Deliverable Lead	CMCC
Deliverable due date	2023/03/31
Status	FINAL
Version	V1.0
Project	SEBASTIEN

Co-financed by the Connecting Europe Facility of the European Union

Agreement: INEA/CEF/ICT/A2020/2373580 **Action**: 2020-IT-IA-0234

DOCUMENT INFORMATION

Title	Deliverable n. 3.1 - Report on HIGHLANDER data platform extension implementing SEBASTIEN data lake
Agreement	INEA/CEF/ICT/A2020/2373580
Action	2020-IT-IA-0234
Creator	CMCC
Deliverable Description	Report on HIGHLANDER data platform extension implementing SEBASTIEN data lake
Contributors	Marco Mancini (CMCC), Valentina Scardigno (CMCC), Giuseppe Trotta (CINECA), Francesco Renzi (Nature4.0), Giovanni Vignali (UNITUS)
Requested deadline	M15
Reviewer	Alfredo Reder (CMCC), Federica Gabbianelli (UNITUS)

Index

1. Introduction 5

2. SEBASTIEN Architecture 6

3. Data Storage 7

4. Data Source Connectors 8

4.1. Copernicus C3S Connector 10

4.2. CMCC DDS Connector 10

4.3. Highlander Connector 11

4.4. Copernicus Open Access Hub Connector 13

4.5. MISTRAL Connector 13

4.6. WEkEO Connector 15

4.7. IoT System Connector 17

4.8. LEO Portal Connector 19

5. Backend 20

6. Catalog 22

6.1. MetaDB 23

6.2. API 24

7. Conclusions 25

8. References 26

1. Introduction

The goal of the document is to provide the specifications of the SEBASTIEN Data Lake (and its extension to the HIGHLANDER Data Platform) that allows the integration and coordination of different data sources, from the Copernicus Services and DIASs to OpenData Portal for livestock and environmental information, and IoT Systems for monitoring animal welfare, to provide to SEBASTIEN Services and Data Portal (see Deliverable 6.1) a unified access to the different datasets identified in the Deliverable D2.2 - "*List of suitable data sources and of newly acquired data*".

The Data Lake design and specifications are defined to facilitate the access and processing of huge volumes of data coming from heterogeneous data sources, overcoming data access and interoperability issues without requiring prior knowledge about data structures, file format and backends of each data source that is integrated. In particular, the design and specifications address the following functional requirements: decouple information from the data sources, provide generic data structures to perform analysis, avoid duplication of datasets provided by external data sources, and provide a unified Application Programming Interface (API) to access, process and store datasets coming from heterogeneous data sources and multi-thematic portals.

The document is structured as follows. Section 2 will introduce the overall SEBASTIEN Platform Architecture and gives an overview of the Data Lake architecture. In the subsequent sections 3,4,5 and 6 the design and specifications of the different components of the Data Lake are reported. Finally, the last section will present the conclusions.

2. SEBASTIEN Architecture

This Section introduces the different components of the SEBASTIEN Platform Architecture, particularly the Data Lake specifications that extend the HIGHLANDER Data platform.

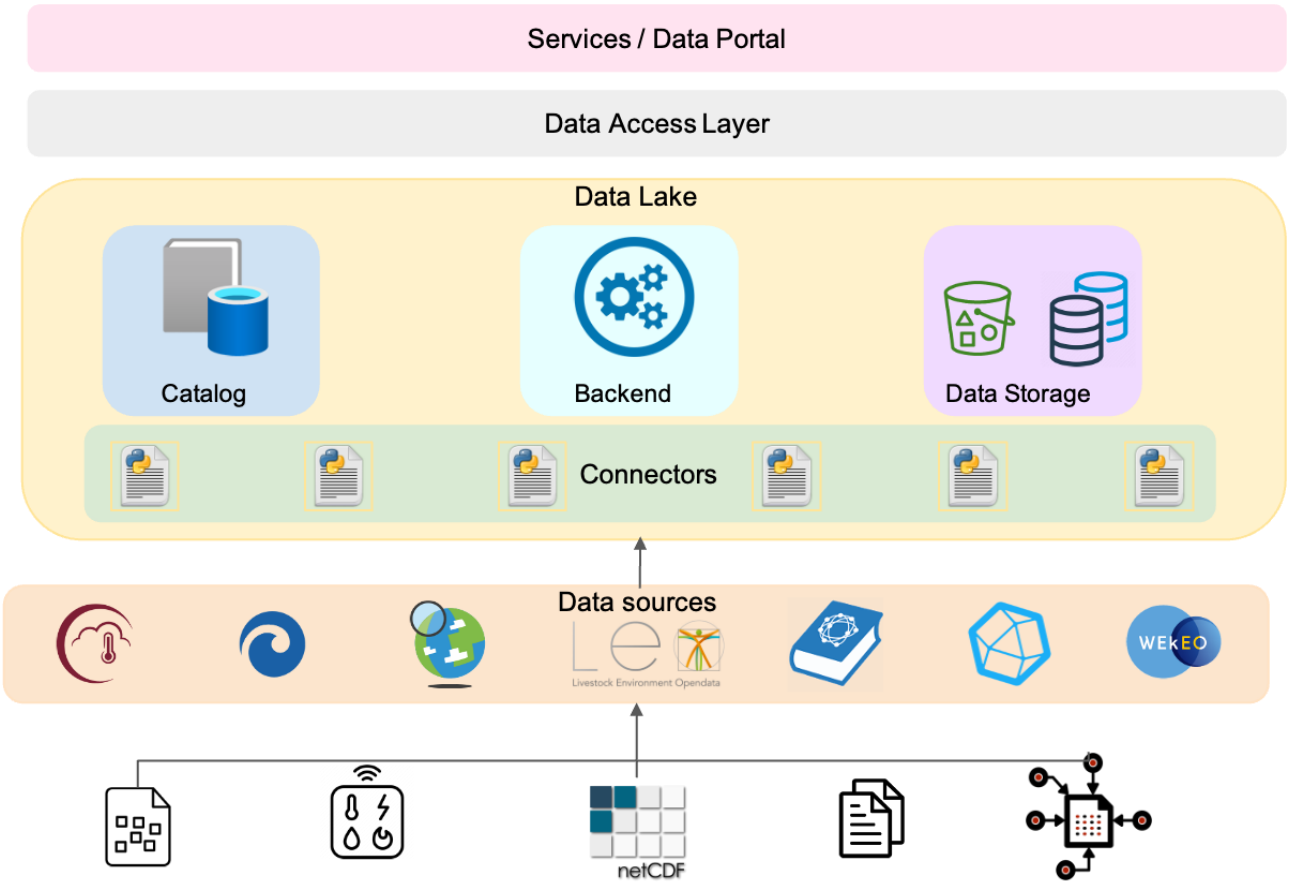

Figure 1: SEBASTIEN Platform Architecture

Figure 1 shows the SEBASTIEN Platform Architecture. The Data Lake is the component that provides the SEBASTIEN Services and Data Portal with a unified way to access the datasets from different and heterogeneous data sources (e.g., Copernicus Services and DIASs, Open Data Portal for livestock information, IoT systems for monitoring animal welfare).

The design and specifications of the Data Lake have been carried out to meet the following requirements: decouple information from the data sources, provide generic data structures to perform analysis, avoid duplication of datasets provided by external data sources and provide a unified way to access, process and store all datasets. SEBASTIEN Services and Data Portal can access the datasets integrated into the Data Lake through the Data Access Layer, whose design and specifications will be described in Milestone 6.

The main components of the Data Lake are the following:

- *Backend* is responsible for managing data requests from the SEBASTIEN clients (Services and Data Portal) through the Data Access Layer.
- *Catalog* is responsible for collecting all the information related to the data source connection and access methods, provided datasets and related metadata (such as dataset identifier and parameters to retrieve data).
- *Connectors* are a set of custom adapters the Backend uses to perform a retrieve query to the right data source when data is requested from the Data Access Layer.
- *Data Storage* is in charge of storing the products developed in SEBASTIEN (indicators and real-time data from IoT sensors) and caching the data of the retrieved requests related to the data sources.

The following sections describe the main elements that constitute the SEBASTIEN Data Lake architecture.

3. Data Storage

The Data Storage is responsible for temporarily or permanently storing data from the different data sources reported in Section 4. In particular, the Data Storage will be used for storing the products developed in SEBASTIEN (indicators and real-time data from IoT Systems for monitoring animal welfare) and for caching data related to the dataset queries from the external data sources such as Copernicus Services and DIASs.

It will be implemented using different technologies according to the data type that has to be stored. In particular, a High Performance Object Storage based on MinIO [1] will be used for storing files in different formats (e.g. netCDF, csv, zip, …) related to the SEBASTIEN indicators and for caching the results of data queries associated with the external data sources, and a time-series DB TimescaleDB [2] for the ingestion and storage of IoT real-time data related to the monitoring of animal welfare produced in SEBASTIEN.

MinIO is an open-source distributed Object Storage released under Apache License v2.0. It is a high-performance cloud-native software designed for large-scale private cloud infrastructure. MinIO aggregates persistent volumes into scalable distributed Object Storage by exposing them using Amazon S3 REST APIs. MinIO uses buckets (as in AWS S3 service) to organize objects (i.e. files), where a bucket is similar to a folder or directory in a filesystem and can hold an arbitrary number of objects, where each object size can range from a few KBs to a maximum of 5TB.

TimescaleDB is an open-source database packaged as a PostgreSQL extension for storing time-series data. The extension model allows the database to take advantage of many of the attributes of PostgreSQL such as reliability, security, and connectivity, to a wide range of third-party tools. At the same time, TimescaleDB leverages the high degree of customization available to extensions by adding hooks deep into PostgreSQL's query planner, data model, and execution engine.

TimescaleDB enables both high ingest rates and real-time analysis queries. It scales by automatically partitioning Hypertable (a single continuous table) into two-dimensional (time and space) proper-sized chunks. Inserts to recent time intervals can be parallelized by placing chunks across clusters or disks based on a specified partition key. Complex queries can be optimized by leveraging the metadata of each chunk.

4. Data Source Connectors

To integrate heterogeneous data sources (i.e. external data portals and services, such as Copernicus Services and DIASs, files stored in Object Storages, and time series from Databases), specifically designed connectors will be devised to access and retrieve data. Multiple techniques and optimizations will be used in the connectors, such as caching, multi-dimensional subsetting on the data source side and efficient in-memory access.

In the following table, we report for each data source the datasets available, the related technologies to build the connector and the format of the retrieved result.

Table 1: Data Sources Connectors

Data Source	Available Datasets	Connector Technologies	Data Format
Copernicus C3S	*E-OBS Gridded Observations* *ERA5 Reanalysis* *ERA5-Land Reanalysis* *UERRA (Regional Reanalysis for Europe from 1961 to 2019)* *CERRA (Sub-daily Regional Reanalysis data for Europe from 1984 to the present)* *C3S Seasonal forecast data* *EURO-CORDEX data*	CDS API Python client (`cdsapi`) [3]	GRIB NetCDF
CMCC DDS	*E-OBS Gridded Observations* *ERA5 Reanalysis* *ERA5-Land Reanalysis* *EURO-CORDEX data* *VHR-REA_IT Dataset* *ITALY8km-CM data* *VHR-PRO_IT Dataset*	DDS Python client (`ddsapi`) [4]	NetCDF

HIGHLANDER	*IOT animal sensor data* *VHR-REA_IT Dataset* *VHR-PRO_IT Dataset*	Python package Requests (`requests`) [5]	NetCDF
Copernicus Open Access Hub	*Sentinel-1* *Sentinel-2*	Python package Requests (`requests`)	NetCDF
MISTRAL	*COSMO-2I: COSMO at 2.2km – Italy area* *COSMO-5M: COSMO at 5km – Mediterranean Region* *Multimodel ensemble forecast by Arpa Piemonte* *Surface Rainfall Intensity from Radar-DPC* *Italy Flash Flood*	Python library S3Fs (`s3fs`) [6]	GRIB
WEkEO	*ERA5 Reanalysis* *ERA5-Land Reanalysis* *UERRA (Regional Reanalysis for Europe from 1961 to 2019)* *C3S Seasonal forecast data* *Sentinel-1* *Sentinel-2* *EU Digital Elevation Model (EU-DEM)*	HDA Python client (`hda`) [7]	NetCDF
IoT	*AnimalTalker data* *TT_air data*	Paho Python client (`paho-mqtt`) [8] PostgreSQL database adapter (`psycopg2`)[9]	Database recordset
LEO Portal	*Climate data* *Precision Livestock Farming (PLFData)* *Laboratory data* *Wellness* *Genetic data* *Collected data*	SPARQL Python client (`sparql-client`) [10]	JSON CSV

The following subsections report the detailed specifications for the different connectors for each data source reported in the previous table.

4.1. Copernicus C3S Connector

The C3S Climate Data Store (CDS) [11] is a one-stop shop for a trusted source of climate data, tools and information at the global and European levels. It includes the datasets containing weather and climate information reported in Table 1.

Copernicus License allows access to datasets in an open, free, and unrestricted way. Users can easily retrieve data using the CDS API Python client. Once the CDS API key and client are correctly installed, they can be used to query datasets listed in the CDS catalog. The following code shows the syntax to use for the API call to request a portion of ERA5 data:

```
import cdsapi
c = cdsapi.Client()
c.retrieve(
    'reanalysis-era5-single-levels',
    {
        'product_type': 'reanalysis',
        'format': 'netcdf',
        'variable': '2m_temperature',
        'year': '2023',
        'month': '01',
        'day': '01',
        'time': '00:00',
    },
    'download.nc')
```

The connector can be implemented as a Python module using the CDS API Client python library to perform the request and uses the Data storage to cache the result.

4.2. CMCC DDS Connector

The CMCC Data Delivery System (DDS) [12] provides a unique, consistent and seamless access point for all data produced and used by CMCC through a unified API interface.

The user can browse the Catalog and the available datasets of seasonal forecasts, decadal predictions, climate change projections and much more through the DDS Web Portal.

Through the DDS Web User Interface, users can browse the Catalog and the available datasets (e.g., seasonal forecasts, decadal predictions, climate change projections, and others). They can quickly build queries by choosing the values to assign to the dataset parameters. Then, according to these criteria, users can access and download data using the DDS Python client. The syntax to use for the API call is similar to this:

```
import ddsapi
```

```
c = ddsapi.Client()
c.retrieve("e-obs", "ensemble-mean",
{
    "variable": [
        "mean_air_temperature"
    ],
    "time": {
        "year": [
            "2021"
        ],
        "month": [
            "1"
        ],
        "day": [
            "1"
        ]
    },
    "version": "v25.0e",
    "format": "netcdf",
    "resolution": "0.1"
},
"e-obs-ensemble-mean.nc")
```

The connector can be implemented as a Python module using the DDS API Client python library to perform the request and uses the Data storage to cache the result.

4.3. Highlander Connector

Highlander aims to build a comprehensive and multi-sector framework for land-management decision-making in Italy. It implements a framework of multi-thematic data, indicators, and tools based on diverse approaches, from remote and in-situ monitoring to analytical tools, numerical models, and machine learning algorithms, to be directly exploited by a wide range of users through dedicated services.

The component that makes those results accessible is the Data Delivery System (DDS) which exposes the collected datasets through a restful API.

Here are the definitions of the APIs currently implemented and available for “data extraction”:

Method	Path	Description
GET	api/datasets	Get all configured datasets
GET	api/datasets/{dataset_id}	Get a dataset by ID

GET	api/datasets/{dataset_id}/image	Get the dataset image thumbnail
GET	api/datasets/{dataset_id}/products/{product_id}	Get a dataset product by ID
POST	api/requests/{dataset_id}	Submit a data extraction request for a dataset
GET	api/requests	Get all user requests
GET	api/download/{timestamp}	Download result data

A request example for the POST in the table above can be:

```
{
  "product":"VHR-REA_IT_1989_2020_hourly",
  "variables":[
        "specific_humidity",
        "lwe_thickness_of_moisture_content_of_soil_layer"
  ],
 "time": {
        "day": ["1", "2"],
        "month": ["1"],
        "year": ["1991"],
  },
   "format":"netcdf"
}
```

Those APIs can be accessed through a web portal upon user registration and can also be exploited from a command line tool using the HL-DDS client.

Users can create a data query by giving the dataset name and the request body as a JSON file and then submit the request as follows:

```
$ hld-dds-cli post DATASET_ID QUERY_FILE
```

The results can be downloaded from successful queries by providing the Request ID:

```
$ hld-dds-cli get REQUEST_ID
```

The connector to the HIGHLANDER data source will be a Python module using the package Requests to communicate with the restful API available for “data extraction” listed above. Once the request is completed, the connector will use the data storage to cache the result.

4.4. Copernicus Open Access Hub Connector

The Copernicus Open Access Hub [13] provides complete, free and open access to Sentinel-1 and Sentinel-2 user products.

The OData interface is a data access protocol built on core protocols like HTTP and methodologies like REST that common web browsers, download-managers or computer programs such as cURL or Wget can handle. The data access mechanism consists of building URI for performing search queries and product downloads. Full authentication is required to access the API.

The OData URI addressing the resource /Products provides the list of entries of the individual entity /Products('Id') corresponding to the data files stored in the Data Hub archive.

To download a full product, the syntax is like this:

https://scihub.copernicus.eu/dhus/odata/v1/Products('ed5a5a5e-bee2-4ae7-bdb9-1d52849f1a7a')/$value

The connector module will use the Python package Requests to communicate with OData API as it allows sending HTTP/1.1 requests extremely easy. After getting the result, it will be cached in the Data Storage.

4.5. MISTRAL Connector

The MISTRAL portal aims to facilitate and foster the re-use of the datasets by the weather community and its cross-area communities to provide added value services through HPC resources, turning them into the level of new business opportunities.
The main Meteo-Hub service allows you to create and download your own collection of meteorological data chosen from different forecast models and measurements from a multitude of weather stations with different parameters and time validity ranges.
There are different ways to download data from Meteo-Hub service.
Registered users can obtain data from one or more datasets using the “Data Extraction” feature through the **Web Portal**. In the first step, the user must select one or more datasets (multiple selections are allowed only with datasets of the same category: observation, forecast, radar). In the second step, the user can filter against the available parameters specific to the dataset category. If the user can use post-processing tools, they can be applied in the third step. The set of post-processing tools is specific to the dataset category, too. Some include “time post-processing”,

“space post-processing” and format conversion. Open data packages can be downloaded for all users, even those not logged in. The packages are produced daily and are available on the Meteo-Hub main page.

The open data from the observational datasets can be downloaded from the map of Observations webpage.

Another way to access data is through a **client application** that allows users to perform the main operations on the Meteo-Hub platform from the command line. The application lets users download data extracted by an immediate or scheduled request prepared interactively on the Meteo-Hub web interface. It also provides a wait-and-download functionality for users accessing an AMQP data-ready queue.

Finally, users can directly use Meteo-Hub APIs to create and schedule data extraction requests and download the output data. The APIs are documented using Swagger and are accessible at this the following url:

https://meteohub.mistralportal.it:7777

In order to create a request for data extraction, the following API has to be used:

```
POST https://meteohub.mistralportal.it/api/data
```

The request model:

```
{
 "request_name": name-of-the-request,
 "reftime": {
   "from": "2022-01-10T11:04:54.615Z",
   "to": "2022-01-10T11:04:54.615Z"
 },
 "dataset_names": [ … ],
  "filters": {
    "area": [],
    "level": [],
    "origin": [],
    "proddef": [ ],
    "product": [],
    "quantity": [],
    "run": [],
    "task": [],
```

```
    "timerange": [],
     "network": []
    },
    "postprocessors": [ … ]
}
```

In the request body, the metadata related to the desired data has to be specified, and the filters are specific to the dataset category. If the data extraction was successful, the fileoutput parameter in the response indicates the name of the output file.

To download the output data the following API has to be used:

```
GET https://meteohub.mistralportal.it/api/data/{filename}
```

where the query parameter filename specifies the file to download.

In case of the SEBASTIEN's services, the ideal solution is to exploit the data extraction request scheduling feature available on the MISTRAL portal. The output files of the scheduled query will be automatically produced on the MISTRAL portal according to the SEBASTIEN services requirements. Using a custom ingestion process to download the output files from the MISTRAL portal, SEBASTIEN will be able to mirror the data into the MinIO Data Storage component.

The MISTRAL Connector will be developed by extending the HIGHLANDER drivers already implemented for netCDF files stored on POSIX-compliant file system to support S3 backend storage such as MinIO; those drivers will be based on the Python library S3Fs.

4.6. WEkEO Connector

The WEkEO DIAS [14] provides users with a single distributed tool for accessing, visualizing and analyzing all Copernicus data and services, including big-data analysis tools, to develop applications tailored to their specific needs.

WEkEO's strength is based on its federated infrastructure built on the existing Copernicus organization in EUMETSAT, ECMWF, Mercator Ocean International and EEA centres. This approach gives users direct access to work with the most up-to-date Copernicus data instead of relying on archive data sets.

The Harmonized Data Access (HDA) protocol allows the user to access the data. It can be used through the API available on the WEkEO portal (web interface), or directly from a virtual machine

or a Jupyter Notebook using python scripts. The simple way is to use the HDA Python client library, which abstracts away the API's details.

Once the library and the configuration file are installed properly, the user can run a Python script to query and download data, as in the following code:

```
from hda import Client
c = Client(debug=True)
query = {
  "datasetId": "EO:ESA:DAT:SENTINEL-3:OL_2_LFR___",
  "boundingBoxValues": [
    {
      "name": "bbox",
      "bbox": [
        -7.499583333333334,
        42.3020216796485,
        11.999333333333333,
        54.75429195536845
      ]
    }
  ],
  "dateRangeSelectValues": [
    {
      "name": "position",
      "start": "2021-01-01T00:00:00.000Z",
      "end": "2021-01-15T00:00:00.000Z"
    }
  ],
  "stringChoiceValues": [
    {
      "name": "productType",
      "value": "LFR"
    },
    {
      "name": "processingLevel",
      "value": "LEVEL2"
    }
  ]
}
matches = c.search(query)
matches.download()
```

The connector can be implemented as a Python module using the HDA Python Client library to perform the request and uses the Data storage to cache the result.

4.7. IoT System Connector

Deep and continuous phenotyping of animals, using IoT (Internet of Things) sensors, is important to evaluate animal welfare and avoid stress conditions. At the same time, sensors can be used to send real-time warnings based on animal and environmental conditions. In particular, as described in Deliverable D2.2, data will be collected from two device types. The AnimalTalker is a device for animal welfare and collects data on movements of the neck and leg, animal position, animal temperature and environmental (air) temperature and relative humidity. The measurement of the heartbeat will be added during the project. The parameters are collected by multiple sensors installed on different parts of the animal body.

On the other hand, TTair is an air quality monitoring device that can analyze gas concentrations in barns or open air. In particular, it collects the concentration of CH_4, CO_2, H_2S NH_3, Particular Matter (1, 2.5 and 10), air temperature and relative humidity. The devices data are published on a Message Queuing Telemetry Transport (MQTT) broker located on the UNITUS server as json formatted message.

The arguments of each AnimalTalker json packet are timestamp, group uuid, sensor uuid, sensor type and measures. A single group can have multiple sensors of various types. The type of the sensor defines the data contained in the measures field. The “timestamp” is provided in Unix time format which is the number of seconds since 00:00:00 UTC on 1 January 1970.

Example of AnimalTalker payload (single value):

```
'{
"timestamp":1600000000,
“animal_id”:00000001,
“sensor_uuid”:00000001,
“sensor_type”: “temperature”,
"measures":{ “value”:10.25 }
}'
```

Example of AnimalTalker payload (multiple values):

```
'{
"timestamp":1600000000,
“animal_id”:00000001,
“sensor_uuid”:00000001,
“sensor_type”: “accelerometer”,
"measures":{“accelerationX”:0,      “accelerationY”:0,      “accelerationZ”:4095,
“stdevX”:50, “stdevY”:10, “stdevZ”:30}
```

```
}'
```

TTair Json packet arguments are device id, timestamp, measurement id and the values collected by the device. The timestamp format is Unix time.

Example of TTAir payload:

```
'{
“device_id”:00000001,
"timestamp":1600000000,
“n_misura”: 3,
“CH4”: 800.52,
“CO2”: 1200.78,
“H2S”: 500.49,
“NH3”: 750.22,
“PM1”:500.78,
“PM2.5”:600.12,
“PM10”:1050.41,
“T”:30.15,
“H”:60.25,
“battery”:12538
}'
```

The MQTT broker has a topic associated to each device type:

1. AnimalTalker
2. TT_air

This solution avoids complicating the broker structure because, given the device type, all the other information can be retrieved from data packets.

In order to allow the connection to the IoT data, it is expected to use the MQTT endpoint provided by the system and to ingest data into a TimescaleDB. This component will be integrated into the SEBASTIEN data lake API to expose real-time data.

This is a simple MQTT client example that performs the subscribing to a topic and then the data ingestion into TimescaleDB:

```
import paho.mqtt.client as mqtt
import psycopg2

CONNECTION = "postgres://username:password@host:port/dbname"
```

```
SQL = "INSERT INTO data (timestamp, animal_id, sensor_uuid, sensor_type, value)
VALUES (%s, %s, %s, %s, %s);"

def write_data(data):
    try:
       conn = psycopg2.connect(CONNECTION)
       cur = conn.cursor()
       cur.execute(SQL, data)
    except (Exception, psycopg2.Error) as error:
        print(error.pgerror)
    conn.commit()
    cur.close()

def on_connect(client, userdata, flags, rc):
   # This will be called once the client connects
   print(f"Connected with result code {rc}")
   client.subscribe("TT_air")

def on_message(client, userdata, msg):
   print(f"Message received [{msg.topic}]: {msg.payload}")
   write_data(msg.payload)

client = mqtt.Client("mqtt-IoT-data") # client ID
client.on_connect = on_connect
client.on_message = on_message
client.username_pw_set("username", "password")
client.connect(url, port)
client.loop_forever()
```

The connector can be implemented as a Python module that allows access to data ingested in the TimescaleDB through the PostgreSQL database Python adapter `psycopg2`.

4.8. LEO Portal Connector

The LEO (Livestock Environment Opendata) Portal [15] collects all the livestock-related information in Italy in a single and open-access database. Its main aim is to support and improve the quality of this sector, reducing, at the same time, the impact on animal wellness and the environment.

The Portal offers the possibility of querying and retrieving data using a SPARQL Endpoint, which is a Point of Presence on an HTTP network capable of receiving and processing SPARQL Protocol requests. The SPARQL Query Language is a Declarative Query Language (like SQL) for performing Data Manipulation and Data Definition operations.

The following code is an example of a query that retrieves the number of farms and slaughterhouses for each province:

```
SELECT ?uriProvincia ?nomeProvincia ?siglaProvincia ?codiceIstatProvincia
?numAziende (count(?macello) as ?numMacelli)
WHERE {
    {
```

```
        SELECT ?uriProvincia ?nomeProvincia ?siglaProvincia ?codiceIstatProvincia
(count(?azienda) as ?numAziende)
        WHERE {
            ?uriProvincia l0:name ?nomeProvincia ;
            clv:acronym ?siglaProvincia ;
            skos:notation  ?codiceIstatProvincia .
            OPTIONAL {
                ?azienda a co:LivestockFarm ;
                co:operatesIn ?uriProvincia .
            }
        }
    }
    OPTIONAL {
        ?macello a co:Slaughterhouse ;
        co:operatesIn ?uriProvincia .
    }
}
ORDER BY ASC(?nomeProvincia)
```

The connector will be based on the SPARQL Python Client (`sparql-client`) since it is able to perform SELECT and ASK queries against a SPARQL endpoint via HTTP. Once the connector has executed the request, the resulting data will be cached in the Data Storage.

5. Backend

The Backend main task is to receive data requests coming from the Data Access Layer and execute them interacting with the other components of the Data Lake.

The retrieval process for accessing the data requested can be implemented both as an asynchronous and synchronous process. To describe how the different components communicate with each other, starting from the user request until the data is provided, a Unified Modeling Language (UML) Sequence diagram is provided for both types of process.

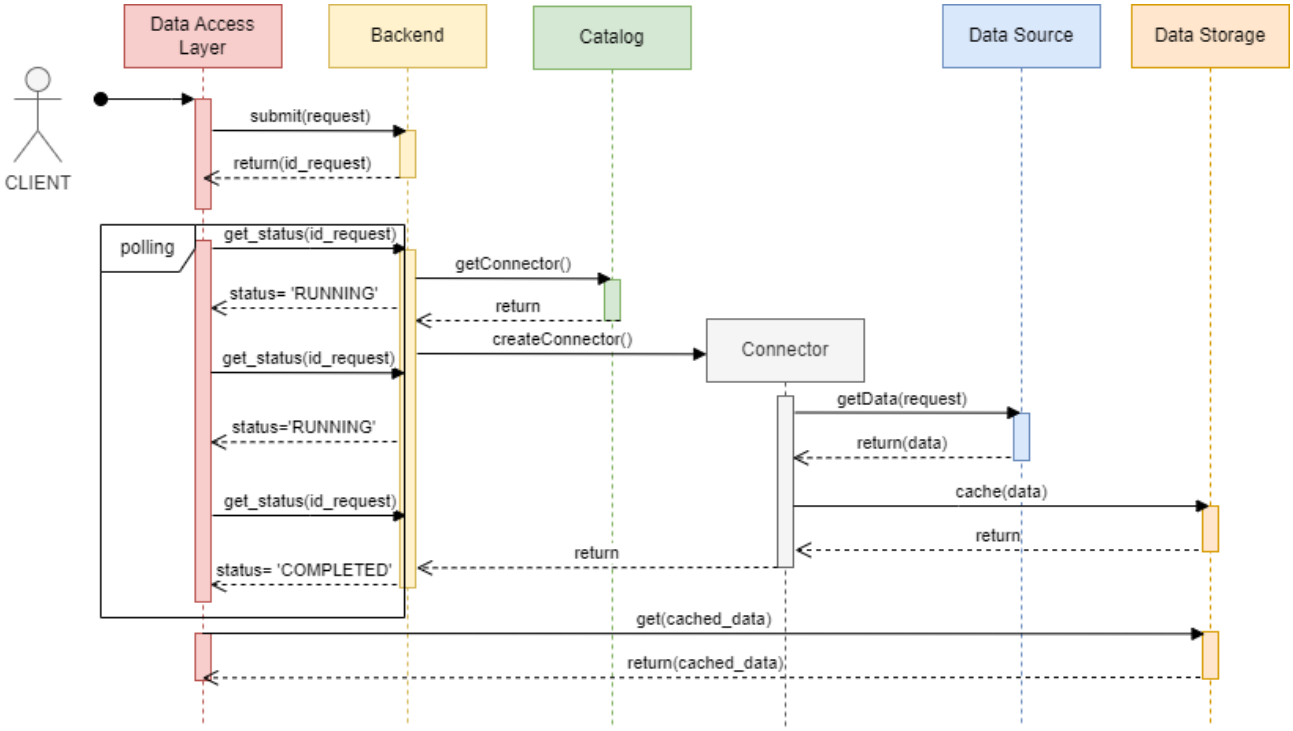

Figure 2: Sequence Diagram for retrieving data from the Data Lake using the asynchronous mode

The sequence diagram related to the asynchronous process is shown in Figure 2, where the operations and communication of the different Data Lake components involved are reported when a data request from the Data Access Layer is related to an external Data Source.

In particular, the Data Access Layer submits the request to the Backend. Then, with the request identifier returned, it will poll the Backend at regular intervals (e.g., every two seconds) to check the request's status until it has been completed.

To manage the request, the Backend interacts with the Catalog to obtain the information related to the data source and the connector for accessing the corresponding dataset. Thus, the Backend creates the Connector object, which will execute the query towards the datasource of the corresponding dataset in the request and cache the result in the Data Storage.

Once the request is completed, the Data Access Layer receives from the Backend information related to the location of the cached data in the Data Storage and proceeds with the download from the MinIO component.

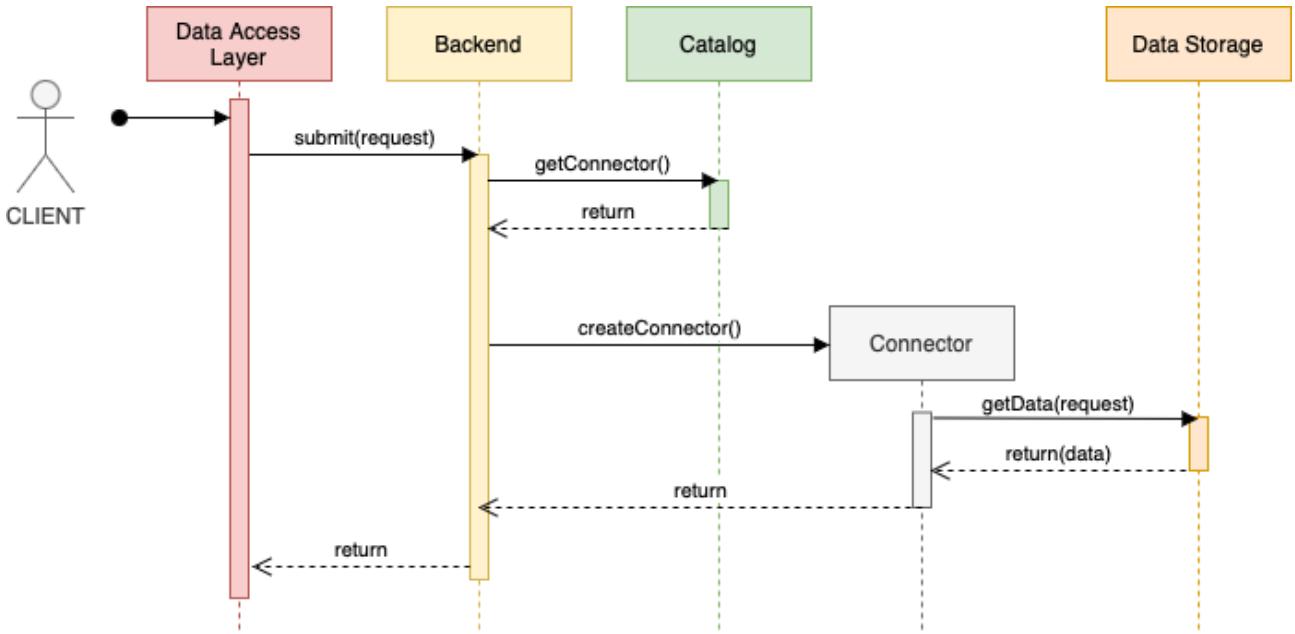

Figure 3: Sequence Diagram for retrieving data from the Data Lake using the synchronous mode

Figure 3 shows the sequence diagram related to the synchronous process; this workflow occurs for accessing and analyzing streamed IoT real-time data stored in TimescaleDB.

A synchronous operation implies that the Data Access Layer submits the request to the Backend and remains blocked until the request is completed. In processing the request, interaction occurs between the Backend and the Catalog to retrieve the information related to the data source and the connector for accessing the corresponding dataset. Thus, the Backend creates the Connector object, which will execute the query directly towards the Data Storage (e.g. TimescaleDB component), and the result is returned to the Data Access Layer.

6. Catalog

This paragraph describes the design and the specifications of the Catalog element within the Data Lake. This component aims to store information about the different data sources described in Section 4: connection and access mechanisms, datasets available in the data source, datasets metadata (such as dataset identifier within the data source and parameters to retrieve data) and the data source connector.

The Catalog stores the information within a database (called MetaDB) and, through an API, exposes different operations that the Backend can use to retrieve information for executing the data requests.

The following paragraph reports detailed specifications about the MetaDB and the Catalog operations.

6.1. MetaDB

From the requirements analysis of the MetaDB, the involved actors that need to be represented as entities are: the data source and the datasets available in each data source. Furthermore, since the metadata schema can differ for each dataset, a NoSQL database is a good candidate for the MetaDB implementation.

MongoDB [16] offers the following potential benefits among the different NoSQL solutions.

- It is used for high-volume heterogeneous data storage, helping organizations store large amounts of data while performing rapidly.
- It doesn't require predefined schemas and stores any type of data. This gives users the flexibility to create any number of fields in a document, making it easier to scale MongoDB databases compared to relational databases.
- It is document-oriented, with the advantage that these objects map to native data types in several programming languages. Having embedded documents also reduces the need for database joins, which can lower costs.

MongoDB stores data as records that are made up of collections and documents. Documents contain the data the user wants to store as objects in the collections; each object in MongoDB is similar to a JSON (JavaScript Object Notation) document composed of field/value pairs. Documents also include a primary key as a unique identifier (called *_id*) within the collection. A document's structure changes by adding or deleting new or existing fields.

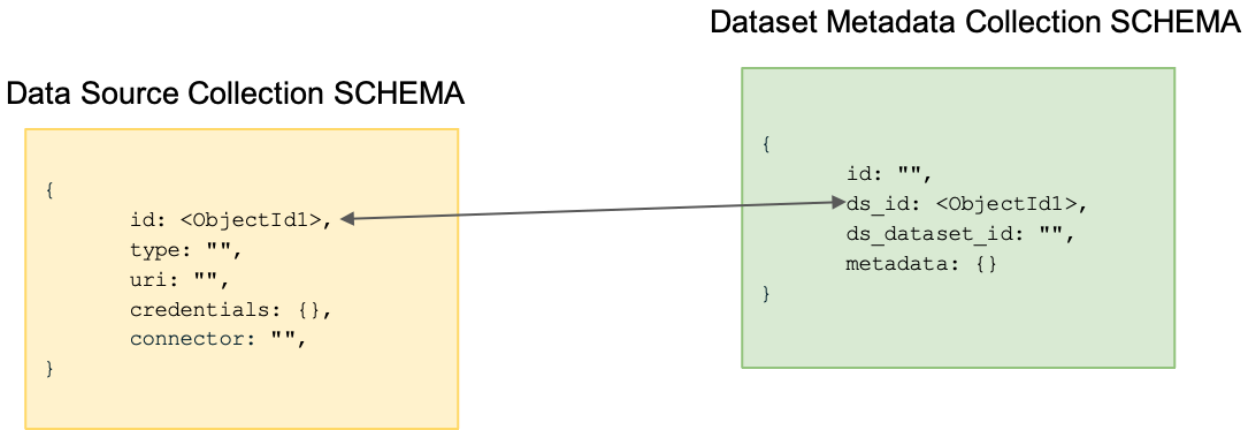

Figure 4: MetaDB Data Model

In Figure 4, the MetaDB Data Model schema is reported using the data abstractions provided by MongoDB.

The Data Source collection contains all the objects related to the data sources with the following schema fields:

- *id*: unique identifier of the data source;
- *type*: type of the data source, i.e., Data Provider (e.g., Copernicus C3S), MinIO Object Storage, TimescaleDB;

- *uri*: the uri for connecting to the data source;
- *credentials*: a dictionary with information about the credentials for the authentication to the data source (e.g. username and password, api-key, and others);
- *connector*: the Python module that the Backend can use to access and retrieve data from the data source.

The Dataset collection contains all the objects related to the information for each dataset with the following schema fields:

- *id*: unique identifier of the dataset;
- *ds_id*: the reference to the data source that provides access to the dataset; (i.e., id in the Data Source collection schema);
- *ds_dataset_id*: the dataset identifier in the data source;
- *metadata*: the dictionary containing key/value pairs related to the parameters (and their values) that are needed to retrieve the dataset from the data sources.

The MetaDB data model can be stored in the MongoDB database as two collections related to the data source and the dataset entities. Each document in the data source collection represents each data source described in Section 4. In contrast, the documents in the dataset collection will contain all the metadata information of the datasets available in the data sources.

In order to ingest the datasets metadata in the MetaDB, two potential methods can be used for indexing.

The first solution uses the data source API if it provides information about the datasets metadata. In the case of Wekeo, CMCC DDS, and HIGHLANDER the API provides endpoints to extract metadata for a particular dataset. For example, in WEkEO, it is possible to use the HDA API to get a JSON document of the full list of attributes for a particular dataset by using `GET /querymetadata/{datasetId}`.

If a data source does not provide an API to access information about its own catalog, the alternative solution is to design and implement a customized web crawler that will search and automatically index and extract metadata from the data source website.

6.2. API

The Catalog provides an API that allows the Backend to obtain the information related to data sources and datasets; in particular, the following operations are provided by the Catalog:

- **get_datasets(id)**: return the list of available datasets for a specific data source identified by *id*;
- **get_datasource(id)**: return the data source related to the dataset identified by *id*;
- **get_connector(id)**: return the connector pertaining to the data source identified by *id*;
- **get_metadata(id)**: return the list of parameters for a given dataset identified by *id*;

- **get_values(id, name)**: return the values of a parameter identified by *name* related to the dataset identified by *id*.

The Catalog can perform those operations using the Mongo Client available in PyMongo [17]; it allows easy interaction between Python and MongoDB through the main query clauses, such as *find* and *distinct*, to retrieve information about datasets and data sources.

7. Conclusions

This document provides the design and specifications of the SEBASTIEN Data Lake and its extension with respect to the HIGHLANDER Data Platform. The main innovation in comparison with HIGHLANDER is the enhancement of the Catalog abstraction. This latter allows us to integrate multiple data sources, avoiding data duplication and facilitating the development of the SEBASTIEN Services and Data Portal, as it allows the usage of a uniform environment for accessing and processing heterogeneous datasets.

The document describes in detail the specifications of all the architecture components that will be developed during the project in Activity 3. A prototype will be released based on the already identified technologies considered in the architecture design, allowing the access, management and processing of large amounts of heterogeneous data provided by different types of data sources for developing and delivering the SEBASTIEN Services and Data Portal.

8. References

[1] MinIO, website: https://min.io

[2] TimescaleDB, website: https://www.timescale.com

[3] CDS API Python Client, website: https://anaconda.org/conda-forge/cdsapi

[4] DDS API Python Client, website: https://pypi.org/project/ddsapi

[5] Python Requests, website: https://pypi.org/project/requests

[6] S3Fs, website: https://s3fs.readthedocs.io/en/latest/

[7] HDA Python Client, website: https://www.wekeo.eu/docs/hda-python-lib

[8] Paho Python Client, website: https://pypi.org/project/paho-mqtt

[9] PostgreSQL database Python adapter, website: https://pypi.org/project/psycopg2

[10] SPARQL Python Client, website: https://pypi.org/project/sparql-client

[11] Climate Data Store - Copernicus, website: https://cds.climate.copernicus.eu

[12] CMCC Data Delivery System - DDS, website: https://dds.cmcc.it

[13] Copernicus Open Access Hub, website: https://scihub.copernicus.eu

[14] WEkEO DIAS, website: https://www.wekeo.eu

[15] LEO Portal, website: https://opendata.leo-italy.eu/portale/sparql

[16] MongoDB, website: https://www.mongodb.com

[17] Mongo Python Client, website: https://pymongo.readthedocs.io/en/stable