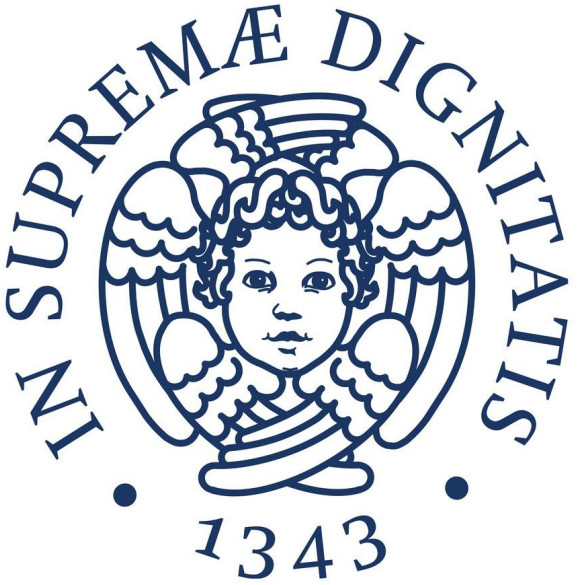
Master degree in Computer Engineering

Cloud Computing



UNIVERSITÀ DI PISA

Multi-tier Cloud Application

September 2020

Federico Cappellini, Andrea Lelli, Alberto Lunghi, Lorenzo Susini

1. App Description **3**

2. App Architecture **4**

2.1 Logical view 4

2.1.1 HAProxy 4

2.1.2 Frontend 5

2.1.3 RabbitMQ 5

2.1.4 Zookeeper 6

2.1.5 Backend 6

2.1.6 MySQL 6

2.2 Physical view 6

3. Deployment **8**

3.1 HAProxy 8

3.2 Frontend 9

3.3 RabbitMQ 9

3.4 Backend 10

3.5 MySQL 10

3.5.1 Manual deployment 10

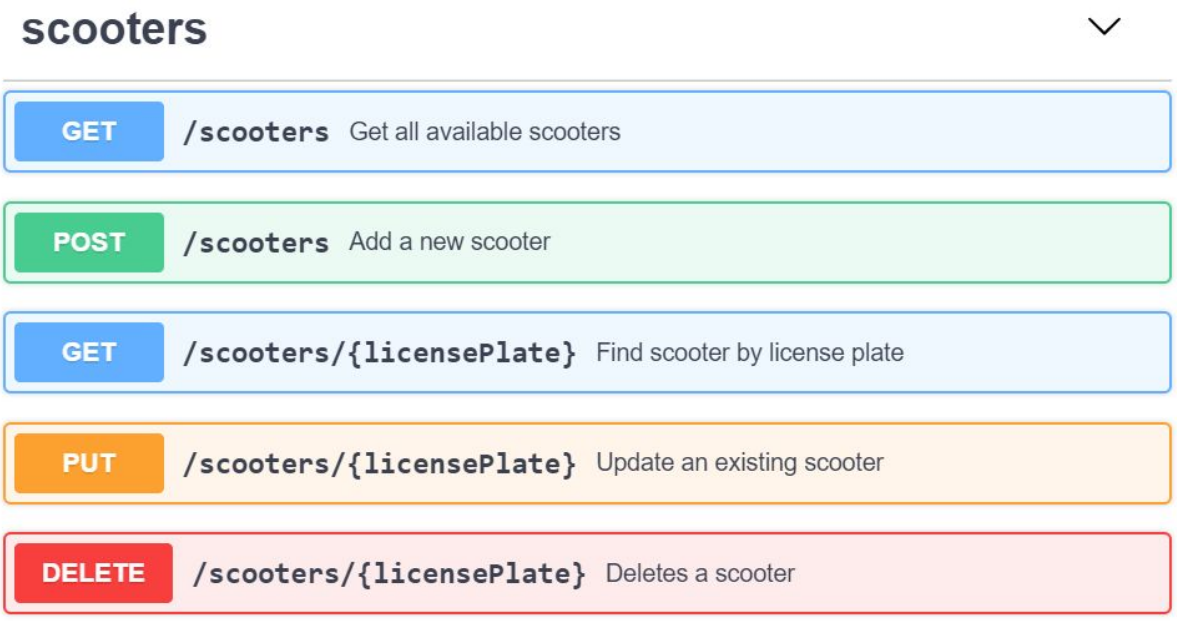
3.5.1 Automatic deployment 11

3.6 Zookeeper 11

3.7 Fault recovery 12

1. App Description

The goal of this cloud application is to manage an electric scooter sharing platform. Users can book a scooter and then rent it. A REST interface was implemented in order to keep track of the position, the battery percentage and the status of the scooters that are spreaded throughout an hypothetical city. The figures below explain completely the REST interface that was implemented.



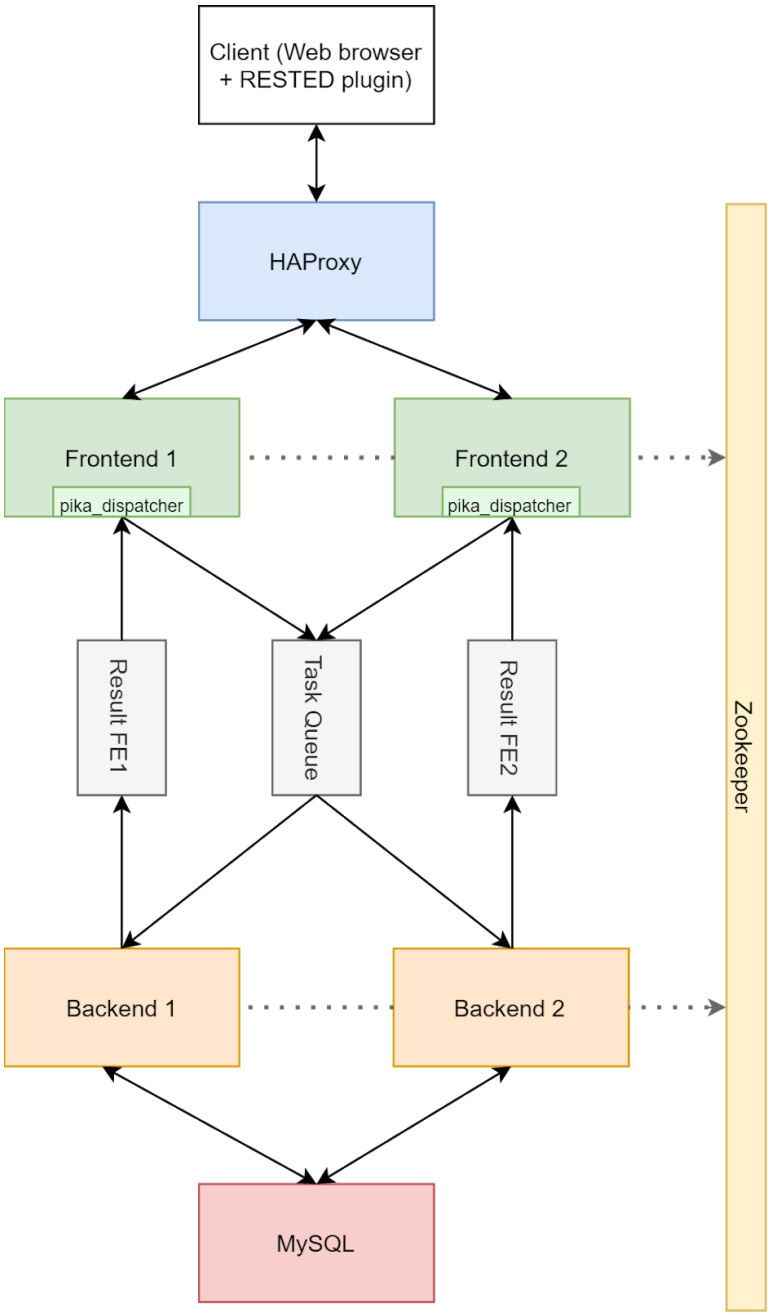
Furthermore, these are the attributes describing a scooter and that are stored in a database.

- license plate,
- latitude,
- longitude,
- battery percentage,
- status

2. App Architecture

2.1 Logical view

The architecture is a multi-tiered one and can be summarized in the following figure.



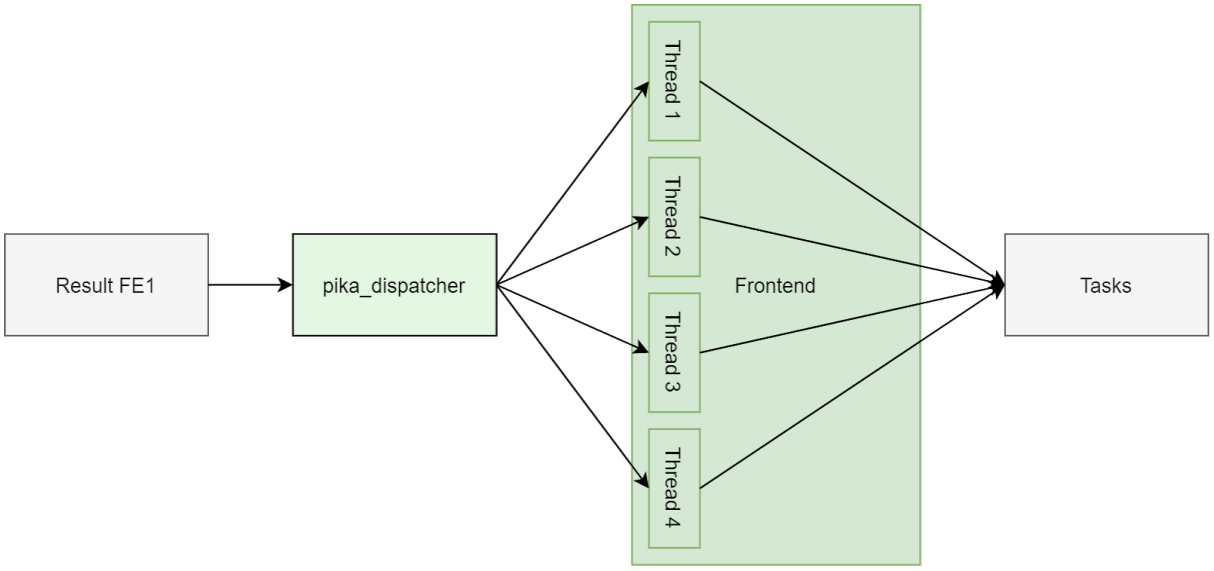
2.1.1 HAProxy

A client sends a request, of course through an HTTP request, since a REST interface is taken into account. The request can be done using the RESTED plugin. Then, a

load-balancer, implemented using HAProxy, is responsible for dispatching the requests to two different instances of the frontend layer.

2.1.2 Frontend

Each frontend instance is implemented using Flask and it is multithreaded. When a request arrives from a client, the frontend generates a *request id,* unpacks the HTTP packet and by inspecting its header, body and the endpoint that it is referring to, it can understand the operation that client wants to perform on the database. Then, at this point, a message containing the *operation, request id, result queue and other parameters* is enqueued in the task queue, in order to be consumed by a backend instance. The *request id* is also present in the message containing the result. This field is used by the *Pika Dispatcher* in order to send the message back to the correct frontend thread. The *result queue* field is used in order to specify where the backend has to put the result.



2.1.3 RabbitMQ

In order to ensure decoupling in time and space in the communication between the front end and the back end layer, RabbitMQ is used. In this way, we ensure scalability of the application: backend and frontend instances can be instantiated according to the application needs. New frontend instances must declare their new queue for the results, instead new backend instances must consume the messages coming from the task queue and then send the result back to the front end using the queue that was specified in the message of the needed operation. Each instance of the back end is responsible for receiving the message and performing the query on a MySQL database. The result of the query is put in a message and then it is sent through the result queue of the front end that has to respond to that request.

2.1.4 Zookeeper

All the configuration parameters needed by each component of the application are stored in Zookeeper. Since the absence of a valid Zookeeper instance due to a failure could result in a totally unworking system, the application is composed of three instances of Zookeeper with a primary server and two secondary servers. In case of a failure of the primary server, the Zookeeper protocol can elect a new primary server.

2.1.5 Backend

All the clients' requests are in the end handled by a backend instance.
Every backend instance consumes messages from the "tasks" queue: each message is an operation request from a frontend instance that the backend computes reading (and writing) data from (to) the MySQL database.
The result of the operation is sent back to the correct frontend instance due to the fact that each frontend instance has its own response queue and the request message carries the information on the name of the correct response queue.

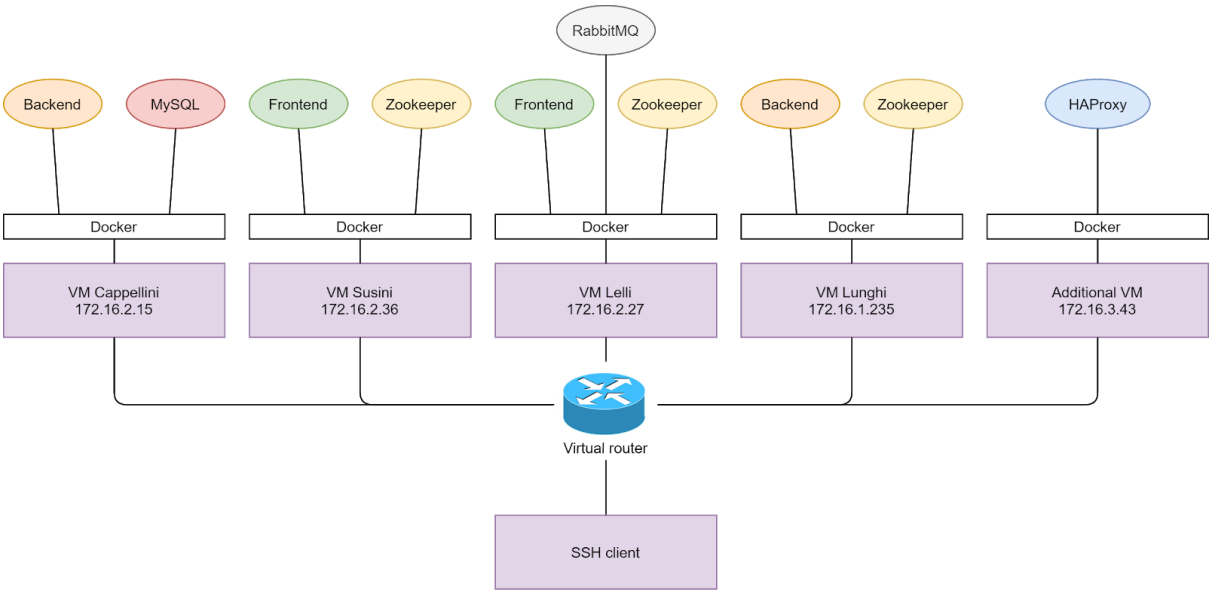
2.1.6 MySQL

All the application data are stored in a MySQL database. The database is named *scooterdb* and has only one table, named *scooters*, responsible to hold all the information about the scooters and composed as follow:

licensePlate	latitude	longitude	batteryPercentage	status
AAA	99,90000153	99,90000153	41	BOOKED
CCC	55,50000000	55,50000000	13	FREE
X4CNP4	10,30000019	11,30000019	80	BUSY
X4CNPC	10,00000000	11,00000000	80	BUSY
X8CNPC	44,29999924	80,59999847	75	FREE

2.2 Physical view

To release and run the multi-tier application, a pool of 5 virtual machines from the cloud infrastructure of the University of Pisa is at our disposal.
The application is deployed in such a way that the workload is as distributed as possible and there is a given degree of redundancy to support the failure of a virtual machine and/or some containers. In particular, containers are deployed in this way:


RabbitMQ
Backend
MySQL
Frontend
Zookeeper
Frontend
Zookeeper
Backend
Zookeeper
HAProxy
Docker
Docker
Docker
Docker
Docker
VM Cappellini
172.16.2.15
VM Susini
172.16.2.36
VM Lelli
172.16.2.27
VM Lunghi
172.16.1.235
Additional VM
172.16.3.43
Virtual router
SSH client

3. Deployment

To avoid manual configurations of the virtual machines that are error-prone and require specialized personnel, the project folder contains a set of scripts that automatically create the images for Docker and can run them in a proper way.
All the configuration scripts are located in the *script* folder inside the project folder.

- HAProxy: **deployHAProxy.sh**
- RabbitMQ: **deployRabbitmq.sh**
- MySQL: **deployMySQL.sh OR db_image_install.sh**
- Zookeeper: **deployZookeeper.sh**
- Zookeeper (parameters init): **zookeeper_init.py**

3.1 HAProxy

HAproxy was used as a load-balancer. The configuration script is located in the *script folder* of the project and it is structured like this

```
#!/bin/sh
cd ../haproxy
docker build -t myhaproxy .
docker run -d --name myhaproxy1 --network host myhaproxy
```

In the Dockerfile we only need to copy the configuration file in the correct directory.

```
FROM haproxy
COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
EXPOSE 80
EXPOSE 9999
```

The configuration file defines the address and the port on which requests to the applications must be received, which are the servers to load balance and the policy of the load balancer. In particular, in our configuration we have used a round robin policy. Moreover, using the *check* keyword, we are enabling the load balancer to perform health checks on the servers, in order to avoid forwarding packets to a server if it is down. HAProxy also allows to monitor the traffic through a web interface, which is useful to verify that the service is working.

```
global
      maxconn 256
      debug
      log /dev/log local0 debug

defaults
      mode http
      timeout connect 5000ms
```

```
        timeout client 50000ms
        timeout server 50000ms
        log global

listen stats
        bind *:9999
        stats enable
        stats hide-version
        stats uri /stats
        stats auth admin:admin@123

frontend myApp
        bind *:80
        default_backend myAppBackEnd

backend myAppBackEnd
        balance roundrobin
        mode http
        server myAppServer1 172.16.2.36:8080 check
        server myAppServer2 172.16.2.27:8080 check
```

3.2 Frontend

The Frontend image is built based on the official *python:3-alpine* image from Docker's repository:

```
#!/bin/bash
cd ../flask-server
docker build -t myfrontend .
docker run -d -p 8080:8080 --name frontend myfrontend
```

The *flask-server* folder contains, indeed, the *Dockerfile*, all the required files for *swagger_server* and the application Python scripts.

3.3 RabbitMQ

RabbitMQ is run in this way:

```
#!/bin/sh
cd ../rabbit-mq
docker build -t some-rabbit .
docker run -p 4369:4369 -p 15671:15671 -p 15672:15672 -p 5671:5671 -p
5672:5672 -p 15691:15691 -p 15692:15692 -p 25672:25672 --hostname rabbit
--name rabbit -d some-rabbit
```

This script will install a custom RabbitMQ image equipped with a plugin to expose a web management interface.

3.4 Backend

The Frontend image is built based on the official *python* image from Docker's repository:

```
#!/bin/sh
cd ../back-end
docker build -t mybackend .
docker run -d --name mybackend1 mybackend
```

The *back-end* folder contains, indeed, the *Dockerfile* and all the required files.

3.5 MySQL

3.5.1 Manual deployment

The MySQL container is built on top of the official Docker's MySQL image pulled from the repository.
By default the database

- has only the root user that can not log-in from remote
- the root user has an unknown password
- must be configured with the application data

To fix all of these problems, the deploy script launches the container, then reads the default root password from the Docker logs and lets us launch a bash shell from inside the container.

```
#!/bin/sh
docker pull mysql mysql server
docker run --name mydb -d -p 3306:3306 mysql/mysql-server
docker logs mydb 2>&1 | grep GENERATED
docker exec -it mydb bash
```

With the discovered password we can log-in into MySQL and change the configuration with the following commands:

```
mysql -uroot -p<discovered password>
    ALTER USER 'root'@'localhost' IDENTIFIED BY 'root';
    use mysql;
    UPDATE user SET host='%' WHERE host='localhost' AND user='root';
    FLUSH PRIVILEGES;
    CREATE SCHEMA `scooterdb`;
       USE scooterdb
    DROP TABLE IF EXISTS `scooters`;
    CREATE TABLE `scooters` (
      `licensePlate` varchar(45) NOT NULL,
```

```
      `latitude` float(10,8) DEFAULT NULL,
      `longitude` float(10,8) DEFAULT NULL,
      `batteryPercentage` int DEFAULT NULL,
      `status` varchar(45) DEFAULT NULL,
      PRIMARY KEY (`licensePlate`)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
    LOCK TABLES `scooters` WRITE;
    INSERT INTO `scooters` VALUES ('AAA',99.90000000,99.90000000,41,'BOOKED'),
('CCC',55.50000000,55.50000000,13,'FREE'),
('X4CNP4',10.30000000,11.30000000,80,'BUSY'),
('X4CNPC',10.00000000,11.00000000,80,'BUSY'),
('X8CNPC',44.30000000,80.60000000,75,'FREE');
    UNLOCK TABLES;
```

3.5.1 Automatic deployment

To avoid dealing with the previous manual installation, an already built MySQL image is bundled with the directory of the project.
In this case the root user is already reachable from every address and its password is simply “root”.
To perform this kind of installation, you can run the script *db_image_install.sh* inside the *docker_images* subdirectory.

3.6 Zookeeper

The Zookeeper container is built from Docker’s official container, pulled from the repository, and the configuration is given by the following launch script:

```
#!/bin/sh
docker run -d \
   -p 2181:2181 \
   -p 2888:2888 \
   -p 3888:3888 \
   --name myzk-1 \
   -e ZOO_MY_ID=1 \
   -e ZOO_SERVERS='server.1=0.0.0.0:2888:3888;2181
server.2=172.16.2.27:2888:3888;2181 server.3=172.16.2.36:2888:3888;2181' \
    --restart always \
    zookeeper
```

The shown script refers to the one that is run on the primary server of Zookeeper. In the other two servers, where the remaining instances of Zookeeper are deployed, the ip addresses are specified in order to have the *0.0.0.0* address always for the current machine, and the address *172.16.1.235* for server 1.

3.7 Fault recovery

In order to be sure that each container is always up and running, we run every container with the option *--restart unless-stopped*. In this way, even if a container goes down due to an error, it is always restarted, assuring availability for the application. Also in case of a reboot of the machine, the container will be run again.
According to the official Docker documentation, It also possible to use different restart policy:

Flag	Description
`no`	Do not automatically restart the container. (the default)
`on-failure`	Restart the container if it exits due to an error, which manifests as a non-zero exit code.
`always`	Always restart the container if it stops. If it is manually stopped, it is restarted only when Docker daemon restarts or the container itself is manually restarted. (See the second bullet listed in restart policy details)
`unless-stopped`	Similar to `always`, except that when the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts.