Temperature Store

1. Problem Description

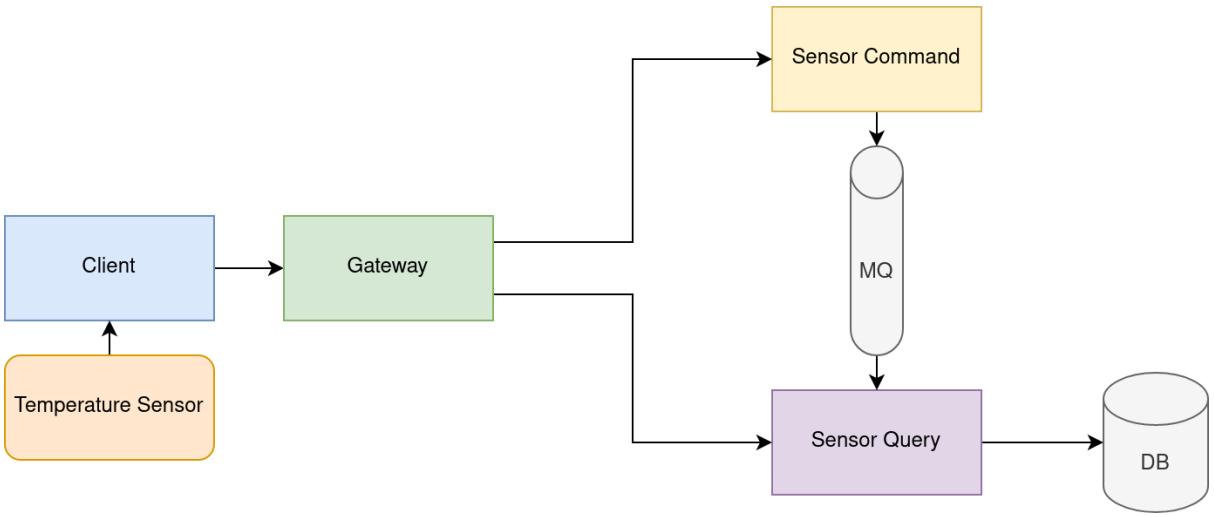
A client connected to a temperature sensor sends temperature data arbitrarily to the backend. The same client is also responsible for gathering aggregated data from the backend. So the requirement from the backend team is providing REST API for sending bulk data and gathering daily or hourly aggregated data. These requests should be as fast as possible, it will directly affect the UI and irresponsive application or long waiting times might directly affect the user experience in a bad way.

2. Architecture

2.1. Initial Notes

From the backend point of view the summary of design:

- Sensor data storing operations can be in an eventually consistent way. Backend might get the event ASAP but processing or storing to the database might take a couple of seconds.
- Daily or Hourly data fetchings should be based on pre-aggregated results. In that case, every query will be a simple query with where conditions instead of heavy join or aggregate operations.
- An Even Store can be used for possible event replays or new microservices implementations.
- For possible nested document and time-series data, a NoSQL database like MongoDB or Influx DB might be better comparing RDBMS.
- The same document will be updated so an optimistic locking mechanism with a version column can be a better approach.



2.2. Final Design

CQRS (Command-Query Responsibility Segregation) approach is used for implementation. API Gateway redirects store operations to command microservice and redirects query operations to query microservice. Command microservice converts incoming bulk reading data to series of events and sends it to Kafka. Kafka is both an MQ and an Event Store. So all our events will be stored safely and no need for an additional event store database.

Query microservice contains a Kafka Listener. This handler consumes the Sensor data events. All pre-aggregate calculations performed here. Average, min, max and sample count is stored to 2 different collections dailyData and hourlyData and indexed by clientId and day/hour.

Average calculation is done by using the rolling statistical average method. It might be easily converted to a stream processor in the future.

The application code base is a mono repo and a multi-module Gradle project. It contains 4 different sub-modules:

- common
- gateway
- sensor-command
- sensor-query

Common module contains common event Pojo which is used by command and query microservices.

Gateway is an API Gateway implementation by default it uses 8080 port on the local environment. Internally it uses Spring Cloud Gateway. Netflix Zuul is deprecated and SCG seems a better alternative for this operation. In a possible Kubernetes deployment scenario it can be easily replaced by Kubernetes ingress rules. Because its only purpose is reverse-proxying the content

sensor-command microservice is a lightweight spring boot application, its main aim is converting data requests to events and publishing them to Kafka.

Sensor-query microservice is another spring-boot application, additionally, it has a MongoDB database connection. Its main objective is consuming events from Kafka and computing the aggregate operations and then storing them to the respective feature tables/collections. It uses optimistic locking for preventing data loss and used indexes for query parameters.

Code metrics and checks:

- Method cyclomatic complexity is below 6 for every method.

- Package by feature used instead of package-by-layer (controller, service, repository packages) for detecting inter-package dependencies and calculating coupling between packages
- Kotlin null-safe interoperability annotations used for better IDE support and NPE prevention on implementation.

3. Possible Future Improvements

- Using Reactive Streams/Spring Webflux and Reactive database drivers for stream sensor data uploading and reading and also for better performance
- Using MQTT-like IoT protocols and RabbitMQ like MQs which support MQTT by design.
- Data partitioning and sharding on MongoDB for better performance and scalability
- Using Kafka Streams for pre-aggregate calculations
- Retry mechanisms, Dead-letter-queue for MQ operations.
- Cursor-based stream pagination, GUI can use infinite scroll and steam data for better performance

4. Development

Needed installations

- Zookeper
- Kafka
- MongoDB

Alternatively, docker containers can be used for the development

```
docker network create ts

docker run -d --name ts-mongo \
      --network tardis \
      -p 27017:27017 \
      mongo

docker run -d --name ts-zookeeper \
      --network tardis \
      -p 2181:2181 \
      zookeeper

docker run -d --name ts-kafka \
      --network ts \
      -e KAFKA_ZOOKEEPER_CONNECT=ts-zookeeper:2181 \
      -e KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092 \
      -e KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR=1 \
      -p 9092:9092 \
      confluentinc/cp-kafka
```

4.1. Compiling and Running

Gradle wrapper or local Gradle installation can be used for testing

```
./gradlew clean build -x test
./gradlew :gateway:bootJar
java -jar gateway/build/libs/gateway.jar
```

Run in a different terminal

```
./gradlew :sensor-command:bootJar
java -jar sensor-command/build/libs/sensor-command.jar
```

Run in a different terminal

```
./gradlew :sensor-query:bootJar
java -jar sensor-query/build/libs/sensor-query.jar
```

5. API

5.1. Store Operation

Bulk sensor data store coming from the client.

URI	/api/store/{clientId}
Path Variables	clientId: UUID Identifier of connected client
HTTP Method	POST
Content-Type	application/json
Content	readings: List of Sensor Readings temperature: measured temperature coming from sensor time: timestamp of temperature measurement
Sample Content	{"readings": [{"temperature": 10, "time": "2021-07-01T01:20:12"}]}
Response	200: Sensor Data Accepted for Storing 400: Bad Request

Sample Curl operation

```
curl -X POST \
```

```
      -d '{"readings": [{"temperature": 10, "time": "2021-07-01T01:20:12"}]}' \
      -H 'Content-Type: application/json' \
      http://localhost:8080/api/store/0526b10f-95c6-4ee8-8198-68380a748d4a
```

5.2. Daily Data Query

Bulk sensor data store reading

URI	/{clientId}/daily/{startDate}/{endDate}
Path Variables	clientId: UUID Identifier of connected client startDate: start date criteria of daily data limit (yyyy-MM-dd) endDate: end date criteria of daily data limit (yyyy-MM-dd)
HTTP Method	GET
Content-Type	application/json
Response	200: List of sensor data

Sample Curl operation

```
curl -H 'Content-Type: application/json'
http://localhost:8080/api/query/0526b10f-95c6-4ee8-8198-68380a748d4a/daily/2021-06-15/20
21-07-02
```

5.2. Hourly Data Query

Bulk sensor data store reading

URI	/{clientId}/hourly/{startDateTime}/{endDateTime}
Path Variables	clientId: UUID Identifier of connected client startDateTime: start date time (yyyy-MM-dd'T'HH:mm:ss) endDateTime: end date time (yyyy-MM-dd'T'HH:mm:ss)
HTTP Method	GET
Content-Type	application/json
Response	200: List of sensor data

Sample Curl operation

```
curl -H 'Content-Type: application/json' \

http://localhost:8080/api/query/0526b10f-95c6-4ee8-8198-68380a748d4a/hourly/2021-06-15T1
2:34:22/2021-07-02T13:45:34
```