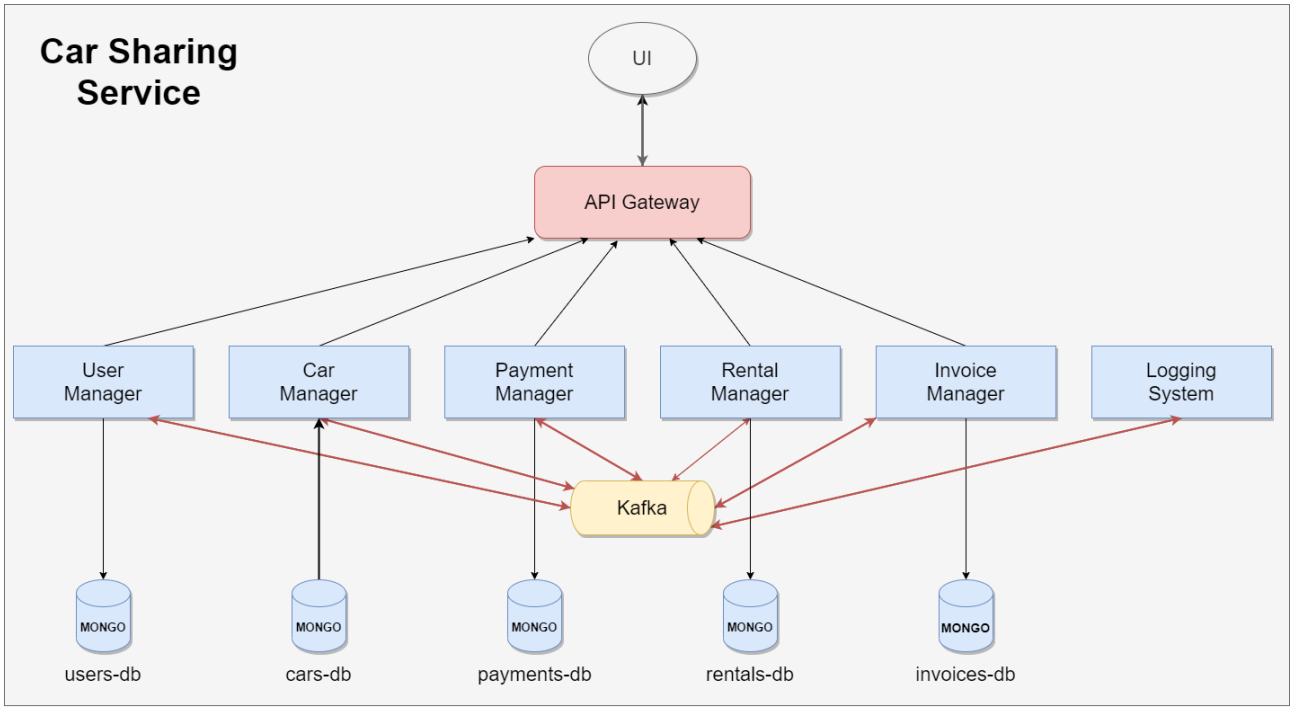
CAR SHARING

Distributed Systems and Big Data

Christian Cavallo e Noemi Buggea

1 SOMMARIO

2 INTRODUZIONE



L'idea consiste nel realizzare un semplice sistema distribuito per gestire un servizio di car sharing simile ai servizi esistenti a Catania (enjoy, amigo). In pratica vi è un app che consente di effettuare un noleggio temporaneo del veicolo. Al termine del noleggio, il veicolo viene automaticamente "chiuso" e si passa alla fase di pagamento (minuti di noleggio * tariffa). I servizi da sviluppare sono i seguenti:

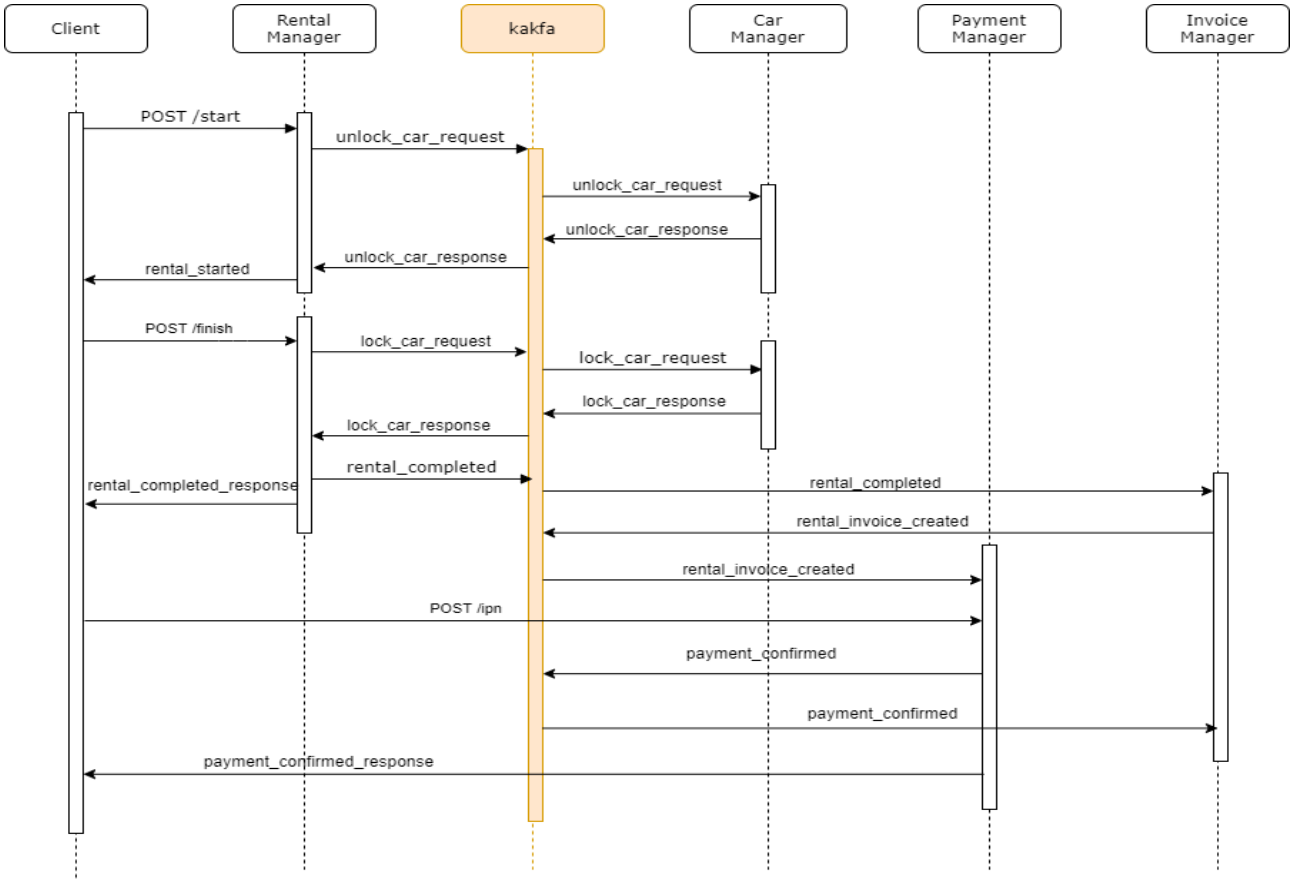
- **User Manager**: si occupa della registrazione e login degli utenti. È anche implementato un sistema di autenticazione che utilizza i JWT (JSON Web Tokens).
- **Car Manager**: si occupa della simulazione "blocco/sblocco" dei veicoli su richiesta di un utente.
- **Payment Manager**: si occupa di gestire il pagamento di una fattura relativa ad un noleggio. Utilizza un IPN Simulator appositamente sviluppato per simulare la presenza di un server PayPal.
- **Rental Manager**: si occupa della gestione di un noleggio (creazione, terminazione).
- **Invoice Manager**: si occupa di generare una fattura al momento della terminazione di un noleggio.
- **Logging System**: si occupa di ricevere i messaggi sul topic "log" e memorizzarli in un formato adatto all'interno di un DB.
- **Gateway**: si occupa di direzionare le richieste verso il corretto micro servizio (mediante discovery).

Ogni micro servizio distribuito è un progetto Spring Boot. Per il local testing usiamo la piattaforma Docker con Docker-Compose, mentre per il Distributed Testing usiamo Kubernetes con Minikube.

GitHub: CarSharing

2.1 PROCEDURA DI NOLEGGIO

In seguito è mostrato il diagramma 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 dà inizio ad una comunicazione request-reply sincrona tra il Rental Manager e il Car Manager mediante Kafka:

1. Il RM richiede lo sblocco del veicolo mediante una synchronous request di kafka e resta in attesa.
2. Il CM riceve la request ed effettua lo sblocco del veicolo (se possibile).
3. Il CM inoltra la response nel canale di risposta alla request di partenza.
4. Il RM gestisce la risposta del CM ed inoltra il risultato al Client di partenza.

La terminazione comporta una transazione simile, ma nel caso in cui il blocco del veicolo non vada a buon fine, è previsto uno stato di "Congelamento" del noleggio per evitare che il contatore temporale incrementi. Nel caso in cui la terminazione si concluda con successo, il RM invia un messaggio kafka che verrà ricevuto dall'Invoice Manager.
Ciò dà inizio a diverse transazioni locali gestite mediante il SAGA Pattern (Choreography-based):

1. L'Invoice Manager crea una nuova fattura nello stato di Pending e trasmette un messaggio di avvenuta creazione ad un topic Kafka
2. Il Payment Manager riceve tale messaggio e crea una richiesta di pagamento in stato di Pending per tale fattura.

A questo punto si attende che il teorico utente confermi il pagamento. Tale evento consiste nella ricezione di una post all'API "/ipn" (eseguita mediante il simulatore appositamente programmato).
A seguito della conferma di pagamento, lo stato della fattura e della richiesta di pagamento passa a "Paid".

3 PANORAMICA DEI MICRO SERVIZI

Proponiamo in seguito una panoramica dei singoli micro servizi. Per ognuno di essi sono descritte:

- API
- Entità
- Messaggi Kafka trasmessi/ricevuti
- Eventuali procedure per la Business Logic.

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

Per la gestione degli errori, i messaggi di errore vengono inviati su un topic kafka "**logging**" nel seguente formato:

```
key = http_errors
value = {
    timestamp: unixTimestamp,
    sourceIp: sourceIp, // ex: 192.168.1.1
    service: products,  // ex: User Manager
    request: method + path,  // ex: POST /register
    error: error    // ex: 404
}
```

Se l'errore è di tipo 50x, error è una la stringa con lo stack trace della exception generata o una spiegazione dell'errore. Se l'errore è di tipo 40x, error è lo status code in formato numerico.

Per il check della liveness, ogni servizio espone una API **"GET /ping"** che risponde con **Pong**.
Ciò è utile anche per l'utilizzo del Fault Detector di Kubernetes (Ping ACK Protocol).

È presente un amministratore del sistema il cui User Id è 0.
In docker e kubernetes, l'amministratore è automaticamente creato con quell'id, nel caso in cui non esista già.

Le richieste alle API devono possedere i seguenti headers ove richiesti:

- **X-User-ID** contenente l'id dell'utente che esegue la richiesta.
- **Authorization** che contiene il JWT (JSON Web Tokens) ottenuto durante il login.

3.1 USER MANAGER

Servizio di gestione degli utenti e di autenticazione.
Entità definita:

```
1.  public class User {
2.      private String id;
3.      private String name;
4.      private String email;
5.      private String password;
6.      private List<String> roles;
7.  }
```

Offre le seguenti API:

1. **POST /register**: registra un nuovo utente.
2. **GET /id/{id}:** ritorna un utente registrato in base all'id.
3. **GET /email/{email}:** ritorna un utente registrato in base all'email. Funziona solo nel caso in cui sia l'amministratore o il proprietario di tale email.
4. **GET /users?per_page=X&page=Y**: ritorna un array json di utenti solo se presente è l'amministratore. Gestire anche la paginazione.
5. **POST /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 (hashed) in una POST.
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", in quanto JWT utilizza l'algoritmo HMAC.

Produce i seguenti messaggi Kafka sul topic "**user**" solo per scopi informativi, dato che nessun altro servizio è interessato attualmente a questi messaggi:

```
1.  key = user_registered
2.  value = {
3.      user_infos
4.  }
```

3.2 CAR MANAGER

Entità coinvolte:

```
1.  public class Car {
2.      String carId;               // Id
3.      String code;                // Code to unlock the car
4.      String location;            // City
5.      Double lat;                 // Latitude
6.      Double lon;                 // Longitude
7.      CarStatus carStatus;        // Status of the car
8.  }
```

Il gestore auto necessità delle seguenti API:

1. **POST /add**: se è presente l'header con user-id dell'amministratore allora aggiunge una nuova auto i cui parametri sono passati in json all'interno del body. Ritorna il json dell'auto completo.
2. **GET /cars?loc=X:** restituisce un array Json di auto sulla base della località. Vengono ritornate solo le auto "noleggiabili" e non quelle attualmente in uso.

Bisogna tenere in considerazione della possibilità di concorrenza nella fase di sblocco del veicolo. È infatti possibile che 2 utenti cerchino contemporaneamente di noleggiare il veicolo. Perciò bisogna implementare la logica di sblocco/blocco come una transazione garantendone l'isolamento per evitare inconsistenze.

Gli eventi inerenti il blocco/sblocco di un veicolo sono gestiti in modo sincrono sfruttando il meccanismo request-reply di Kafka. Praticamente il Rental Manager manda delle richieste sincrone, che vengono ricevute dal Car Manager mediante KafkaListener con topic "**car_requests**". Le response vengono inviate poi al topic "**car_responses**".

Il blocco/sblocco del veicolo richiede 2 condizioni:

1. L'utente deve trovarsi in prossimità del veicolo (soglia parametrizzata).
2. L'utente deve fornire un codice disponibile all'interno del veicolo.

Ciò è necessario per prevenire l'apertura del veicolo ad una distanza non sicura o per prevenire utilizzi illeciti del servizio.

Entità delle requests e responses:

```
1.  public class CarRequest {
2.      private String car_id;
3.      private String user_id;
4.      private String rental_id;
5.      private CarOperation operation;      // Lock or Unlock
6.      private Double lat;                  // User's Latitude
7.      private Double lon;                  // User's Longitude
8.      private String carCode;              // Code to unlock/lock the car
9.  }
```

```
1.  public class CarResponse {
2.      private String car_id;
3.      private Boolean success;
4.      private String message;
5.  }
```

Il servizio è produttore dei seguenti messaggi kafka sul topic "**car**" (utile per debugging):

```
key = car_unlocked
value = {
    car_id: id,
    rental_id: id,
    user_id: id,
    timestamp: t,
}
```

```
key = car_locked
value = {
    car_id: id,
   timestamp: t,
   extraArgs: ...
}
```

In caso di errore durante il blocco/sblocco del veicolo, viene generato un errore nel topic "**logging**":

```
key = car_unlock_failure
value = {
    car_id: id,
   timestamp: t,
   extraArgs: ...
}
```

Se l'errore è inerente al blocco, si usa la `key="car_lock_failure".`
Ciò porta all'annullamento del noleggio nel caso si tratti di un errore nell'apertura del veicolo, mentre porta ad un congelamento temporaneo del noleggio se invece è un problema inerente il blocco.

3.3 PAYMENT MANAGER

Si ipotizza la possibilità di effettuare pagamenti mediante il sistema PayPal.
L'entità coinvolta è:

```
public class Payment {
    private String id;                  // Payment Id
    private String rentalId;            // Rental Id
    private String userId;              // User Id
    private Double amount;              // Amount to pay
    private String currency;            // Currency ($, €)
    private String business;            // Business Email
    private String payer_email;         // Client Email
    private PaymentStatus paymentStatus; // Created or Paid
    private Long timestamp;
}
```

Le API da implementare sono le seguenti:

1. **POST /ipn**: simula la notifica di avvenuto pagamento.
2. **GET /transactions?fromTimestamp=unixTimestamp1&endTimestamp=unixTimestamp2**: recupera un array json di transazioni effettuate comprese nell'intervallo di tempo tra "fromTimestamp" e "endTimestamp". Ma richiede che sia presente un header con X-User-ID=0 (Amministratore).

All'arrivo del messaggio dal topic "**Invoice**" con `key "rental_invoice_created",` viene generato un nuovo pagamento e viene posto nello stato "**CREATED**".

All’arrivo della notifica inerente l’API 1, vengono effettuate le seguenti operazioni:

1. Verifica della validità della richiesta, ovvero che venga da paypal e rispecchi la transazione.
2. Se la verifica di prima passa, si verifica che la richiesta abbia il campo “business” del body uguale ad una email impostata come variabile d’ambiente (ad esempio MY_PAYPAL_ACCOUNT = "mybusiness@mycompany.tld")

In caso di avvenuto pagamento, si invia un messaggio kafka sul topic “**invoice**”:

```
key = rental_paid
value = {
    rental_id: id,
    user_id: id,
    amount_paid: amountPaid,
}
```

Praticamente è l’IPN SIMULATOR di paypal a richiamare /ipn.
Si risponde sempre con un messaggio vuoto e codice 200, altrimenti viene ritrasmesso, e si salva la transazione nel DB. Dopo di che si verificano i dati della richiesta e si trasmette una POST a https://ipnpb.sandbox.paypal.com/cgi-bin/webscr con le seguenti proprietà:

1. i parametri della richiesta precedente più cmd=_notify-validate.
2. Il content type impostato su **application/x-www-form-urlencoded**

La risposta ottenibile può essere “VERIFIED” o “INVALID”.
È stato sviluppato anche un ipotetico Paypal Server in Python a cui inviare tale richiesta.

In caso di errore si trasmette sul topic “**logging**” il seguente messaggio:

```
key = rental_payment_failure
value = {
    rental_id: id,
    user_id: id,
    timestamp: t,
   extraArgs: …
}
```

Nel caso in cui fallisca la **POST /ipn** nel punto 1, si usa la *key=”bad_ipn_error”.* Se fallisce il punto 2 si usa la *key=”received_wrong_business_paypal_payment*”.

3.4 RENTAL MANAGER

Questo micro servizio è responsabile della creazione, avvio e terminazione dei noleggi degli utenti.
È definita la seguente entità:

```
1.  public class Rental {
2.      private String id;                  // Rental Id
3.      private String car_id;              // Car Id
4.      private String userId;              // User Id
5.      private Long startTimestamp;        // Start time
6.      private Long stopTimestamp;         // End time
7.      private Double price_per_minute;    // Rate per Minute
8.      private Double amount_to_pay;       // Total amount to pay
9.      private RentalStatus status;        // STARTED, COMPLETED or FROZEN
10. }
```

Espone le seguenti API:

1. **GET /{id}:** ottiene il noleggio con l'id specificato solo se lo userId presente come header coincide con quello presente nel noleggio o è 0.
2. **POST /start:** consente di iniziare un nuovo noleggio.
   Richiede che vengano forniti i seguenti dati:
   a. Latitudine e Longitudine dell'utente.
   b. Codice di sblocco (solitamente inserito in un punto visibile del veicolo).
   c. Header X-User-ID per identificare l'utente.
3. **POST /stop:** consente di terminare il noleggio. Richiede gli stessi parametri dell'api di start.
4. **GET /rentals?per_page=x&page=y:** ritorna un array json di noleggi relativi all'utente con id espresso nell'header "X-User-ID". Se è 0, ritorna tutti i noleggi. È gestita la paginazione.

Quando viene richiesto lo start di un nuovo noleggio, deve essere verificato se il veicolo sia effettivamente disponibile. Può succedere che 2 utenti richiedano lo stesso veicolo nello stesso momento, portando ad una inconsistenza sullo stato dell'auto (disponibile o non disponibile?).
Questo problema è gestito dal Car Manager, il quale implementa l'operazione di sblocco/blocco come una Transazione che rispetta l'isolamento.

Vengono trasmesse le richieste definite nel capitolo del Car Manager (3.2).

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". Il contatore inerente il tempo di utilizzo del veicolo viene fermato e l'utente ha il dovere di contattare un teorico "Servizio di assistenza" per risolvere il problema (NON IMPLEMENTATO).

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

```
1.  key = rental_accepted
2.  value = {
3.      rental_id: id,
4.      car_id: id,
5.      user_id: id,
6.     timestamp: t,
7.  }
```

```
1.  key = rental_completed
2.  value = {
3.      rental_id: id,
4.      car_id: id,
5.      user_id: id,
```

```
6.      start_timestamp: t1,
7.      end_timestamp: t2,
8.     price_per_minute: rate,
9.  }
```

In caso di errore si trasmette sul topic "**logging**" il seguente messaggio:

```
1.  key = …
2.  value = {
3.      rental_id: id,
4.      user_id: id,
5.      timestamp: t,
6.  }
```

Nel caso in cui l'errore sia dovuto ad un veicolo non disponibile, la key è *"rental_failure_car_not_available".* Se invece vi è un problema nella terminazione del noleggio, è necessario impostare lo stato del noleggio in "Frozen" (congelato) e la key è *"rental_car_locking_failure*".

3.5 INVOICE MANAGER

Micro servizio per la gestione delle fatture.
La fattura è rappresentata dalla seguente entità:

```
1.  public class Invoice {
2.      private String id;
3.      private String rentalId;
4.      private String carId;
5.      private String userId;
6.      private Integer count;                          // Invoice's counter
7.      private Double totalAmount;                     // Amount to pay
8.      private Double price_per_minute;                // Rate per minute
9.      private InvoiceStatus invoiceStatus;            // PENDING or PAID
10.     private Date start_timestamp, end_timestamp;    // Start and End time
11. }
```

Offrire le seguenti API:

1. **GET /invoices/{id}:** Risponde con la rappresentazione Json della fattura con id {id} se lo user id della fattura è uguale a quello fornito o è 0. Altrimenti 404 (se la fattura non esiste o non è associata allo user id fornito).
2. **GET /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. Gestire anche la "paginazione".
   Ad esempio, **GET /invoices?per_page_10&page=2** indica la pagina 2 con massimo 10 risultati.

Inoltre il sistema è consumatore del seguente messaggio Kafka sul topic **rental**:

```
1.  key = rental_completed
2.  value = {
3.    rental_id: id,
4.    car_id: id,
5.    user_id: id,
6.    start_timestamp: t1,
7.    end_timestamp: t2,
8.    price_per_minute: rate,
9.    totalAmount: amount,
10. }
```

All’arrivo del precedente messaggio, viene creata la fattura rispettiva fattura. La fattura generata sarà posta in stato di PENDING (in attesa di essere pagata).

Viene trasmesso il seguente messaggio sul topic “**invoice**”:

```
key = rental_invoice_created
value = {
    invoice
   …
}
```

Lo stato della fattura viene modificato all’arrivo del messaggio di avvenuto pagamento sul topic “**invoice**”:

```
key = rental_paid
value = {
    rental_id: id,
    user_id: id,
    amount_paid: amountPaid,
}
```

Per farlo, è necessario cercare la fattura associata a (rental_id, user_id, amountPaid). Se esiste, lo stato della fattura viene impostato su Paid e si genera un numero fattura incrementale.
Il numero fattura deve ricominciare ad ogni 1 gennaio. Quindi, ad ogni generazione, si prende il max dei numeri tra le fatture dell'anno (se nessuna fattura è ancora stata emessa sarà 0) e si incrementa di uno.
Se non esiste la fattura si invia nel topic **logging** il messaggio:

```
key= invoice_unavailable
value = {
    invoiceId: id,
    user_id: id,
    timestamp: t,
}
```

Se invece si riceve il messaggio di pagamento fallito, la fattura resta in stato di PENDING e viene trasmesso il seguente messaggio:

```
key = rental_payment_failure
value = {
    rental_id: id,
    user_id: id,
    timestamp: t,
    amount_paid: amountPaid
}
```

3.6 LOGGING MANAGER

Questo micro servizio è sottoscrittore kafka del topic **logging**.
Alla ricezione di ognuno dei possibili messaggi, salva sul database il loro contenuto.
L'entità deve avere una struttura adatta a gestire e memorizzare i differenti messaggi:

```
1.  public class Log {
2.      private String id;
3.      private String key;         // Message Key
4.      private String service;     // Service Name
5.      private String payload;     // Message Content
6.      private Long timestamp;     // Message Timestamp
7.   }
```

Se non è disponibile un timestamp, viene aggiunto automaticamente. Il formato scelto consente di implementare le seguenti API:

1. **GET /keys/{key}?from=unixTimestampStart&end=unixTimestampEnd:** recupero di tutti i log message con key = {key} che abbiano tempo compreso fra i due forniti nei parametri della GET (obbligatori)
2. **GET /http_errors/services/{service}?from=unixTimestampStart&end=unixTimestampEnd**: retrieve di tutti i log message di tipo http_errors associati al service {service}.

3.7 GATEWAY

Il gateway è una parte fondamentale del progetto.
Esso svolge le seguenti funzionalità:

- Reindirizza le richieste verso il corretto micro servizio sfruttando un servizio di Discovery.
- Filtra le API che richiedono l'autenticazione.
- Verifica la validità dei Json Web Tokens.

Considerando il testing su Docker, è possibile semplicemente sfruttare la rete che docker compose crea tra i container del progetto. Ovvero basta impostare delle apposite route per ciascun micro servizio.
Ma visto che esistono delle soluzioni alternative, abbiamo deciso di utilizzare **Eureka** come discovery server per l'esecuzione su Docker, mentre sfruttiamo il **Kubernetes Discovery Service** per l'esecuzione sul cluster minikube.
Il gateway implementa entrambe le alternative grazie all'annotazione comune: **@EnableDiscoveryClient**.
La scelta se attivare o meno il discovery con Eureka/Kubernetes è parametrizzata mediante due variabili d'ambiente. In tal modo, quando è necessario eseguire su Docker, basta abilitare Eureka e disabilitare K8S Discovery e viceversa, in modo da non creare conflitti.

Riguardo l'autenticazione, il file *application.properties* contiene due variabili "**services**" e "**api_services**" corrispondenti ai nomi dei servizi e rispettivi API Base Path (es. /user o /rental) che richiedono l'autenticazione. A partire da queste variabili, sono automaticamente generate le route.
A ciascuna route è associato un medesimo **AuthenticationFilter,** il quale verifica la presenza di un **Authorization Header** e ne verifica la validità.
Nel caso in cui si tratta di un token invalido, risponde con 403 *Unauthorized*. Se invece il token è valido, vi estrae lo User Id ed aggiunge l'header **X-User-ID** (necessario alle API per identificare l'utente) automaticamente alla richiesta.

Un ulteriore variabile d'ambiente "**open_endpoints**" contiene una lista di endpoints accessibili senza la necessità di fornire un Authorization Token (es. /login, /register e /ping).

4 DOCKER

Nello sviluppo dei vari micro servizi, abbiamo deciso di dividere equamente il lavoro complessivo. Docker e compose ci hanno permesso di testare singolarmente ciascun micro servizio in modo completamente indipendente.
Ciascun micro servizio ha un DockerFile per effettuare il building dell'immagine:

```
1.  FROM maven:3-jdk-8 as builder                       # Imposta il builder con jdk-8
2.  WORKDIR /project                                    # Spostamento della cartella project
3.  COPY ./users_microservice/pom.xml ./pom.xml         # Copia del pom.xml
4.  RUN mvn dependency:go-offline -B                    # Download delle dipendenze
5.  COPY ./users_microservice/src ./src                 # Copia della cartella src
6.  RUN mvn package                                     # Building del file jar
7.
8.  FROM java:8-alpine                                  # Imposta java-8 come executer
9.  WORKDIR /app                                        # Spostamento in app
10. COPY --from=builder /project/target/users_microservice-0.0.1-SNAPSHOT.jar
    ./users_microservice.jar                            # Copia del jar file appena creato
11. CMD java -jar users_microservice.jar                # Esecuzione del jar
```

Abbiamo 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 comune a tutti (**commons.env**).
Riportiamo in seguito il ***docker-compose-user.yml*** come esempio:

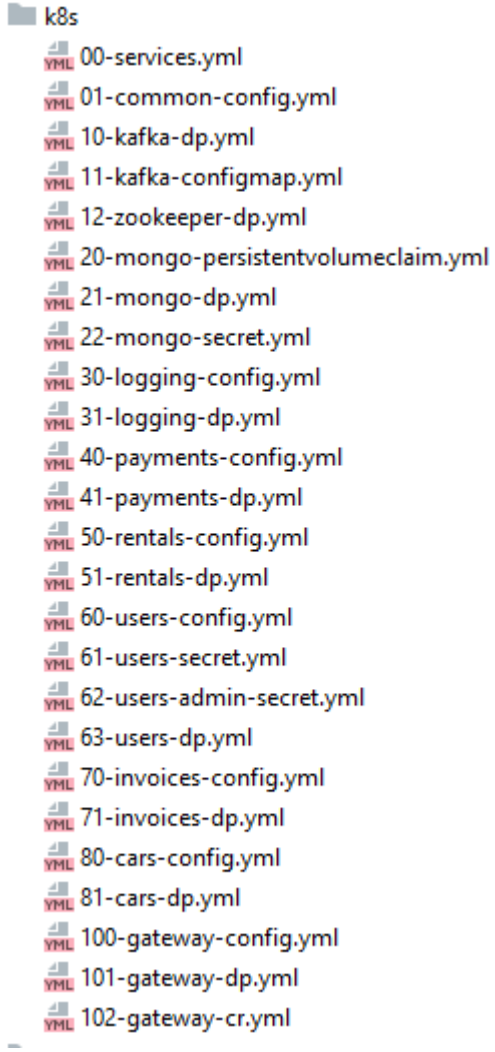
```
1.  version: '3.4'
2.
3.  services:
4.    mongo_db:
5.      image: mongo
6.      volumes:
7.        - mongodb:/var/lib/mongo
8.
9.    users-service:
10.     build:
11.       context: .
12.       dockerfile: ./users_microservice/Dockerfile
13.     #ports:
14.     #  - "2225"
15.     #  - "2225:2225"
16.     restart: always
17.     env_file:
18.       - user.env
19.       - commons.env
20.
21. volumes:
22.   mongodb:
```

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

```
1.    users-service:
2.      extends:
3.        service: users-service
4.        file: docker-compose-user.yml
5.      depends_on:
6.        - kafka-service
7.        - mongo-db
```

5 KUBERNETES

Per un testing su singolo cluster, abbiamo utilizzato (come richiesto) **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.
- **Logging**: deployment + ConfigMap
- **Payment**: deployment + ConfigMap
- **Rental**: deployment + 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 abbiamo deployment e ConfigMap.
- **Invoice**: deployment + ConfigMap
- **Cars**: deployment + 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 un Hub Remoto Docker da cui poter effettuare il Pull delle immagini, abbiamo 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, abbiamo definito "**readinessProbe**, **livenessProbe**", sfruttando il ping. Per il Car Manager ad esempio:

```
1.  readinessProbe:
2.    httpGet:
3.     path: /car/ping
4.     port: 2227
5.    initialDelaySeconds: 20
6.    failureThreshold: 30
7.    periodSeconds: 10
8.  livenessProbe:
9.    httpGet:
10.    path: /car/ping
11.    port: 2227
12.   initialDelaySeconds: 60
13.   failureThreshold: 5
14.   periodSeconds: 15
```