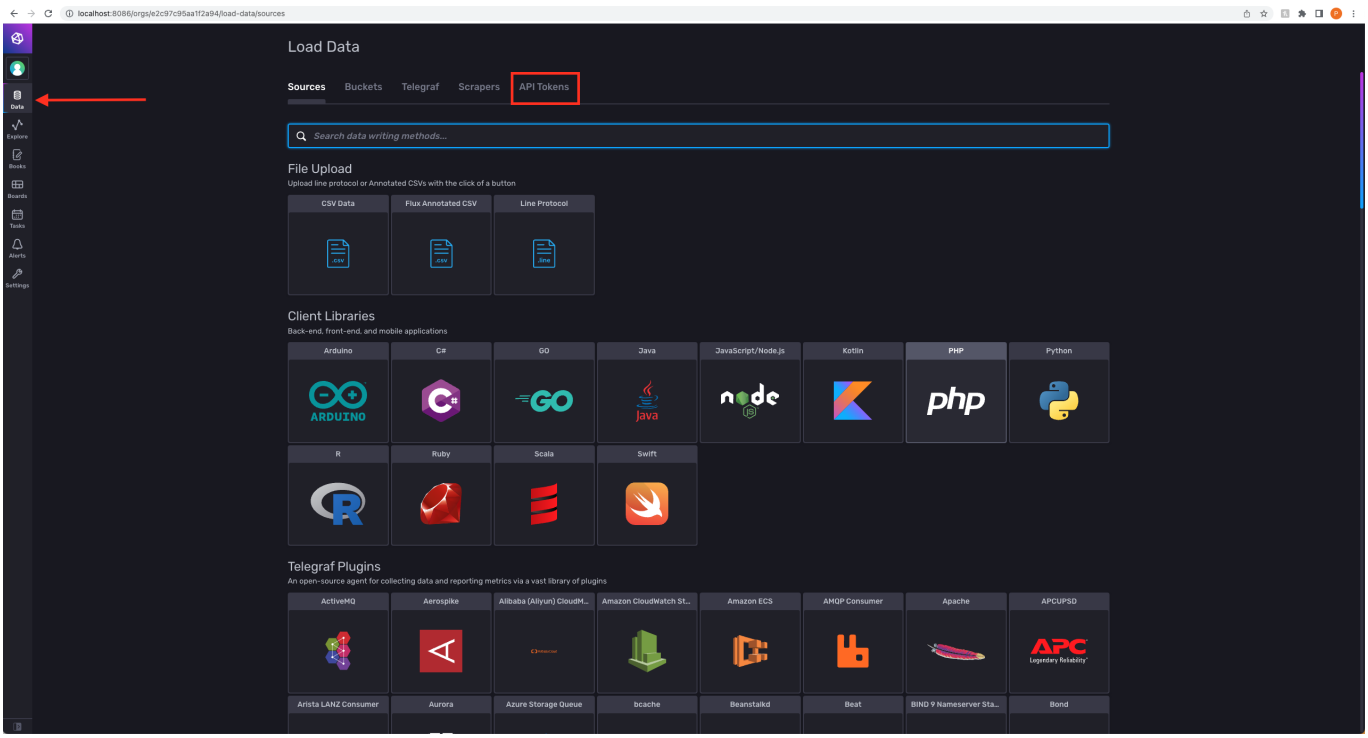
To access the InfluxDB web interface, go to `localhost:8086`. Press 'Get Started' to begin the setup procedure. The setup then asks for a login, password, organization name, and an initial bucket name in the next stage. The password for `root` is `password`, and the initial organization name is my tag `pmoritzer`. The bucket name must be set initially, but the bucket is not needed because our data processing unit python application will create a bucket dynamically in the future. The information should be saved because

it will be required to log in to the web interface later on and the organization name will serve as an identifier in the client application.

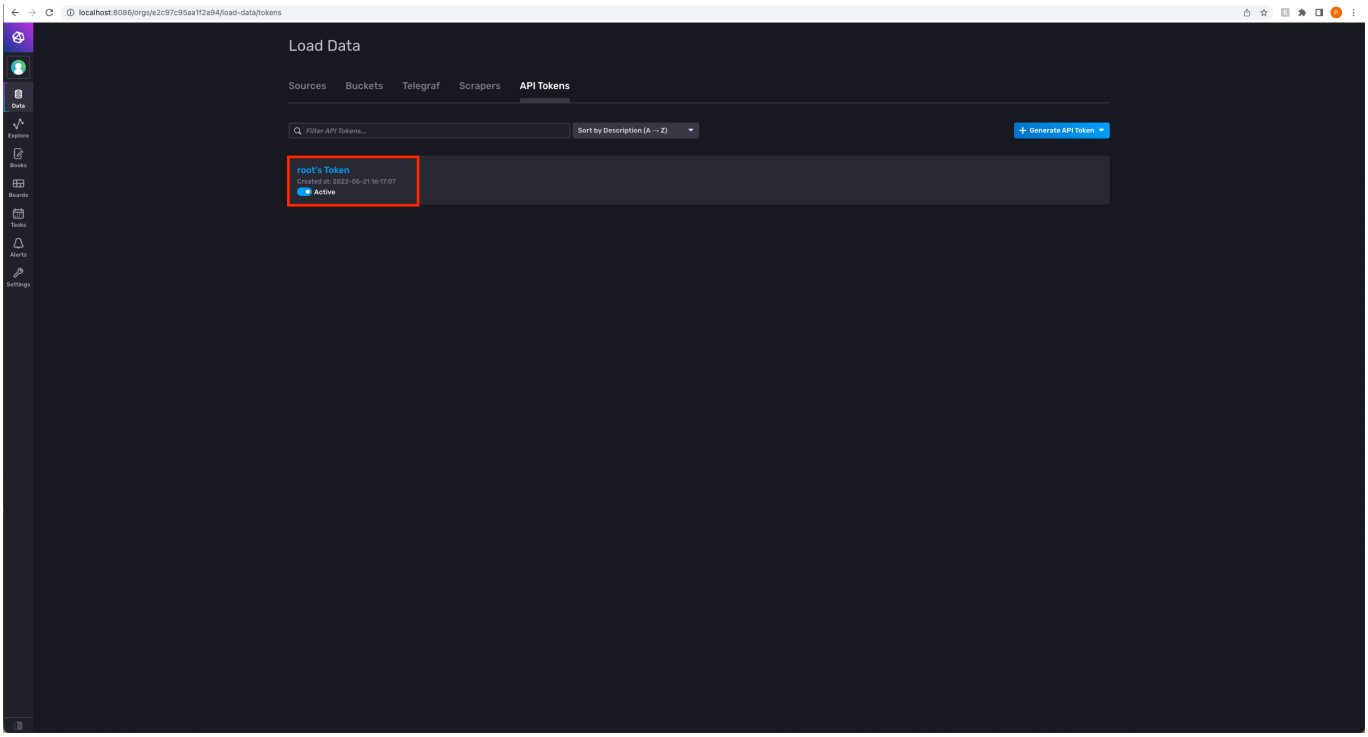



*First setup step for InfluxDB*

Because we want to inject data manually, simply select "Configure later" in the following step. The final step in setting up InfluxDB is to obtain the necessary API token. To do so, select the Data tab in the left sidebar, followed by the "API Tokens" tab in the tab view on top. A window will appear when you click on root's Token including the API Token. To access the InfluxDB from an external application, the token should be saved somewhere. Because security is not an issue in this proof of concept, we can use the root token; however, in a production environment, separate users with access rights should be set up.




*InfluxDB API-Token 1/2*




*InfluxDB API-Token 2/2*

## Writing the client application

### Setting up the environment

First, environment variables that will be required to connect to the local instance of InfluxDB will be set. Using an environment file, a folder called `<project-root>/env/` will be created within the project root directory. A file called `env.app` will be created that looks as follows:

```
INFLUX_URL=http://influxdb:8086
INFLUX_TOKEN=<root-api-token>
INFLUX_ORG=<org-name>
```

Replace the `INFLUX_TOKEN` property to equal the generated API-Token from the setup and the `INFLUX_ORG` properties with the organization name set. The data processing client application is able to connect to the database by loading these definded properties.

### Setting up the application

Create a file called `main.py`in the project's root directory that will serve as an entrypoint for the client application. First we will do a simple output to check whether our application works.

```
print ("Hello World")
```

*main.py*

```
$ python3 main.py
# Output: Hello World
```

For the dependencies a `requirements.txt`file is created in the project's root directory with following content to install the python client library for InfluxDB using pip. To make sure they are locally available, run the follwing command:

```
influxdb-client == 1.29.0
```

```
$ pip3 install -r requirements.txt
```

Next, we're going to dockerize the application. To do so, a Dockerfile will be created. Docker ensures that the application can run with the dependencies defined regardless of the system's environment.

```
# syntax=docker/dockerfile:1
FROM python:3.10.4-slim-bullseye

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip3 install -r requirements.txt

COPY . .

CMD ["python3","-u","./main.py"]
```

Create a `run.sh` script (or `.bat` for Windows Users) that executes following commands in a row:

```
$ docker build --no-cache -t influxdb-sample .
$ docker network create project_network1
$ docker run --network=project_network1 --env-file ./env/env.app influxdb-
sample
```

This script builds our application into a Docker Image, creates a network and starts the container while making sure it can communicate with the services on the same network, `network1`.

```
$ ./run.sh
# Output: Hello World
```

Every time we want to start the application locally we can run the `run.sh`script.

### Implementing functionality

First a function for connecting to InfluxDB will be implemented. create a folder `app/` including a file `connect_to_influx.py`. The content of the file will be as follows:

```
import time
from influxdb_client import InfluxDBClient

def connect_to_influxdb(url, token, org, retries=10, tried=0) ->
InfluxDBClient:
    print("Connecting to InfluxDB on " + url)
    client = InfluxDBClient(url=url, token=token, org=org, debug=True)
    health = client.health()

    if health.status == "pass":
        print("Connected to InfluxDB on " + url + "/")
        return client
    else:
        if tried < retries:
            print("Connection to {} refused, retrying {}
times.".format(url, (retries-tried)))
            tried += 1
            time.sleep(10)
            return connect_to_influxdb(url, token, org, retries, tried)
        else:
            raise ConnectionError("Connection to influxdb failed.")
```

The file contains a function that returns an InfluxDBClient connection object. The method invokes the constructor with the required information (URL, API-Token, organisation). If the connection is successful and healthy, this custom wrapper method returns the InfluxDBClient object. If it is not, it tries to connect to the given InfluxDB instance as many times as specified. If it fails, the application will exit with a connection error.

The application logic for writing our sample data to InfluxDB must now be implemented. First, we'll import the database connection function we just wrote, as well as any other imports required for this logic. Among these are the InfluxDB client library, csv for handling CSV data, and the `rx` functional programming library.

```
from app.connect_to_influx import connect_to_influxdb

from influxdb_client import Point, WriteOptions

from csv import DictReader
from collections import OrderedDict
from decimal import Decimal
import os

# rx for functional programming
```

```
import rx # functional programming library
from rx import operators
```

The parameters for connecting to the database will be loaded using environment variables in the following step. It must be ensured that the application is started using Docker with env-files. The script will fail if these environment variables are not specified.

```
url = os.environ['INFLUX_URL'] or 'http://localhost:8086'
token = os.environ['INFLUX_TOKEN'] or "<token>"
org = os.environ['INFLUX_ORG'] or 'pmoritzer'
```

Following that, a function will be defined to parse each row of the CSV containing the bird migration data. We will need the following fields for this analysis:

- timestamp
- event-id
- lon
- lat
- manually-marked-outlier
- individual-taxon-canonical-name
- tag-local-identifier
- individual-local-identifier

The remaining columns have little value for data analysis, so they are not considered in this project. The InfluxDB interface's Point-Object is returned by the following function. To create the dynamic data structure, we can use the provided Builder-Pattern. Each Point will later be stored as a dataset in an InfluxDB bucket.

```
def parse_row(row: OrderedDict):

    return Point("migration-point").tag("type", "migration-
value").measurement("migration") \
        .field("event-id", row['event-id']) \
        .field("lon", Decimal(row['location-long'])) \
        .field("lat", Decimal(row['location-lat'])) \
        .field("manually-marked-outlier", row['manually-marked-outlier'])
\
        .field("individual-taxon-canonical-name", row['individual-taxon-
canonical-name']) \
        .field("tag-local-identifier", row['tag-local-identifier']) \
        .field("individual-local-identifier", row['individual-local-
identifier']) \
        .time(row['timestamp'])
```

The function above is then called for each row of the csv file using functional programming. It must be ensured that the.csv dataset mentioned earlier is located in the project's root directory and is named `migration_original.csv`.

```
data = rx \
    .from_iterable(DictReader(open('migration_original.csv', 'r'))) \
    .pipe(operators.map(lambda row: parse_row(row)))
```

The command above maps each row from the source data set to an Ordered Dictionary entry and parses it to a. Afterwards we are left with a dictionary that is readable by the InfluxDB data client.

> Example adapted from: https://github.com/influxdata/influxdb-client-python/blob/master/examples/import_data_set.py (visited: June 4th, 22:00)

The code below establishes a connection to InfluxDB, creates a new bucket using the Bucket API, and writes data to InfluxBD using the Write API, all of which are provided by the Python library.

```
with connect_to_influxdb(url, token, org) as client:
    bucket_name = 'bird-migration'
    bucket_api = client.buckets_api()
    old_bucket = bucket_api.find_bucket_by_name(bucket_name=bucket_name)
    if old_bucket:
        try:
            bucket_api.delete_bucket(old_bucket)
        except:
            exit()
    bucket = bucket_api.create_bucket(bucket_name=bucket_name, org=org)

    with client.write_api(write_options=WriteOptions(batch_size=50000,
flush_interval=10000)) as write_api:
        write_api.write(bucket="bird-migration", record=data)

    query = 'from(bucket:"bird-migration")' \
            ' |> range(start: 0, stop: now())'
    result = client.query_api().query(query=query)
    print()
    print("=== results ===")
    print()
```

The results will be validated by committing the following flux query using the Python code:

```
from(bucket:"bird-migration") |> range(start: 0, stop: now())
```

Using the InfluxDB web client, run the same flux query to see if the data is in our database.

## Setting up Grafana

python3 -m pip install influxdb

### Python

### Grafana

### Run sample project

## Summary

## Sources