Vehicle Flow Monitoring

Mini Project Documentation

Asztrik Bakos (1527997)
e1527997@student.tuwien.ac.at

Vehicle Flow Monitoring

The proposed miniproject is a simple service supporting city traffic. The key idea is to protect privacy, while providing real-time traffic status data.

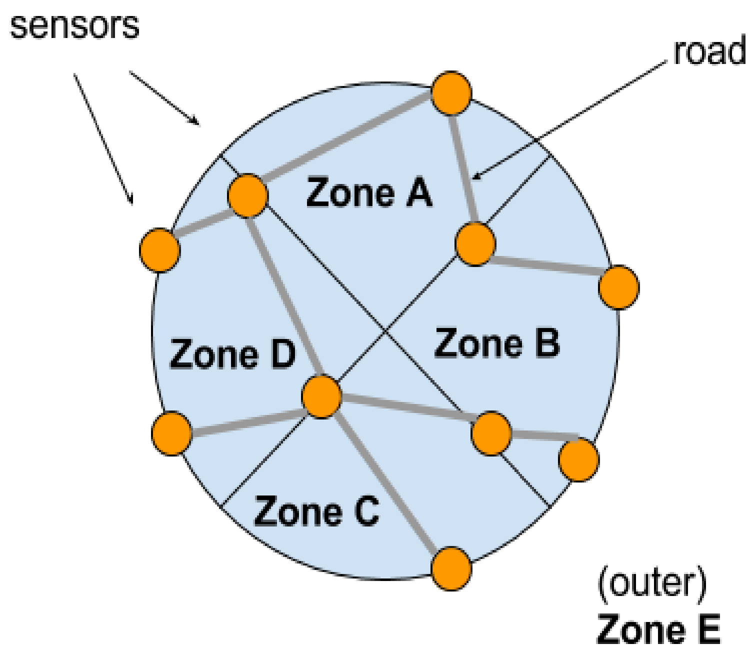
Vehicle Flow Monitoring simply counts cars passing through sensors, thus maintaining the actual car count for a given area.

For more on this project, the idea, architecture and planning, see the proposals.

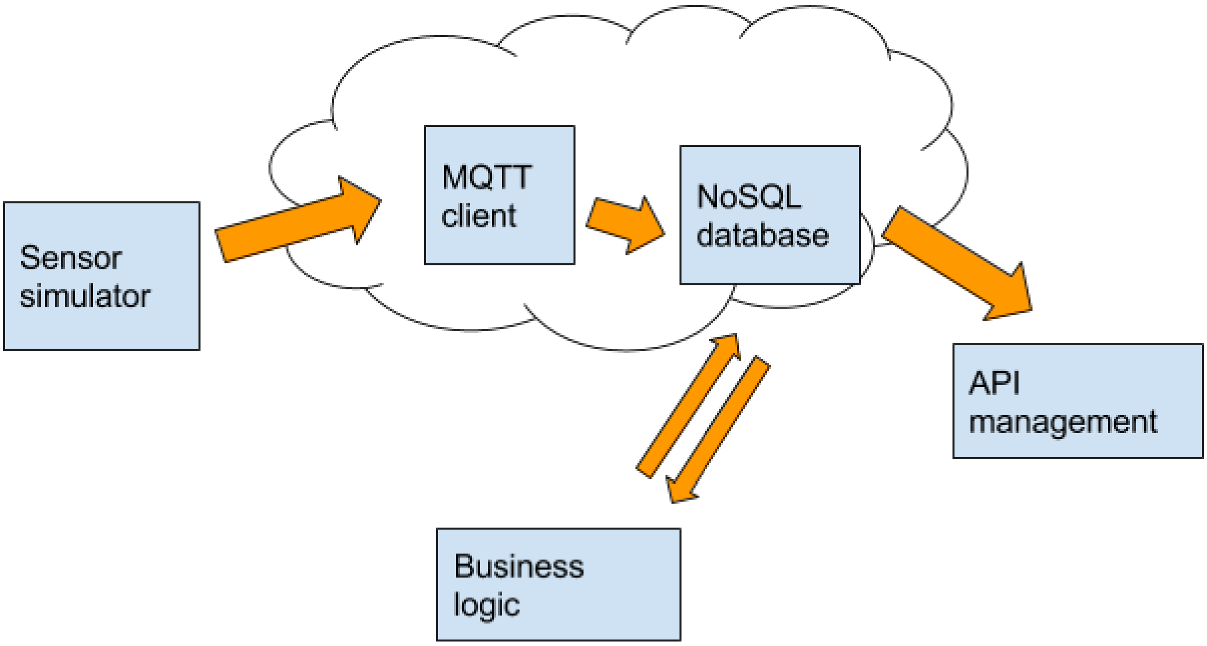
Miniproject Implementation

The demo

In the miniproject demonstration a small, 10 sensor system shows the basic idea:



Each time a car passes by a sensor, a signal is sent to the central (with regards to the direction of the passing). This passing gets registered and translated into an increment / decrement of the car count in a given zone.



Preparations

As in the real world, first a backend must exist, to be able to receive the signals from the peripherals. In our case it is a set of Amazon Web Services:

- **MQTT Client of AWS Iot** - which is a very light telemetry transport protocol client used in IoT systems. The client needs an Id, for the demo it is **18979**.
- **Dynamo DB** - a NoSQL database, transfer from the MQTT is configured by an action (a select statement) and governed by a policy. The database needs the following tables:
    - CityMap - contains which sensor is ordered to which zone. It is a fix table, created on installation of the sensors
    - sensorData - is the table the MQTT client uses to forward the sensor data into, also it is the source for the business logic. It contains a timestamp value too for each incoming sensor data packet.
    - zoneData - contains the cyclic updated car count values for each zone. It is updated by the business logic and accessed by the API

If all set up, the MQTT client can connect with the id, and subscribe to a topic **'tf'** - as in traffic flow.

Running the sensors

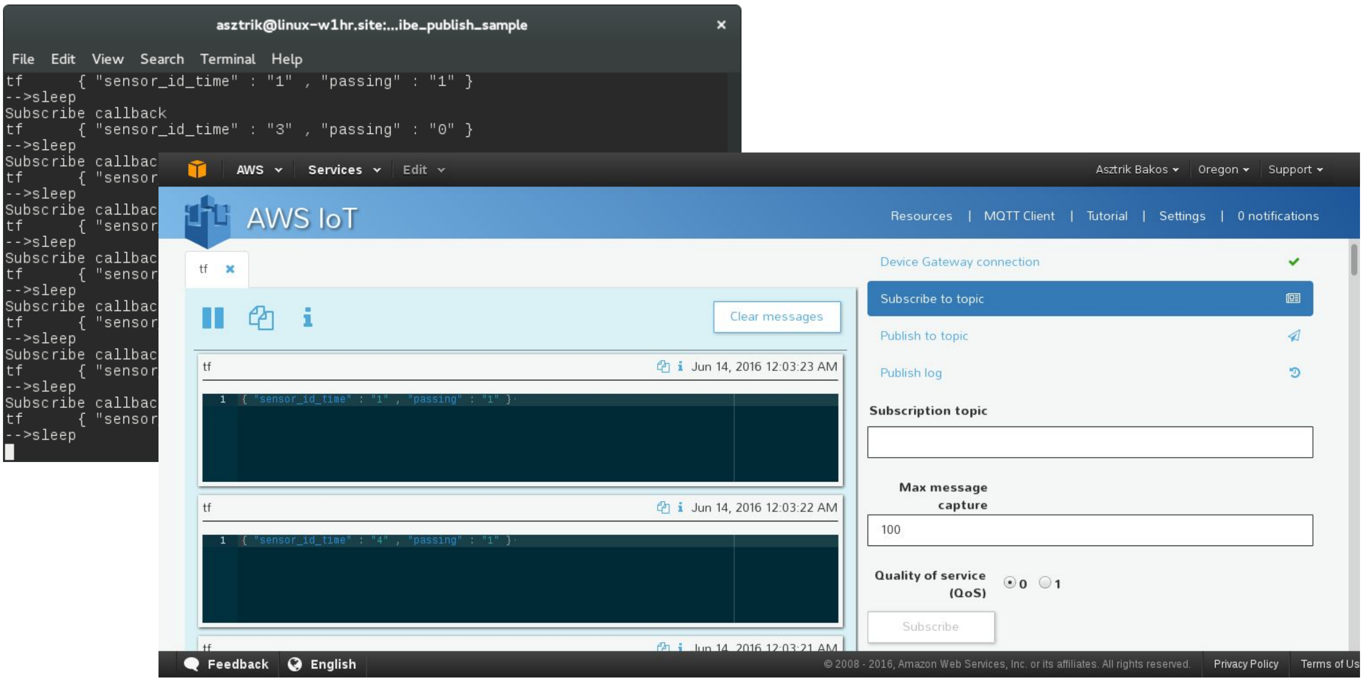
The sensors are simulated by a simple C program, which is based on the MQTT examples of Amazon. The example was modified so that the number of sensors is variable by a parameter **-i**. After starting it like

```
./subscribe_publish_sample -i 4
```

It begins to send regularly small JSON objects to the MQTT topic 'tf':

```
tf    { "sensor_id_time" : "4" , "passing" : "0" }
```

Which is also observable on the AWS IoT MQTT Console.



Business Logic

Starting from Eclipse, it is a simple Java Application with the following classes:

- **AppRunner** - starts the app and executes the methods. Formerly it was the whole app, but later it was "promoted" to be a controller. Now it is responsible for the length of the periods (if the sleep is shorter, the process gets faster), and it implements the logic which ensures that only fresh (older than the last timestamp) sensor data gets processed.
- **LoadCityMapData** - connects to the Dynamo DB's **CityMap** table and receives its contents:

sensor_id	zone_id
1	A
1	B
2	A
2	B

Each sensor ID has two zone ID-s, but in order to be able to monitor the passing directions of the cars, the sensors have to be directed too. This means, that when the map is read into a List of **SensorMap** objects, each sensor has an upper, and a lower zone based on the alphabetical order. For example on the table above sensor 1 has zone A as upper, and zone B as lower zones.

- **LoadSensorData** - connects the DynamoDB regularly and fetches the sensors and filters out the old ones (with timestamps older than the current last). Again it builds a list consisting of **RawSensorData** objects containing timestamp, sensor ID and passing direction
- Finally **ZoneLogic** gets the actual **SensorMap** and **RawSensorData** lists and calculates the aggregated car count changes for each zone. This means simply incrementing / decrementing an integer value regarding to the direction. It creates **ZoneChange** objects containing the +/- car flow for each zone. When both arrays were processed, the zoneData table is scanned from the Dynamo DB database and, its according values are modified using API methods, so that the car count is actual again.
- Additionally for possible **Human interaction** a **Mailer** service is implemented. Each time an email will be sent if a **VFMException** is thrown, or a query on the zoneCount table returns a ***negative*** value.

This part runs in an endless loop, keeping the database updated.

API Management

Now with the zoneCounts present in the database, the users can access it, by sending a GET message to a REST API:

```
{
  "tableName": "zoneData",
  "operation": "list",
  "payload": {"TableName" : "zoneData"}
}
```

As a result, a JSON object is received:

```
...
 {
   "zone": "B",
   "carcount": 16
 },
 {
   "zone": "E",
   "carcount": 119
 },
```

. . .

This format is easy to precess for a mobile application, and so the primary functionality of the project is fulfilled. It is also possible to push updates to the Dynamo DB as a mobile user's observation - but if has to be analyzed and validated first, but the validation involves machine learning techniques to find out the parameters of a "trustable" user submission. As a temporary solution, a basic filtering on the API management side could be implemented.

ApiGee configuration

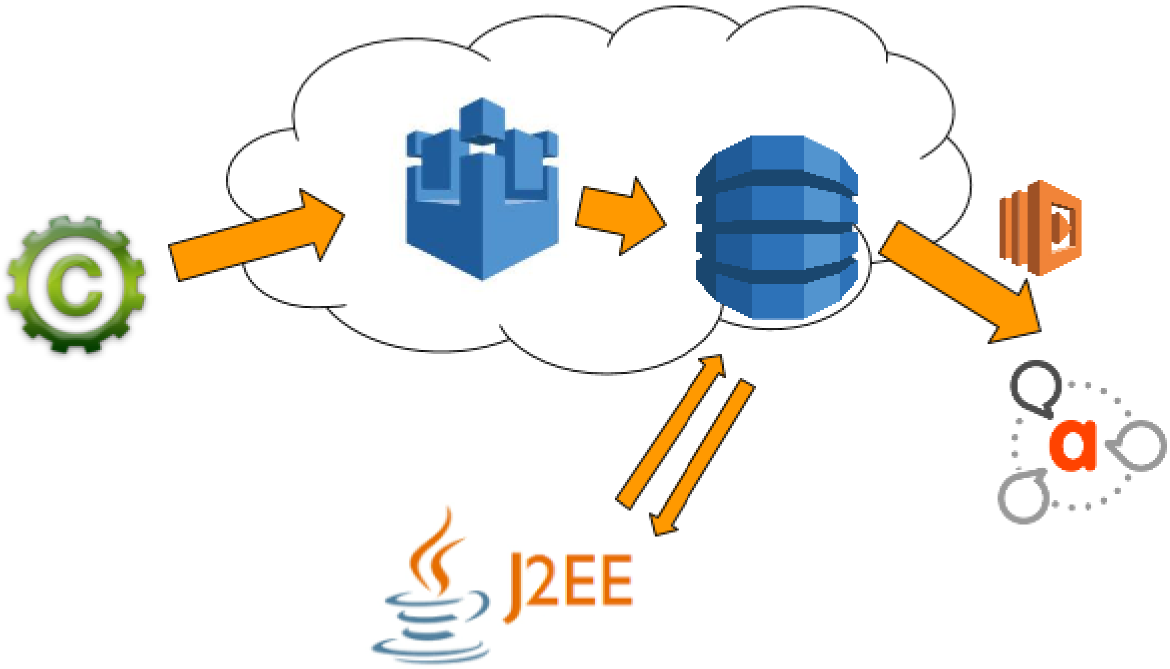
Aim is to make the DynamoDB tables **publicly** reachable on a link similar to the following: http://toczee1527997-test.apigee.net/vfm/zones because it would mean, that I have control over the API requests. So it is possible to differentiate the services provided to various users. First the API access count and syncing time will differ for
Free users

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Quota async="false" continueOnError="false" enabled="true" name="VFM-free"
type="calendar">
        <DisplayName>VFM free</DisplayName>
        <Properties/>
        <Allow count="100" countRef="request.header.allowed_quota"/>
        <Interval ref="request.header.quota_count">1</Interval>
        <Distributed>false</Distributed>
        <Synchronous>false</Synchronous>
        <TimeUnit ref="request.header.quota_timeout">month</TimeUnit>
        <StartTime>2016-6-19 12:00:00</StartTime>
        <AsynchronousConfiguration>
        <SyncIntervalInSeconds>20</SyncIntervalInSeconds>
        <SyncMessageCount>5</SyncMessageCount>
        </AsynchronousConfiguration>
</Quota>
```

And **premium** users

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Quota async="false" continueOnError="false" enabled="true" name="Quota-1"
type="calendar">
        <DisplayName>VFM premium</DisplayName>
        <Properties/>
        <Allow count="20000" countRef="request.header.allowed_quota"/>
        <Interval ref="request.header.quota_count">1</Interval>
        <Distributed>false</Distributed>
        <Synchronous>false</Synchronous>
        <TimeUnit ref="request.header.quota_timeout">month</TimeUnit>
        <StartTime>2016-6-19 12:00:00</StartTime>
        <AsynchronousConfiguration>
        <SyncIntervalInSeconds>5</SyncIntervalInSeconds>
        <SyncMessageCount>10</SyncMessageCount>
        </AsynchronousConfiguration>
</Quota>
```

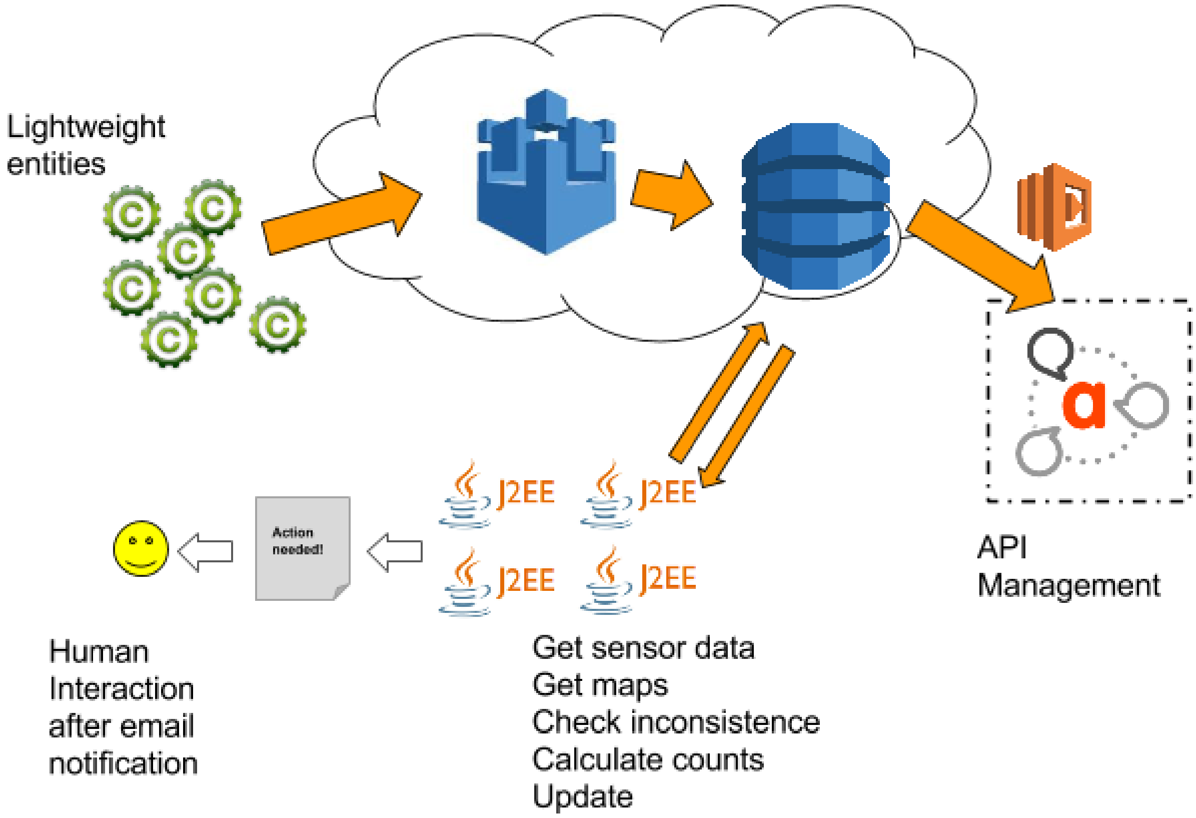
A seen in the XML configs, the premium users have a higher syncing rate and a higher request limit than free users.



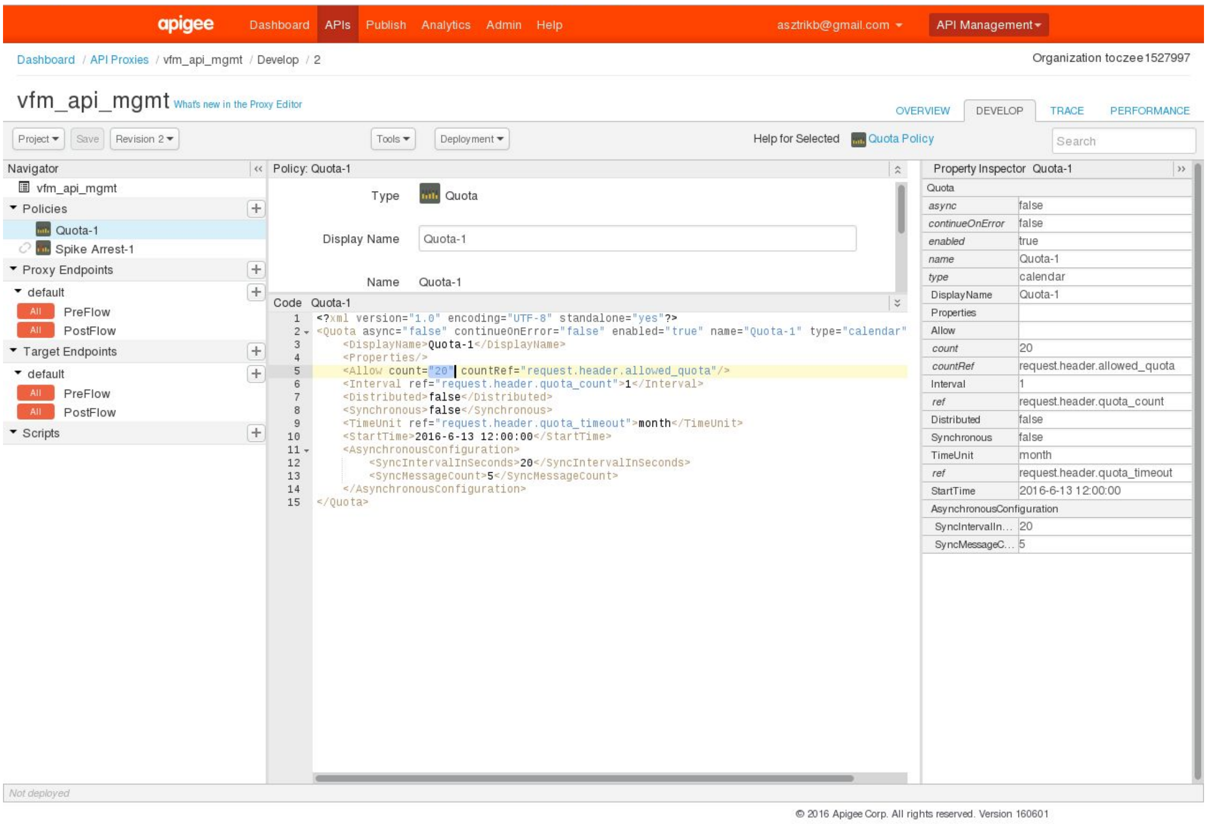
The final implementation/architecture view is as seen on the image:

- C sensor simulators
- AWS IoT backend
- AWS DynamoDB as NoSQL database
- Java Application as business logic
- AWS Lambda for implementing a basic API for the database
- Apigee for API management

Other functions



- Elasticity Support
  - Number of sensors is easily scalable, for a better city map granularity a way bigger amount of sensors is manageable with the same infrastructure
  - The Java application backend is scalable too:
    - It has several modules, which can run parallel, even on different performance VM-s. So for thousands of sensors the LoadSensorData module can run on bigger machines, while the ZoneLogic which handles only some hundred records doesn't require big capacity
    - The city can split up: LoadSensorData can run in multiple instances, so that it filters not only the sensor timestamp, but only a sensor ID range. After processing the raw data ZoneLogic is able to aggregate it again.
- Human interaction
  - Is possible on the API management interface (see below)
  - Also the AppRunner part of the business logic can set the timer intervals for refreshing the zone counts
  - By running a resnc method, the zoneCount table can be reset from the Java app
- API Management
  - Makes possible to limit access for querying and posting zoneCounts by users:



Pitfalls and lessons learned

- The development was much easier with the previous four assignments, the goals and the core principles cleared left only some technical difficulties
- **Dynamo DB** is very well structured, however sometimes it is way easier to use the Java API than the user interface for eg. deleting items due to the error messages. Also creating the sufficient policies to allow IoT and Lambda to access the database
- **Creating an API** was also very hard, because first it needs an **API Gateway** then a **Lambda** python script to expose the DB table. In my solution I didn't use any security, because my main goal was to keep the access simple
- **Accessing the API** was easy in a browser, but when I tried to use an API Management Service like **ApiGee** or **WSO2** I hit walls due to an SSL incompatibility of some Amazon resources.