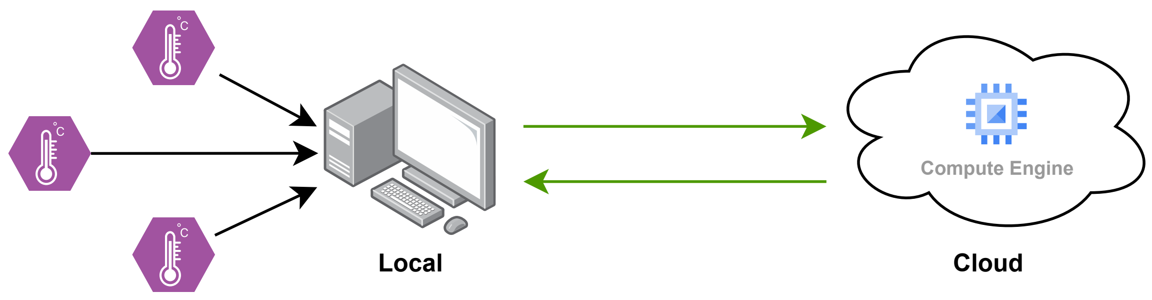
# Fog Computing
## TU Berlin - Summer Semester 2022

Prototype Documentation

B4rtware, Re-Krass

13.07.2022



Repository link:
https://github.com/fog-computing-prototype/fog-computing-prototype

## Introduction

The goal was to design and implement an application that has to cope with various fog-specific challenges. The main objective was to manually implement the reliable delivery of messages. For this purpose, we have implemented a local component and a cloud component.

## Local component & Cloud component

The whole application sends data back and forth between two clients, i.e. between the local component and the cloud component (see Fig. 1). The local component, which in our case runs on our own computer and receives data from virtual sensors.

The cloud component runs on the Google Cloud Platform (GCP), or more precisely, it runs there on a Google Compute Engine (GCE) instance and is made publicly available via firewall rules. The implemented shell scripts enable easy deployment of the cloud application to GCE. Thus, it fulfills the 1st requirement (Fig. 2).

## Data generation (Sensors)

Various sensors are simulated within the local component. In our case, we implemented this via inheritence. With the help of a YAML file (`sensor.yaml`, Fig. 1) it is possible to define individual sensors. Each of these sensors has a freely selectable name and an interval in seconds that specifies how often data should be produced. Two sensor classes have been defined via the sensor class. A temperature sensor and a humidity sensor (Fig. 1). The 2nd requirement is therefore also fulfilled (Fig. 2).

## Data transmission (Local $\Longleftrightarrow$ Cloud)

Each of these sensors periodically sends data to the cloud component via the local component at a predefined interval (in our case: every 10 seconds (Code 4 l. 70)). The cloud component, in turn, regularly sends statistical messages to the local component (in our case: every 5 seconds). Also, the cloud responds to any message from the local component with the sequence received.

JSON objects are sent in both directions via ZeroMQ (Code 4 l. 29 in local component). The sensor data sent by the local component and received by the cloud component are shown in Code 2. In the other direction the scheme in Code 3 is used. Thus, the 3rd requirement is fulfilled (Fig. 2).

## Disconnects (Preserving data)

Each of these components uses a local cache database to cache messages to be sent (Fig. 1). This helps against its own failure. For example, if the local component fails, the cloud component continues to produce statistical data, that is sent in full to the local component on reconnect, when the local component reconnects to the cloud component. To realize this, a database is used as a cache to persistently store messages to be sent. Once the response with the correct sequence is received from the local component, the message to be sent is deleted (similar to the local component in Code 4 l. 36, l. 51).

This procedure is also used for the other direction to ensure reliable messaging. In this case, however, a sequence is incremented for all sensors so that each sensor value gets a unique sequence. Thus, the 4th requirement of reliable messaging is also fulfilled (Fig. 2).

## Technology stack

The application was implemented with the help of various technologies. We use the JavaScript framework Vue.js together with TypeScript for the dashboard (Fig. 3). The names of the sensors are listed on the left and the graphical evaluation of the last 100 data points on the right. On the settings page (Fig. 4) you can set the API endpoint URL.

The local and cloud components were developed using the Python programming language. For the communication between the two components we use the universal messaging library ZeroMQ. The cloud component also has a small REST API (Fig. 5) to provide access to the persistently stored sensor data that the dashboard accesses to display the data. The cache and persistent storage were implemented using a database. To keep the complexity low, we decided to use SQLite.

In order to be able to test the application comfortably locally, a docker-compose file is located in the repository, which executes all three components together. To be able to start each component individually with docker-compose, there is an additional docker-compose file in each component. The images are each built by a docker file, which is also located in the component folders.

The exact list of all technologies we use can be found in Fig. 6.

## Appendix: Figures

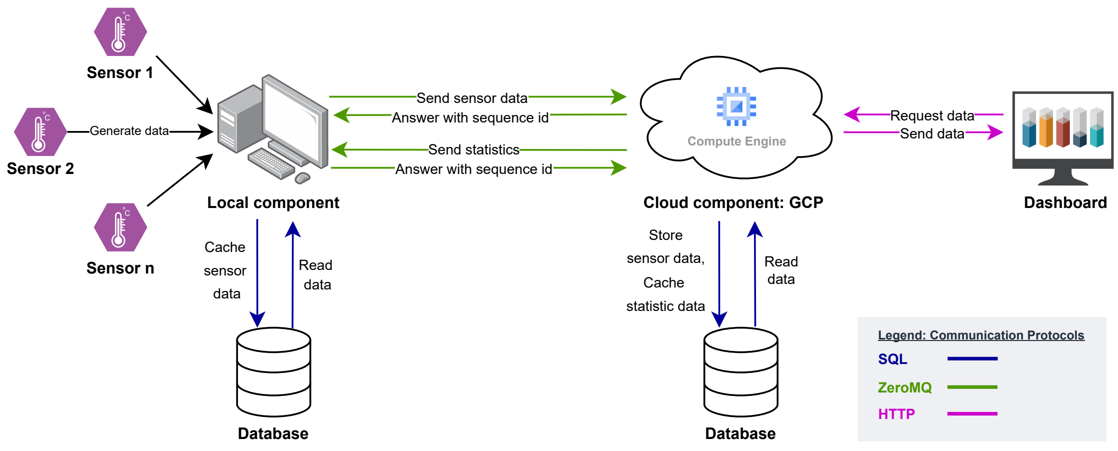
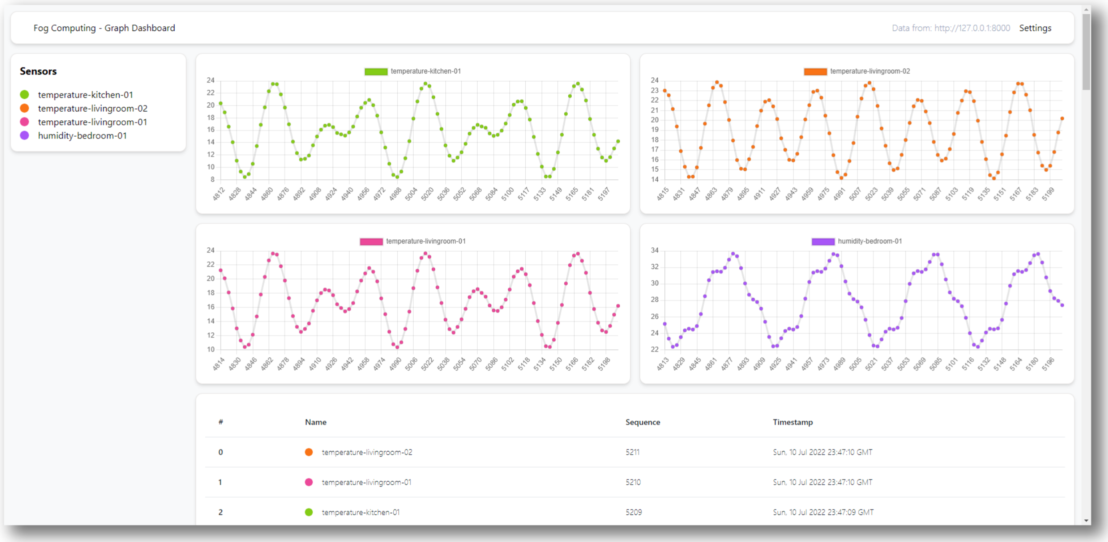


Figure 1: Architecture

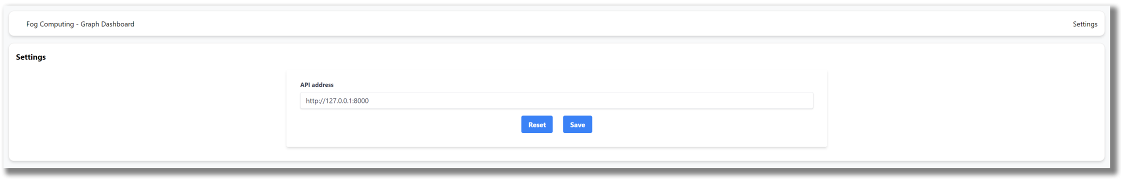
1. The application must comprise a local component that runs on a own machine, and a component running in the Cloud, e.g., on GCE.
2. The local component must collect and make use of (simulated) environmental information. For this purpose, design and use a minimum of two virtual sensors that continuously generate realistic data.
3. Data has to be transmitted regularly (multiple times a minute) between the local component and the Cloud component in both directions.
4. When disconnected and/or crashed, the local and Cloud component have to keep working while preserving data for later transmission. Upon reconnection, the queued data needs to be delivered.

Figure 2: Requirements for the prototype


Fog Computing - Graph Dashboard
Data from: http://127.0.0.1:8000
Settings
Sensors
temperature-kitchen-01
temperature-livingroom-02
temperature-livingroom-01
humidity-bedroom-01

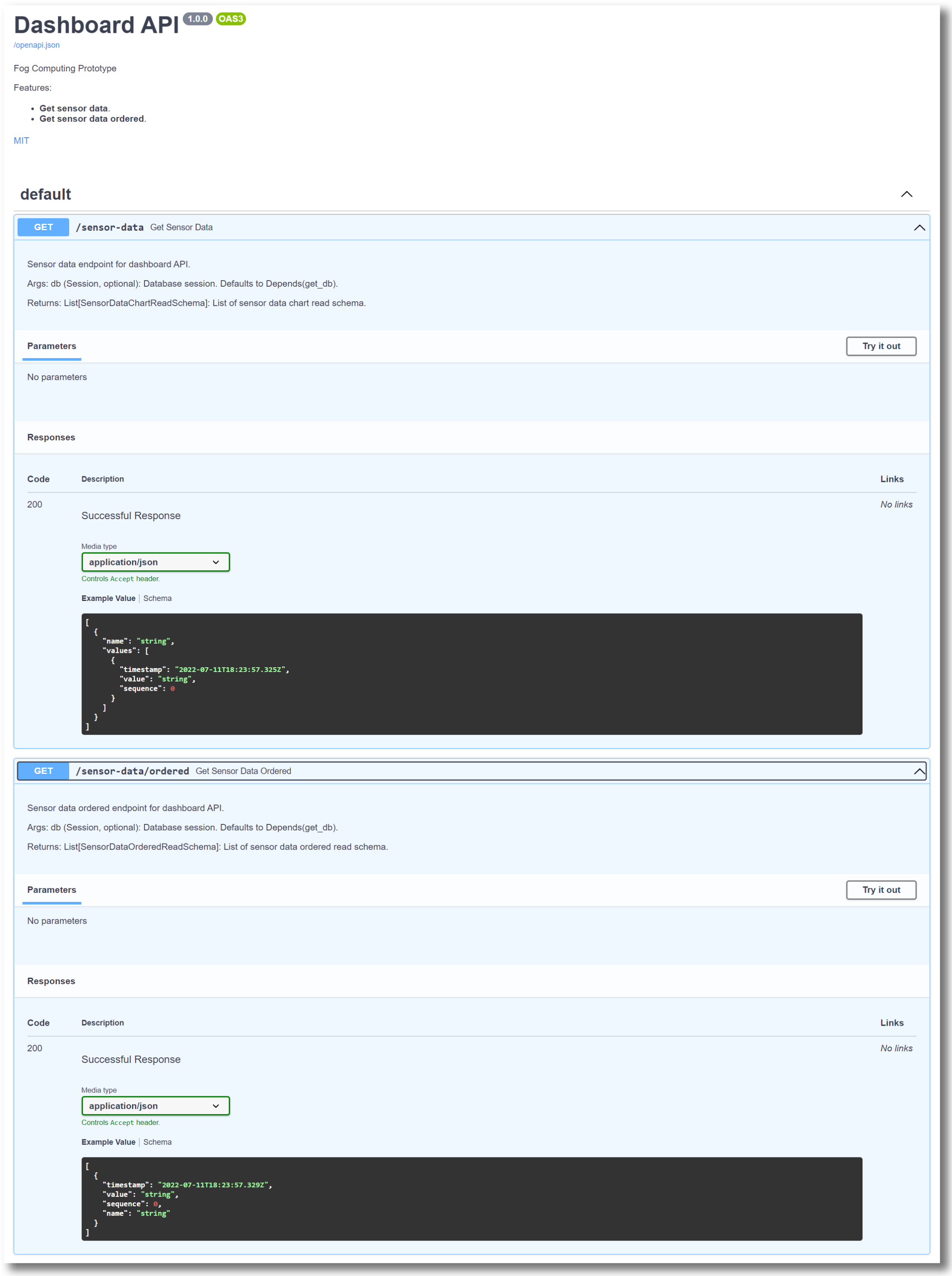
| # | Name | Sequence | Timestamp |
|---|---|---|---|
| 0 | 🟠 temperature-livingroom-02 | 5211 | Sun, 10 Jul 2022 23:47:10 GMT |
| 1 | ● temperature-livingroom-01 | 5210 | Sun, 10 Jul 2022 23:47:10 GMT |
| 2 | 🟢 temperature-kitchen-01 | 5209 | Sun, 10 Jul 2022 23:47:09 GMT |

Figure 3: Dashboard - Homepage



API address: http://127.0.0.1:8000

Figure 4: Dashboard - Settings



Dashboard API 1.0.0 OAS3

/openapi.json

Fog Computing Prototype

Features:

- **Get sensor data**.
- **Get sensor data ordered**.

MIT

default

GET `/sensor-data` Get Sensor Data

Sensor data endpoint for dashboard API.

Args: db (Session, optional): Database session. Defaults to Depends(get_db).

Returns: List[SensorDataChartReadSchema]: List of sensor data chart read schema.

**Parameters**

Try it out

No parameters

**Responses**

| Code | Description | Links |
|---|---|---|
| 200 | Successful Response | *No links* |

Media type: application/json

Controls `Accept` header.

**Example Value** | Schema

```
[
  {
    "name": "string",
    "values": [
      {
        "timestamp": "2022-07-11T18:23:57.325Z",
        "value": "string",
        "sequence": 0
      }
    ]
  }
]
```

GET `/sensor-data/ordered` Get Sensor Data Ordered

Sensor data ordered endpoint for dashboard API.

Args: db (Session, optional): Database session. Defaults to Depends(get_db).

Returns: List[SensorDataOrderedReadSchema]: List of sensor data ordered read schema.

**Parameters**

Try it out

No parameters

**Responses**

| Code | Description | Links |
|---|---|---|
| 200 | Successful Response | *No links* |

Media type: application/json

Controls `Accept` header.

**Example Value** | Schema

```
[
  {
    "timestamp": "2022-07-11T18:23:57.329Z",
    "value": "string",
    "sequence": 0,
    "name": "string"
  }
]
```

Figure 5: Dashboard API - OpenAPI documentation

- Local
  - Python
  - Pydantic
  - SQLAlchemy
  - ZeroMQ (PyZMQ)
  - SQLite
  - Poetry
  - Docker
  - Docker compose
- Dashboard
  - Vue.js
  - Vite
  - TypeScript
  - Tailwind CSS
  - vue-chartjs
  - Axios
  - pnpm
  - Docker
  - Docker compose
- Cloud
  - Python
  - Pydantic
  - SQLAlchemy
  - ZeroMQ (PyZMQ)
  - SQLite
  - Poetry
  - Docker
  - Docker compose
  - FastAPI
  - Uvicorn
  - Shell script
  - Google Cloud Platform (GCP)
  - Google Compute Engine (GCE)

Figure 6: Technology stack

## Appendix: Code

```yaml
temperature_sensors:
  - name: temperature-kitchen-01
    interval: 5

  - name: temperature-livingroom-01
    interval: 5

humidity_sensors:
  - name: humidity-bedroom-01
    interval: 5
```

Code 1: Sensors as yaml file: `sensors.yaml`

```json
{
    "name": "temperature-livingroom-01",
    "timestamp": "2022-07-02T23:56:08.209678",
    "value": "23.0",
    "sequence": 2
}
```

Code 2: Example sensor data JSON object to be sent to the cloud component

```json
{
    "sensor_data_count": 345,
    "sequence": 10
}
```

Code 3: Example statistic data JSON object to be sent to the local component

```python
# inspired by: https://zguide.zeromq.org/docs/chapter4/#Client-Side-Reliability-Lazy-Pirate-Pattern
async def start_reliable_zmq_sender(ctx: zmq.asyncio.Context, endpoint: str):
    """Start reliable ZeroMQ sender.

    Args:
        ctx (zmq.asyncio.Context): Asynchronous ZeroMQ context.
        endpoint (str): Endpoint URL.
    """

    # create zero mq socket connection
    socket = ctx.socket(zmq.REQ)
    socket.connect(endpoint)
    LOG.info(f"Connected to socket at: '{endpoint}'")

    while True:
        # access database cached data
        with Session(engine) as session:
            cached_sensor_data = [
                SensorDataCachedReadSchema.from_orm(data)
                for data in get_all_cached_sensor_data(session)
            ]

        LOG.info(f"Loaded '{len(cached_sensor_data)}' sensor data from cache.")

        # send message to cloud
        for sensor_data in cached_sensor_data:

            message = sensor_data.json().encode("utf-8")
            await socket.send(message)

            while True:
                if (await socket.poll(3000) & zmq.POLLIN) != 0:
                    answer = await socket.recv()

                    try:
                        received_sequence = int(answer)
                    except ValueError:
                        LOG.error(
                            f"Received answer from cloud is not a sequence: '{answer}'!"
                        )
                        continue

                    if received_sequence != sensor_data.sequence:
                        LOG.error(
                            f"Received sequence from cloud does not match with "
                            f"required sequence: '{received_sequence} != {sensor_data.sequence}'"
                        )
                        continue

                    with Session(engine) as session:
                        delete_cached_sensor_data(
                            session, sensor_sequence=sensor_data.sequence
                        )

                    LOG.info(
                        f"Send sensor data: '{sensor_data.name}' "
                        f"sequence: '{sensor_data.sequence}'"
                    )
                    break

                LOG.warning("No response from server received!")
                socket.close(linger=0)
                LOG.info("Trying to reconnect to server...")
                socket = ctx.socket(zmq.REQ)
                socket.connect(endpoint)
                LOG.info(f"Resending data: '{message}'")
                await socket.send(message)

        LOG.info("Waiting for new sensor data for 10 seconds...")
        await asyncio.sleep(10)
```

Code 4: Reliable sender function for the local component