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

Instituto Superior Técnico – MEIC-A

*Sprint 4 Report*

In this Sprint I had to create the Payment Service to debit the prepaid amount from the user account and to trigger the dunning process whenever any client exhausts the plafond to meet the integration between rating and dunning process requirement. In addition, I created the operator provision workflows to meet the providers provision requirements. The improvement that I made was to create a Load Account workflow that loads the user account and stops the dunning process when the user balance becomes bigger than 0 again.

1. **Definition of the mobility operators and respective messages**

Mobility as a Service is an analogy of the usual Software as a Service model popularized by the Cloud for the software industry. The idea is the same, people will use the transport network they see fitting better their needs, be it the Public Transport Operators like Metro or Buses, Taxis or the new players such as Uber or Cabify and all other innovative alternatives for personal transportation like rental bikes, scooters, motorcycles, etc.

The innovation is the seamless use of all of them without the usual difficulties of different ticketing and payments systems, and the negative incentive for such use due to incompatibilities between cards, apps, tariffs, monthly subscriptions, etc.

The main idea of Maas is that one can take any transportation system and in the background his usage is being registered and one will pay for mobility according to the schema that best suits his needs.

As part of the proposal, I aim to integrate all the different transportation operators that provide services in Lisbon. For that purpose, I have defined the architecture of the Maas Operator based on an analysis I made of the types of services that are provided by each operator. I can generalize these services by gathering all of them in three types:

* **Type 0**: Check in and check out method
  + This method is used by Metropolitano de Lisboa and CP, for example
  + The user must have a ticket or a pass that is validated when he enters and when he leaves the transport and the price is fixed

Types of messages:

{

    "event": {

        "eventType": "t0-check-in",

        "operator": "Metro",

        "info": {

            "Id": "1",

            "Station": "Odivelas",

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

        }

    }

}

{

    "event": {

        "eventType": "t0-check-out",

        "operator": "Metro",

        "info": {

            "Id": "1",

            "Station": "Alameda",

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

        }

    }

}

* **Type 1**: Distance and time dependent method
  + This method is used by Uber and Cabify, for example
  + The price of the trip is mainly calculated using the distance and time of the trip

Types of messages:

{

    "event": {

        "eventType": "t1",

        "operator": "Uber",

        "info": {

            "Id": "1",

            "Price": "20.63",

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

        }

    }

}

* **Type 2**: Time dependent method
  + This method is used by GIRA and Lime, for example
  + The price of the trip depends on how much time do you spend using the transport

Types of messages:

{

    "event": {

        "eventType": "t2",

        "operator": "Gira",

        "info": {

            "Id": "1",

            "Time":"3600",

            "Price": "12.60",

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

        }

    }

}

To have a representative example of how the system would work in the “real world”, I chose one operator of each type to make sure that the system would cope with every type of public transportation operating in Lisbon. I chose **Metro, Uber and GIRA**.

1. **Definition of the event queueing integration: Topics, Partitions**

In my system I have defined that every operator has a topic because of two main reasons:

* One topic for company enforces the event load to be more distributed, one topic for all operators would have too much load
* It helps identifying the operator that sent the message

In addition, I have identified the type of service in the name of the topic: **T0\_Metro**, **T1\_Uber** and **T2\_GIRA**. This helps the consumer services to know the type of processing that is needed for the event simply by reading the name of the topic.

The **TripCosts** topic will be used for the TripCost 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.

To provide some parallelism to my system I have created all the topics with 3 partitions and a replication factor of 3.

1. **Event flow diagram**

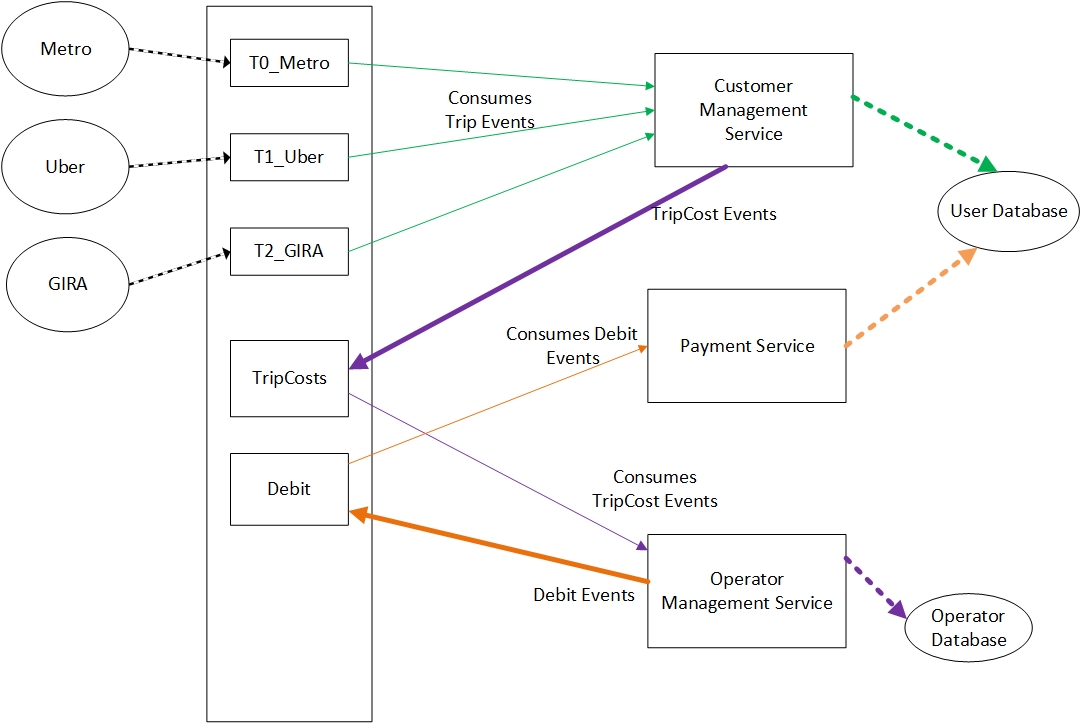


Fig. 1 – Taxation Event flows diagram

1. **Microservices input and output**

Customer Management Service:

**Input**

This service takes as input 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 id of the customer, station, and a timestamp.

Examples:

Check-in in metro

**JSON**

{

    "event": {

        "eventType": "t0-check-in",

        "operator": "Metro",

        "info": {

            "Id": "1",

            "Station": "Odivelas",

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

        }

    }

}

**XML**

<event>

  <eventType>t0-check-in</eventType>

  <info>

      <Station>Odivelas</Station>

      <Timestamp>2020-02-29 18:23:41.278</Timestamp>

      <Id>1</Id>

  </info>

  <operator>Metro</operator>

</event>

Check-out in metro

**JSON**

{

    "event": {

        "eventType": "t0-check-out",

        "operator": "Metro",

        "info": {

            "Id": "1",

            "Station": "Alameda",

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

        }

    }

}

**XML**

<event>

  <eventType>t0-check-out</eventType>

  <info>

      <Station>Alameda</Station>

      <Timestamp>2020-02-29 18:23:47.718</Timestamp>

      <Id>1</Id>

  </info>

  <operator>Metro</operator>

</event>

* **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 id of the customer, price, and a timestamp.

Example:

**JSON**

{

    "event": {

        "eventType": "t1",

        "operator": "Uber",

        "info": {

            "Id": "1",

            "Price": "20.63",

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

        }

    }

}

**XML**

<event>

  <eventType>t1</eventType>

  <info>

      <Price>20.63</Price>

      <Timestamp>2020-02-29 19:45:58.638</Timestamp>

      <Id>1</ Id >

  </info>

  <operator>Uber</operator>

</event>

* **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 id of the customer, time spent with the vehicle, price of the ride and a timestamp.

Example:

**JSON**

{

    "event": {

        "eventType": "t2",

        "operator": "Gira",

        "info": {

            "Id":"1",

            "Time":"3600",

            "Price": "12.60",

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

        }

    }

}

**XML**

<event>

  <eventType>t2</eventType>

  <info>

      <Price>12.60</Price>

      <Time>3600</Time>

      <Timestamp>2020-02-29 20: 57: 10.294</Timestamp>

      <Id>1</Id>

  </info>

  <operator>Gira</operator>

</event>

**Output**

This service has as output a TripCost Event. This event has the goal provide information to the Operator Management Service to compute 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: cost of the trip (null if operator type is t0), id of the user, plan type, operator name and timestamp.

Example:

**JSON**

{

    "event": {

        "eventType": "trip-cost",

        "info": {

            "cost": "23",

            "id": "69c594cfdeeaedd220",

            "planType": "pre-paid",

            "operatorName": "Uber",

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

        }

    }

}

**XML**

<event>

  <eventType>trip-cost</eventType>

  <info>

      <cost>23</cost>

      <operatorName>Uber</operatorName>

      <planType>pre-paid</planType>

      <timeStamp>2020-02-29 20:57:10.294</timeStamp>

      <id>1</id>

  </info>

</event>

Operator Management Service:

This service has as input TripCost Events. This event has the goal provide information to the Operator Management Service to compute 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 id of the user.

Example:

**JSON**

{

    "event": {

        "eventType": "debit",

        "info": {

            "id": "1",

            "planType": "pre-paid",

            "amount": "20"

        }

    }

}

**XML**

<event>

  <eventType>debit</eventType>

  <info>

      <amount>20</amount>

      <planType>pre-paid</planType>

      <id>1</id>

  </info>

</event>

Payment Service:

This service has as input Debit Events. This event has the goal to debit the amount in the event from the user account considering its plan type.

**Input**

* + **Debit Events**

(Described before)

The microservices described before are just used for the taxation event flow handling. I had to create other microservices as Lambda functions for the executable business processes:

* + Create Discount Service
  + Create Operator Service
  + Blacklist User Service
  + Load Account Service
  + Check User Service (checks if user exists in the system)
  + Get Email Service
  + Remove User Service
  + Unique Id Service (checks if the user already exists in the system)
  + User Registration Service

(The inputs and outputs of these microservices are present in the Swagger files folder and the code is in the Microservices folder)

1. **Definition of the fault tolerance requirements for Kafka**

In order to provide **high availability** in my system and as every topic has 3 partitions (number of partitions has to be less or equal to the number of brokers), I’ve created **3 brokers** in the Maas operator, added a **replication factor** of 3 to every topic and configured a **minimum of 2 ISR**. This way I can have a **leader of each topic in every broker** and **I guarantee that there will be another ISR even if the leader fails**, providing a normal service to the users even if two of the brokers fail.

The Kafka cluster durably persists all published records using a configurable retention period. I have set the **retention period** for 48 hours, so for the two days after the record is published, it is available for consumption, after which it will be discarded to free up space. I think that 48 hours more than enough for the records to be consumed.

1. **Kafka configuration**

* **3 brokers were created**

cp /usr/local/kafka/config/server.properties /usr/local/kafka/config/server-1.properties

cp /usr/local/kafka/config/server.properties /usr/local/kafka/config/server-2.properties

cp /usr/local/kafka/config/server.properties /usr/local/kafka/config/server-3.properties

sudo nano /usr/local/kafka/config/server-1.properties

sudo nano /usr/local/kafka/config/server-2.properties

sudo nano /usr/local/kafka/config/server-3.properties

|  |  |  |
| --- | --- | --- |
| Broker-1:  config/server-1.properties:  **broker.id=0**  **listeners=PLAINTEXT://*<Public DNS>*:9092**  **offsets.topic.replication.factor=3**  **transaction.state.log.replication.factor=3**  **transaction.state.log.min.isr=2**  **log.dir=/tmp/kafka-logs-0**  **log.retention.hthes = 48**  **zookeeper.connect=localhost:2181** | Broker-2:  config/server-2.properties:  **broker.id=1**  **listeners=PLAINTEXT://*<Public DNS>*:9093**  **offsets.topic.replication.factor=3**  **transaction.state.log.replication.factor=3**  **transaction.state.log.min.isr=2**  **log.dir=/tmp/kafka-logs-1**  **log.retention.hthes = 48**  **zookeeper.connect=localhost:2181** | Broker-3:  config/server-3.properties:  **broker.id=2**  **listeners=PLAINTEXT://*<Public DNS>*:9094**  **offsets.topic.replication.factor=3**  **transaction.state.log.replication.factor=3**  **transaction.state.log.min.isr=2**  **log.dir=/tmp/kafka-logs-2**  **log.retention.hthes = 48**  **zookeeper.connect=localhost:2181** |

**(The parameters not mentioned here remained with the default values)**

**broker.id –** broker id

**listeners –** the address the broker socket listens on

**offsets.topic.replication.factor -** specify the replication factor for the \_\_consumer\_offsets topic. This topic stores information about committed offsets for each topic:partition per group of consumers

(I’ve set this value to 3 to take advantage of having 3 brokers, providing more redundancy for this information)

**transaction.state.log.replication.factor -** the replication factor for the transaction topic. Internal topic creation will fail until the cluster size meets this replication factor requirement.

(I’ve set this value to 3 to take advantage of having 3 brokers, providing more redundancy for this information)

**transaction.state.log.min.isr -** minimum ISR for this topic

(All the topics will have at least the leader and one replica in sync to continue to provide service)

**log.dir** **–** the directory in which the log data is kept

**log.retention.hthes** **-** the number of hthes to keep a log file before deleting it

(As mentioned before we’ve set it to 48, the messages will be kept for 48 hthes before they are deleted)

**zookeeper.connect -** ZooKeeper connection string

(Contains the addresses for the zookeeper nodes)

* **Opened the in-bound ports 9092, 9093 and 9094 in the AWS EC2 console for the Kafka brokers.**

1. **Zookeeper configuration**

* **I created new directory with the command**

sudo mkdir -p /var/lib/zookeeper

* **I created the baseline zookeeper server configuration with the command**

cat > /usr/local/zookeeper/conf/zoo.cfg << EOF

* **And then wrote the following content to the file directly in the command line**

tickTime=2000

dataDir=/var/lib/zookeeper

clientPort=2181

EOF

* **I opened the in-bound ports 2181 in the AWS EC2 console for the zookeeper nodes**

1. **Topics configuration**

sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 -replication-factor 3 --partitions 3 --topic T0\_METRO

sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 -replication-factor 3 --partitions 3 -–topic T1\_UBER

sudo /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 -replication-factor 3 --partitions 3 –-topic T2\_GIRA

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

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

1. **Database configuration**

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

I also added an inbound rule in the security group on port 3306 for the database connections

1. **User database creation script**

DROP DATABASE IF EXISTS userdb;

CREATE DATABASE IF NOT EXISTS userdb;

USE userdb;

DROP TABLE IF EXISTS userInfo;

CREATE TABLE userInfo

(

    token VARCHAR(100) NOT NULL,

    nif VARCHAR(9) UNIQUE NOT NULL,

    email VARCHAR(50) NOT NULL,

    firstName VARCHAR(20) NOT NULL,

    lastName VARCHAR(20) NOT NULL,

    planType VARCHAR(20) NOT NULL,

    address VARCHAR(100) NOT NULL,

    CONSTRAINT pk\_userInfo PRIMARY KEY (token)

);

DROP TABLE IF EXISTS userBalance;

CREATE TABLE userBalance

(

    token VARCHAR(100) NOT NULL,

    balance INT NOT NULL,

    blackListed BOOLEAN NOT NULL,

    CONSTRAINT pk\_userBalance PRIMARY KEY (token),

    CONSTRAINT fk\_userInfo\_userBalance FOREIGN KEY (token) REFERENCES userInfo(token) on DELETE CASCADE

);

DROP TABLE IF EXISTS history;

CREATE TABLE history

(

    tripID VARCHAR(100) NOT NULL,

    token VARCHAR(100) NOT NULL,

    operatorName VARCHAR(30) NOT NULL,

    time\_stamp DATETIME NOT NULL,

    CONSTRAINT pk\_history PRIMARY KEY (tripID, time\_stamp),

    CONSTRAINT fk\_userInfo\_history FOREIGN KEY (token) REFERENCES userInfo(token) on DELETE CASCADE

);

DROP TABLE IF EXISTS T0\_History;

CREATE TABLE T0\_History

(

    tripID VARCHAR(100) NOT NULL,

    time\_stamp DATETIME NOT NULL,

    station VARCHAR(15) NOT NULL,

    isCheckIn BOOLEAN NOT NULL,

    CONSTRAINT pk\_historyt0 PRIMARY KEY (tripID, time\_stamp),

    CONSTRAINT fk\_historyt0 FOREIGN KEY (tripID, time\_stamp) REFERENCES history(tripID, time\_stamp) on DELETE CASCADE

);

DROP TABLE IF EXISTS T1\_History;

CREATE TABLE T1\_History

(

    tripID VARCHAR(100) NOT NULL,

    time\_stamp DATETIME NOT NULL,

    price DECIMAL (4, 2) NOT NULL,

    CONSTRAINT pk\_historyt1 PRIMARY KEY (tripID, time\_stamp),

    CONSTRAINT fk\_historyt1 FOREIGN KEY (tripID, time\_stamp) REFERENCES history(tripID, time\_stamp) on DELETE CASCADE

);

DROP TABLE IF EXISTS T2\_History;

CREATE TABLE T2\_History

(

    tripID VARCHAR(100) NOT NULL,

    time\_stamp DATETIME NOT NULL,

    time BIGINT NOT NULL,

    price DECIMAL(4, 2) NOT NULL,

    CONSTRAINT pk\_historyt2 PRIMARY KEY (tripID, time\_stamp),

    CONSTRAINT fk\_historyt2 FOREIGN KEY (tripID, time\_stamp) REFERENCES history(tripID, time\_stamp) on DELETE CASCADE

);

1. **Operator database creation script**

DROP DATABASE IF EXISTS operatordb;

CREATE DATABASE IF NOT EXISTS operatordb;

USE operatordb;

CREATE TABLE operator(

    operatorName VARCHAR(100) NOT NULL,

    operatorType VARCHAR(2) NOT NULL,

    price DECIMAL (4, 2),

    CONSTRAINT pk\_operator PRIMARY KEY (operatorName)

);

CREATE TABLE discount(

    discountId VARCHAR(100) NOT NULL UNIQUE,

    discountName VARCHAR(100) NOT NULL UNIQUE,

    value INT NOT NULL,

    beginAt DATETIME NOT NULL,

    endAt DATETIME NOT NULL,

    CONSTRAINT pk\_discount PRIMARY KEY (discountId)

);

CREATE TABLE planType(

    plan VARCHAR(20) NOT NULL,

    CONSTRAINT pk\_planType PRIMARY KEY (plan)

);

CREATE TABLE discount\_planType

(

    discountId VARCHAR(100) NOT NULL,

    plan VARCHAR(20) NOT NULL,

    CONSTRAINT pk\_discount\_planType PRIMARY KEY (discountId,plan),

    CONSTRAINT fk\_planType FOREIGN KEY (plan) REFERENCES planType(plan) on DELETE CASCADE

);

CREATE TABLE operator\_discount(

    operatorName VARCHAR(100) NOT NULL,

    discountId VARCHAR(100) NOT NULL,

    CONSTRAINT pk\_operator\_discount PRIMARY KEY (operatorName,discountId),

    CONSTRAINT fk\_operator FOREIGN KEY (operatorName) REFERENCES operator(operatorName) on DELETE CASCADE,

    CONSTRAINT fk\_discount FOREIGN KEY (discountId) REFERENCES discount(discountId) on DELETE CASCADE

);

1. **Kong configurations**

**--CREATES USER REGISTRATION SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=UserRegistration' --data 'url=https://7zp5uskhi8.execute-api.us-east-1.amazonaws.com/default/UserRegistration'

**--ADDS ROUTE FOR NEW USER CREATION**

curl -i -X POST --url http://localhost:8001/services/UserRegistration/routes --data 'hosts[]=new-user.com'

**--CREATES USER UNIQUE ID VALIDATION SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=UniqueIDValidation' --data 'url=https://8gyz42fgd6.execute-api.us-east-1.amazonaws.com/default/UniqueIDValidation'

**--ADDS ROUTE FOR UNIQUE ID VALIDATION**

curl -i -X POST --url http://localhost:8001/services/UniqueIDValidation/routes --data 'hosts[]=unique-id.com'

**--CREATES LOAD ACCOUNT SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=LoadAccountService' --data 'url=https://zhx0o69m0i.execute-api.us-east-1.amazonaws.com/default/LoadAccountService'

**--ADDS ROUTE FOR ACCOUNT LOADING**

curl -i -X POST --url http://localhost:8001/services/LoadAccountService/routes --data 'hosts[]=load-account.com'

**--CREATES BLACKLIST USER SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=BlacklistUserService' --data 'url=https://z62m3l4rh2.execute-api.us-east-1.amazonaws.com/default/BlackListUser'

**--ADDS ROUTE FOR USER BLACKLISTING**

curl -i -X POST --url http://localhost:8001/services/BlacklistUserService/routes --data 'hosts[]=blacklist-user.com'

**--CREATES USER REMOVAL SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=UserRemovalService' --data 'url=https://hbjv9al4p4.execute-api.us-east 1.amazonaws.com/default/UserRemoval'

**--ADDS ROUTE FOR USER REMOVAL**

curl -i -X POST --url http://localhost:8001/services/UserRemovalService/routes --data 'hosts[]=remove-user.com'

**--CREATES CHECK USER SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=CheckUserService' --data 'url=https://qokp7xifaf.execute-api.us-east-1.amazonaws.com/default/CheckUser'

**--ADDS ROUTE FOR USER CHECKING**

curl -i -X POST --url http://localhost:8001/services/CheckUserService/routes --data 'hosts[]=check-user.com'

**--CREATES USER EMAIL SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=GetUserEmailService' --data 'url=https://xgtr8btwm8.execute-api.us-east-1.amazonaws.com/default/GetUserEmail'

**--ADDS ROUTE FOR USER EMAIL RETRIEVAL**

curl -i -X POST --url http://localhost:8001/services/GetUserEmailService/routes --data 'hosts[]=get-email.com'

**--CREATES OPERATOR CREATION SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=OperatorCreationService' --data 'url=https://ysuo5wgpg1.execute-api.us-eaault/OperatorCreation'

**--ADDS ROUTE FOR OPERATOR CREATION**

curl -i -X POST --url http://localhost:8001/services/OperatorCreationService/routes --data 'hosts[]=create-operator.com'

**--CREATES DISCOUNT CREATION SERVICE**

curl -i -X POST --url http://localhost:8001/services/ --data 'name=DiscountCreationService' --data 'url=https://5jgqj5i8z3.execute-api.us-eaault/DiscountCreation'

**--ADDS ROUTE FOR DISCOUNT CREATION**

curl -i -X POST --url http://localhost:8001/services/DiscountCreationService/routes --data 'hosts[]=create-discount.com'

1. **Executable processes**

**Costumer Provision process**:

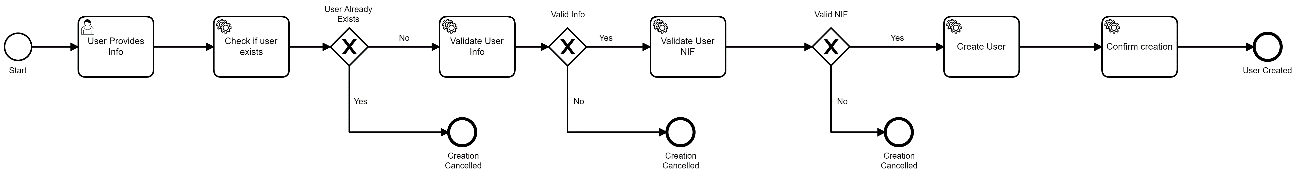


Fig. 2 – User Registration Process

**Dunning Process:**

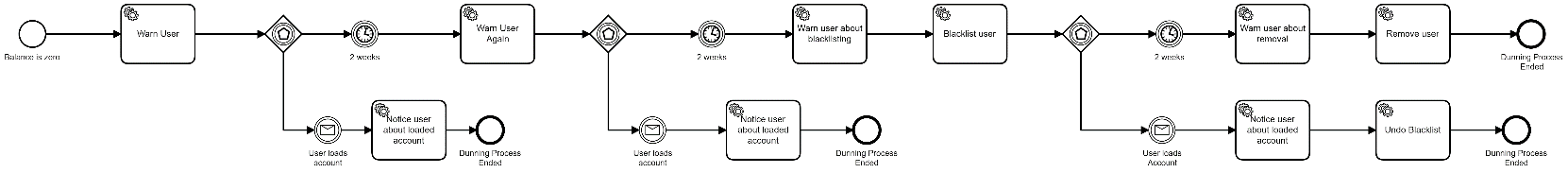
****

Fig. 3 – Dunning Process

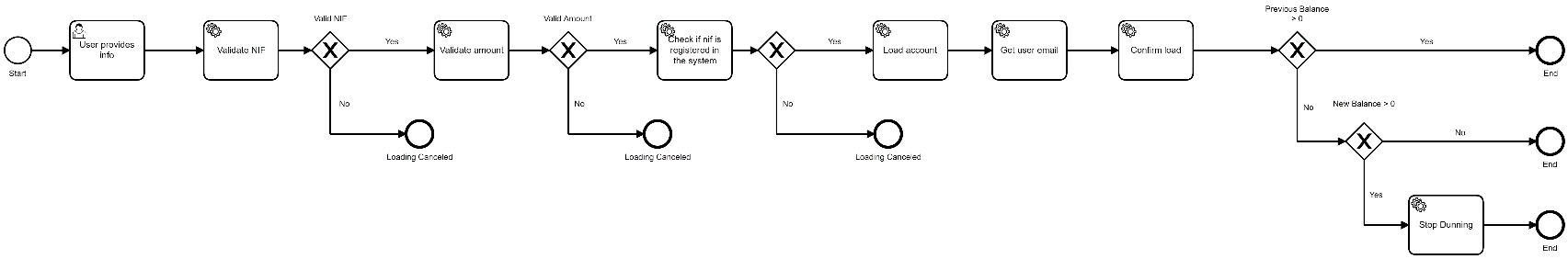
**Account Loading Process:**

Fig. 4 – Account Loading Process

**Operator Creation Process:**

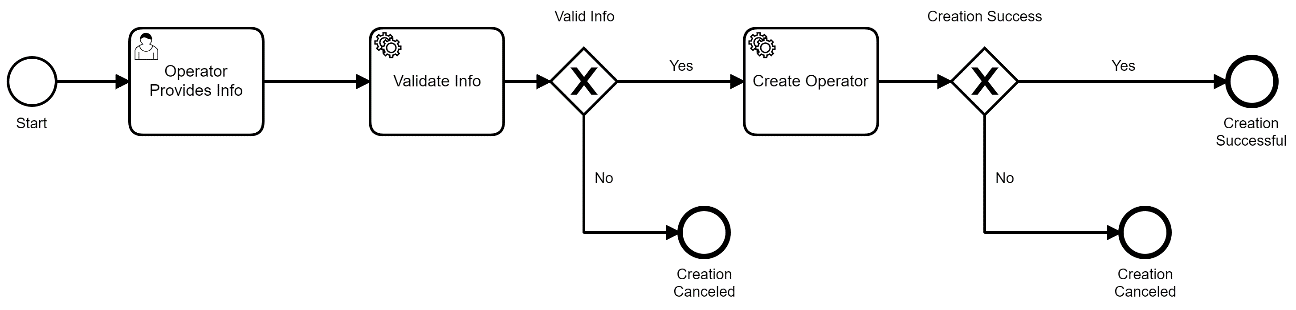


Fig. 5 – Operator Creation Process

**Discount Creation Process:**

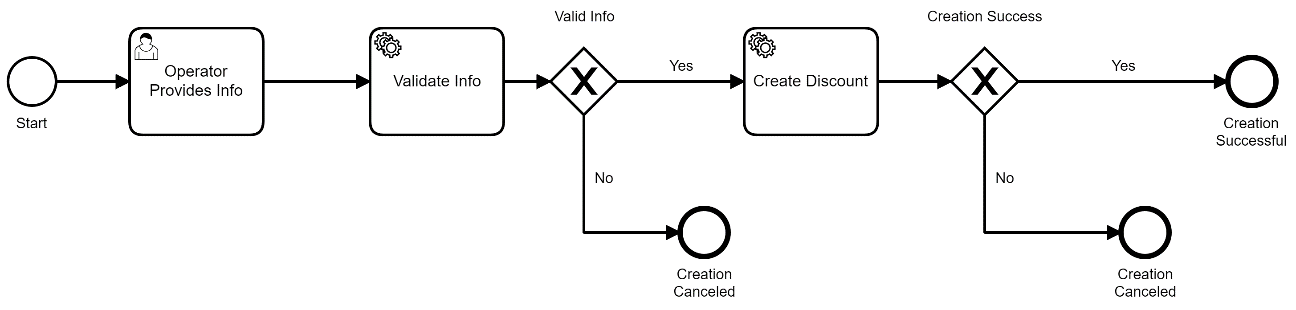
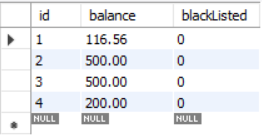


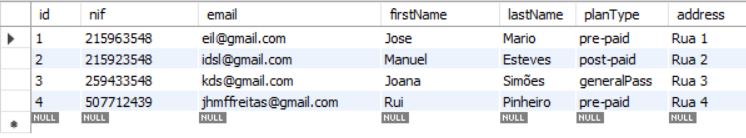
Fig. 6 – Discount Creation Process

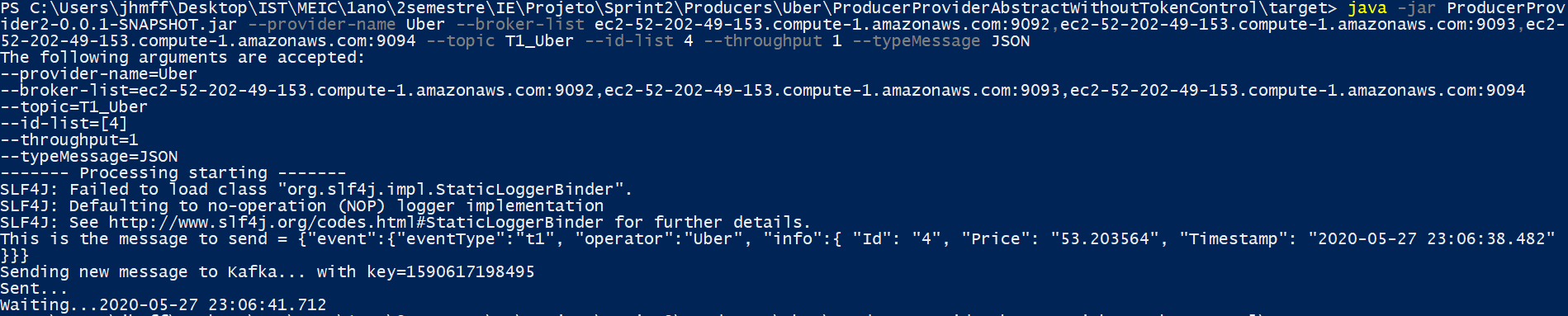
1. **Functional Testing**

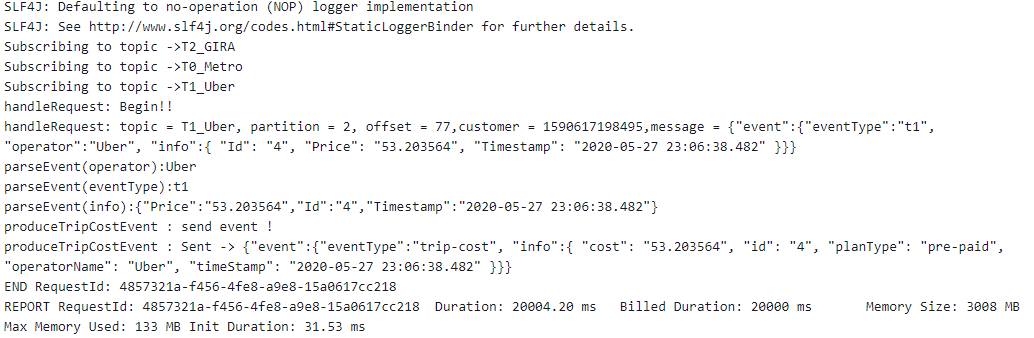
**Trip Event Handling**

Database Initial State:

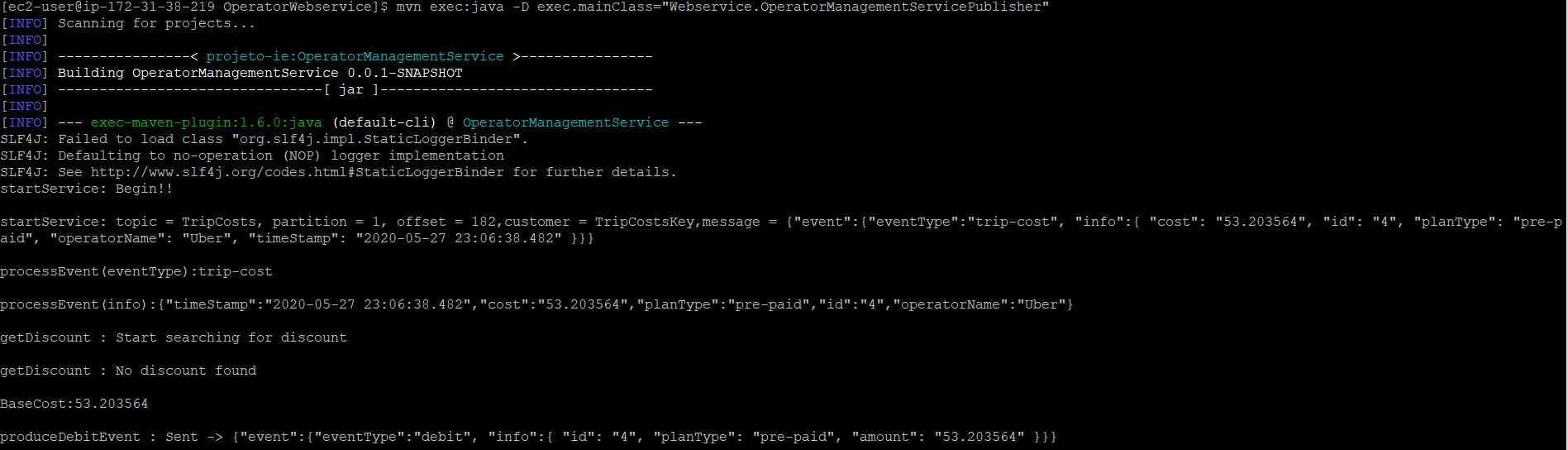
****

****

1. I started by creating an Uber Producer and produced a trip event for user with id 4
2. Then I started the UserManagementService to consume this event and produce a TripCost Event

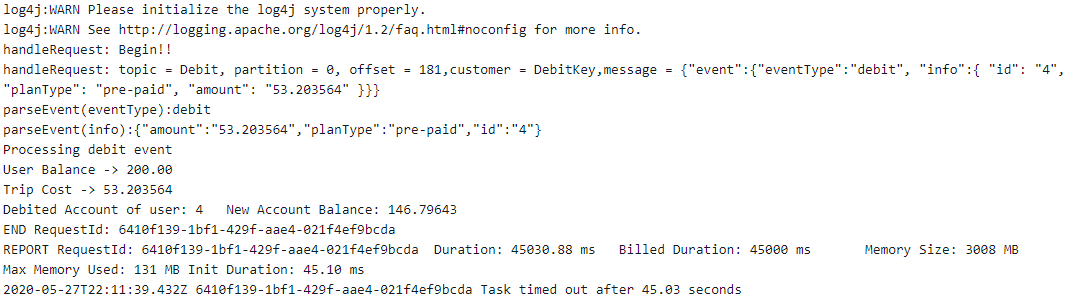


It successfully gathered all the correct information and sent the TripCost event to the OperatorManagementService

1. Then I started the OperatorManagementService to consume this event and produce a Debit Event

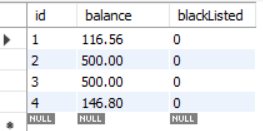
It gathered the correct information and sent the Debit event to the PaymentService

1. Finally, I started the PaymentService to consume this event and debit 53.20€ from the user account

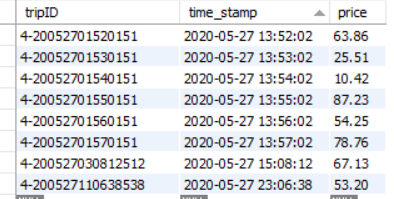


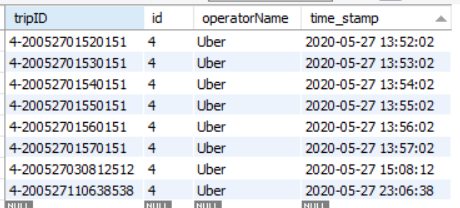
Database final state:

The amount was debited from the user account:

****

The user history was also updated:

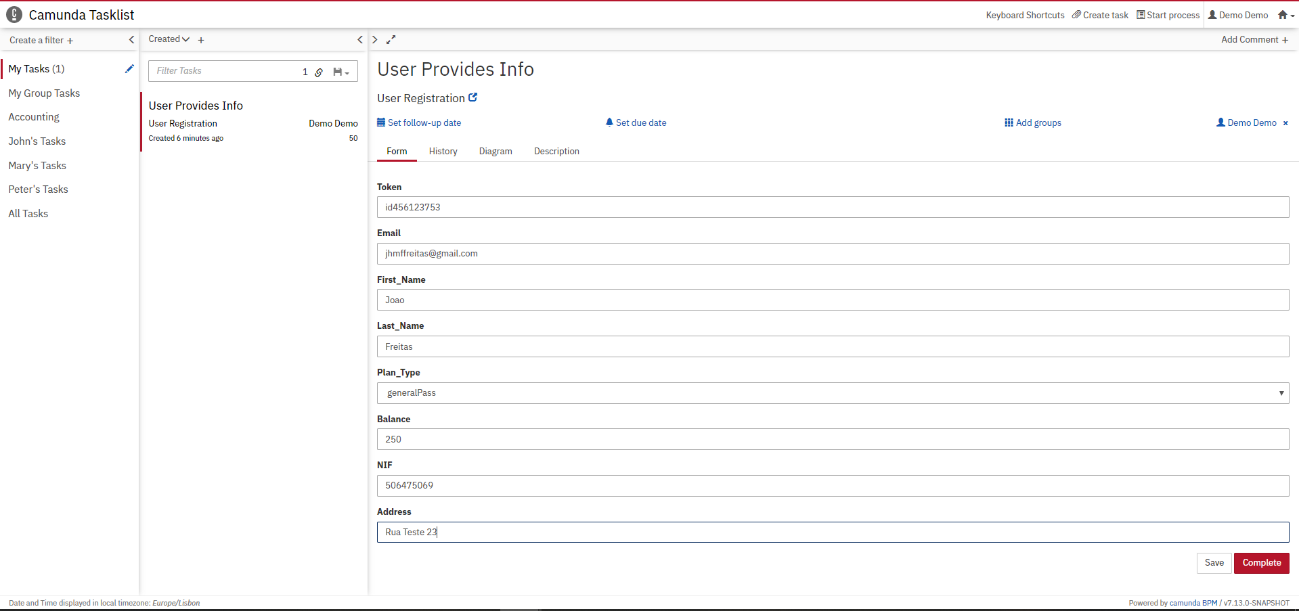




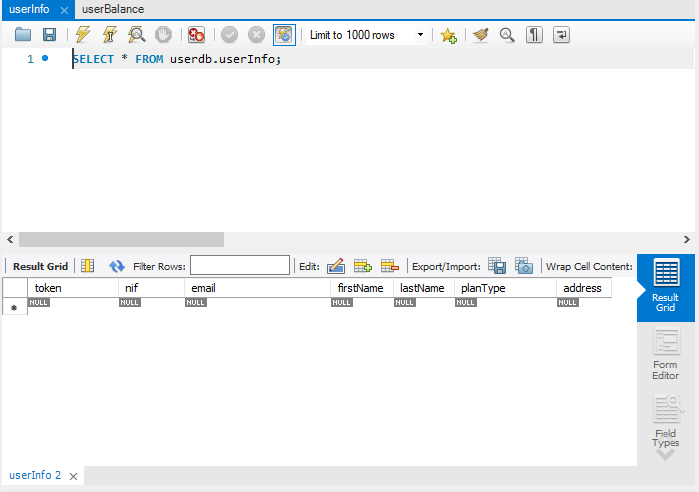
**Customer Provision - User Registration Process**

* User is created successfully:

1. When I start the process, the following form shows up on Camunda Tasklist, this corresponds to the User Provides Info Task. In this case I will provide correct inputs to all fields to show how this process works when everything goes well

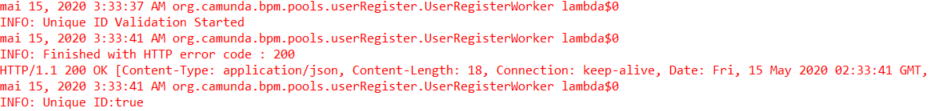


At the beginning of the process the database tables were empty:



1. The first task in the workflow is to check if the user exists in the database:

This image shows the console output when I ran the validation client



This is image shows part of the request made to the validation service retrieved from the CloudWatch Logs, which confirms the call made to the service simply by analyzing the timestamp and the request body.



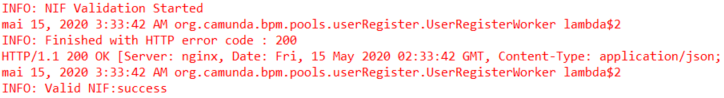
1. The next task is to validate the userInfo by checking if there are any empty or null fields:

I can see in the console that the validation is made and succeeds



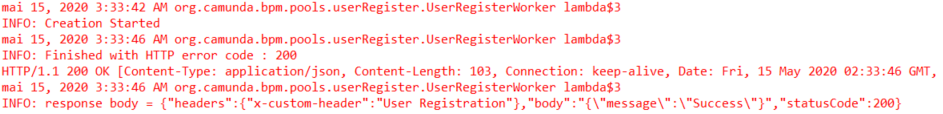
1. Next, I have the validation of the NIF task:

In the console I can verify that indeed this validation was successful, thus can be confirmed by checking the last log line. The “success” printed in extracted directly from the webservice response result.

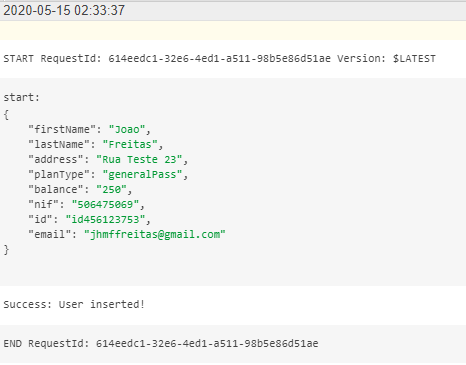


1. Finally, I have the creation of the user task:

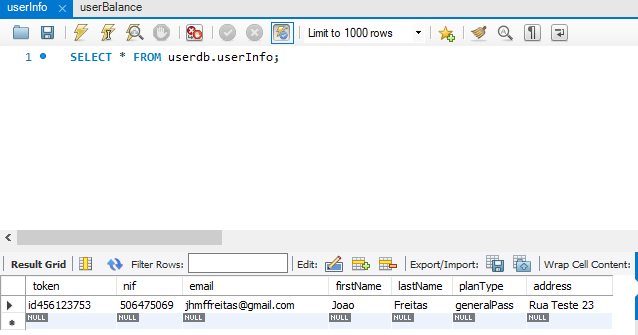
In the console I can see that the operation succeeded.

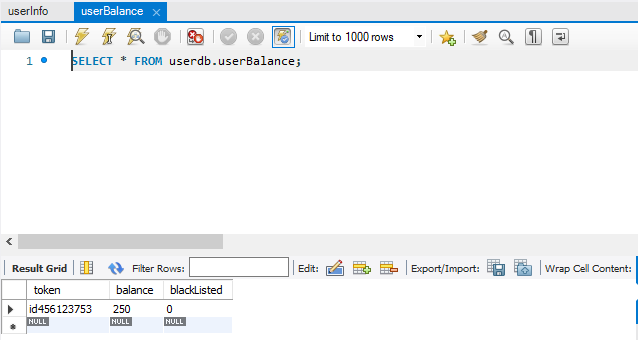


For confirmation purposes I present here the CloudWatch Log of the corresponding request:



Also here are the entries created in the database:



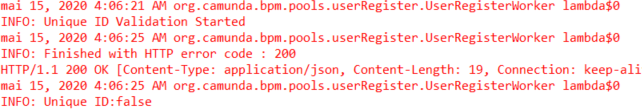


Now I am going to present the failure cases:

* User already exists

I will start the process again and try to insert a user with the same data as the previous one.

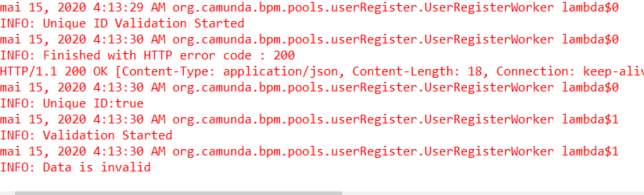
As I can see, as the user already existed, the process ended after the first task by following the Yes branch in the “User Already Exists” gateway



* Invalid data is provided

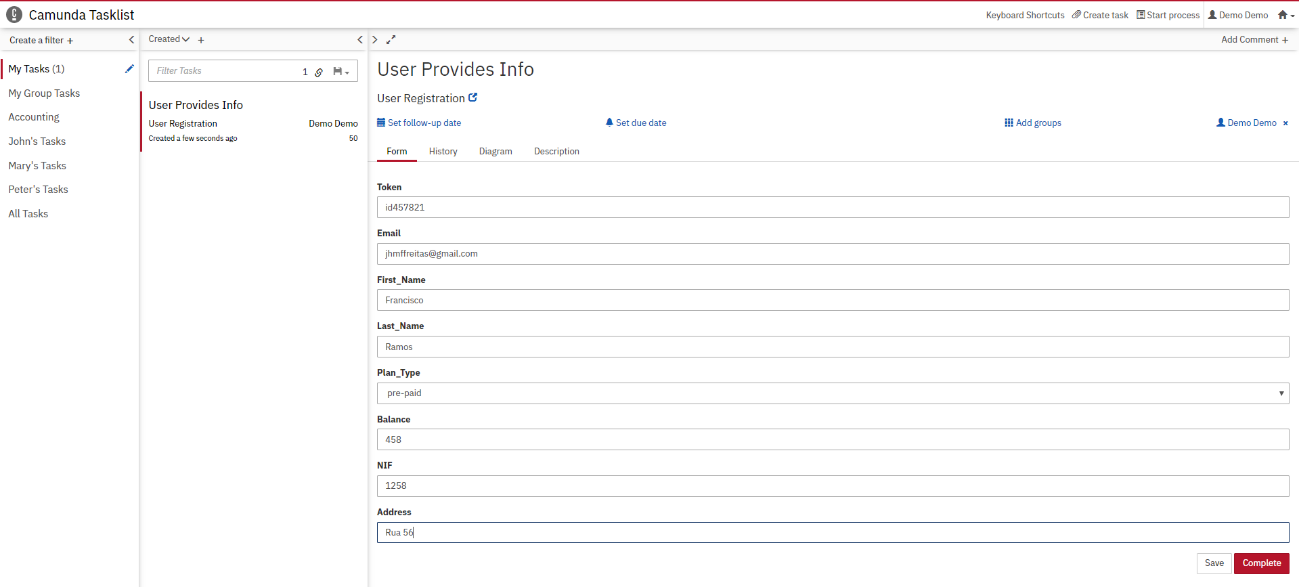
I will start the process again and try to insert a user with different token but with empty fields in the first name and last name for example

As I can see, because of the empty fields, the process ended after the second task by following the No branch in the “Valid Info” gateway

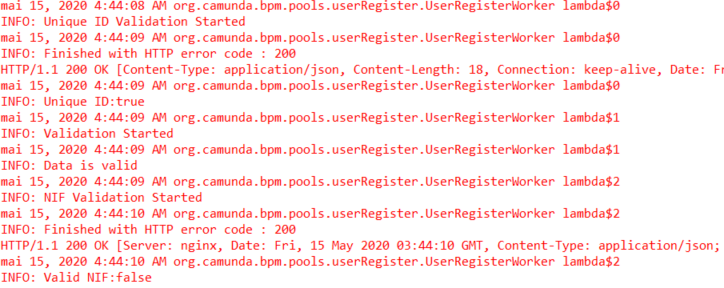


* Invalid NIF is provided

I will start the process again and try to create a user with an invalid NIF but the other parameters are valid

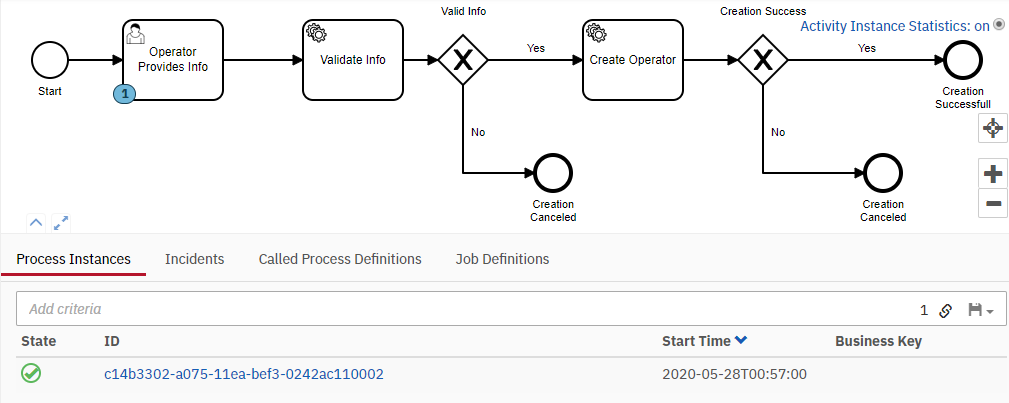


As I can see, as the NIF is invalid, the process ended after the third task by following the No branch in the “Valid NIF” gateway

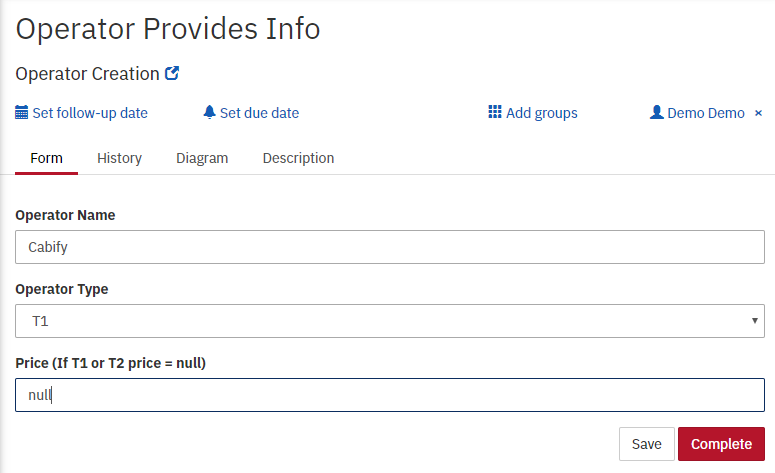


**Operator Provision – Operator Creation Process**

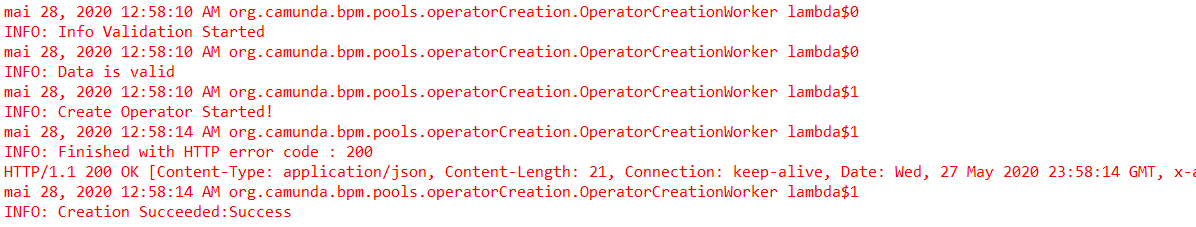
1. I started the process:

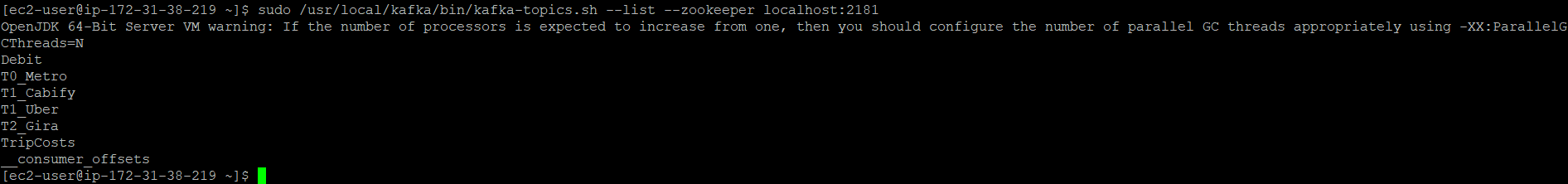


1. Then I provided the information needed:

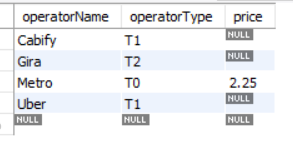


1. Then I checked the logs of the worker, it seems that the creation succeeded



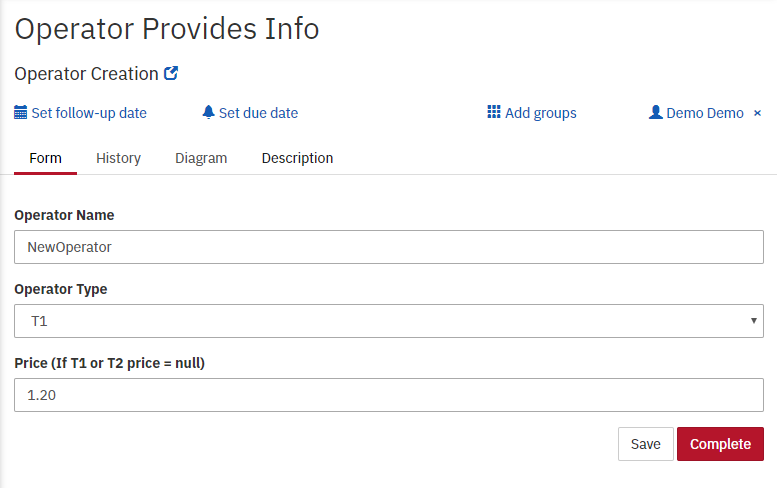
It created the topic in kafka:

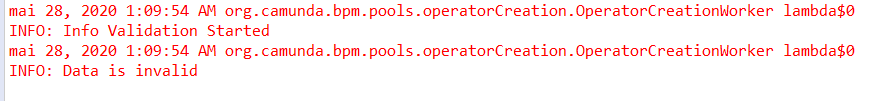
It also created the operator in the database:



* Invalid data is provided

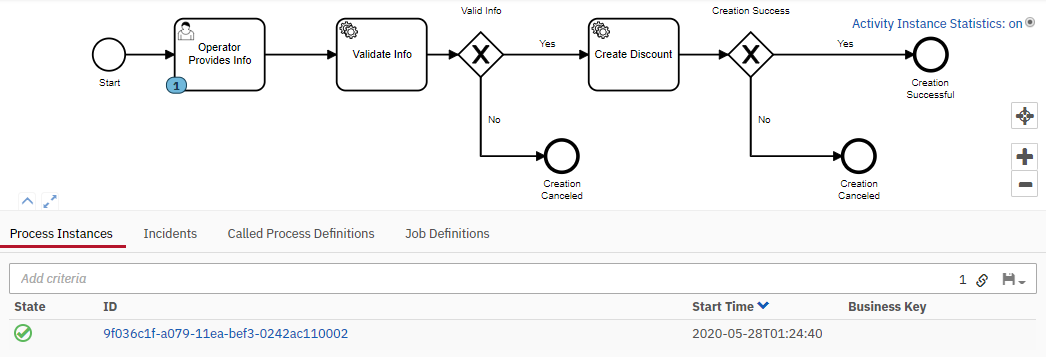
If I try to create an Operator of type T1 with a non null value for the price for example, it detects invalid data and cancels the process



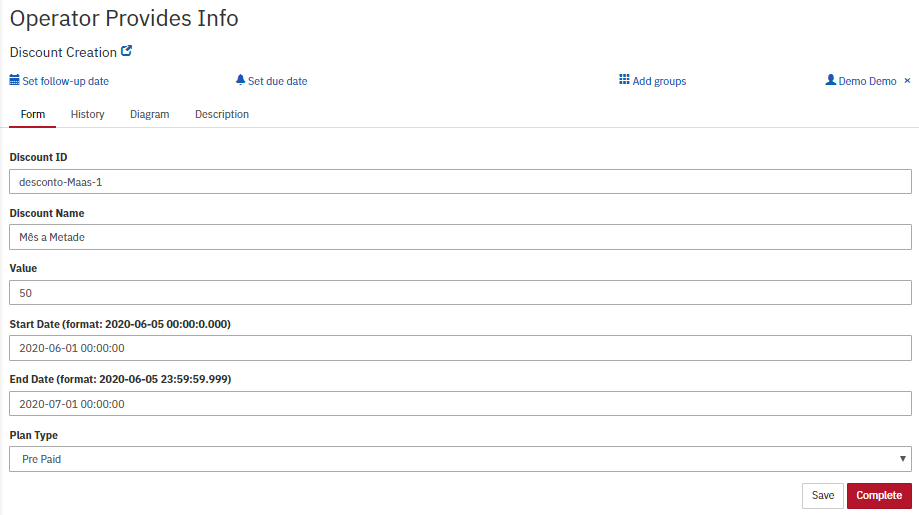


**Operator Provision – Discount Creation Process**

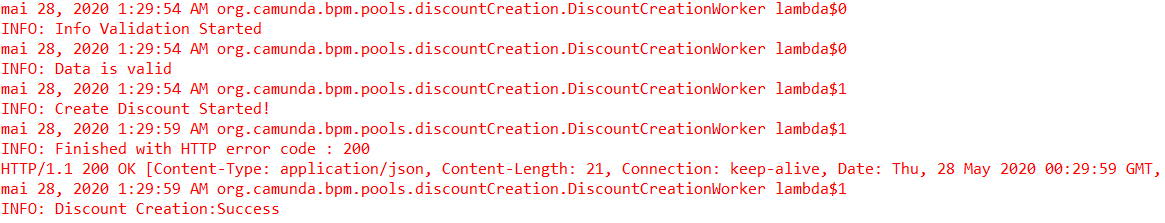
1. I started the process:



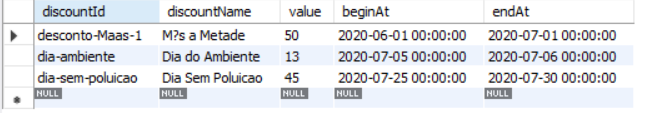
1. Then I provided the information needed:



1. Then I checked the log:

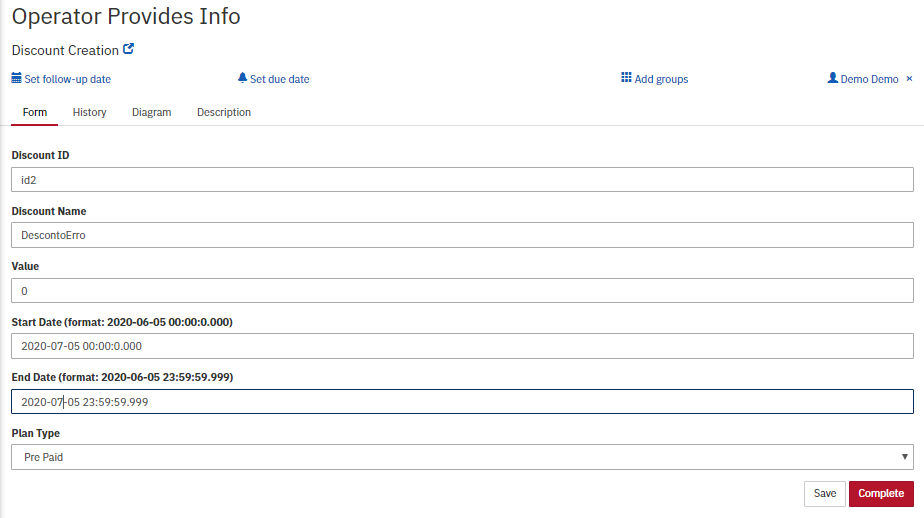


The creation succeeded, then I checked the database:

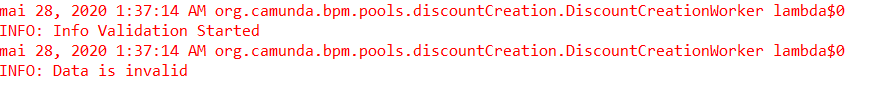


* Invalid data is provided

1. I tried to create a discount with a value equal to 0:

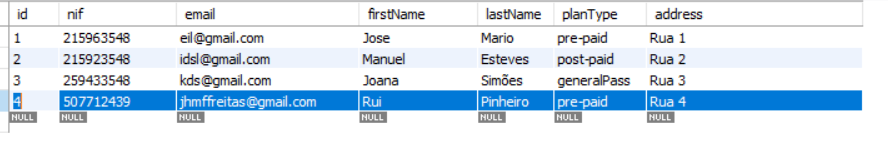


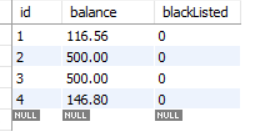
1. The creation was cancelled:



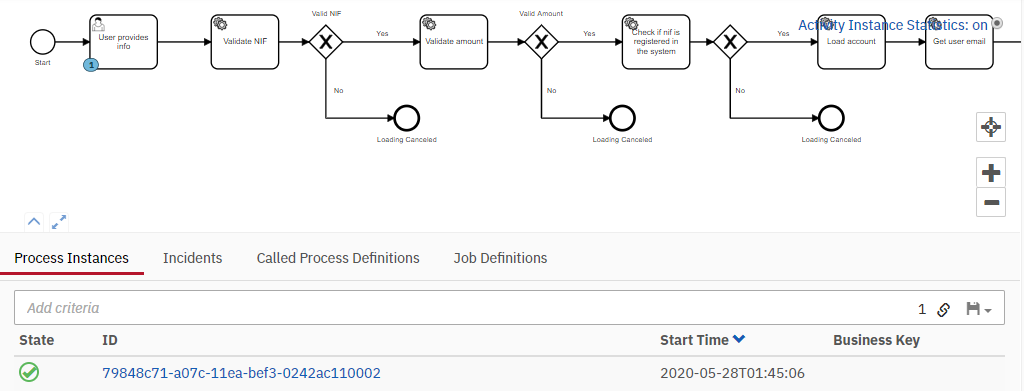
**Load Account Process**

I used the information from user with id 4 for this test:

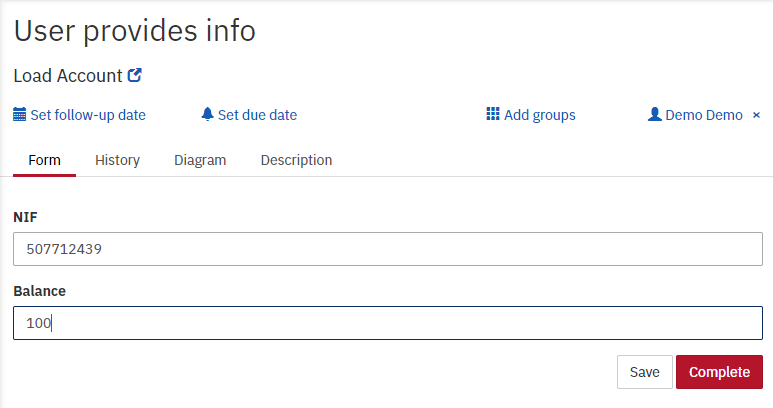
****

****

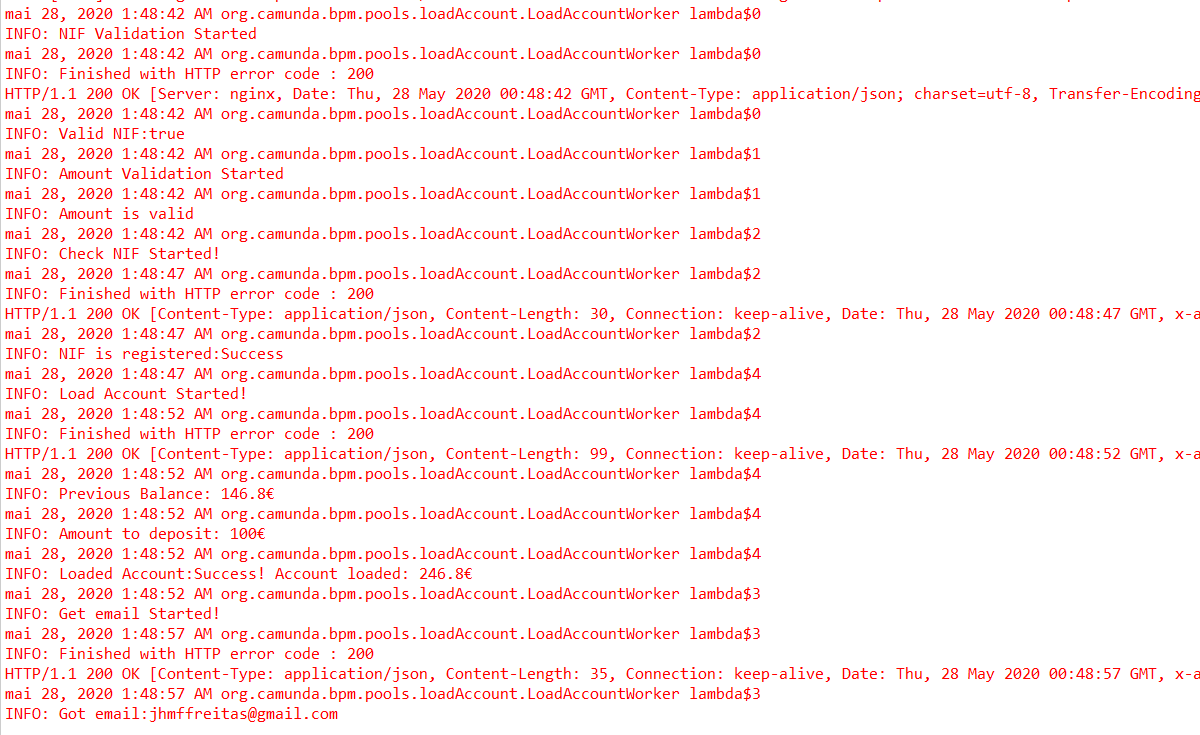
1. I started the process:



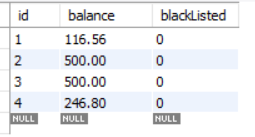
1. Provided the needed info:



1. Then I checked the log:



The creation succeeded, then I checked the database to verify the user balance:



I also received the confirmation email:



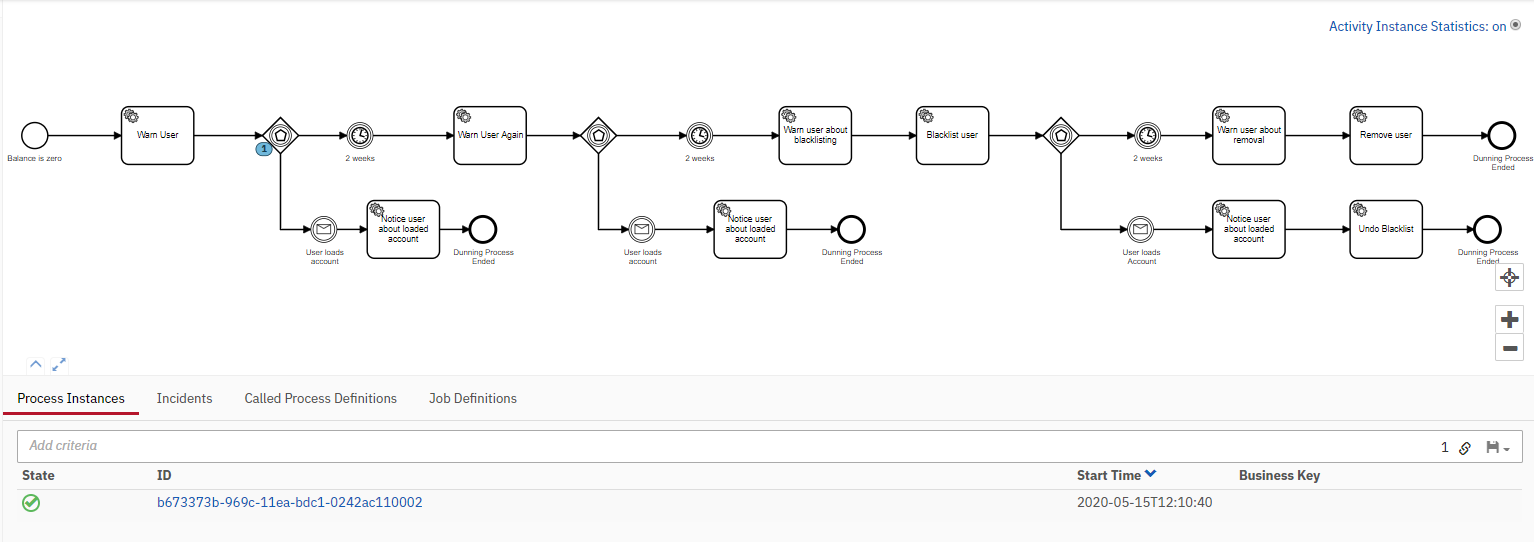
**Dunning Process**

* User never loads the account

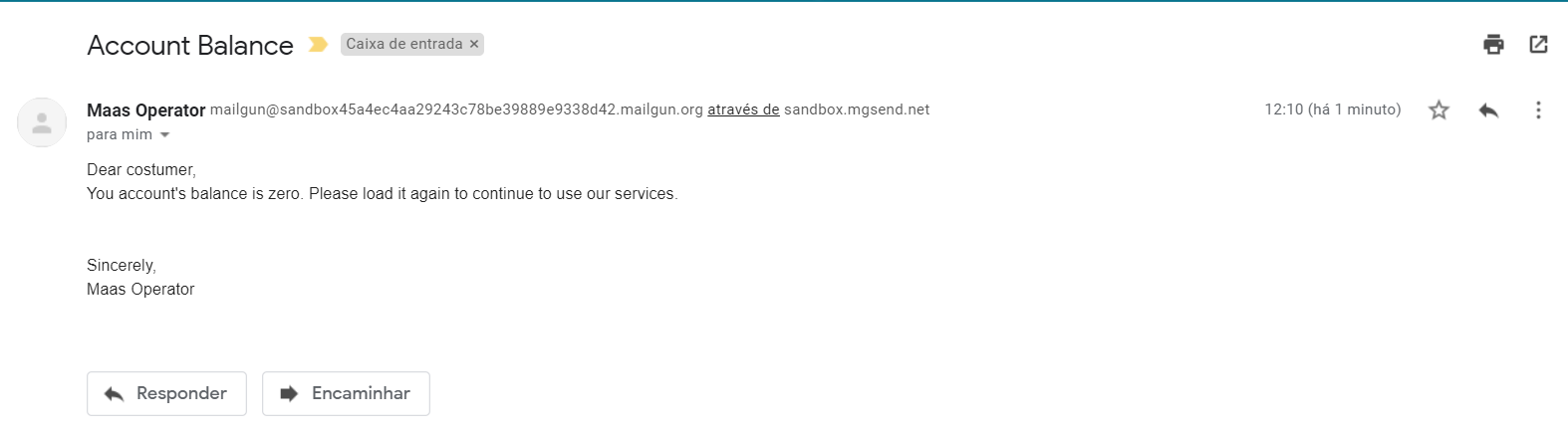
1. I will use the previously created user to execute de dunning process. In addition, I will use my personal email to check if the emails are sent.
2. I started the process with this command:

curl -H "Content-Type: application/json" -X POST -d '{"variables": {"fromEmail": {"value":"Maas Operator <mailgun@sandbox45a4ec4aa29243c78be39889e9338d42.mailgun.org>"},"email":{"value":"jhmffreitas@gmail.com"},"token": {"value":"id456123753"} } }' <http://192.168.99.100:8080/engine-rest/process-definition/key/dunning-process/start>

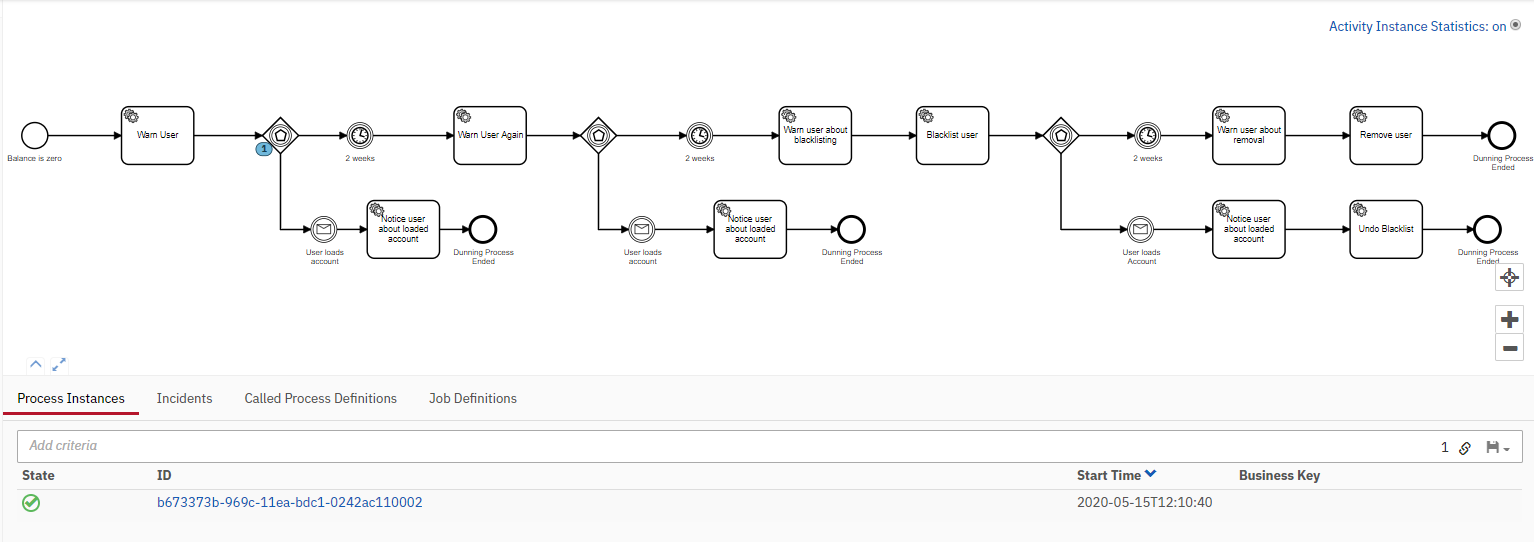
1. The process started at 12:10:



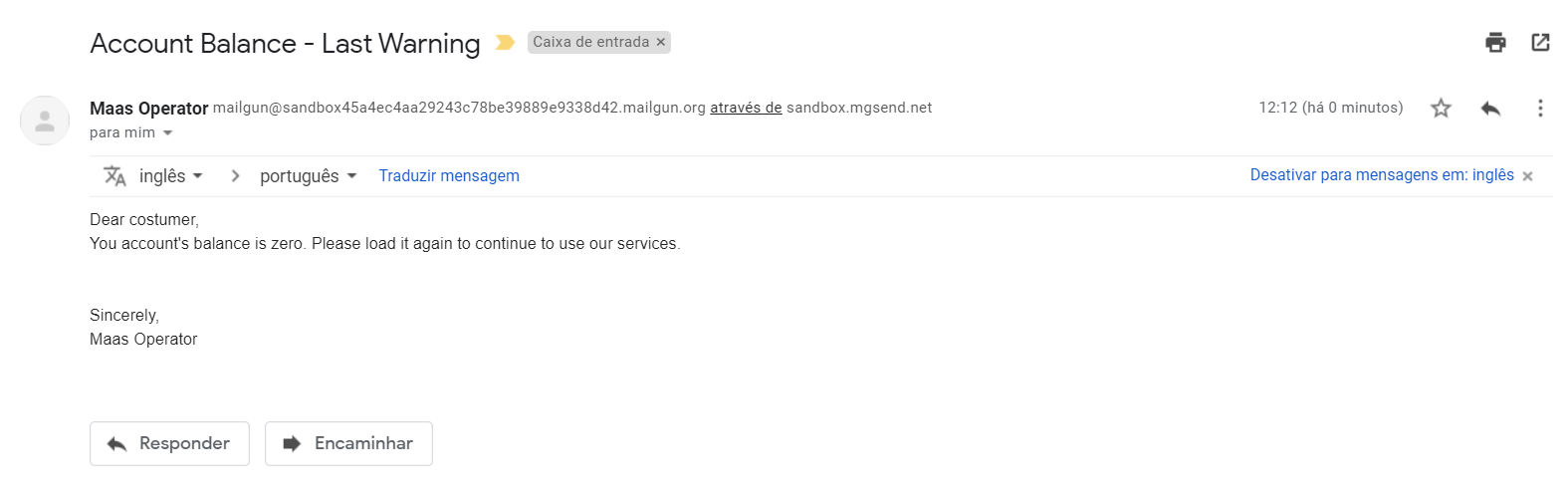
When the process started, I received this email:



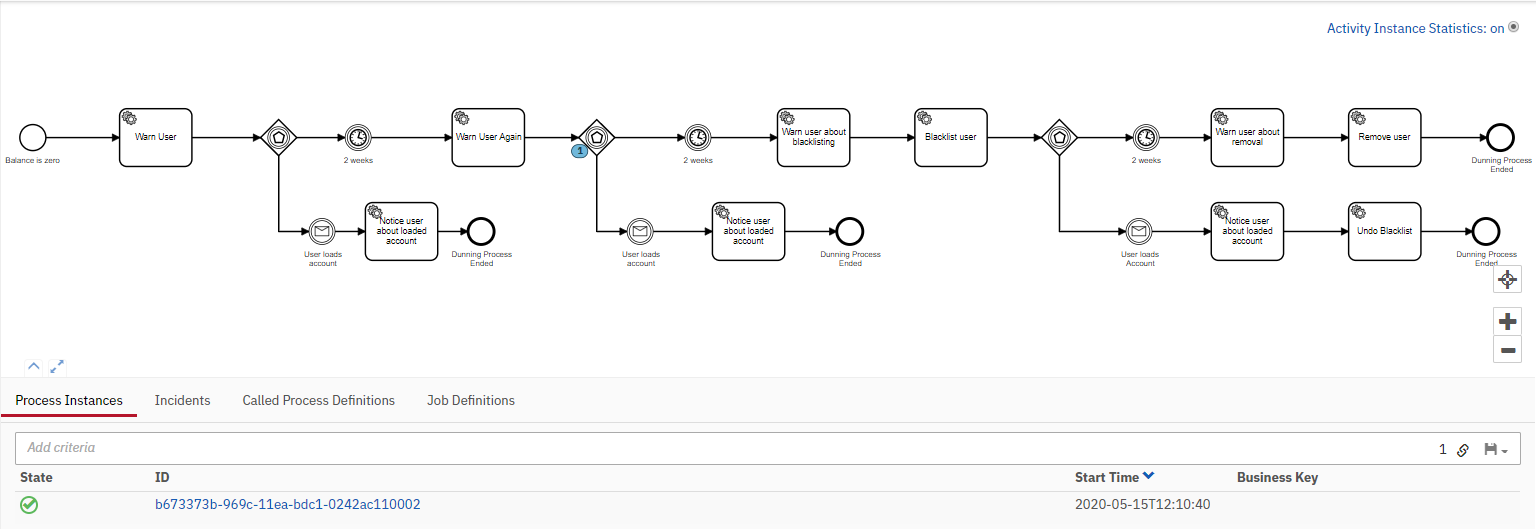
Then it remained waiting for 2 weeks to pass (60 seconds):



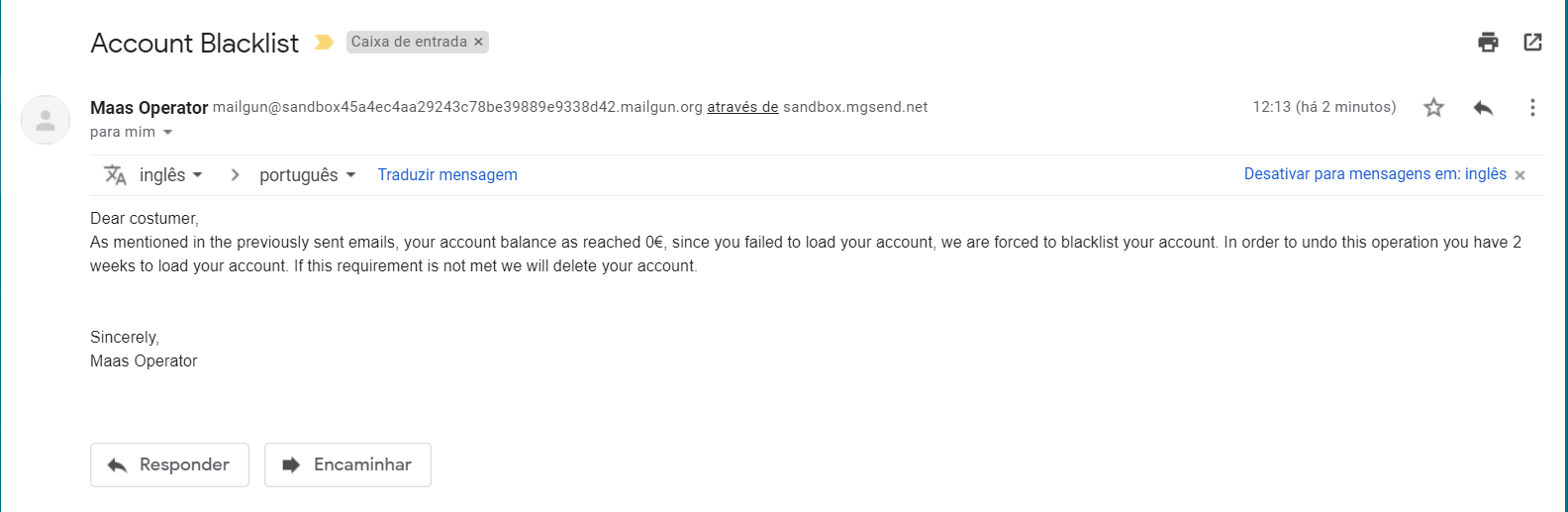
1. As I did not load the account it carried on warning the user again:



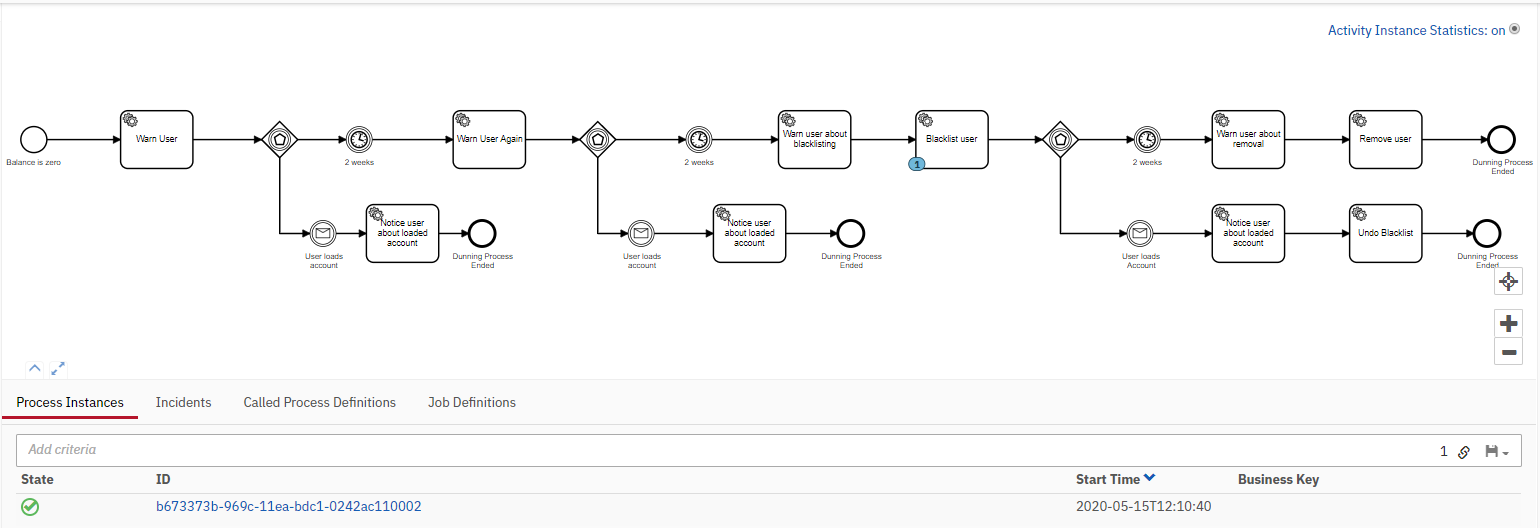
Then it remained waiting for 2 weeks to pass (60 seconds):



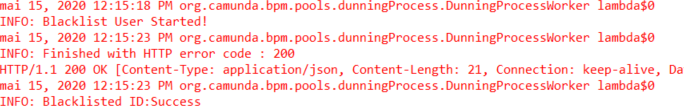
1. As I did not load the account again it sent the user a notice about blacklisting the account:



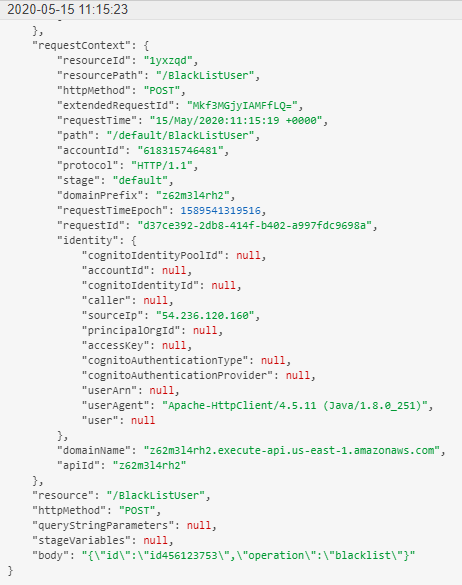
Then started the blacklisting task:



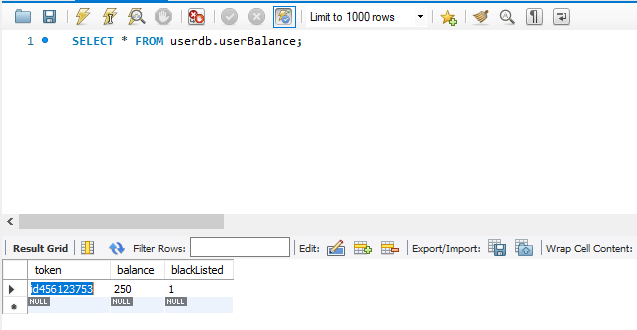
I can check in the console:



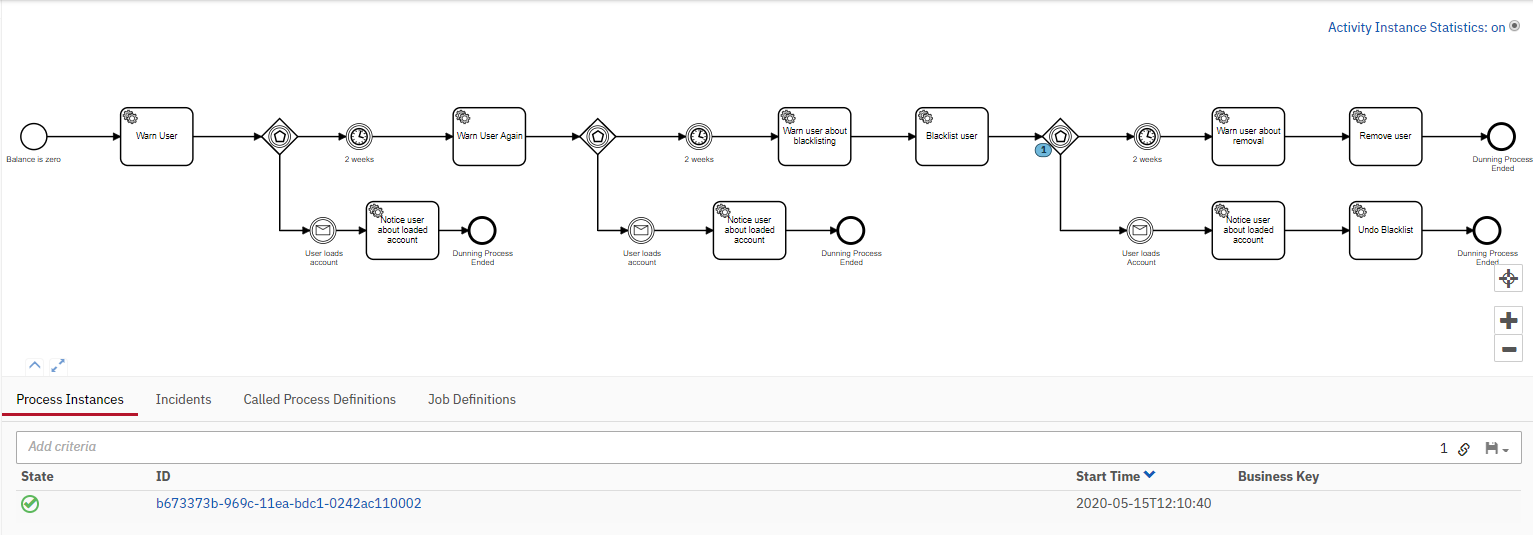
Here is the CloudWatch Log of the Blacklisting Service:



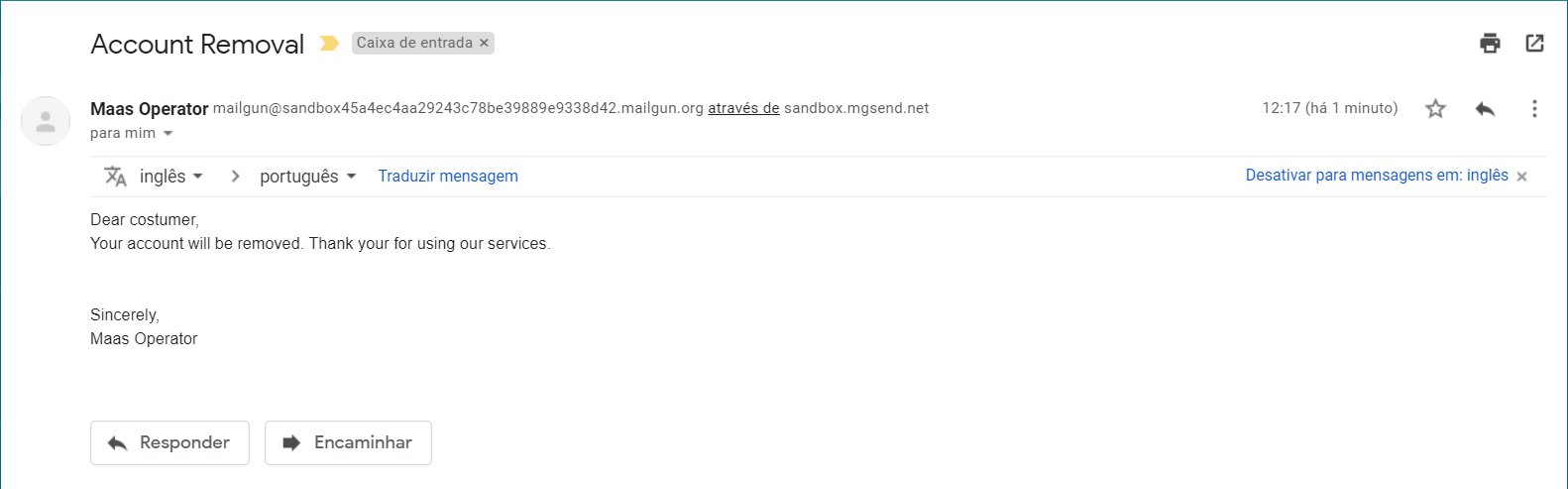
After this I can check the database, the blacklist flag is set to true:



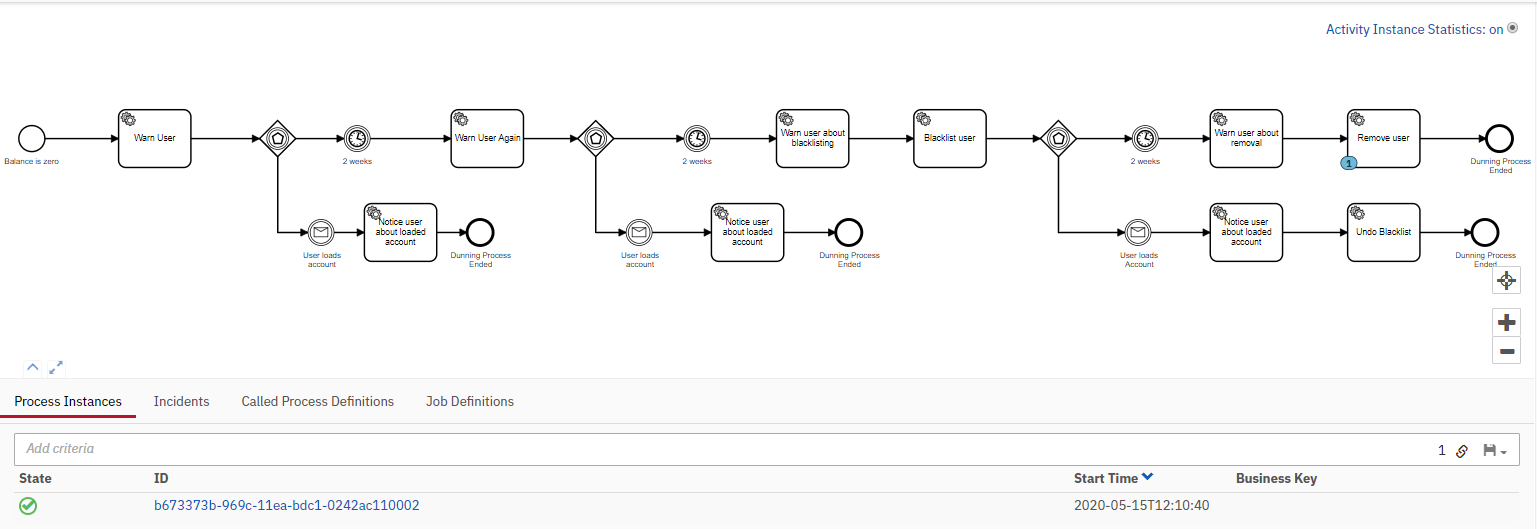
After this the process waited for 2 weeks (60 seconds):



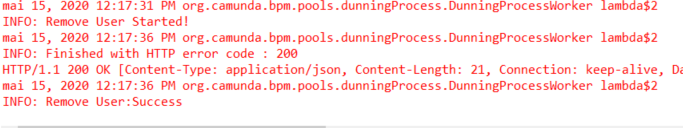
1. As I did not load the account again during this period, it warned the user about the removal:



Then it started the removal task:



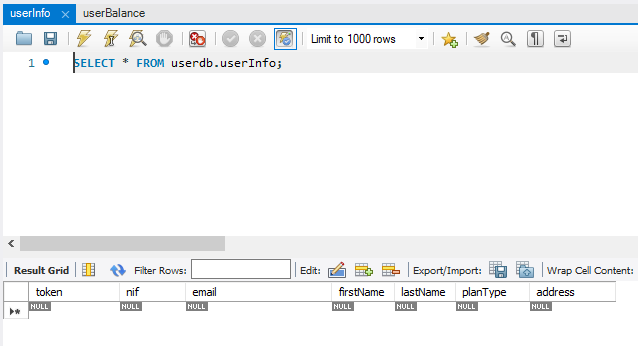
I can check in the console that the operation was successful, and the process ended:



Here is the CloudWatch Log of the Removal Service:

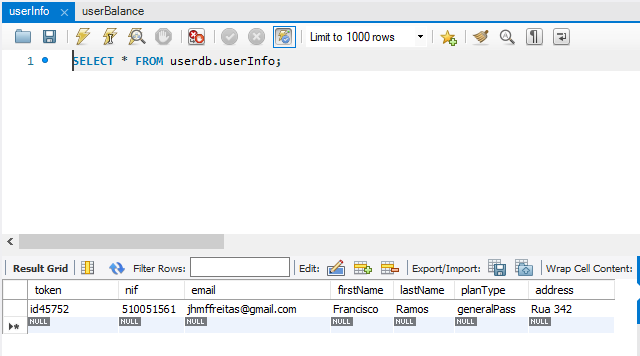


The user is not in the database anymore:



* User Loads account after first warning

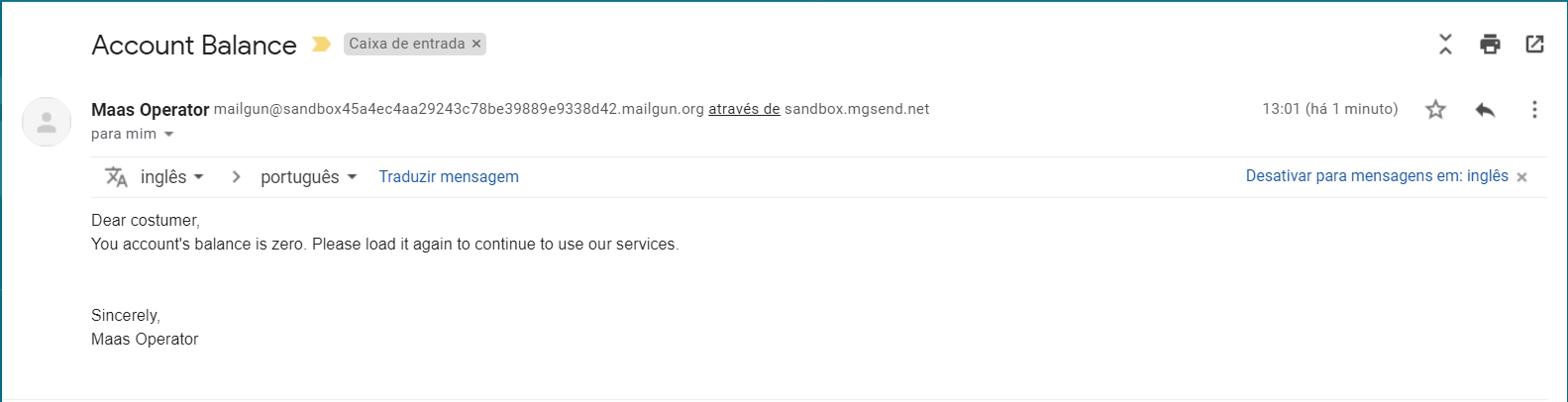
1. I created a new user in the database:



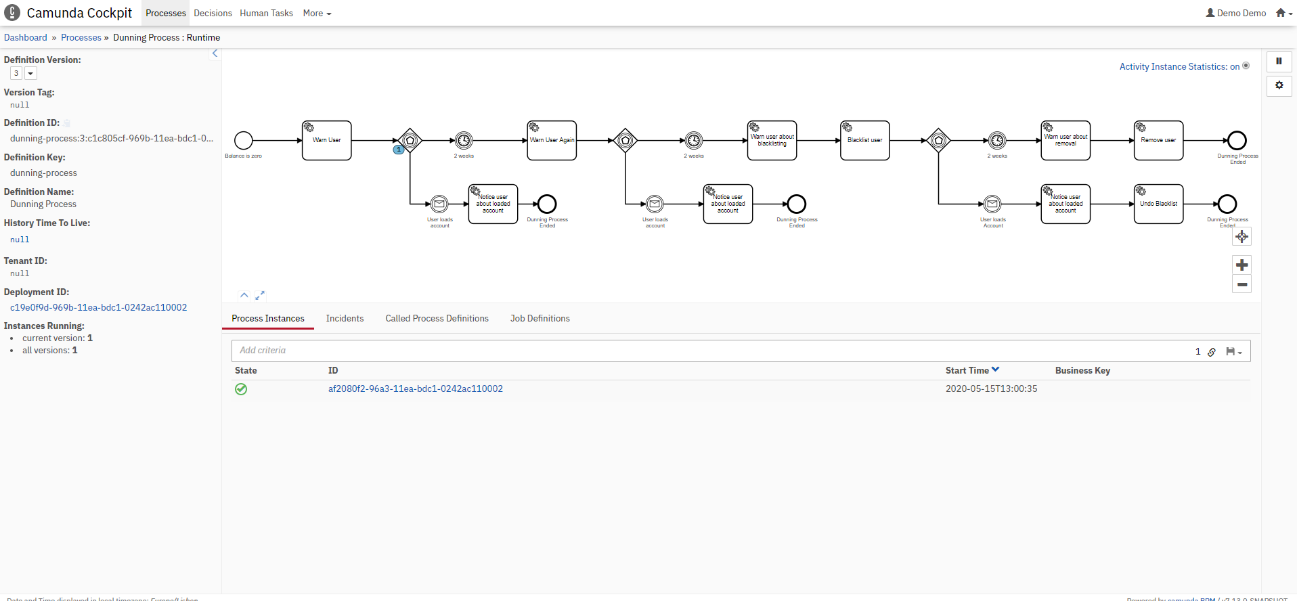
I started the process again with the command:

curl -H "Content-Type: application/json" -X POST -d '{"variables": {"fromEmail": {"value":"Maas Operator <mailgun@sandbox45a4ec4aa29243c78be39889e9338d42.mailgun.org>"},"email":{"value":"jhmffreitas@gmail.com"},"token": {"value":"id45752"} } }' <http://192.168.99.100:8080/engine-rest/process-definition/key/dunning-process/start>

I received the email:



Then the process started waiting for the two weeks:



1. When the process was waiting for the two weeks to pass, I ran this command:

curl -H "Content-Type: application/json" -X POST --data @userLoadsAccount.json <http://192.168.99.100:8080/engine-rest/message>

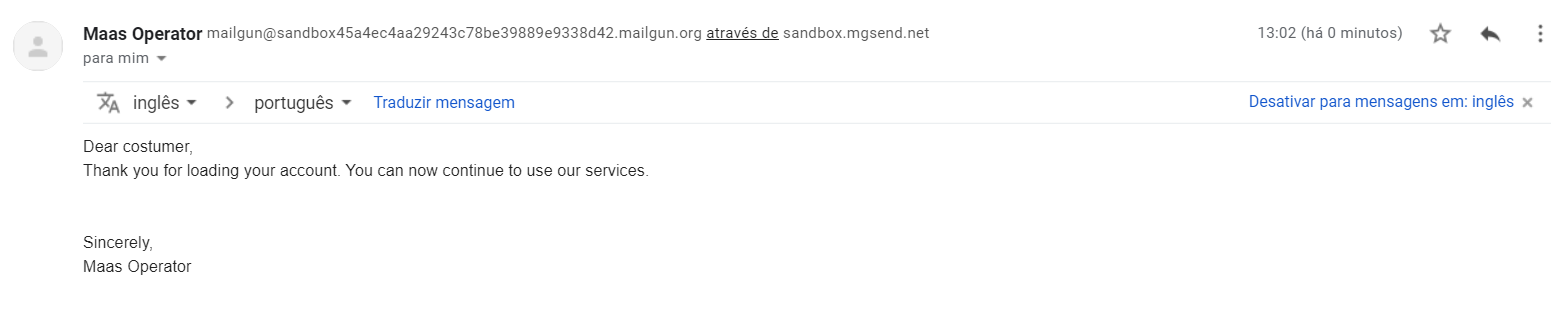
userLoadsAccount.json content:

{

    "messageName": "UserLoadsAccountMessage"

}

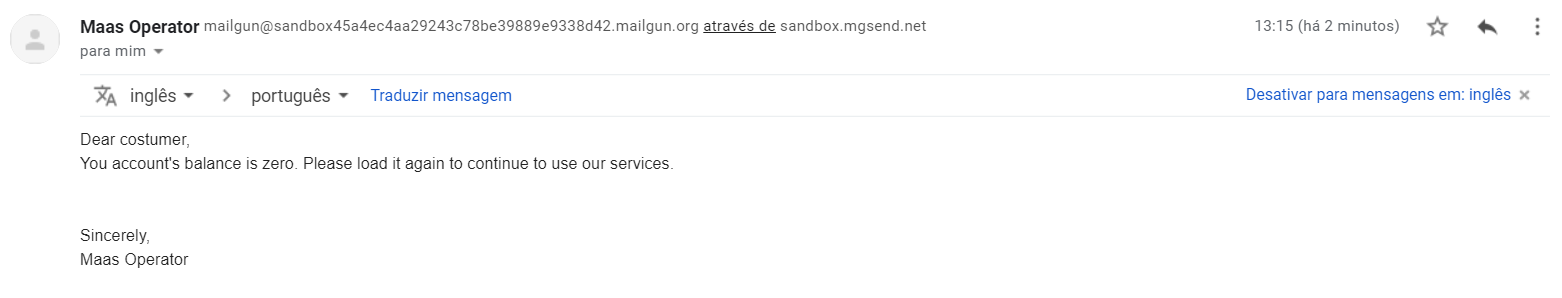
After this message was sent, I received an email saying this:

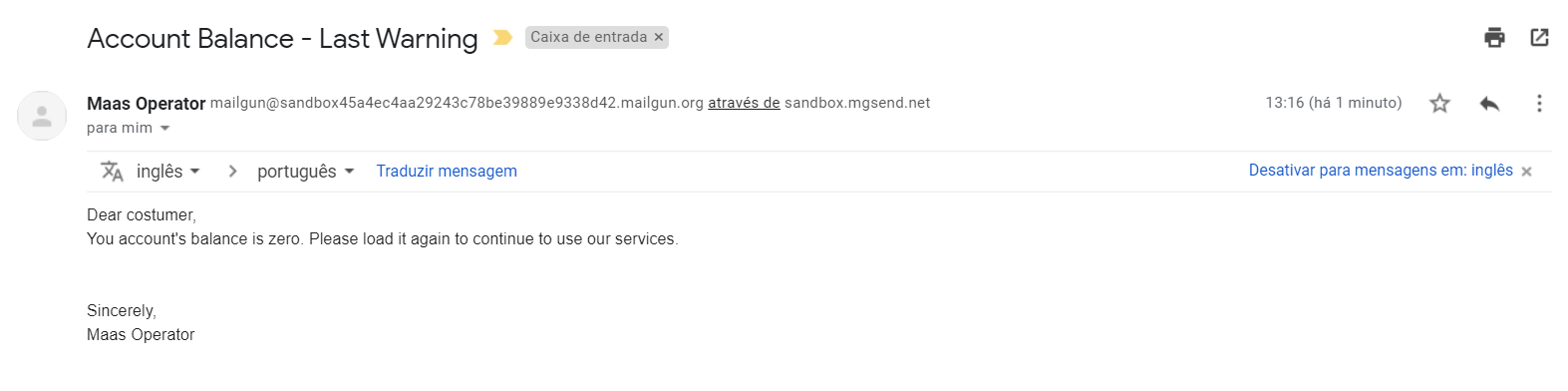


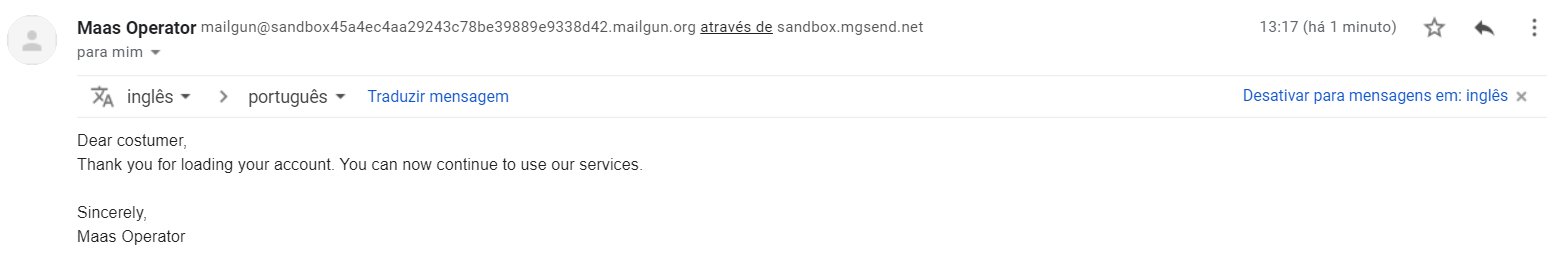
This message will be used by the Payment Service when the user loads an account to end the dunning process. After this the process ended.

* User Loads account after second warning

This case is the same as before, with the difference that the user receives two warnings instead of one. For this case I used the user created in the previous case as it was not deleted from the database. Just to illustrate I will show the emails received with the corresponding timestamps:

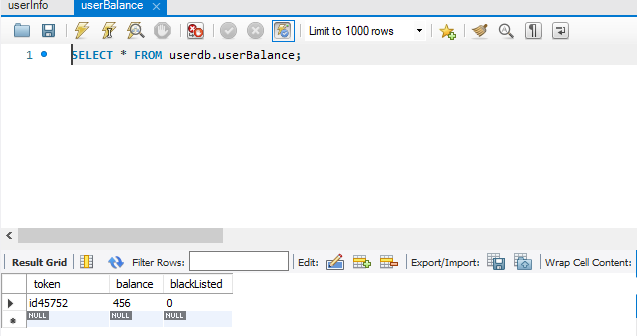




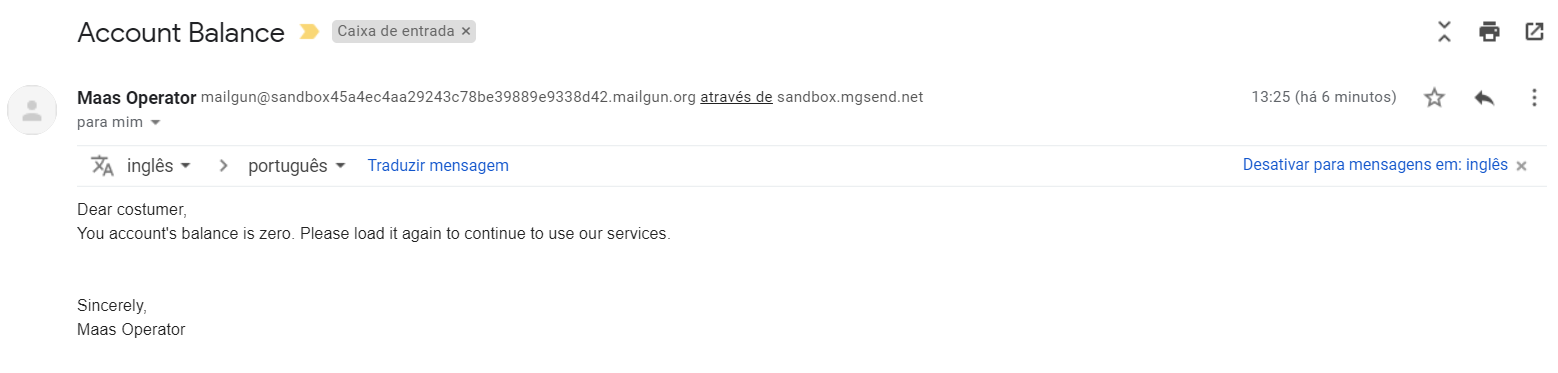


* User loads account after blacklist

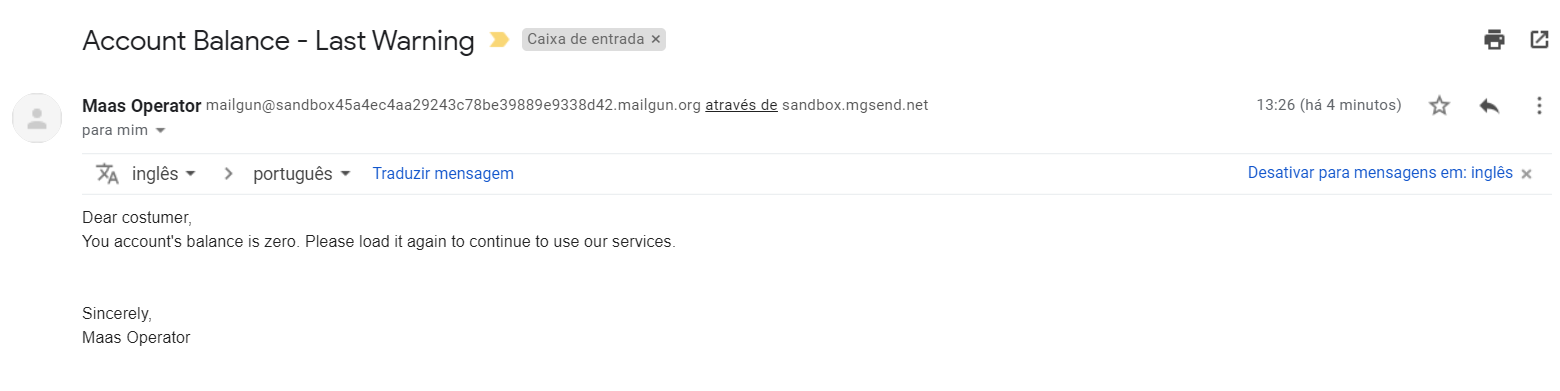
In the beginning the user was not blacklisted:



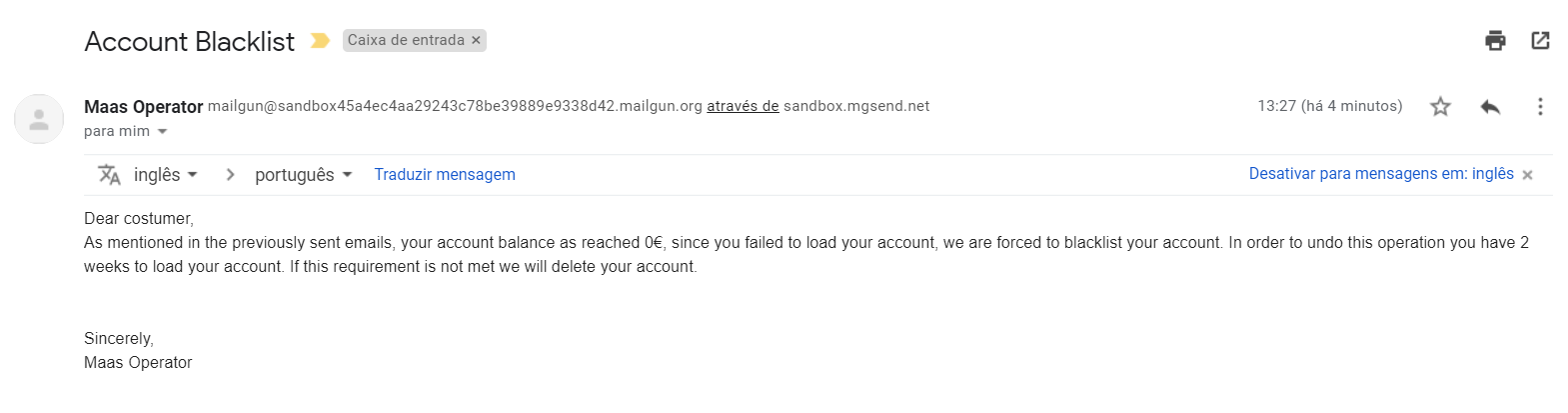
1. I started the process with the previous command and received the first email:



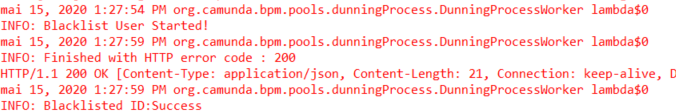
1. After 60 seconds I received the second one:



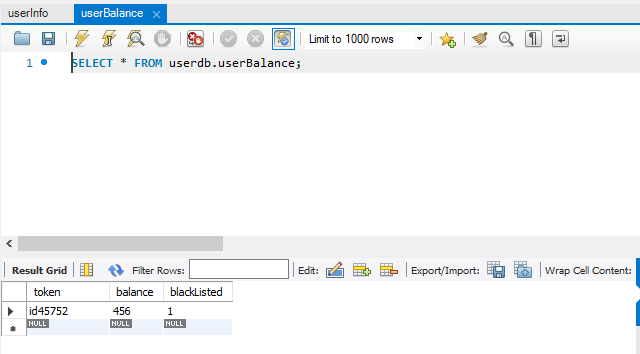
1. Then I received a warning about the blacklist operation



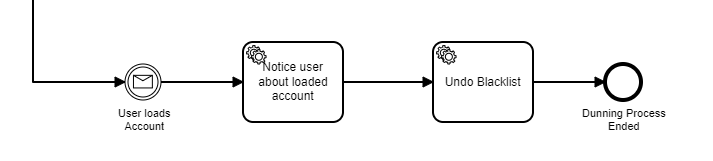
I checked the console:



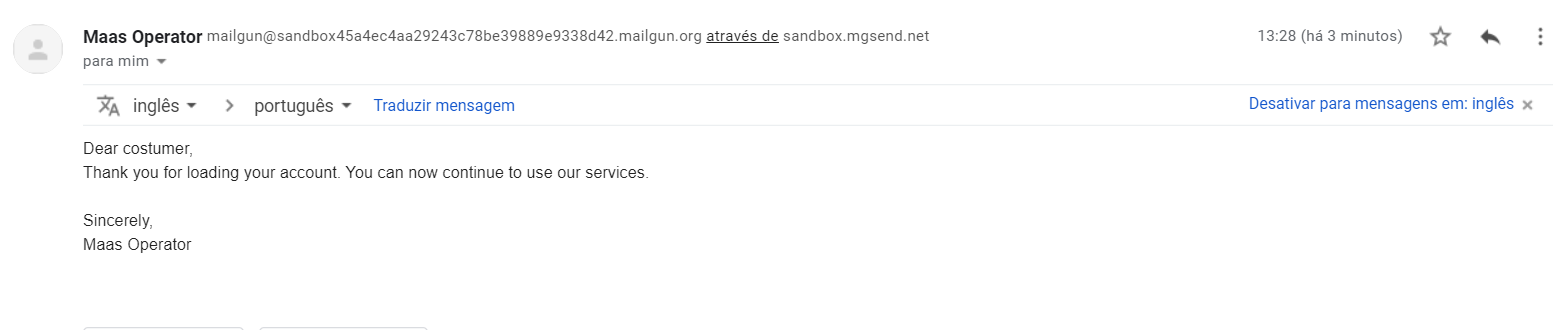
Then I checked the database, the user is blacklisted:



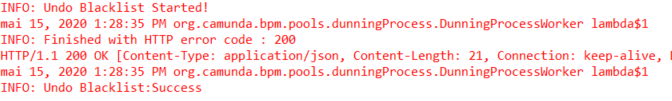
1. After this the process was waiting for the next 60 seconds but I sent a loadAccountMessage, so it went to this branch:



I received the following email:



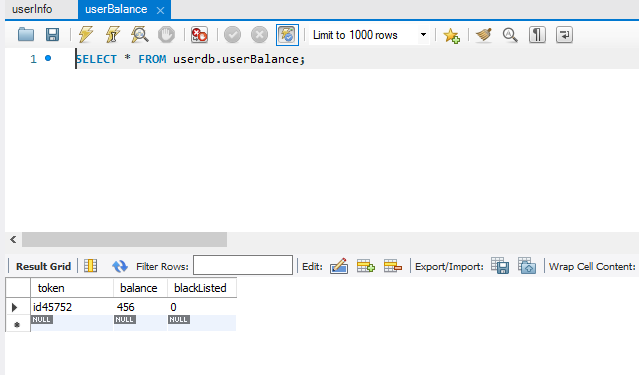
Then I checked the console:



Here is the CloudWatch Log of the undo-blacklist operation:

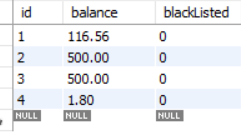


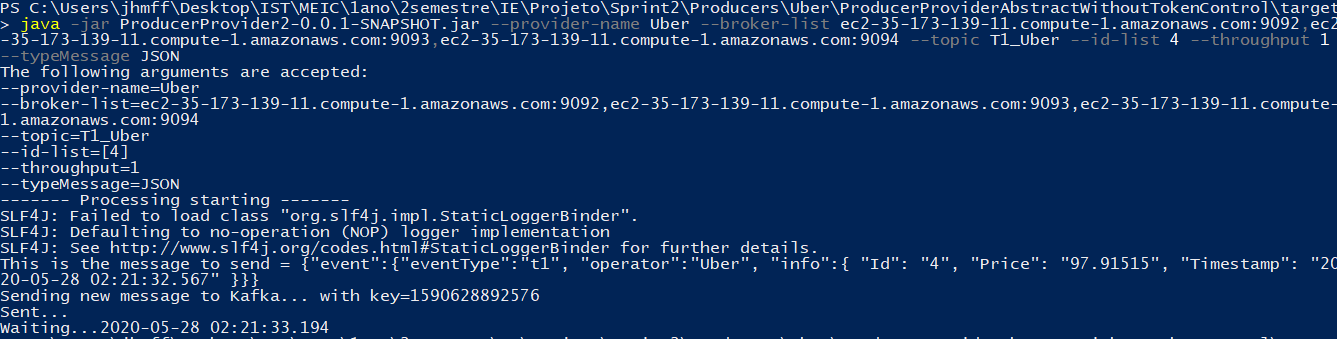
Finally, I checked the database again, the user was not blacklisted anymore:



**Automatic start of Dunning Process and stop with Load Account Process**

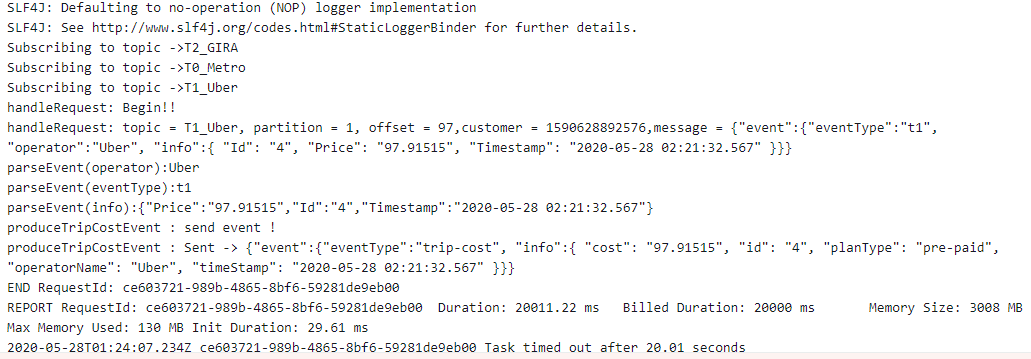
1. I started by having a user in the database with a low balance and produced a trip event for that user

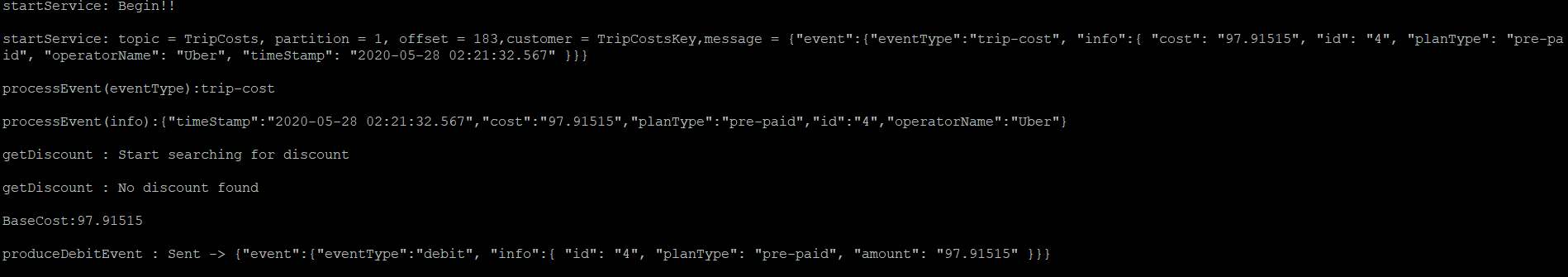




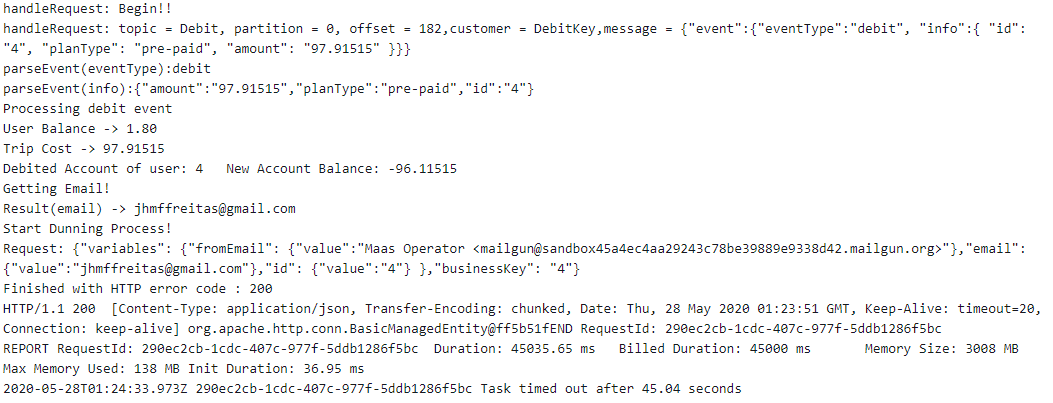
1. The trip event was successfully handled as I can see by the logs :

UserManagementService:

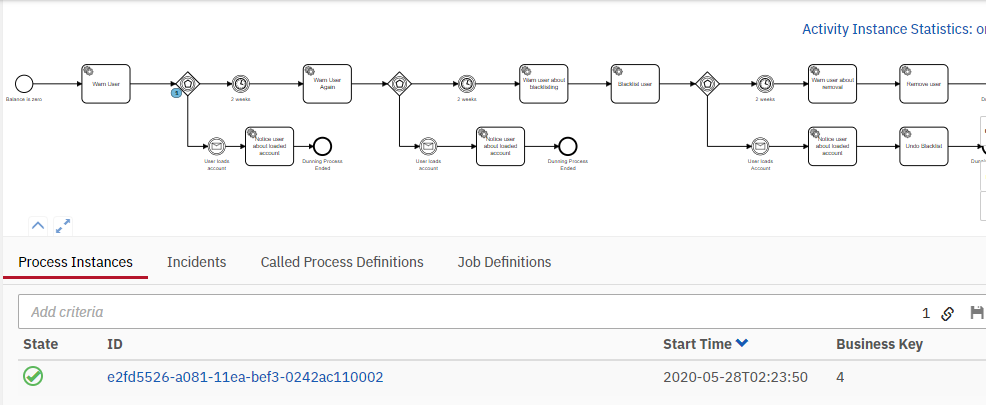


OperatorManagementService:

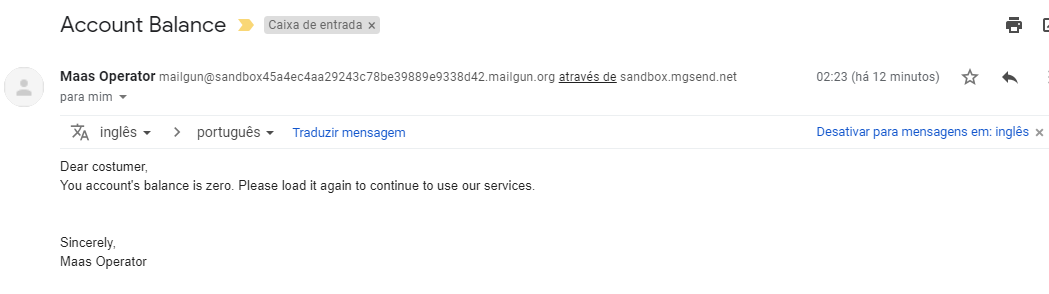
PaymentService:



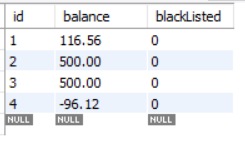
As we can see the PaymentService noticed that the balance was below zero, so it started the dunning process automatically.



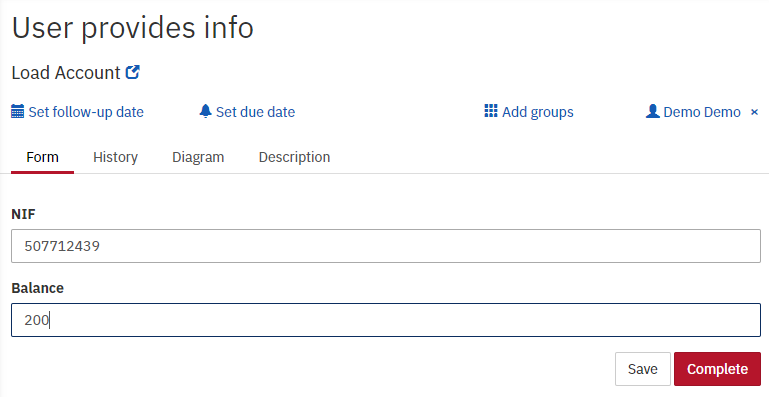
I received this email:



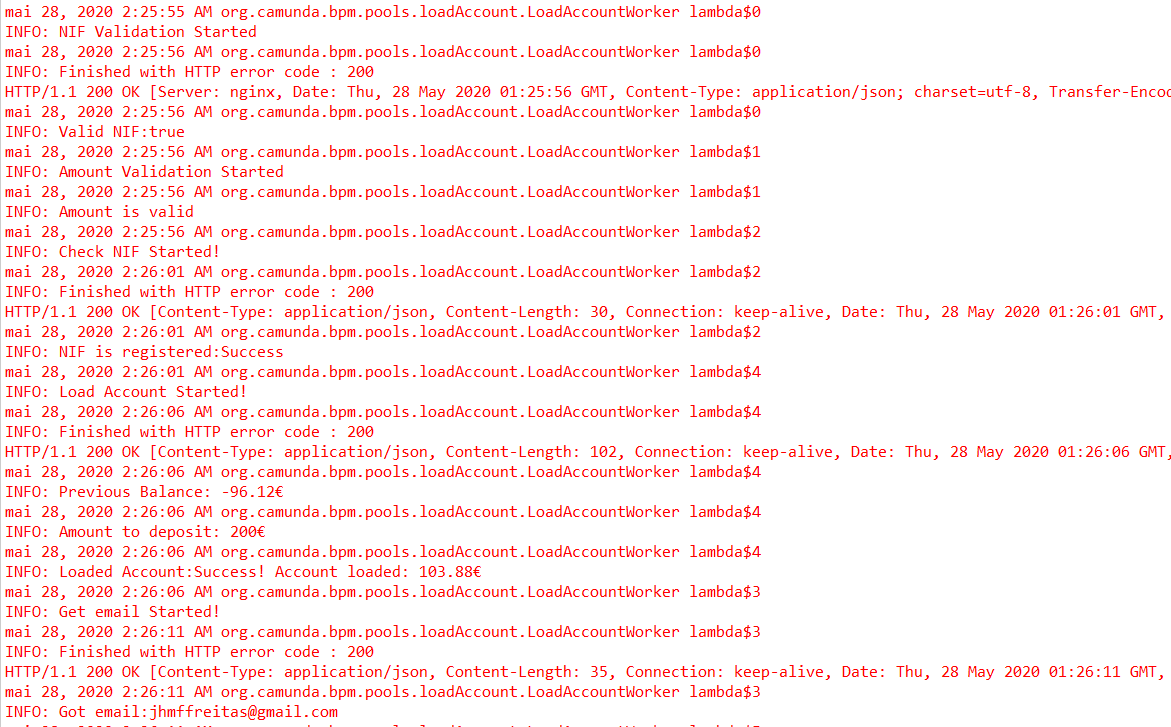
Database:



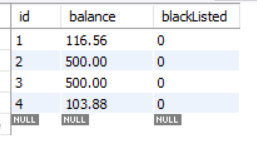
1. I started a Load Account Process with this info:



1. I checked the log and the loading succeeded:



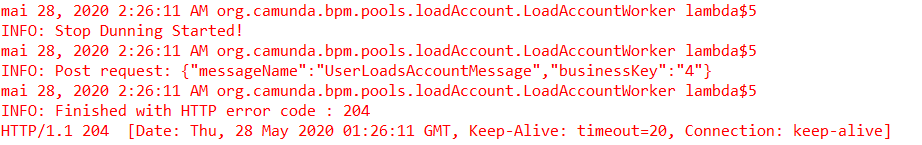
The user now has positive balance:



I also received this email:



1. It also detected that the dunning process was active, so it sent a message to stop it:



After this the dunning process stopped and I received this email:

