DISTRIBUTED SYSTEMS AND BIG DATA

Ingegneria Informatica LM-32

A.A. 2021/2022

eRent

Sviluppo di un'applicazione per il noleggio di monopattini elettrici

Studenti:

Amenta Daniele
Matricola: 1000027022

D'Agosta Daniele
Matricola: 1000027035

GitHub: eRent

Sommario

1. Introduzione

Lo scopo dell'elaborato è quello di realizzare un sistema distribuito per gestire il noleggio di monopattini elettrici, servizio ormai diffuso in gran parte delle città. Il funzionamento prevede la presenza di alcune chiamate che permettono di effettuare, sulla base dei monopattini disponibili, il noleggio temporaneo di quest'ultimi. Al termine del noleggio, il mezzo viene nuovamente bloccato e si passa alla fase conclusiva del flusso, ovvero la generazione della fattura per il noleggio concluso.

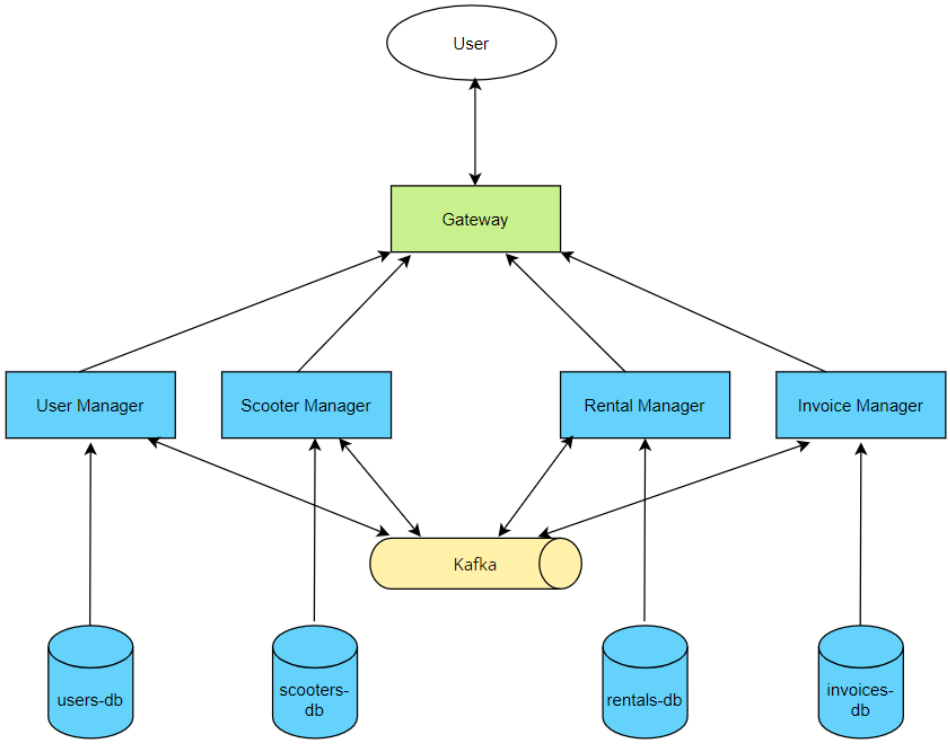
I servizi che sono stati sviluppati sono i seguenti:

- **User Manager**: si occupa della registrazione e del login degli utenti. È stato anche implementato un servizio che permette l'autenticazione dell'utente tramite JWT; difatti, al seguito del login, verrà restituito il token all'utente, che sarà necessario per il suo riconoscimento nelle successive chiamate;
- **Scooter Manager**: si occupa della simulazione del blocco o dello sblocco dei monopattini a seguito della richiesta di inizio noleggio da parte dell'utente. Gestisce anche l'inserimento di nuovi monopattini da parte dell'amministratore del sistema, oltre a restituire la lista di tutti i monopattini disponibili per il noleggio (*ovvero quelli nello stato LOCKED*);
- **Rental Manager**: si occupa della gestione dei noleggi, ovvero della creazione e della terminazione;
- **Invoice Manager**: si occupa di generare la fattura al termine del noleggio, nonché della visualizzazione delle fatture precedentemente generate;
- **Gateway**: si occupa di direzionare le richieste verso il corretto micro-servizio.

Ogni micro-servizio distribuito è un progetto Spring Boot. Per il local testing si è fatto uso di Docker con Docker-Compose, mentre per il Distributed Testing è stato usato Kubernetes con Minikube.

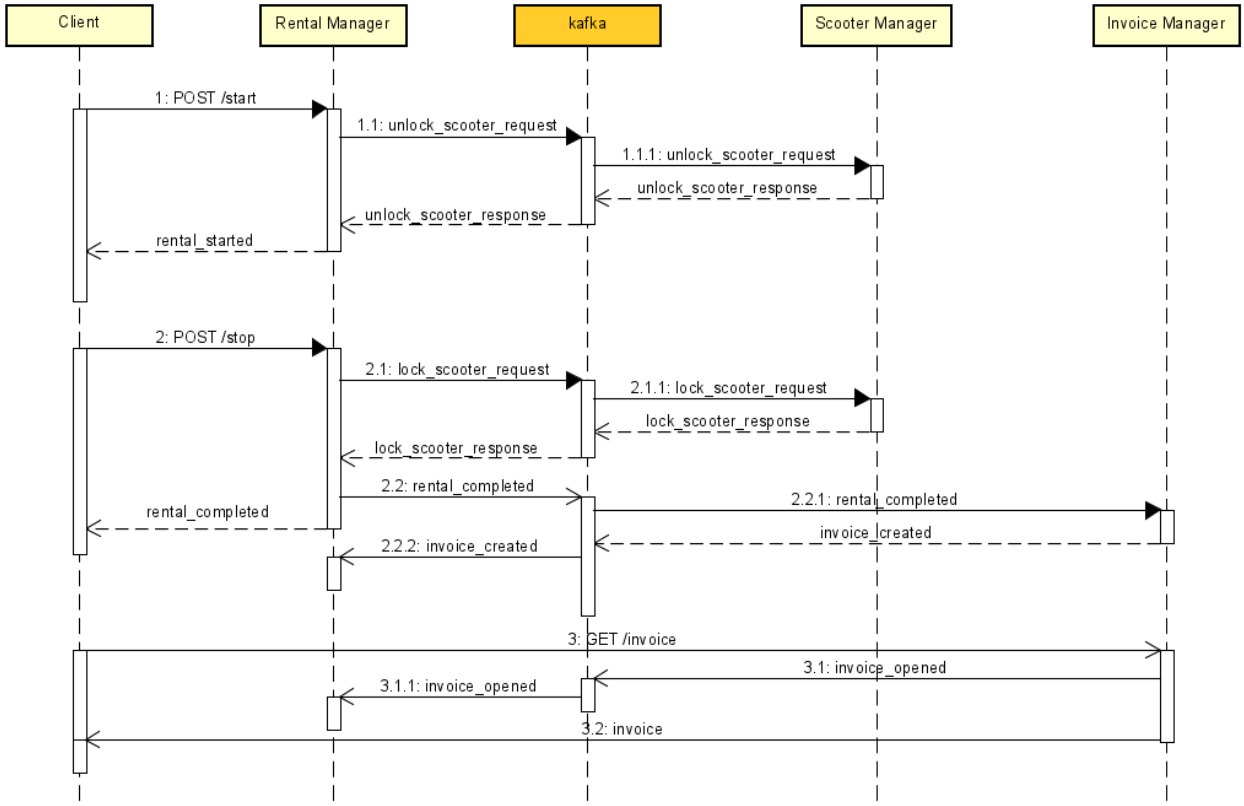
La persistenza dei dati è effettuata con MongoDB.

In figura è possibile osservare lo schema architetturale del sistema distribuito.


User
Gateway
User Manager
Scooter Manager
Rental Manager
Invoice Manager
Kafka
users-db
scooters-db
rentals-db
invoices-db

2. Procedura di noleggio

Di seguito viene mostrato il diagramma di sequenza con le richieste e i messaggi scambiati dalle varie parti per avviare e concludere correttamente un noleggio.



La creazione di un noleggio avviene richiamando l'API "/start" che da inizio ad una comunicazione request-reply sincrona tra il Rental Manager e lo Scooter Manager mediante Kafka:

1. Il RM richiede lo sblocco del mezzo mediante una richiesta sincrona di Kafka e resta in attesa;
2. Lo SM riceve la request ed effettua lo sblocco del mezzo se possibile; non sarà possibile sbloccare un mezzo se già sbloccato o se la posizione dell'utente è al di fuori dell'area coperta dal servizio;
3. Lo SM inoltra la response sul canale in risposta alla request;
4. Il RM gestiste la risposta del SM ed inoltra il risultato al client.

Si è scelto di usare una richiesta sincrona poiché non ha senso che nell'attesa dello sblocco un utente possa effettuare altre operazioni, tra cui richiedere altri noleggi.

La terminazione comporta una transazione simile, ma nel caso in cui il blocco del mezzo non vada a buon fine è previsto uno stato "FROZEN" al fine di evitare che il contatore temporale incrementi in modo indefinito. Nel caso in cui la terminazione si concluda con successo, il RM invia un messaggio Kafka che verrà ricevuto dall'Invoice Manager che provvederà a generare la fattura.

Questo dà inizio ad una serie di transazioni locali gestite mediante il SAGA Patter (Choreography-based):

1. L'Invoice Manager crea una nuova fattura per il noleggio appena concluso e trasmette un messaggio di avvenuta creazione ad un topic Kafka;

2. Tale messaggio verrà ricevuto dal Rental Manager che, per quel noleggio, annoterà tramite una variabile booleana che la fattura è stata generata.

A questo punto, l'utente può procedere a visualizzare la fattura per quell'ordine, utilizzando l'API "/invoice/{id_rental}". Chiamando questa API:

1. L'Invoice Manager trasmetterà un messaggio di visualizzazione avvenuta su un topic Kafka;
2. Tale messaggio verrà ricevuto dal Rental Manager che, per quel noleggio, annoterà tramite una variabile booleana che la fattura è stata visualizzata.

3. Panoramica dei micro-servizi

Di seguito una paronamica dei singoli micro-servizi. Per ognuno di essi sono descritte:

- API;
- Entità;
- Messaggi Kafka trasmessi e ricevuti;
- Procedure di business logic.

Sono presenti delle caratteristiche comuni a tutti i micro-servizi:

- Per il check della liveness, ogni microservizio espone una API "**GET /ping**" che risponde con Pong. Ciò è utile anche per l'utilizzo del Fault Detector di Kubernetes.
- L'amministratore del sistema, per convenzione, ha UserID pari a 0; Sia in Docker che in Kubernetes l'amministratore viene automaticamente creato con quell'ID se non esiste già;
- Le richieste delle API devono possedere i seguenti header:
    - **X-User-ID** che contiene l'id dell'utente che esegue la richiesta;
    - **Authorization** che contiene il JWT (Json Web Tokens) che è stato ottenuto durante il login alla piattaforma.

3.1 User Manager

Servizio che si occupa della gestione degli utenti e dell'autenticazione.

Entità

```
public class User {

    @Id
    private String id;

    private String name;
    private String email;
    private String password;
    private List<String> roles;

    public User() {
        this.roles = new ArrayList<>();
    }

}
```

API

Espone le seguenti API:

- **POST /user/register**: registra un utente alla piattaforma; se roles è pari a [] (*oggetto vuoto*), l'unico ruolo assegnato sarà USER;
- **GET /user/{id}**: ritorna un utente registrato in base all'id;
- **GET /user/email/{email}**: ritorna un utente registrato in base all'email. La richiesta andrà a buon fine solo nel caso in cui sia l'amministratore o il proprietario di tale email;
- **GET /user/users**: ritorna un array JSON di utenti solo se colui che fa la richiesta è l'amministratore;
- **POST /user/login**: in caso di esito positivo, ritorna il JWT che l'utente deve usare per le future richieste autenticate.

Questo servizio si occupa quindi dell'autenticazione. Il suo funzionamento è abbastanza semplice:

1. L'utente richiede di effettuare il login fornendo username e password in una POST. La password verrà codificata in SHA-256 per poi procedere alla ricerca di tale utente nell'users-db.
2. In caso di esito positivo, viene generato un Json Web Token che include il suo ID.

La generazione del tokens richiede di dover specificare un secret necessario per la verifica di "Autenticità".

Kafka

Questo micro-servizio non prevede la scrittura su nessun topic.

3.2 Scooter Manager

Servizio che si occupa della simulazione del blocco o dello sblocco dei monopattini a seguito della richiesta di inizio noleggio da parte dell'utente.

Entità

```
public class Scooter {

    @Id
    private String scooterId;

    private String code;

    private Double lat;
    private Double lon;

    private ScooterStatus scooterStatus;
}
```

API

Espone le seguenti API:

- **POST /scooter/add**: se è presente l'header con user ID dell'amministratore allora aggiunge un nuovo monopattino con i dati passati nel JSON all'interno del body. Non va inserita latitudine e longitudine, poiché verranno automaticamente impostate quelle del centro dell'area. Restituisce il JSON del monopattino inserito;
- **GET /scooter/scooters**: ritorna un array JSON di monopattini disponibili (*stato LOCKED*);

Bisogna tenere in considerazione della possibilità di concorrenza nella fase di sblocco del mezzo. È infatti possibile che due utenti cerchino contemporaneamente di noleggiare il mezzo. È stata quindi implementata la logica di sblocco/blocco come una transazione garantendo l'isolamento per evitare inconsistenze.

Il blocco e lo sblocco del mezzo richiedono due condizioni:

1. L'utente deve trovarsi all'interno dell'area coperta dal servizio (4km di raggio dal centro dell'area);
2. L'utente deve fornire il codice del veicolo, che è una stringa. Questa è una simulazione del codice QR che si trova attaccato ai monopattini in giro per le città per essere noleggiati.

Kafka

Gli eventi inerenti il blocco e lo sblocco di un monopattino sono gestiti in modo sincrono sfruttando il meccanismo request-reply di Kafka. Praticamente il Rental Manager manda delle richieste sincrone, che vengono ricevute dallo Scooter Manager mediante KafkaListener con topic "**scooter_requests**". Le response vengono inviate poi automaticamente al topic "**scooter_responses**".

Entità delle **request** e delle **response**:

```
public class ScooterRequest implements Serializable {

    private String scooter_id;
    private String user_id;
    private String rental_id;
    private ScooterOperation operation;
    private Double lat;
    private Double lon;
    private String scooterCode;

}


public class ScooterResponse implements Serializable {
    private String scooter_id;
    private Boolean success;
    private String message;
}
```

Il servizio è produttore dei messaggi kafka sul topic "**scooter**", utili a fine di debug:

- scooter_locked;
- scooter_unlocked.

3.3 Rental Manager

Servizio che si occupa della gestione dei noleggi, ovvero della creazione e della terminazione.

Entità

```
public class Rental {
    @Id
    private String id;

    private String scooterId;
    private String userId;

    private Long startTimestamp;
    private Long stopTimestamp;

    private Double price_per_minute;
    private Double price_per_start;
    private Double amount_to_pay;

    private RentalStatus status;

    private Boolean invoiceGenerated;
    private Boolean invoiceOpened;
}
```

API

Espone le seguenti API:

- **GET /rental/{id}**: ottiene il noleggio con l'id specificato solo se lo user ID presente come header coincide con quello presente nel noleggio o è 0 (user ID dell'amministratore);
- **POST /rental/start**: consente di iniziare un nuovo noleggio. Richiede che vengano forniti i seguenti dati:
  - Latitudine e Longitudine dell'utente.
  - Codice di sblocco (solitamente inserito in un punto visibile del veicolo).
  - Header X-User-ID per identificare l'utente.
- **POST /rental/stop**: consente di terminare il noleggio. Richiede gli stessi parametri dell'API di start.
- **GET /rental/rentals**: ritorna un array JSON di noleggi relativi all'utente con id espresso nell'header "X-User-ID". Se è 0, ritorna tutti i noleggi.

Quando viene richiesto lo start di un nuovo noleggio, deve essere verificato se il mezzo sia effettivamente disponibile. Può succedere che due utenti richiedano lo stesso veicolo nello stesso momento, portando ad una inconsistenza sullo stato del monopattino (disponibile o non disponibile?). Questo problema è gestito dallo Scooter Manager, il quale implementa l'operazione di sblocco/blocco come una Transazione che rispetta l'isolamento.

Nel caso in cui la terminazione non vada a buon fine, bisogna bloccare il reale tempo di utilizzo del veicolo per evitare di attribuire costi aggiuntivi a un utente. A tal motivo è presente lo stato "FROZEN".

Kafka

È produttore dei seguenti messaggi sul topic "**rental**":

- rental_accepted (*utile per scopi di debug*);
- rental_completed.

È consumatore del topic "**invoice**", in particolare dei messaggi ***invoice_created*** e ***invoice_opened***, per gestire l'aggiornamento delle variabili relative alla fattura generata e alla fattura visualizzata dall'utente. A tale scopo, si elencano le entità delle **request** e delle **response** che vengono utilizzate:

```
public class RentalRequest implements Serializable {

    private String scooterId;
    private String user_id;
    private String rental_id;

}
```

```
public class RentalResponse implements Serializable {

    private String rental_id;
    private Boolean success;
    private String message;

}
```

L'aggiornamento della variabile *invoiceCreated* sarà possibile solo se la variabile è ancora settata a false.

L'aggiornamento della variabile *invoiceOpened* sarà possibile solo se la variabile *invoiceCreated* è settata a true e *invoiceOpened* è ancora settata a false.

Non è stato gestito un eventuale errore nel caso di questa fase di aggiornamento, ma è eventualmente possibile visto la presenza dell'entità *RentalResponse*.

3.4 Invoice Manager

Servizio che si occupa di generare la fattura al termine del noleggio, nonché della visualizzazione delle fatture precedentemente generate.

Entità

```
public class Invoice {

    @Id
    private String id;

    private String rentalId;
    private String scooterId;
    private String userId;

    private Double total;
    private Double price_per_minute;
    private Double price_per_start;

    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE_TIME)
    private Date start_timestamp, end_timestamp;
}
```

API

Espone le seguenti API:

- **GET /invoice/{id}**: Risponde con la rappresentazione JSON della fattura con rental_id fornito se lo user id della fattura è uguale a quello fornito o è 0, ovvero è l'amministratore del sistema. Altrimenti 404 Not Found (quindi se la fattura non esiste o non è associata allo user id fornito);
- **GET /invoice/invoices**: Risponde con la rappresentazione JSON dell'array delle fatture associate allo user id fornito o di tutti gli utenti se lo user id è 0.

Quando si invoca la GET /{id} fornendo l'id del noleggio, viene aggiornata la variabile relativamente alla fattura visualizzata solamente se l'id è quello dell'utente; un amministratore è quindi libero di visualizzare le fatture, senza avviare la procedura di aggiornamento gestita tramite Kafka.

Kafka

È produttore dei seguenti messaggi sul topic "**rental**":

- invoice_created;
- invoice_opened.

È consumatore del topic "**rental**", in particolare del messaggio ***rental_completed***, che avvierà la procedura che consente di aggiornare le variabili condivise gestite dal Rental Manager.

4. Docker

Ciascun micro-servizio ha un **Dockerfile** per effettuare il building dell'immagine. Di seguito il Dockerfile di ***users_microservice***.

```
FROM maven:3-jdk-8 as builder
WORKDIR /project
COPY ./users_microservice/pom.xml ./pom.xml
RUN mvn dependency:go-offline -B
COPY ./users_microservice/src ./src
RUN mvn package

FROM java:8-alpine
WORKDIR /app
COPY --from=builder /project/target/users_microservice-0.0.1-SNAPSHOT.jar ./users_microservice.jar
#ENTRYPOINT ["/bin/sh", "-c"]
CMD java -jar users_microservice.jar
```

Le operazioni effettuate sono le seguenti:

- Impostare il builder con jdk-8;
- Spostamento nella cartella project;
- Copia del pom.xml;
- Download delle dipendenze;
- Copia della cartella src;
- Building del file jar;
- Impostare l'executer con java-8;
- Spostamento nella cartella app;
- Copia del jar file appena creato;
- Esecuzione del file jar.

È stato creato un docker-compose file per ciascun micro-servizio, in modo da poter testare ognuno di essi in modo autonomo durante lo sviluppo. Ogni micro-servizio presenta un proprio file per le variabili d'ambiente, ma è anche presente un file di variabili d'ambiente comuni a tutti (**commons.env**). Riportiamo in seguito il **docker-compose-user.yml** come esempio:

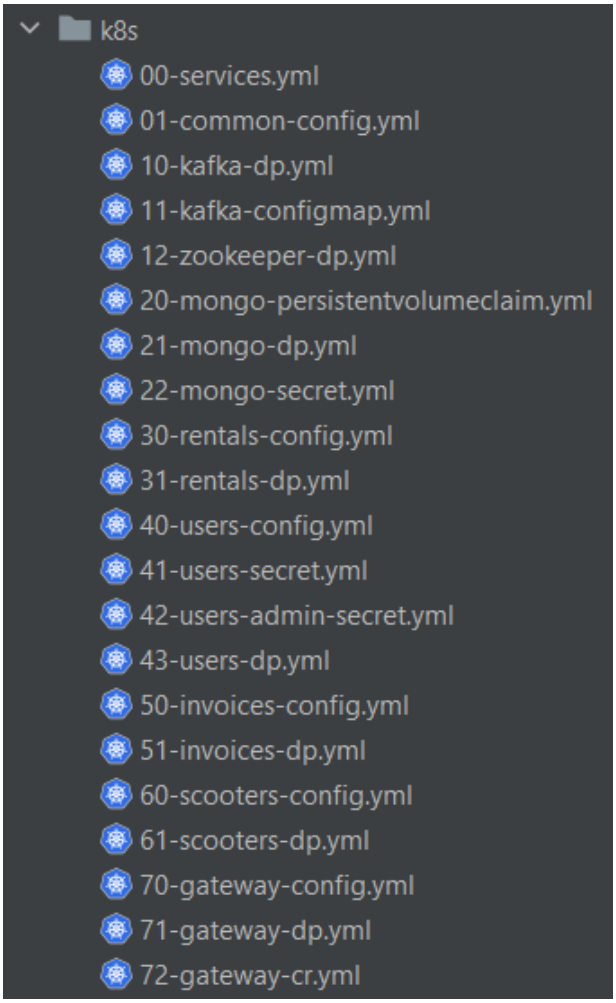
```yaml
version: '3.4'

services:
  mongo_db:
    image: mongo
    volumes:
      - mongodb:/var/lib/mongo

  users-service:
    build:
      context: .
      dockerfile: ./users_microservice/Dockerfile
    # ports:
    #   - "2225"
    #   - "2225:2225"
    restart: always
    env_file:
      - user.env
      - commons.env

volumes:
  mongodb:
```

Docker consente anche di effettuare l'estensione di un servizio a partire da un altro docker-compose file. Questo ha permesso di semplificare il **docker-compose** file finale. A seguire si riporta parzialmente il contenuto di **docker-compose.yml** per rendere l'idea:

```yaml
users-service:
    extends:
      service: users-service
      file: docker-compose-user.yml
    depends_on:
      - kafka-service
      - mongo-db
```

5. Kubernetes

Per un testing su singolo cluster, è stato utilizzato **Minikube** su **Docker**. Il progetto contiene una cartella "**k8s**" con tutti i file necessari all'esecuzione del progetto su Kubernetes:



- **Services**: contiene la definizione di tutti i servizi necessari. Ognuno espone la propria porta di lavoro come tipo "ClusterIP". Solo il gateway utilizza il tipo "LoadBalancer" per l'accesso esterno;
- **Common-config**: è una ConfigMap contenente le variabili d'ambiente comuni a tutti i micro-servizi;
- **Kafka files**: contengono il deployment di kafka con la rispettiva config map. Il deployment è di tipo "StatefulSet" in quanto Kafka deve mantenere uno stato consistente e non è stateless;
- **Zookeeper-dp**: deployment di zookeeper;
- **Mongo**: il suo deployment è di tipo "StatefulSet". È utilizzato un "Secret" per la creazione di un utente utilizzato per l'autenticazione dei micro-servizi. Inoltre un PersistentVolumeClaim si occupa di definire un volume;
- **Rental**: deployment e ConfigMap;
- **User**: sono necessari diversi file. Per l'autenticazione, è definito un Secret contenente la JWT Key. Per la creazione dell'amministratore, è definito un Admin-Secret contenente username e password. Infine si ha deployment e ConfigMap;
- **Invoice**: deployment e ConfigMap;
- **Scooter**: deployment e ConfigMap;
- **Gateway**: oltre al deployment e alla ConfigMap, è necessario definire una ClusterRoleBinding per ottenere i privilegi necessari ad utilizzare il servizio di Discovery.

Non avendo fatto uso di un Hub Remoto Docker da cui poter effettuare il Pull delle immagini, è effettuato il building dei micro-servizi all'interno dell'istanza Docker contenuta in Minikube, per poi specificare "**ImagePullPolicy: Never**" in ciascun Deployment file.

Ai fini di sfruttare il Fault Detector di Kubernetes, sono stati definiti i valori di readinessProbe e livenessProbe, andando a sfruttare il ping dei micro-servizi. Di seguito un esempio nel caso di "**scooter-dp**":

```yaml
livenessProbe:
  httpGet:
    path: /scooter/ping
    port: 2227
  initialDelaySeconds: 300
  failureThreshold: 5
  periodSeconds: 15
readinessProbe:
  httpGet:
    path: /scooter/ping
    port: 2227
  initialDelaySeconds: 150
  failureThreshold: 5
  periodSeconds: 15
```