### **Event Controller**

It has the logic of the /event request.

There is a PayloadService, that processes the request Payload and transforms it into a List of DataEvent objects, and an EventService, that is storing the relevant data into the data structures

### **Stats Controller**

It has the logic of the /stats request.

We use the EventService to calculate the stats at the moment of receiving the request, using the stored data.

## **Data Structures & Relevant Variables**

lastUpdate Timestamp variable, that stores the timestamp of the last POST request.

int[] countX Array of ints of size 60, that stores in the position i the number of x data that belongs to the second i, with i between 0 and 60 seconds in the past, with respect to lastUpdate.

int[] countY Same as countX but for y data.

double[] sumX Array of doubles of size 60, where in the position i will be stored the sum of values of x that belongs to second i.

long[] sumY Array of longs of size 60, same as sumX but for y data.

## **Algorithms**

### **Save data**

* Get difference (diffPrevUpdate) of time in seconds between the lastUpdate and now. If lastUpdate is null, then this is the first time data will be saved, so we'll set lastUpdate as now and diffPrevUpdate will be zero.
* Then, we will prepare the data structures for loading the data.
  + If diffPrevUpdate is greater than 60, then the data previously stored is too old and we won't need it anymore, so we can set all the arrays (countX, countY, sumX, sumY) back to 0.
  + If diffPrevUpdate is between 0 and 60, then there is some data stored that we need to keep and some of it is already outdated.
    - We first need to find out which is the lastValid position of the array, and that is the difference between 60 seconds and diffPrevUpdate.
    - The "old" data that is still on time considering the current timestamp is stored between the positions 0 and lastValid of the arrays. We need to shift those values to the right of the array, to reflect that fact that the time has advanced and to return the arrays to a consistent status.
    - After that, we also need to fill the positions between 0 and diffPrevUpdate with zeros, as these positions (timestamps) didn't exist in previous payloads.
* After this, we can take care of the new data from the payload, just by calculating how many seconds apart each Data Event timestamp is from now.
* For every Data Event that is in a range of 60 seconds, we will sum the value of x and y, to sumX and sumY respectively, and increment the count in countX and countY for the corresponding positions.
* Lastly, we'll update the lastUpdate timestamp.

#### **Complexity**

This method has a time complexity of O(N), where N is the number of lines in the request payload. This is because it has to read each of the lines, parse it and verify if the data point should be included or not, by comparing the timestamps. At the beginning of the algorithm it also checks if the data stored in the arrays needs to be updated, and updates them accordingly. All of these operations can be done in a constant time because the arrays have a fixed length of 60. Nevertheless, the overall complexity is still O(N) + O(1) = O(N). The space complexity of this method is constant. We are not creating new data structures, we just create a few variables of type Integer.

### **Get stats**

Total : Sum of countX, as this is the total number of Data Events stored. Returns an int.

Sum x : Sum of values in sumX. Returns a double.

Sum y : Sum of values in sumY. Returns a long.

Avg x : Sum x / Total. Returns a double. *Important*: For more than 1000000 Data Events (Total > 1000000) this could overflow.

Avg y : Sum y / Total. Returns a double.

#### **Complexity**

This method has a time complexity of O(1) as it only performs a sum operation on the Arrays countX, sumX, and sumY, which have a fixed length of 60. Then, it performs some divisions and creates an object of type Stats containing the 4 values of the response. All these operations are performed in constant time. The space complexity of this method is also constant, since we are not creating extra data structures, we just create some variables of type int, long and double.

## **Concurrency & Thread Safety**

This application was developed with Spring Boot. Spring automatically embedded a web container (Tomcat by default) and it can handle requests simultaneously (up to 200 threads by default). On the other hand, each thread that is created by the web container to process every new request will be modifying the state of the arrays inside the EventService, so the application is not thread safe. To solve this problem I decided to make the methods updateData and getStats from the class EventService synchronised. This ensures that only one thread can access those methods at a time. Another possibility would be to implement a lock that is synchronised, and every time a thread wants to access the arrays, it needs to obtain the lock.