**Enterprise Integration (MEIC-A, 2019-20, 2º semestre)**

Instituto Superior Técnico – MEIC-A

*Sprint 2 Report*

1. **Definition of the microservices needed for the MaaS functionality**

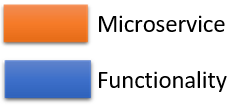


Fig. 1 – Microservices and functionalities

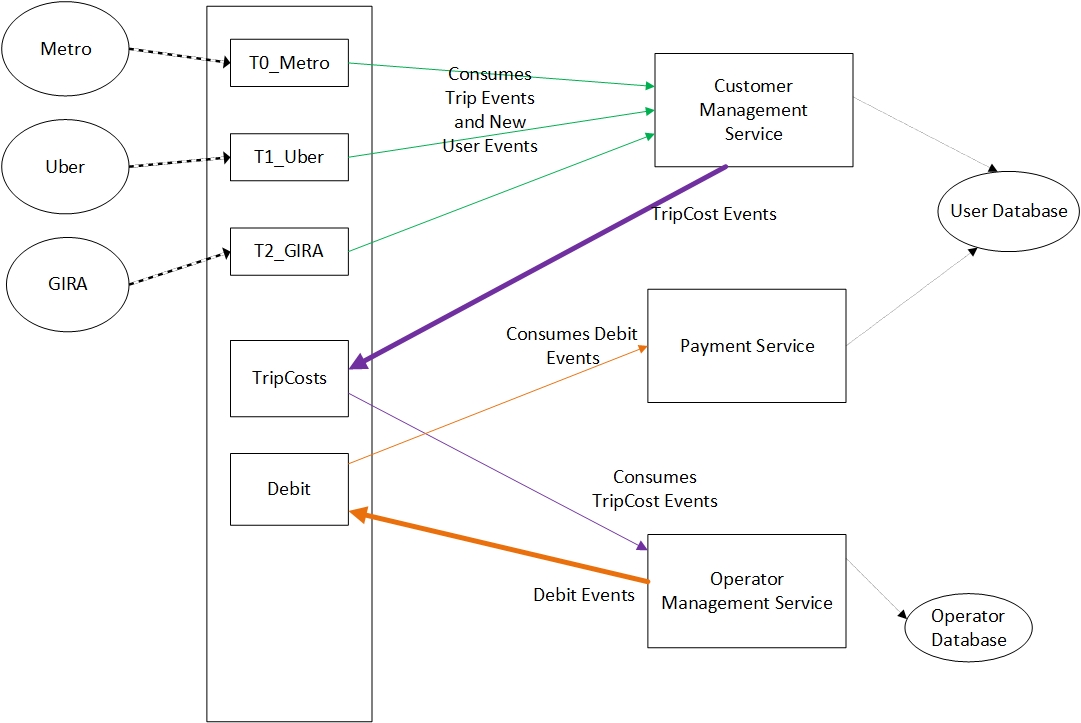


Fig. 2 – Main Event flows diagram

1. **Chosen microservices**

Our project plans the usage of 5 microservices to provide all the needed functionalities:

* **Customer Management Service:** this service aggregates all the functionalities that have to do with the users like consuming trip events, provide information about the trips to Operator Management Service, creation of users and maintain user history about the trips
* **Payment Service:** this service has the responsibility of debiting from the user accounts considering all the business rules
* **Operator Management Service:** this service aggregates all the functionalities that have to do with the operators, services and discounts. It deals with the creation of all them and consumes the events produced by the User Management Service, processes them by applying the available discounts and produces a debit event for the Payment Service to consume
* **Inspector:** this service has the responsibility of checking if there is any abnormality in the system like for example a user making a checkout without having checked in
* **Customer Support:** this service handles complaints and provides help to the user when there is some problem

We’ve decided to implement the User Management Service and the Operator Management Service because we consider them the core of the Maas Operator. These services comprise the functionalities that deal with customer accounts, services, operators, discounts and the processing of trip events.

1. **Microservices input and output**

Customer Management Service:

**Input**

This service takes as input NewUser events and Trip events that are consumed from the operator topics.

* **Trip Events**

There are 3 types of trip events:

* **Type 0 operators** (Check in and check out method)

These events contain as main fields:

* **eventType** to say if it is a check-in or a check-out
* **operator** that contains the name of the operator
* **info** that contains information about a trip. Inside this field there are fields with the token of the customer, station and a timestamp.

Examples:

Check-in in metro

{

    "event": {

        "eventType": "t0-check-in",

        "operator": "Metro",

        "info": {

            "Token": "t15345",

            "Station": "Odivelas",

            "Timestamp": "2020-02-29 18:23:41.278"

        }

    }

}

Check-out in metro

{

    "event": {

        "eventType": "t0-check-out",

        "operator": "Metro",

        "info": {

            "Token": "t5431",

            "Station": "Alameda",

            "Timestamp": "2020-02-29 18:23:47.718"

        }

    }

}

* **Type 1 operators** (Distance and time dependent method)

These events contain as main fields:

* **eventType** that says “t1”
* **operator** that contains the name of the operator
* **info** that contains information about a trip. Inside this there are fields with the token of the customer, price and a timestamp.

Example:

{

    "event": {

        "eventType": "t1",

        "operator": "Uber",

        "info": {

            "Token": "t243",

            "Price": "70.51.901",

            "Timestamp": "2020-02-29 19:45:58.638"

        }

    }

}

* **Type 2 operators** (Time dependent method)

These events contain as main fields:

* **eventType** that says “t2”
* **operator** that contains the name of the operator
* **info** that contains information about a trip. Inside this field there are fields with the token of the customer, time spent with the vehicle, price of the ride and a timestamp.

Example:

{

    "event": {

        "eventType": "t2",

        "operator": "Gira",

        "info": {

            "Token":"t1",

            "Time":"3600",

            "Price": "12.60",

            "Timestamp":"2020-02-29 20: 57: 10.294"

        }

    }

}

* **New User Events**

These events contain as main fields:

* **eventType** that says “new-user”
* **user** that contains information about the user. This field contains information about a user: id, email, plan type, first name, last name, balance and a field that says if the user has a pass or not. The plan type is used by the Payment Service to define he should charge the costumer right away **(pre-paid)** or in the end of the month **(post-paid)**.

Example:

{

    "event":{

        "eventType": "new-user",

        "info": {

            "id": "69c594cfdeeaedd220",

            "email": "user@gmail.com",

            "planType": "pre-paid",

            "firstName": "Paulo",

            "lastName": "Neves",

            "balance": "500",

            "hasPass": "true"

        }

    }

}

**Output**

This service has as output a TripCost Event. This event has the goal provide information to the Operator Management Service compute the how much money goes to the operator and how much money should be debited from the user account.

* + **TripCost Events**

These events contain as main fields:

* + **eventType** that says “trip-cost”
  + **info** that contains information about a trip and the user that made that trip. The info field contains: base cost of the trip, token of the user, info about whether a user has a pass or not, plan type, trip ID, type of operator, operator name and timestamp.

Example:

{

    "event": {

        "eventType": "trip-cost",

        "info": {

            "baseCost": "23",

            "token": "69c594cfdeeaedd220",

            "hasPass": "true",

            "planType": "pre-paid",

            "tripId": "tod89430d",

            "operatorType": "t1",

            "operatorName": "Uber",

            "timeStamp": "2020-02-29 20:57:10.294"

        }

    }

}

Operator Management Service:

This service has as input TripCost Event, NewOperator events, NewService events and NewDiscount events. This event has the goal provide information to the Operator Management Service compute the how much money goes to the operator and how much money should be debited from the user account.

**Input**

* + **TripCost Events**

(Described before)

**Output**

This service has as output a Debit Event. This event has the goal to provide information to the Payment Service to debit money from the user account.

* **Debit Events**

These events contain as main fields:

* **eventType** that says “debit”
* **info** that contains information about the debit. The info field contains information about the amount to debit, plan type and the user ID.

Example:

{

    "event": {

        "eventType": "debit",

        "info": {

            "token": "69c594cfdeeaedd220",

            "planType": "pre-paid",

            "amount": "20"

        }

    }

}

1. **Functional integration of the two microservices with the previous Kafka topics**

We started by deleting the topic that we created in the first sprint because we won’t need it anymore:

sudo /usr/local/kafka/bin/kafka-topics.sh --zookeeper <Public\_DNS>:2181, <Public\_DNS>:2182, <Public\_DNS>:2183 --delete --topic Discounts

Then we created two topics: TripCosts and Debit

sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper *localhost:*2181, localhost:2182, localhost:2183 -replication-factor 3 --partitions 3 --topic TripCosts

sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper *localhost:*2181, localhost:2182, localhost:2183 -replication-factor 3 --partitions 3 --topic Debit

The TripCosts topic will be used for the TripCosts events, the Customer Management Service will produce this type of events to this topic for them to be consumed by the Operator Management Service.

The Debit topic will be used for the Debit events, the Operator Management Service will produce this type of events to this topic for them to be consumed by the Payment Service.

We decided to create just one topic for each purpose for simplicity, but we added a replication factor of 3 to ensure some redundancy. The 3 partitions are just to allow the same degree of parallelism that the other topics have.

**(Create new topic for new services?)**

1. **Functional integration of the two microservices with the previous Kafka topics**

We’ve decided to use an AWS Lambda function to implement the User Management Service and javax.jws.WebService library to implement Operator Management Service.

**User Management Service pseudo code:**

**function handleRequest:**

startService <- beginEvent

bd\_connect <- connectToDatabase

consumer <- prepareConsumer

**while(true):**

consumerRecords <- consumer.poll()

**for each** record in consumerRecords **do**

message <- record.value

extractedEvent <- parse(message)

**if** extractedEvent != null and bd\_connect == ok **do**

processEvent(extractedEvent)

**else**

throw exception

**endif**

**endfor**

consumer commit offset

**endwhile**

**end**

**function processEvent:**

eventType <- extractedEvent.type

eventInfo <- extractedEvent.info

**switch**(eventType):

**case** "t0-check-in":

insertT0InfoInDB <- eventInfo

break

**case** "t0-check-out":

insertT0InfoInDB <- eventInfo

break

**case** "t1":

insertT1InfoInDB <- eventInfo

break

**case** "t2":

insertT2InfoInDB <- eventInfo

break

**case** "new-user":

insertUserInDB <- eventInfo

break

**default:**

log

break

**endswitch**

**end**

**Note:** in the insertInfo functions a tripCost event will be sent after inserting data in the database

**Operator Management Service pseudo code:**

1. **Implementation of the two microservices**

UserDB Database configurations:

* + Engine Type: MySQL
  + Version: 5.7.22
  + Template: Free Tier
  + DB instance identifier: userdb
  + DB instance size: db.t2.micro
  + Storage type: General Purpose (SSD)
  + Allocated storage: 20 GiB
  + Enable storage autoscaling: true
  + Maximum storage threshold: 1000 GiB
  + Virtual Private Cloud (VPC): Default VPC
  + Subnet group: default-vpc-8af6c4f0
  + Publicly accessible: Yes
  + VPC Security Groups: default and launch-kafka
  + Availability zone: No preference
  + Database port: 3306
  + Database authentication options: Password Authentication

OperatorDB Database configurations:

* + Engine Type: MySQL
  + Version: 5.7.22
  + Template: Free Tier
  + DB instance identifier: OperatorDB
  + DB instance size: db.t2.micro
  + Storage type: General Purpose (SSD)
  + Allocated storage: 20 GiB
  + Enable storage autoscaling: true
  + Maximum storage threshold: 1000 GiB
  + Virtual Private Cloud (VPC): Default VPC
  + Subnet group: default-vpc-8af6c4f0
  + Publicly accessible: Yes
  + VPC Security Groups: default and launch-kafka
  + Availability zone: No preference
  + Database port: 3306
  + Database authentication options: Password Authentication

We also added an inbound rule in launch-kafka security group on port 3306.

**Relational Model of customerManagementDB**

* userInfo(token, email, firstName, lastName, planType, hasPass)
* userBalance(token, balance)
  + token: FK(userInfo)

We’ve decided to separate the balance from the userInfo table because the balance is going to be accessed by Payment Service and the rest of the tables are going to be accessed by User Management Service, this way we can provide some isolation.

* history(tripID, time\_stamp, token, operatorName)
  + token: FK(userInfo)
* T0\_History(tripID, time\_stamp, station, isCheckIn)
  + tripID,time\_stamp: FK(history)
* T1\_History(tripID, time\_stamp, price)
  + tripID,time\_stamp: FK(history)
* T2\_History(tripID, time\_stamp, time, price)
  + tripID,time\_stamp: FK(history)

**Relational Model of operatorManagementDB**

* operator(operatorId, name)
* service(operatorId, serviceId, name, price)
  + operatorId: FK(operator)
* discount(operatorId, serviced, discountId, value)
  + operatorId,serviceId: FK(service)

1. **Functional testing**