The ODA service

The Observable Data Access service (ODA), depicted in Figure 1, is fed by Data Generators that push data acquired by sensors, digital twins, or services, through a streaming broker (Apache Kafka). The ODA stores data in the short term (e.g. a week, a month) and provides data access to Data Consumers in two ways:

- a DC can subscribe to certain topics to get data streamed by generators about such topics, or
- a DC can pull data by invoking a REST API provided by ODA.

Conceptually, the data flow in a system exploiting an ODA service is bottom-up, from the DGs to the DCs.

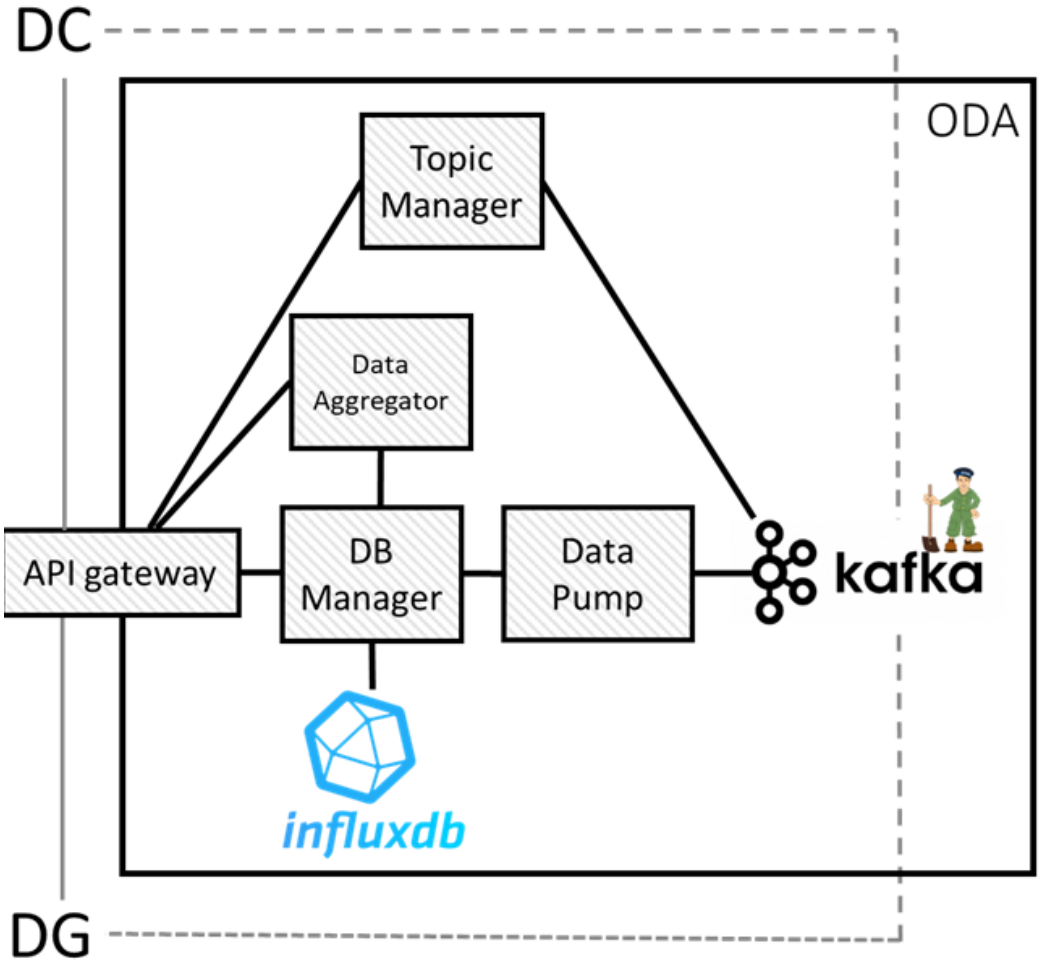

Figure 1: The ODA service

The database of the ODA is a time series database, implemented by exploiting InfluxDB.

Apache Kafka is used as a broker to implement the queue where DGs push data and DCs receive subscribed data. To manage the Kafka broker Apache Zookeeper is used.

The Data Aggregator manager aggregated queries by asking data to the DB manager and aggregating data as requested by the DC.

The Data Pump acts as a consumer of the Kafka broker subscribed to all the topics to store the data in the database and retrieve them when requested.

The Topic manager creates in Kafka the topics registered in ODA by the DGs and provides the list of topics to the DCs during the registration process.

Interacting with ODA

Both DCs and DGs must register to the ODA service through the API gateway, which delivers the endpoint to reach the Kafka broker.

Figure 2 shows how DGs must act to send data to the ODA service, starting with the registration of the topics it will use and then streaming the data to the Kafka broker directly.

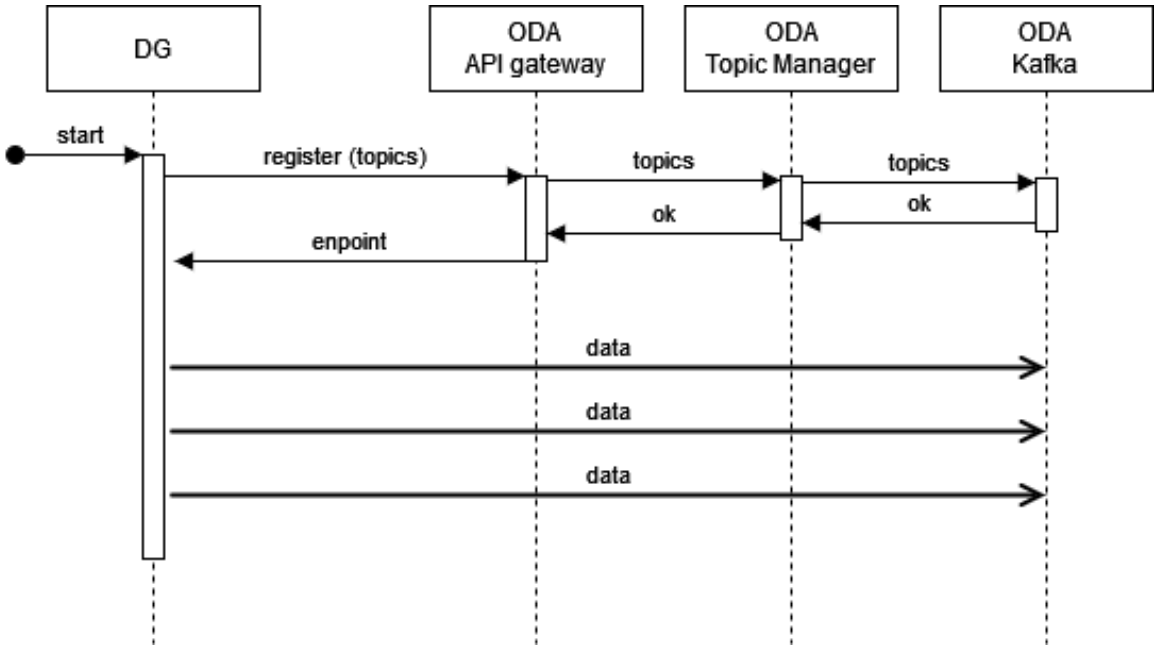

Figure 2: DG streaming data

Figure 3 shows how DCs interested in data streaming must act to receive data from the ODA service. After the registration through the API gateway, DCs receives the list of topics of ODA and the Kafka endpoint. They must subscribe to topics

they are interested in contacting the Kafka broker. From that point, the data streamed to Kafka having such topics will be sent to the DC.

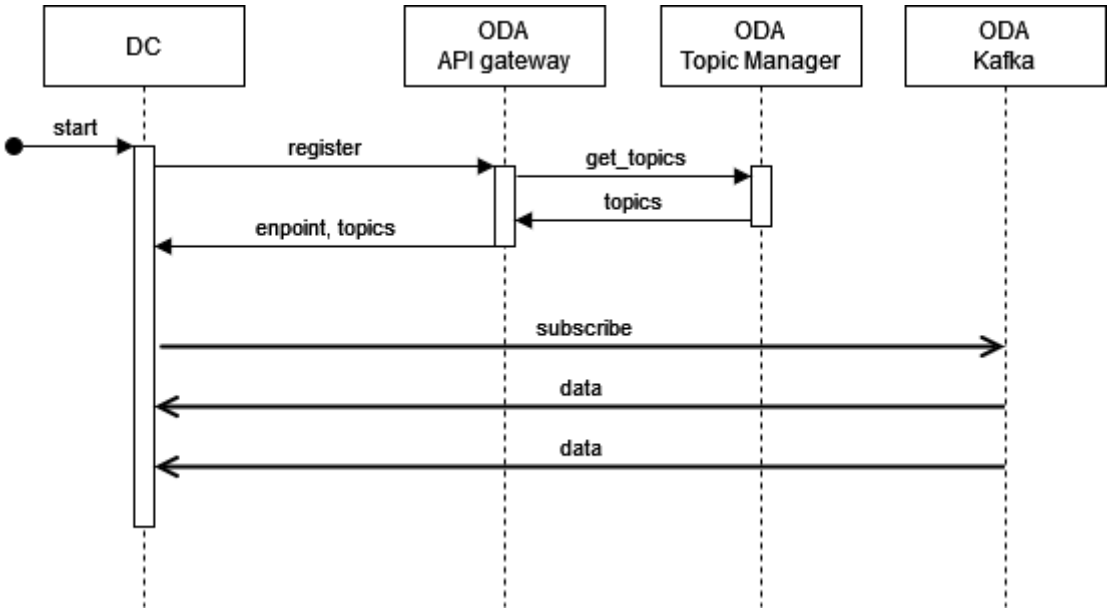

Figure 3: DC receving streamed data

Figure 4 shows how DCs can pull data stored in the DB. They send a query to the API gateway, which propagates the query to the Data Service component. The data retrieved from the database is sent back to the DC along the reverse path.

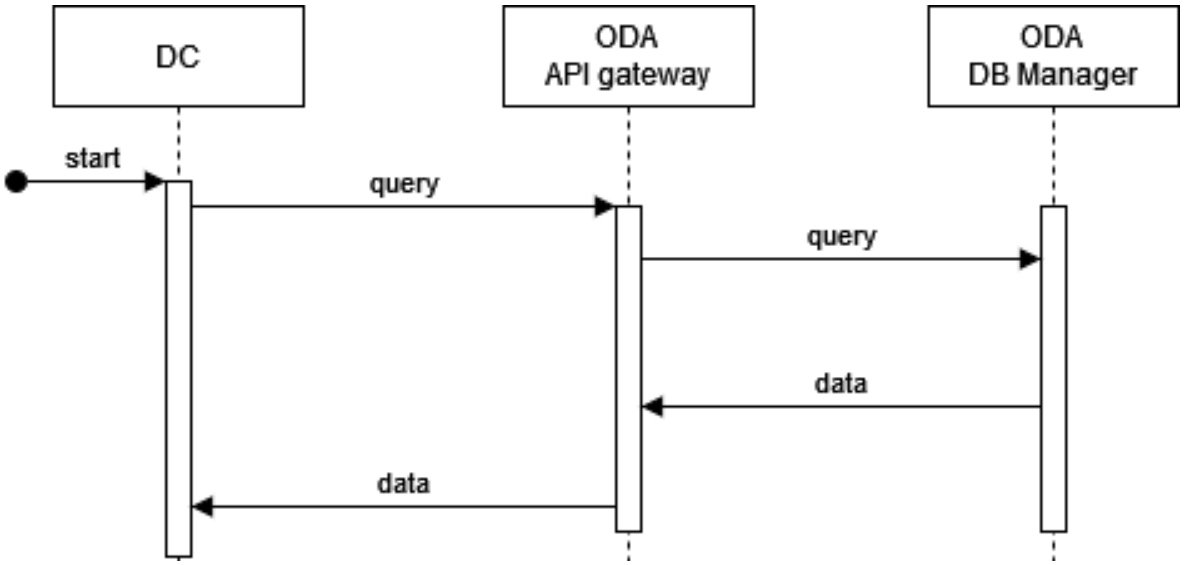

Figure 4: DC queries data stored in the DB

Figure 5 shows how DCs can ask for aggregated data stored in the DB. They send a query to the API gateway, which propagates the query to the Data Aggregator component, which requests data to the data service. The data retrieved from the database is aggregated according to the DC request and sent back to the DC along the reverse path.

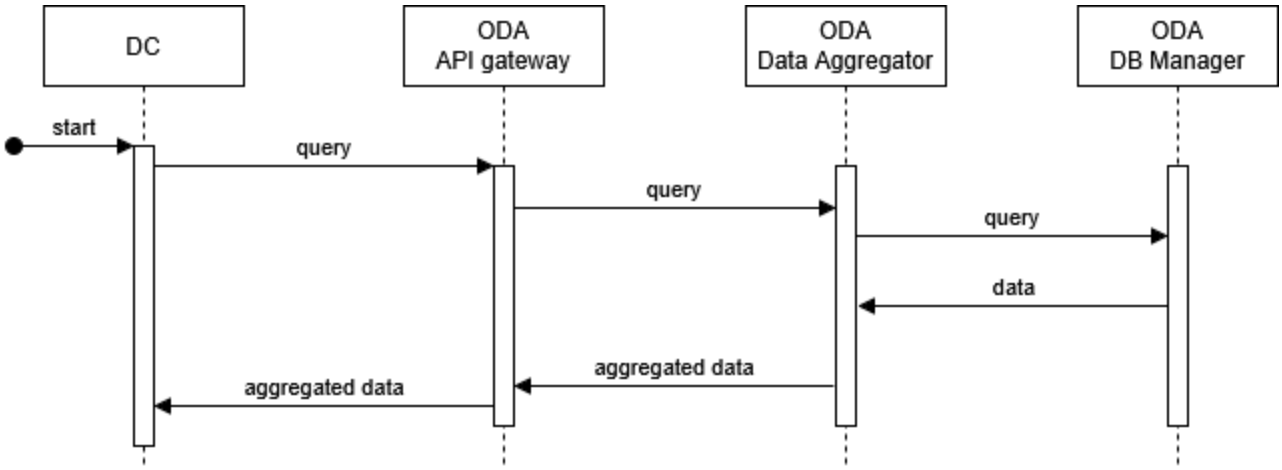

Figure 5: DC queries aggregated data stored in the DB

Data format

The format of data sent to the ODA is extensible to make it easily usable by different DGs. This is obtained by prescribing that all data must include the following mandatory fields:

```
{
  "timestamp": <string>
  "generator_id": <string>
  "topic": <string>
  "data": <string>
}
```

where:

- `timestamp` is the data timestamp (in the ISO 8601 format),
- `generator_id` is a unique identifier string for each ODA representing the generator of the data,
- the `topic` chosen by the DG to label the data, and
- the payload `data` (as a string).

Further structuring of the data can be done by adding further structure within the `data` field.

The de facto standard field `data` used in ODA is proposed by POLIMI:

```
{
 "attribute_name_1": {
  "value": "any supported type",
  "unit": "String or null"
 },
 "attribute_name_2": {
  "value": "any supported type",
  "unit": "String or null"
 },
 ...
}
```

Please refer to the document *ODA_UpTown_DataFormat.pdf* for details.

Aggregation queries

It is possible to query the data stored in ODA and receive aggregated results by adding the "aggregator" field to your query JSON payload.

The "aggregator" field must be an object specifying the aggregation parameters:

```
"aggregator": {
    "fun"          // (Required) Aggregation function
    "field"        // (Required) Name of the field to aggregate
    "unit":        // (Required) Target unit for the result
    "frequency":  // (Optional) Aggregate in time buckets of this size (in minutes)
}
```

The supported value for "fun" are: "sum", "avg", "min", and "max" which are self-explanatory.

When the parameter "frequency" is used, the aggregation is performed in time buckets of the given size in minutes. If it is omitted, aggregation is performed over the entire result set.

Supported Unit Conversions

When the desired target unit is specified in the "unit" field ODA will convert values as needed before aggregation for the following unit:

- Power: W (Watt) ↔ kW (Kilowatt)
- Energy: Wh (Watt-hour) ↔ kWh (Kilowatt-hour)
- Current: A (Ampere) ↔ mA (Milliampere)
- Temperature: Celsius ↔ Kelvin

Example: Aggregated Query (Total Power in kW)

This is an example of an aggregated query for the topic *generic_topic*, with the *sum* of all the *power* fields and converting them in *kW*.

```
curl -X POST http://host:50005/query  \
  -H 'Content-Type: application/json' \
  -d '{
    "topic": "generic_topic",
    "aggregator": {
        "fun": "sum",
        "field": "power",
        "unit": "kW"
    }
}' --output results.gzip
```

The response will be a .gzip archive containing a JSON file with the aggregated results, such as:

```
{
    "timestamp": "2024-10-01T12:00:00Z",
    "generator_id": null,
    "topic": "generic_topic",
    "data": {
        "power": 5.0, // Sum of all power values in kW
        "unit": "kW"
    }
}
```

Example: Aggregated Query with Frequency (Avg. Temperature per 60 Mins)

This is an example of an aggregated query with frequency:

```
curl -X POST http://host:50005/query  \
  -H 'Content-Type: application/json' \
  -d '{
    "topic": "generic_topic",
    "aggregator": {
        "fun": "avg",
        "field": "temperature",
        "unit": "Celsius",
        "frequency": 60
    }
}' --output results.gzip
```

The response will be a .gzip archive containing a JSON file with aggregated results for Each entry in the JSON represents a 60-minute time window and includes the average of all *temperature* readings within that window. Temperatures are converted to *Celsius* before averaging. The timestamp for each object corresponds to the end of the respective 60-minute window. Windows without data will not be included.

For example:

```
{
    "timestamp": "2024-10-01T12:00:00Z",
    "generator_id": null,
    "topic": "generic_topic",
    "data": {
        "temperature": 22.5, // Avg. temperature in Celsius of first 60 minutes
        "unit": "Celsius"
    }
},
{
    "timestamp": "2024-10-01T13:00:00Z",
    "generator_id": null,
    "topic": "generic_topic",
    "data": {
        "temperature": 23.0, // Avg. temperature in Celsius of second 60 minutes
        "unit": "Celsius"
    }
},
{
    "timestamp": "2024-10-01T14:00:00Z",
    "generator_id": null,
    "topic": "generic_topic",
    "data": {
        "temperature": 21.8, // Avg. temperature in Celsius of third 60 minutes
        "unit": "Celsius"
    }
}
```