Lab3 Microservices

Lorin Speybrouck

Preparation

The preparation when very quickly on Windows 11 Home using the winget package manager

```
winget install -e --id Docker.DockerDesktop
# restart
winget install -e --id Kubernetes.kubectl
winget install -e --id Helm.Helm
```

Tutorial

`Error while uploading Lab2 code:`

- In the values.yaml file there was a typo from ingress-api to ingressapi
- This was fixable by editing the file and redeploying the container using the following commands

```
docker build --tag daellhin/ingress-api:latest .
docker push daellhin/ingress-api:latest
helm upgrade "ingress-api" helm
kubectl rollout restart deployment ingress-api-helm
```

Tasks

Task 1: Extending the ingress API

`Result:` see **task1.mp4** for a full overview

`Code`

```python
# main.py
# Posting data
@app.post("/data/", status_code=201, description="Store sensor data in
InfluxDB")
async def post_data(data: RfidDataPoint):
    try:
        write_api   = influx.write_api(write_options=SYNCHRONOUS)

        people_point = Point("people").tag("source",
data.sensor_name).field("value", data.count).time(data.timestamp,
WritePrecision.MS)
        rfid_point = Point("raw_ids").tag("source",
data.sensor_name).field("value", data.rfid_id).time(data.timestamp,
WritePrecision.MS)

        # Write the data point to InfluxDB
        write_api.write(
            bucket=settings.bucket,
            org=settings.influx_org,
            record=[people_point, rfid_point]
        )

        logger.info(f"Data stored: {data}")
        return {"status": "success"}
    except Exception as e:
        logger.error(f"Error storing sensor data: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))
```

- In the POST data endpoint JSON data can be stored in the form of a RfidDataPoint
- First a Influx Point is created(this could also be done with plain Flux queries) but using the Python libraries is more developer friendly. A point is created for people with a field containing the count. Another point is created containing the rfid of the scanned tag
- Both of these point are saved to the database using the influx write_api, here multiple points can be passed as an array
- Error handling was also added wicht wil return a HTTPException with more information to the user

```python
# main.py
# Retrieving data
@app.get("/data/", description="Retrieve recent data")
async def get_data(since: str = "1h") -> RecentData:
    try:
        query_api = influx.query_api()

        count_query = f'''
            from(bucket: "{settings.bucket}")
            |> range(start: -{since})
            |> filter(fn: (r) => r["_measurement"] == "people")
            |> filter(fn: (r) => r["_field"] == "value")
        '''

        rfid_query = f'''
            from(bucket: "{settings.bucket}")
            |> range(start: -{since})
            |> filter(fn: (r) => r["_measurement"] == "raw_ids")
            |> filter(fn: (r) => r["_field"] == "value")
        '''

        count_tables = query_api.query(count_query,
org=settings.influx_org)
        rfid_tables = query_api.query(rfid_query,
org=settings.influx_org)

        # Process count results
        counts = [
            CountEntry(timestamp=int(record.get_time().timestamp()),
sensor_name=record.values.get("source"), count=record.get_value())
            for table in count_tables for record in table.records
        ]

        # Process RFID results
        rfids = [
            TagEntry(timestamp=int(record.get_time().timestamp()),
sensor_name=record.values.get("source"), rfid_id=record.get_value())
            for table in rfid_tables for record in table.records
        ]

        logger.info(f"Retrieved {len(counts)} count records and
{len(rfids)} RFID records since {since}")
        return RecentData(counts=counts, rfid_ids=rfids)
    except Exception as e:
        logger.error(f"Error retrieving recent data: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))
```

- In the GET data endpoint the most recent data(default 1 hour) is returned including the counts and rfid data
- This is done with 2 Flux queries

    - The count query takes the provided time range and filters the measurements on the people field(count)
    - The rfid query takes the provided time range and filters the measurements on the raw_ids field
- Both queries are executed using the query_api, this returns a table of results where the correct data is extracted from with list comprehension

```
# main.py
# Retrieving count
@app.get("/count/", description="Retrieve last count")
async def get_count() -> int:
    try:
        query_api = influx.query_api()

        # Query for the most recent count
        count_query = f'''
            from(bucket: "{settings.bucket}")
            |> range(start: today())
            |> filter(fn: (r) => r["_measurement"] == "people")
            |> filter(fn: (r) => r["_field"] == "value")
            |> sort(columns: ["_time"], desc: true)
            |> limit(n: 1)
        '''

        result = query_api.query(count_query, org=settings.influx_org)

        # Check if there are any records
        if not result or not result[0].records:
            return 0

        # Get the value from the first (and only) record
        latest_count = result[0].records[0].get_value()

        logger.info(f"Retrieved latest count: {latest_count}")
        return int(latest_count)
    except Exception as e:
        logger.error(f"Error retrieving latest count: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))
```

- In the GET count endpoint the most recent count is returned
- This is very similar to getting al count data, but we fiers sort on time and then take only the first result

Additional

```
# main.py
# Delete data in bucket
@app.get("/", description="Example route, also used for healthchecks")
async def root():
    try:
        delete_api = influx.delete_api()

        start = "1970-01-01T00:00:00Z"
        stop = "2099-12-31T23:59:59Z"
        delete_api.delete(start, stop, predicate='',
bucket=settings.bucket, org=settings.influx_org)

        print(f"All data deleted from bucket: {settings.bucket}")
        return {"message": "Data deleted"}
    except Exception as e:
        logger.error(f"Error deleting all data: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))
```

- When developing the API and client application is was useful to be able to reset the influx bucket. For this i also added a method to remove all data from a bucket
- This simply uses the delete_api, specifies a bucket and sets a long enough time frame

Task 2: Collecting sensor data

Results: see **task2.mp4** for a full overview

Code

```
# main.py
HOST = "http://ingress-api.daellhin.cloudandmobile.ilabt.imec.be/"
sensor_name = "RPI"

count = 0
RFID_tags = []

def load_startup_data():
    global count, RFID_tags

    try:
        r_tags = requests.get(url=HOST + "current", params={})
        r_tags.raise_for_status()
        r_count = requests.get(url=HOST + "count", params={})
        r_count.raise_for_status()

        RFID_tags = [int(tag) for tag in json.loads(r_tags.text)]
        count = int(r_count.text)
        print(f"Loaded startup data: {count} people in the building with
tags {RFID_tags}")
    except requests.exceptions.RequestException as e:
        print(f"Error loading startup data: {e}")

def process_RFID_tag(tag_id: int):
```

```
    global count

    if tag_id in RFID_tags:
        count = count - 1
        RFID_tags.remove(tag_id)
        print(f"        {tag_id} left the building")
    else:
        RFID_tags.append(tag_id)
        count = count + 1
        print(f"        {tag_id} entered the building")

    current_milliseconds = int(time.time() * 1000)
    data_point = RfidDataPoint(timestamp=current_milliseconds,
rfid_id=str(tag_id), count=count, sensor_name=sensor_name)

    try:
        r = requests.post(url=HOST + "data", params={},
json=data_point.to_dict())
        r.raise_for_status()
        print(f"        Data posted successfully: {r.status_code}")
    except requests.exceptions.RequestException as e:
        print(f"        Error posting data: {e}")

    print(f"        {count} people in the building")

reader = SimpleMFRC522()
if __name__ == "__main__":
    load_startup_data()
    print("RFID scanner started")
    try:
        while True:
            # -- Actual reading --
            read_id, text = reader.read()
            print(f"Detected {read_id}")
            process_RFID_tag(read_id)
            sleep(1)
            # -- Simulated reading --
            # random_id = random.randint(0, 3)
            # print(f"Detected {random_id}")
            # process_RFID_tag(random_id)
            # sleep(5)
    finally:
        GPIO.cleanup()
        pass
```

- The program stores the count and current rfid tags in the building using two variables
- When the rfid reader detects a tag(a timeout of 1s is used to prevent multiple readings), it calls the process_RFID_tag function. Which checks if the tag is checking in(increase count), or checking out(decrease count). Using the request library a POST request is made to the influx API created in task1
- Additional 1: a small simulation was also added to be able to test without having to manually having to place a tag in front of the sensor

- Additional 2: when the program initialises it gets the current tags and current count from the influx API using 2 GET requests

Task 3: Monitoring sensor data

`Result` see **task3.mp4** for a full overview

`Code`

```
from(bucket: "default")
  |> range(start: -1h)
  |> filter(fn: (r) => r["_measurement"] == "people")
  |> filter(fn: (r) => r["_field"] == "value")
  |> sort(columns: ["_time"], desc: true)
```

- This is a flux query that firs limits it range to the past hour, then it filters on the people measurement with the field name value(where the count is stored), then filters on time

Task 4: Query for active RFID IDs

`Result` demonstrated in previous tasks

`Code`

```
# main.py
@app.get("/current/", description="Get tags currently inside")
async def get_current() -> List[str]:
    try:
        query_api = influx.query_api()
        current_query = f'''
                from(bucket: "{settings.bucket}")
                  |> range(start: today())
                  |> filter(fn: (r) => r._measurement == "raw_ids" and
r._field == "value")
                  |> duplicate(column: "_value", as: "rfid_id")
                  |> group(columns: ["rfid_id"])
                  |> count()
                  |> filter(fn: (r) => r._value % 2 != 0)
                  |> rename(columns: {"_value": "count"})
                  |> keep(columns: ["rfid_id", "count"])
        '''
        current_tables = query_api.query(current_query)

        tags = [ record.values["rfid_id"] for table in current_tables
for record in table.records]
        logger.info(f"Retrieved current tags: {tags}")
        return tags
    except Exception as e:
        logger.error(f"Error retrieving latest count: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))
```

- In the GET current endpoint the rfids of the tags inside the building are returned
- This is done with a Flux query that sets the range to start from today, filters the measurements to include only those with _measurement as "raw_ids" and _field as "value", duplicates the value column as rfid_id, groups the results by rfid_id, counts the occurrences of each rfid_id, filters the results to include only those with an odd count (indicating the tag is currently inside), renames the value column to count, and keeps only the rfid_id and count columns.
- The query results are processed to extract the rfid_id values, which are then returned as a list of tags currently inside the building.

Questions

Question 1

> Which steps are required to add a new container (e.g. an egress API that exposes the time-series data from InfluxDB) to this architecture?

```
docker login
docker build --tag daellhin/egress-api:latest .
docker push daellhin/egress-api:latest
helm install "egress-api" helm
```