

DDP

Docker Databases Pandas



Code and slides

- Slides and code snippets are also pushed to repositories on GitHub
<https://github.com/Giulianini/docker-database-pandas>
- Make a lot of questions (make me feel the Impostor Syndrome)
- **Guideline:**
 - Slides with “→ Example” refer to a practical example on notebook or shell
 - All the “floating code” in slides is present in the form of script.sh inside corresponding folders. Ex docker-intro/scripts/docker-run.sh
- **Examples:**
 - 1 Real example of a sample project → REST API – Controller – Model (DBs)
 - More notebook examples of pandas-database interactions
 - Docker is used in each example, be comfortable with it

Schedule

- ~~Docker refresh~~

- ~~Theory~~

- ~~Docker commands~~

- ~~Dockerfile~~

- ~~Docker compose~~



- Databases

- Relational – SQL like → MySQL, SQLServer, Oracle

- Object-Relational – SQL Like → PostgreSQL

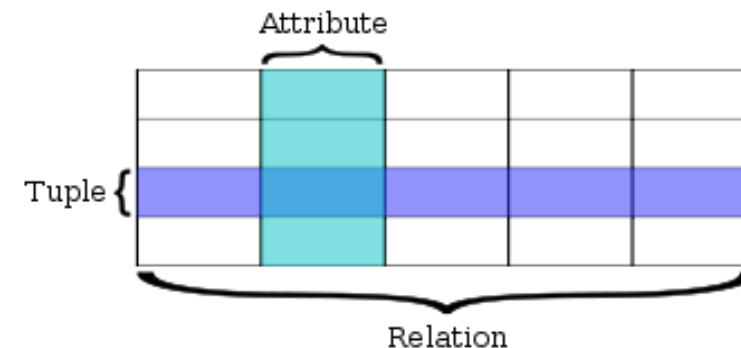
- Non-Relational – NoSQL like → MongoDB

- Temporal – SQL like (custom syntax) → InfluxDB

- Pandas integration → Probably shown in parallel with each DB type

Relational DBs - RDBMS

- **First type DB** (1970) created by IBM: used since that time, pervasive systems
- Data organization with an OOP perspective:
 - **Tables:** relations of columns and rows with a unique **id** for each row. *Like **Classes** in OOP*
 - → **Type of Entity** (Person, Customer, ecc)
 - **Column:** attributes of the type. *Like **fields** in OOP languages*
 - **Row:** tuples, records of multiple attributes of same relation. *An **instance of a Class** in OOP*
 - **Keys:**
 - each instance must be unique and identified by the **smallest set of attributes** → **Primary Key**
 - relations between **A** and **B** are created using the **primary key of B** inside **A**. **Foreign key**
- **Modelling**
 - We use the **Entity Relationship** model (ER) → like UML
 - **Entity:** the tables with attributes
 - **Relationship:** connection between keys
 - 1-1 → Person - Anagraphic
 - 1-* → Person - Gadget
 - *-1 → Person - Bus
 - *-* → Person - Course



My SQL (SQL Like) – A Company-wise view

- **Commercial factors:**

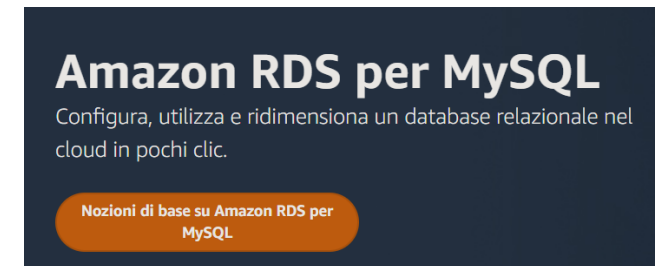
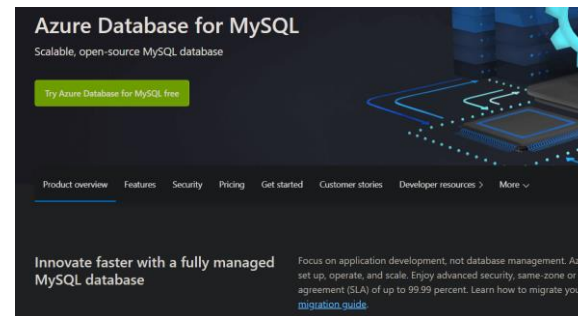
- **Most famous RDBMS:** client and server
- Free software licensed under **GNU GPL** → **copyleft** not a business-friendly license. Company need to **release all source-code** to meet **GPL open-source transitive requirements!**
- **Alternatives** → **MariaDB**. But licensed under **GPL** too!
- **Totally free alternatives** → **SQL Lite (Public domain license)**, **PostgreSQL (BSD license)**, hence the fact that Postgres became the most used relational DB in the business world.

- **Servers (MySQL, Maria DB):**

- Working on port **3306 TCP**
- Generally **self-hosted** but is sometimes present in *SAAS, PAAS* environments with preconfigured connectors.
- Amazon AWS, Azure Database, ecc
- **Dockerized** nature: simplify deployment, scaling, resource allocation, balancing.

- **Clients ([Beekeeper](#)):**

- We use beekeeper to view database and make fast query.
- Download the portable version



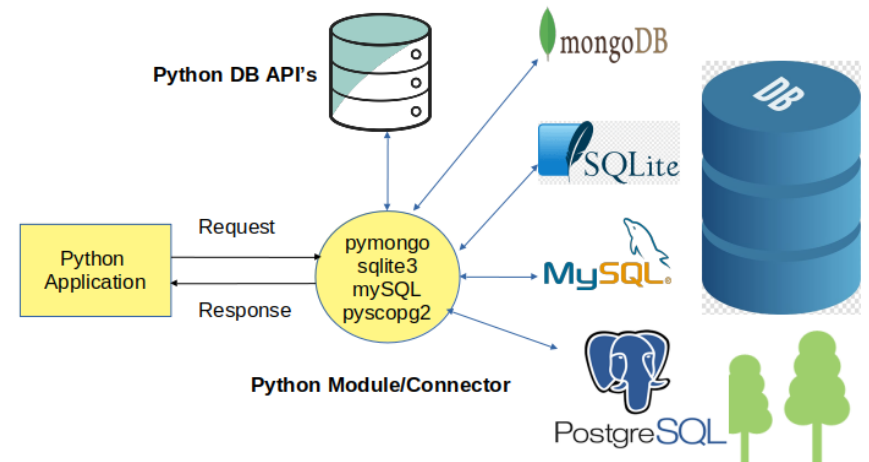
Databases and Python - Connection stack

- **The recipe for a DB connection in Python** → other languages behave in a similar way (Java JDBC)

1. **DB:** The DB itself. Communications happen through TCP connection.

2. **Connector:** A database connector is a **driver** that works like an **adapter** that connects a software interface to a specific database vendor implementation. Must Implement **DB API v2.0**:

1. PyMySQL → Our choice for MySQL connection
2. MySQLdb → MySQL
3. Psycopg2 → PostgreSQL
4. SuperSQLite → SQLite

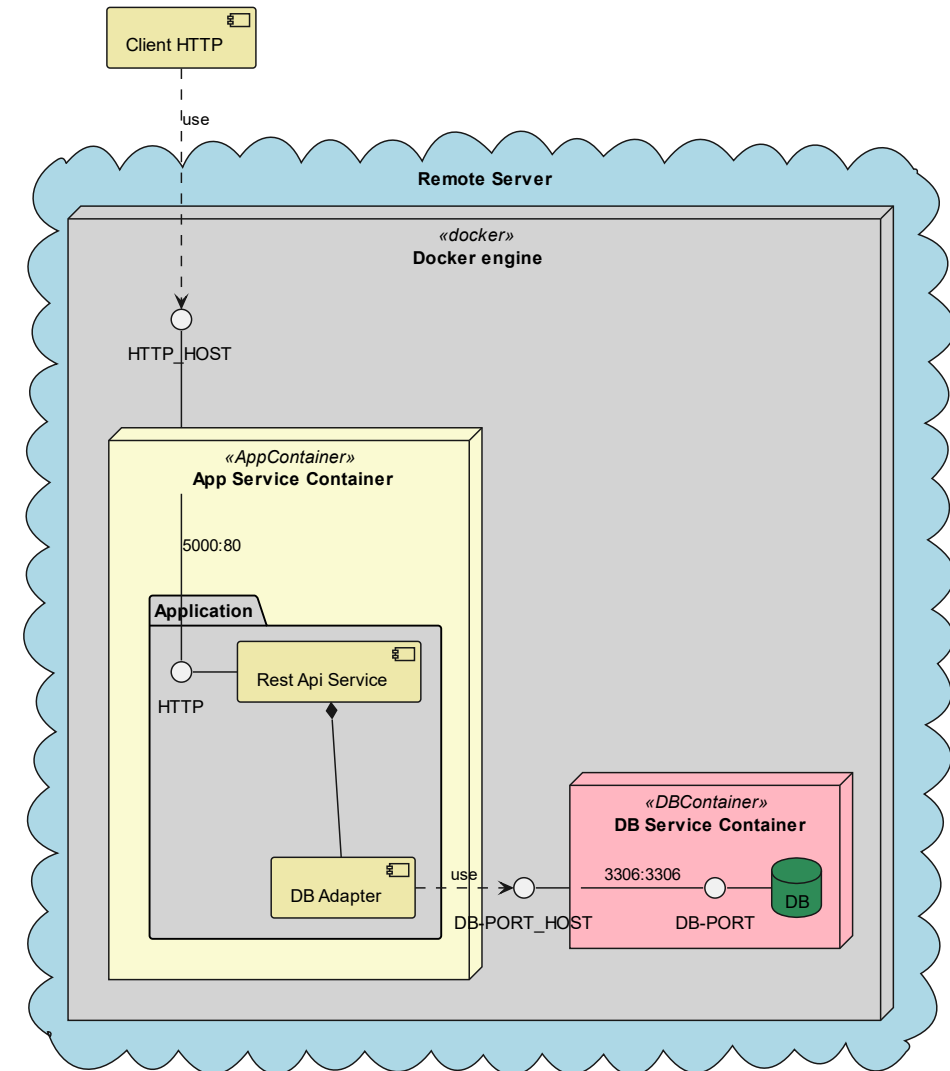


- **DB-API v2.0:** Python's standard database interface **specification** is **Python DB-API**. It's documented in [PEP 249](#). Nearly all Python database connectors such as Sqlite3, PyMySQL, and MySQL-Python **conform** to this interface.
- **PyMySQL** is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and contains a pure-Python MySQL client library.

3. **SQLAlchemy:** A **Python SQL toolkit** and **Object Relational Mapper (ORM)** that gives application developers the full power and flexibility of SQL. SQLAlchemy supports many **dialects/DBAPI options**.

Rest API Service – SQL version → Example

- Multi service dockerization and internal connection
 - **App Service Container:**
 - **REST API:** It's a Flask REST API server listening on localhost:80. See Dockerfile entrypoint.
 - **DB adapter:** it's a python module that maps REST method to SQL Queries. Queries are made using SQLAlchemy with pyMySQL connector driver.
 - **Mapping:** port 80 is mapped to 5000 on the host network because port 80 was already in use
 - **DB connection:** App service connects to the DB service using Docker internal DNS naming resolution. sql-service→172.0.0.3
 - **Restart on failure:** initially a connection error is raised because DB container is not completely initialized.
 - **DB Service Container**
 - **MySQL:** server listening on port 3306. No need for a remapping.
 - **Initialization:** tables initialized from init.sql script
 - **Configuration:** loaded from environment variables.
 - From **dotenv file** if outside docker
 - Passed to **docker-compose env-file** if inside docker
- Exercise:** Add other REST-API and update DB accordingly → Test with Postman



MySQL - Pandas Integration → Example

- SQLAlchemy introduces **extension methods** for DB interactions: `to_sql` e `read_sql`
- **DB configuration**
 1. Install pymysql driver and sqlalchemy db tool
 2. Database connection string with pymysql driver
 3. Engine creation and connection
- **Reading cvs from Pandas**
 - Dataframe creation
- **Writing dataframe to SQL table**
 - Using sqlalchemy extension methods
- **Updating tables connections**
 - Need to remap Primary Key
 - Need to remap Foreign key
- **Execute sql query directly on DB**

SQLite - Pandas Integration → Example

- **SQLite Example**

- No need for a driver → SQLite db is a file loaded in memory

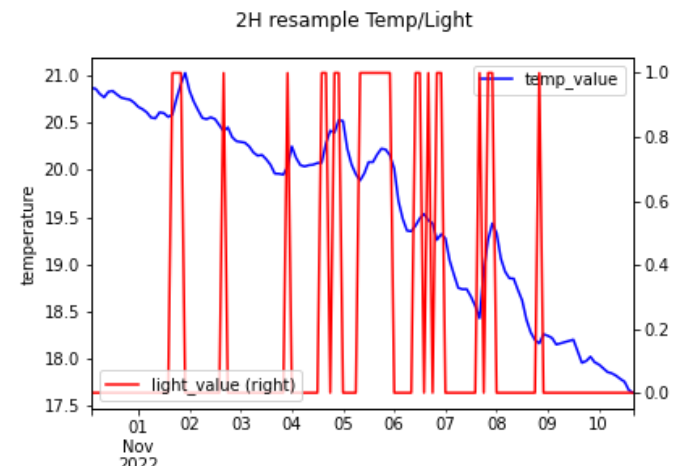
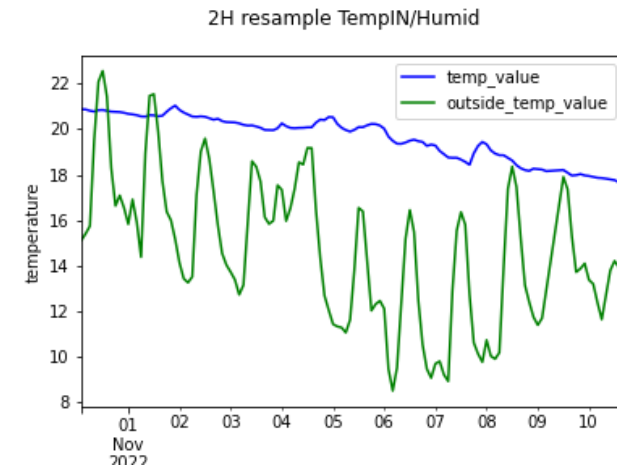
- **Home Assistant Examples**

- Read temperature values for inside and outside

1. Define **SQL queries** and retrieve data
2. **Process** data on Pandas side
3. **Resample** data and **interpolate** holes
4. **Plotting**

- Compare temperature and light condition

1. Light as a **presence indicator**
2. **Forward fill** data for resampling
3. **Merging** data
4. **Correlation** between presence and temperature



Schedule

- ~~Docker refresh~~

- ~~Theory~~
- ~~Docker commands~~
- ~~Dockerfile~~
- ~~Docker compose~~



- Databases

- ~~Relational – SQL like → MySQL, SQLServer, Oracle~~
- ~~Object Relational – SQL Like → PostgreSQL~~
- Non-Relational – NoSQL like → MongoDB
- Temporal – SQL like (custom syntax) → InfluxDB



- Pandas integration → Probably shown in parallel with each DB type

NoSQL Databases

- **NoSQL is no relation?**

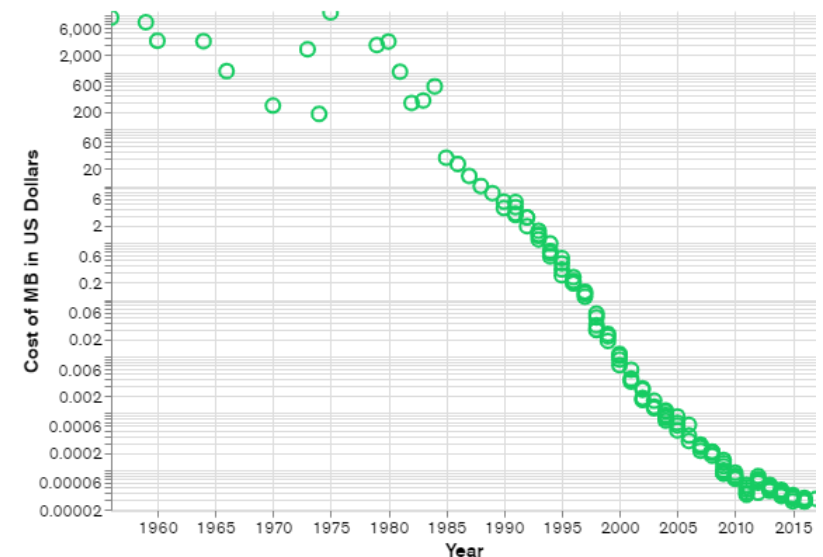
- NO! NoSQL only means that relations are memorized using **non-table structures**.
- Connections are expressed in a different **simple** way → **Single data structure** with **multiple nesting**

- **Why NoSQL?**

- SQL DBs aim to reduce memory allocation through **schema normalization** and **optimization**
- Today memory allocation costs are not companies' main problem → **developers effort** is the true cost now, reduce using a simpler DB
- **Agile-friendly DBs** → Agile methodologies need fast iterations hence fast DBs

- **NoSQL DB types**

- **Document based:** Data is memorized in the form of **objects** (ex JSON) with keys and values.
 - **Data types:** many datatypes to choose. Also new types can be defined.
 - **Scaling:** Horizontal scaling each time we need to add more data
 - **Examples:** MongoDB is the primary example, but also Dynamo DB, Redis are widely used
- **Wide-column:** each row couldn't have the same columns.
 - **Usage:** when we **know query types** and we have to save **large amount of data**. IOT usages
 - **Examples:** Cassandra, Hbase
- **Graphic:** use the concepts of **Nodes** and **Edges**. Nodes save information about persons, places and things while Edges save information about relations between nodes.
 - **Usage:** for social network models, fraud detection and recommendation engines.
 - **Examples:** Neo4j, JanusGraph

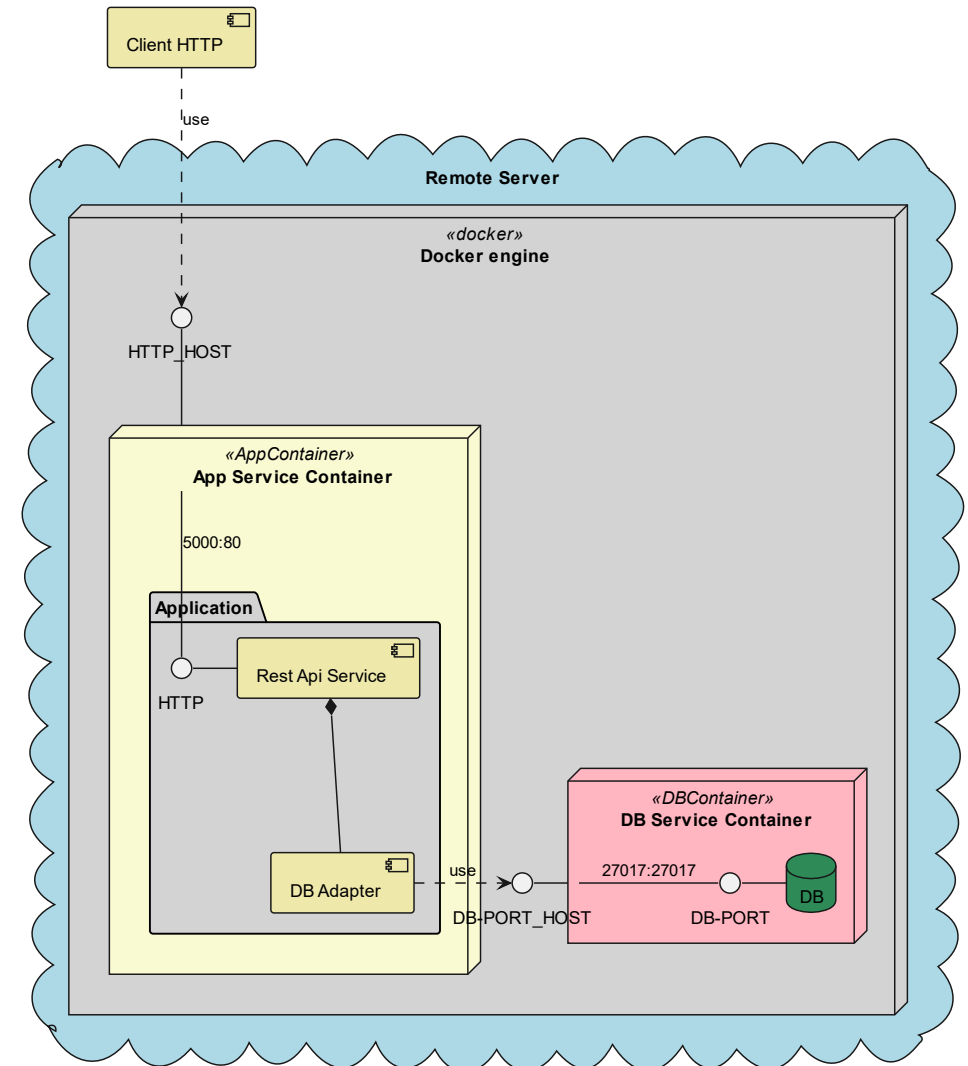


MongoDB – the document base king

- **NoSQL:** most famous NoSQL database → **Documental DB**
- **Documental form:** Mongo uses **BSON** a dynamic schema version of **JSON**. It's simple JSON for the user
- **History:**
 - **2007 – PAAS:** born as PAAS service component for the company 10gen
 - **2009 – Open Source:** source open and 10gen giving commercial support for the product
 - **2009 - now → MongoDB:** change the name and include other services like *Compass, Atlas, ecc*
- **Commercial factors:**
 - **Community version**, Licensed under **SSPL** (*Server Side Public License*): **AGPLv3 + rewrite of section 13** “anyone who offers the functionality of SSPL-licensed software to third-parties as a service must release the entirety of their source code, including all software, APIs, and other software that would be required for a user to run an instance of the service themselves” legit cos companies reselled MongoDB as a Service without giving MongoDB nothing. Now they have to release the source code.
 - Like MySQL **cannot be sold** in a production environment
 - **Enterprise version**, licensed under **MongoDB Inc. licenses**: when we want to sell the DB along with our solution.
- **Servers (MongoDB):**
 - Working on port **27017 TCP**
 - Generally **self-hosted** but is sometimes present in *SAAS, PAAS* environments with preconfigured connectors.
 - Amazon AWS, Azure Database, ecc
 - **Dockerized** nature: simplify deployment, scaling, resource allocation, balancing. Even more than SQL solutions
- **Clients ([MongoDB Compass](#)):**
 - We use Compass to view database and make fast query.

Rest API Service – MongoDB version → Example

- Multi service dockerization and internal connection
- **App Service Container:**
 - **REST API:** It's a Flask REST API server listening on localhost:80. See Dockerfile entrypoint.
 - **DB adapter:** it's a python module that maps REST method to MongoDB "Queries".
 - **Mapping:** port 80 is mapped to 5000 on the host network because port 80 was already in use
 - **DB connection:** App service connects to the DB service using Docker internal DNS naming resolution. mongo-service→172.0.0.3
 - **Restart on failure:** initially a connection error is raised because DB container is not completely initialized.
- **DB Service Container**
 - **MongoDB:** server listening on port 27017. No need for a remapping.
 - **Initialization:** tables initialized from mongo-init.js script. (not needed)
 - **Configuration:** loaded from environment variables.
 - From **dotenv** file if outside docker
 - Passed to **docker-compose env-file** if inside docker
- **Exercise:** Add other REST-API and update DB accordingly → Test with Postman



MongoDB - Pandas integration

Using MongoDB with Pandas is very straightforward

```
import pymongo
import pandas as pd
from pymongo import MongoClient

client = MongoClient()

#point the client at mongo URI
client = MongoClient('Mongo URI')
#select database
db = client['database_name']
#select the collection within the database
collection = db['collection_name']
#convert entire collection to Pandas dataframe
df = pd.DataFrame(list(collection.find()))
```

- Import pymongo
- Start a MongoClient with connection string
- Fetch the database
- Fetch the collection
- Get the data
- Wrap data into a list and pass to a Dataframe

MongoDB – Dump and Restore a backup → Example

- **Why?**

- We want to **load an existing dataset dump** to make some experiments.
- We want to learn how to **backup and restore a mongoDB**

- **Sample collections [Link](#):**

- Repository that contains a **dump** of Mongo sample collections
- Loading must occur **after mongo startup!**
- **2 solutions:**
 - **Manual solution:**
 1. Attach to container instance with a bash terminal: `docker exec -it mongo bash`
 2. Moving into db folder where dump is stored: `cd /data/db`
 3. Call the mongo restore with authentication: `mongorestore -u root -p my_password --nsInclude="SampleCollections.*" --authenticationDatabase admin`
 - `-u <user> -p <password>` work on the **authentication db** → we need to **tell mongo where are auth infos**
 - `--nsInclude="db.collection"` → what **db/collections** we want to **backup/restore**
 - `/data/db/dump/` → the backup **entry folder**
 - **Automatic solution:**
 1. Define a **mongo-init.sh** script into: `./init/mongo-init.sh /docker-entrypoint-initdb.d/`
 2. Insert the manual steps command inside the script
 3. Mongo will **execute** the script **after DB creation**
 4. Env variables are always **accessible** → safe

- **Exercises:**

1. **Explore DB data** → With MongoDBCompass db explorer
2. **Show pokemon images** → With pandas
3. **Experiment on other collections** → Fetch data, insert data, integrate in pandas

Schedule

- ~~Docker refresh~~

- ~~Theory~~
- ~~Docker commands~~
- ~~Dockerfile~~
- ~~Docker compose~~



- Databases

- ~~Relational – SQL like → MySQL, SQLServer, Oracle~~
- ~~Object Relational – SQL Like → PostgreSQL~~
- ~~Non-Relational – NoSQL like → MongoDB~~
- Temporal – SQL like (custom syntax) → InfluxDB



- Pandas integration → Probably shown in parallel with each DB type

Time Series – A machine learning prespective

- **Pattern:** some type of **data** (simple or complex) that we want to **classify**.
 - Hand-writing, movement, photo, fingerprint, vibration frequency, ecc → can be **classified** in a **predetermined SET** of abstract boxes, containers, logics aka **classes**. Ex if classes cat=1, dog=2 a photo with a cat is classified with 1
 - **Static vs Dynamic** → **Static** = time independent (photo, fingerprint) **Dynamic** = time dependent (movement, hand-writing)
 - **Unique** in some **Features** → pixels in photos, 3D position in movement, minutiae in fingerprint. **So, a List/Vector of Features**
 - **Classification** = **f: pattern->class** hence a pattern must be converted to a numeric representation, a numerical vector of features called **Feature Vector**. Static: **f: (Int, ...) -> Int** Dynamic: **f: [(Int, ...) ...] -> Int**
- **Time series pattern:** a Dynamic pattern composed by **multiple features vectors** in the number of time series length.
 - To classify a hand gesture, track the position of the hand during a time window (ex 1 sec=30 FPS) and record the features vector at each frame. A feature vector is **v = [x, y, z]**, for 30 FPS it's a vector of [x, y, z] of length = 30. **[v1, v2, ..., v30]**
- **Standard DB and Time Series**
 - A SQL DB fails to store all the data because it's relational and it need to store a new row for every time-frame. Can be optimized
 - A NoSQL DB is sometimes used to store time series data in a big data environment
 - In a **telemetry driven environment** must optimized according to data type, time series nature. Compression of data, optimize
 - Need a support for **time series operations** and query. Different from a normal DB query. Ex interpolate, resample, aggregate
 - **Integration** to commonly used big-data **time-series framework** and tools. Ex Pandas

Influx DB – Intro 1

- **Why**

- Support for **time series operations** and **query**
- A **high-performance** time series **engine**
- A **powerful API & toolset** for real-time applications
- A **massive community** of open-source developers
- **Static schema** → **most performance** in data retrieve

- **Data Element**: list of data elements with correspondent column in table

- **Timestamp[_time]**: it stores timestamp in a nanosecond form according to **RFC3339 UTC**
- **Measurement[_measurement]**: a string tag for measurement. Container for tags, fields and timestamps
- **Fields[_field, _value]**: field key _field and a field value _value. Key=[string], Value=[string, float, int or boolean]. NOT INDEXED
- **Field set**: collection of fields(key, value) pairs associated to same timestamp
- **Tags**: customizable column. Key=name of column, value=the value of the row in that column. INDEXED
- **Tag set**: like Field set
- **Bucket**: All InfluxDB data is stored in a bucket. A **bucket** combines the concept of a **database** and a **retention period** . Explicit **bucket schema** is required for each measurement.
- **Series**: A **series key** is a **collection of points** that share a **measurement**, **tag set**, and **field key**. A series includes **timestamps** and **field values** for a given **series key**.
- **Point**: A **point** includes the **series key**, a **field value**, and a **timestamp**
- **Organization**: An InfluxDB **organization** is a **workspace** for a **group of users**. All dashboards, tasks, buckets, and users belong to an organization.

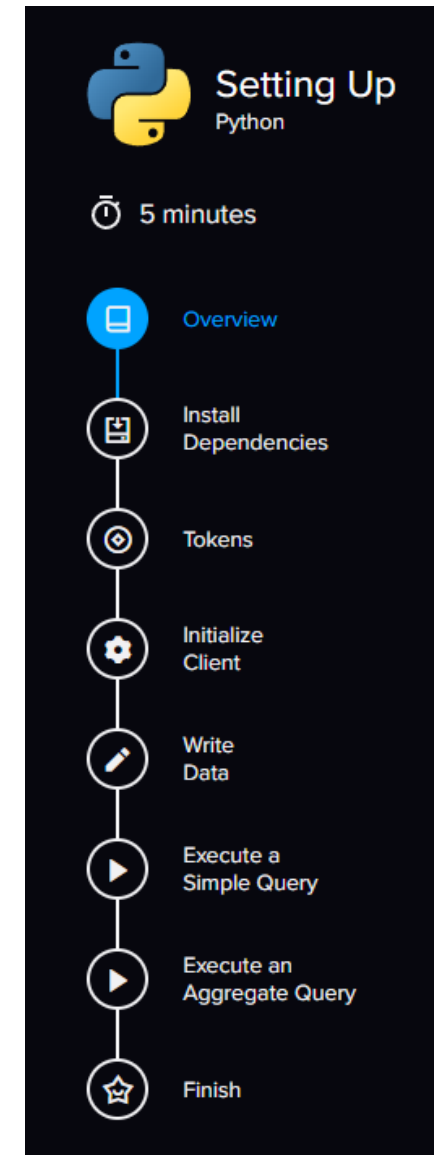
_time	_measurement	location	scientist	_field	_value
2019-08-18T00:00:00Z	census	klamath	anderson	bees	23
2019-08-18T00:00:00Z	census	portland	mullen	ants	30
2019-08-18T00:06:00Z	census	klamath	anderson	bees	28
2019-08-18T00:06:00Z	census	portland	mullen	ants	32

Influx DB – Intro 2

- **Internal Structure:** data elements are stored in **time-structured merge tree (TSM)** and **time series index (TSI)** files to efficiently compact stored data.
- **Data Schema:** InfluxDB also provides a **tabular data schema** (used to raw view and CSV export) that includes the following:
 - **Annotation rows:** Describe column properties
 - **Header row:** Defines column labels that describe data in each column: (table, _time, _value, _field, _measurement, ...)
 - **Data Row:** Each data row contains the data specified in the header row for one point.
 - **Other columns:** Optional columns: annotation, result, table
 - **Group keys:** grouping records that share common values in specified columns
- **Design Principles:** InfluxDB implements optimal design principles for time series data.
 - **Time ordered data:** improve performance
 - **Strict update and delete permissions:** InfluxDB tightly restricts **update** and **delete** permissions. Deletes generally only affect data that isn't being written to.
 - **Read and Write queries first:** InfluxDB prioritizes read and write requests over strong consistency. If frequency is high a read could not return most recent data.
 - **Schemaless design:** InfluxDB uses a schemaless design to better manage discontinuous data. Ex start-report-stop devices
 - **Datasets over individual points:** points are differentiated by timestamp and series, no IDS → aggregate is a common operation
 - **Conflict resolution:** If a new field value is submitted for a point, InfluxDB updates the point with the most recent field value.

Influx DB – Explore the basics → Example

- **Setting up with python:**
 - **Examine docker-compose:** Spin the instance
 - **Connect to localhost:8086:** Examine the GUI
 - **Python integration:** Influx DB include how to connect to influx directly inside the instance so, we only must copy API tokens edit server and follow the tutorial.
 - **Select a bucket:** we must have a container for our data
 - **Write to the bucket:** write data to the bucket using examples in tutorial
 - **Try in GUI:** visualize your data in GUI
- **Loading my Home Assistant dump:**
 - **Spin a clean instance:** delete volumes
 - **Attach to container bash:** `docker exec -it influxdb_container bash`
we can find in container the backup/ folder
 - **Restore the dump:** `influx restore <folder>`
 - **How could we automate the backup??**
 - **Find new buckets:** homeassistant and homeassistant-aggregated are created and they are on a different organization, switch organization and find them.
 - **Examine buckets:** examine the two buckets, find differences, explain what they do
 - **Make some visual queries on GUI:** make some queries and understand query textual form
 - **Look at textual queries generated by visual queries** -> Helps a lot understanding queries
 - **Then we will see textual queries**



Influx DB - Queries

```
from(bucket: "example-bucket")
  |> range(start: -1h)
  |> filter(fn: (r) => r._measurement == "example-measurement" and r.tag == "example-tag")
  |> filter(fn: (r) => r._field == "example-field")
```

- **Flux:** is the language used by Influx DB. It's functional, stream-like → a flux
- **Data flow up-down:** from source to simple element
- **From:** define the source bucket
- **Range:** define the time range. Start and Stop of type date. Also: 1s, 1m, 1d, 1we, 1mo, 1y, ecc
- **Group:** we have group-by

```
|> group(columns: ["host"], mode: "by")
```
- **Sort:** not so used

```
|> sort(columns: ["host", "_value"])
```
- **Limit:** limit the output data

```
|> limit(n: 4)
```
- **Aggregate:** automatic aggregate on window size with operation

```
|> aggregateWindow(every: 20m, fn: mean)
```
- **Map:**

```
|> map(fn: (r) => ({ r with _value: r._value * r._value })) "by"
```
- **Moving average:** compute the moving average over n samples

```
|> movingAverage(n: 5)
```
- **Timed moving average:** over time

```
|> timedMovingAverage(every: 2m, period: 4m)
```
- **Derivative:** for rate change

```
|> derivative(unit: 1m, nonNegative: true)
```
- **Histogram:** compute the histogram with upper bound over some defined bins

```
|> histogram(
  column: "_value",
  upperBoundColumn: "le",
  countColumn: "_value",
  bins: [100.0, 200.0, 300.0, 400.0],
)
```
- **Fill:** forward fill or static fill of null

```
|> fill(usePrevious: true) |> fill(value: 0.0)
```
- **Median/quantile:** median and percentile

```
|> median() |> quantile(q: 0.99, method: "estimate_tdigest")
```
- **Cumulative sum:**

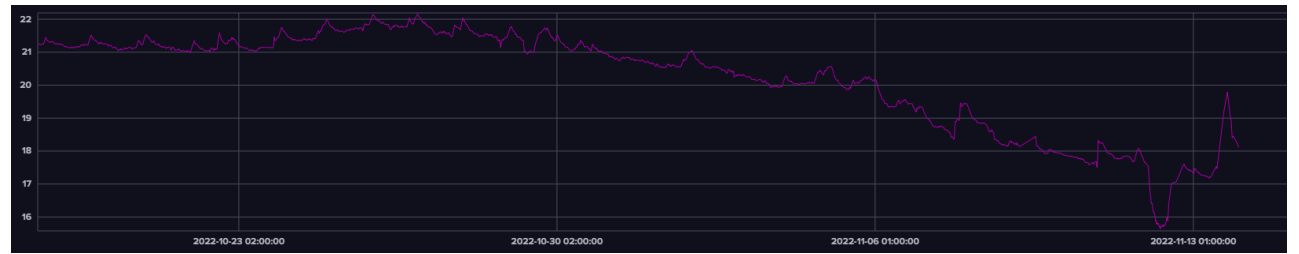
```
|> cumulativeSum()
```

Pandas Influx DB – Integration → Example

- Influx supports pandas from scratch: `query_data_frame()` function
- **Use homeassistant-aggregated bucket!!!** → homeassistant bucket has a huge amount of data it will destroy your CPU “uomo avvertito...”

1. Try queries on home assistant aggregated

```
Query 1 (0.11s) +
1 from(bucket: "homeassistant-aggregated")
2   |> range(start: -30d)
3   |> filter(fn: (r) => r["domain"] == "sensor")
4   |> filter(fn: (r) => r["entity_id"] == "basement_bed_pht_temperature")
5   |> filter(fn: (r) => r["_field"] == "value")
```

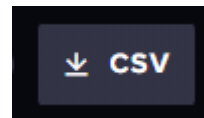


2. Visualize graphs or Raw data each time:

- Try with different graphs types, customize duration

3. Export a query on CSV

- Analyze the csv



4. Exercises: some exercises hard and easy with influx DB queries.

- Influx has many functions probably all functions you will ever need

END

- Good luck for the course