Relazione tecnica progetto CarPooling (*Gestore Utenti*)

Glossario

Requisiti iniziali

Il progetto consiste nello sviluppo di un' applicazione per il *carpooling,* nella quale un utente è in grado di registrarsi come *client* (alla ricerca di un percorso da poter prenotare) oppure come *proprietario* di una macchina, quindi è lui stesso che mette a disposizione i percorsi che possono essere prenotati.
Un' altra entità importante nell' architettura è il *car controller*, che può essere visto come un dispositivo hardware (come un semplice Raspberry ad esempio) che viene installato sulla propria macchina, e se abilitato, consente di inviare periodicamente la propria posizione al *Gestore Percorsi* per poter registrare e memorizzare il percorso che si sta svolgendo.
Nel momento in cui un utente *client* (il quale si distingue da un utente proprietario per il semplice fatto che non ha alcun car controller associato al proprio account) decide di prenotare un percorso, tramite un semplice sistema di messaggistica il proprietario del percorso viene notificato, e può decidere se accettare oppure rifiutare la richiesta.

Requisiti riscritti

Utente

Per gli utenti identificati da un ID rappresentiamo:

- nome
- cognome
- email
- username
- password (*informazione che verrà custodita nell' auth server*)
- data di nascita

Interazione col sistema

- Un utente deve registrarsi/loggarsi per accedere al servizio. Al momento della registrazione vengono chiesti tutti i dati precedentemente menzionati, ad esclusione dell' ID ovviamente che viene gestito dal sistema. Al momento del login il sistema richiede solamente username e password per l' autenticazione.
- Un utente registrato può visionare i percorsi disponibili inserendo partenza e arrivo. Per ogni percorso disponibile l' utente può visionare proprietario e macchina che offrono tale percorso. Da qui si deduce inoltre che ad ogni percorso è associato un ID controller, legato alla macchina che offre il percorso ed un ID dell' utente proprietario.
- Un utente può accordarsi con un altro utente proprietario tramite messaggi, per un passaggio su un percorso già tracciato e visibile pubblicamente, tramite un sistema di notifica basato su messaggi di *richiesta* e *conferma*. Questo ovviamente può declinare o accettare la richiesta e l' utente ne verrà notificato
- Un utente proprietario può mettere a disposizione una sola auto, questo vuol dire che può aver associato al massimo un unico car controller
- Un utente proprietario può visionare e rimuovere i percorsi generari dalla sua macchina e anche impedire le prenotazioni su un certo percorso, impostandone la visibilità

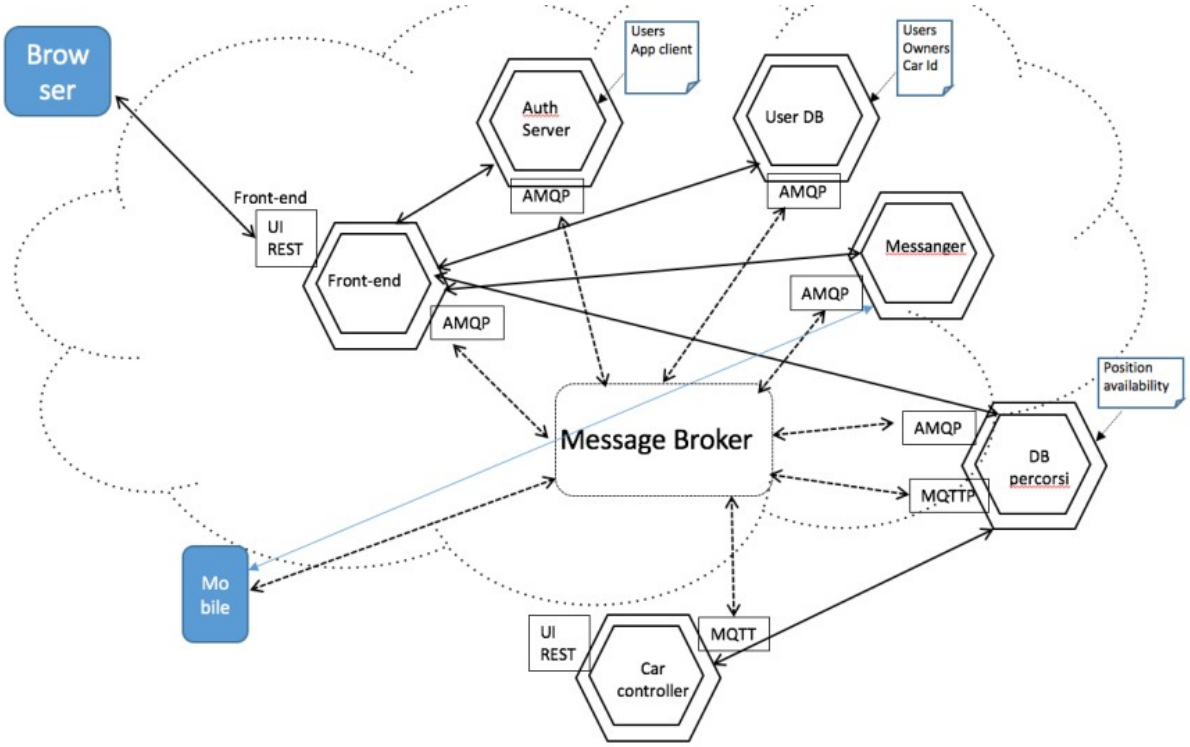
- Un utente proprietario può aggiungere una *descrizione della macchina* su cui ha installato il car controller
- Un utente proprietario può abilitare un car controller ad inviare posizioni per una sua macchina

Car controller
Per i car controller identificati da ID rappresentiamo:
- ID utente proprietario
- Campo descrizione del veicolo
- Numero dei posti presenti sulla vettura
- Status (ON/OFF)

Il car controller viene installato su una singola automobile.

Interazione col sistema
Un car controller per poter funzionare deve essere abilitato da un utente.
Il car controller può essere in due stati (ON/OFF), in stato di ON questo è abilitato ad inviare la posizione attuale del controller al Gestore Percorsi periodicamente.

Descrizione generale dell' architettura



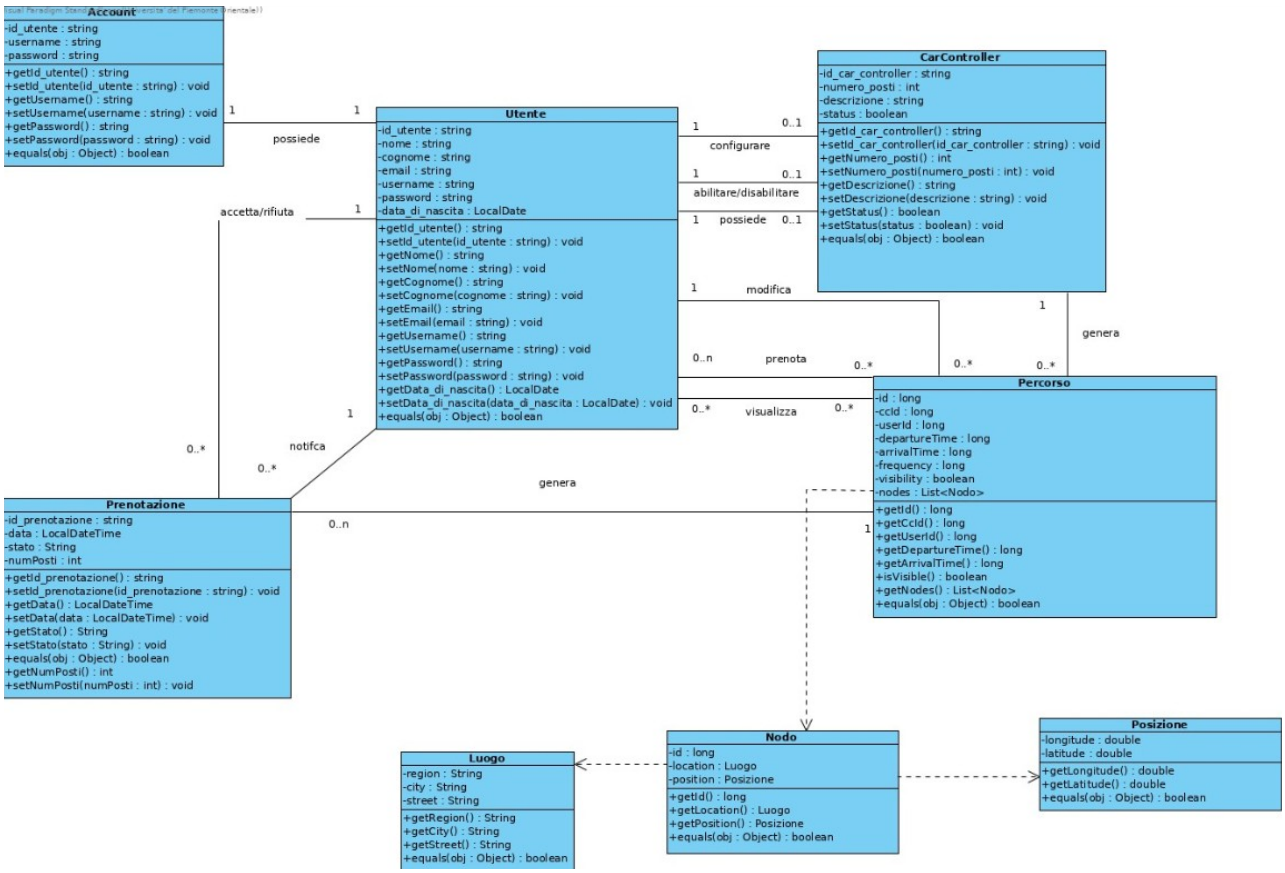
La seguente architettura descrive un servizio di carpooling, il quale permette a diversi utenti di poter effettuare varie azioni, oltre alle due principali: *prenotare un passaggio* e *mettere a disposizione la propria macchina per un viaggio*.
Il sistema offre, ad esempio, la possibilità agli utenti di potersi accordare tramite un sistema di messaggistica basato su notifiche.
L' architettura è la classica architettura su microservizi, dove ciascun microservizio viene gestito ed implementato in maniera autonoma dagli altri secondo quanto concerne lo schema di programmazione agile.
L' architettura comprende:

- **Front-End:** microservizio destinato ad interfacciarsi con l' utente, si occupa dell' autenticazione di questo e risulta essere il soggetto principale dell' architettura
- **Auth-server:** server utilizzato per l'autenticazione degli utenti e per lo storaggio dei dati sensibili di questi come la passsword
- **Gestore Utenti (+ DB):** si occupa dello storaggio delle informazioni relative ad utenti e car controller (DB), e collabora con il Front-End per restituire i dati richiesti dagli utenti (Gestore)
- **Messanger (+ DB):** si occupa del sistema di messaggistica precedentemente menzionato
- **Gestore Percorsi (+ DB):** si occupa dello storaggio delle informazioni relative ai percorsi (DB), e collabora con il Front-End e il Car controller per la gestione di questi
- **Car controller:** dispositivo che si occupa di inviare le proprie posizioni al Gestore Percorsi, ogni macchina viene gestita da un solo car controller, per semplicità si è scelto che ciascun utente può essere proprietario al massimo di un solo car controller

Questi microservizi comunicano fra di loro tramite un sistema di messaggistica publish/subscribe che può essere AMQP oppure MQTT.

Diagramma delle classi

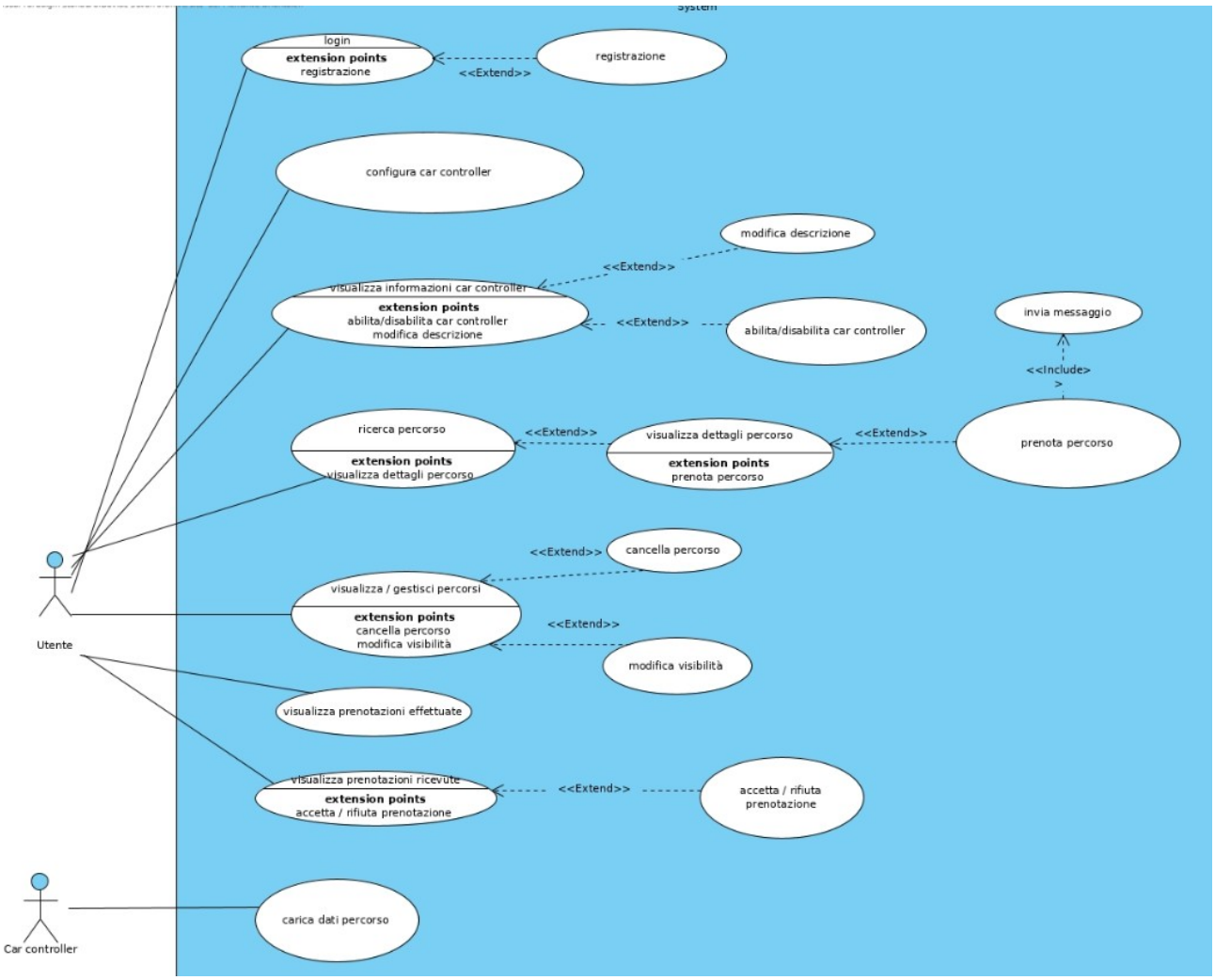
La prima parte della progettazione, è consistita nel progettare il *diagramma delle classi*, quel diagramma cioè che riassume le principali classi che vengono utilizzate per modellare al meglio i concetti e gli attori necessari alla corretta implementazione dell' architettura.



Questo ha permesso di capire meglio, anche a noi sviluppatori, come questi collaborassero fra di loro.

Diagramma dei casi d'uso

Successivamente abbiamo analizzato le azioni che potevano essere intraprese dagli attori principali dell' architettura ovvero gli *utenti* e i *car controller*.
Per comprendere al meglio questo, e facilitare la fase di implementazione, seguendo il processo di ingengnerizzazione del software, abbiamo provveduto alla realizzazione dei *diagrammi dei casi d'uso*.
Questo ci ha permesso di comprendere più a fondo anche come più azioni potessero essere facilmente correlate fra di loro.

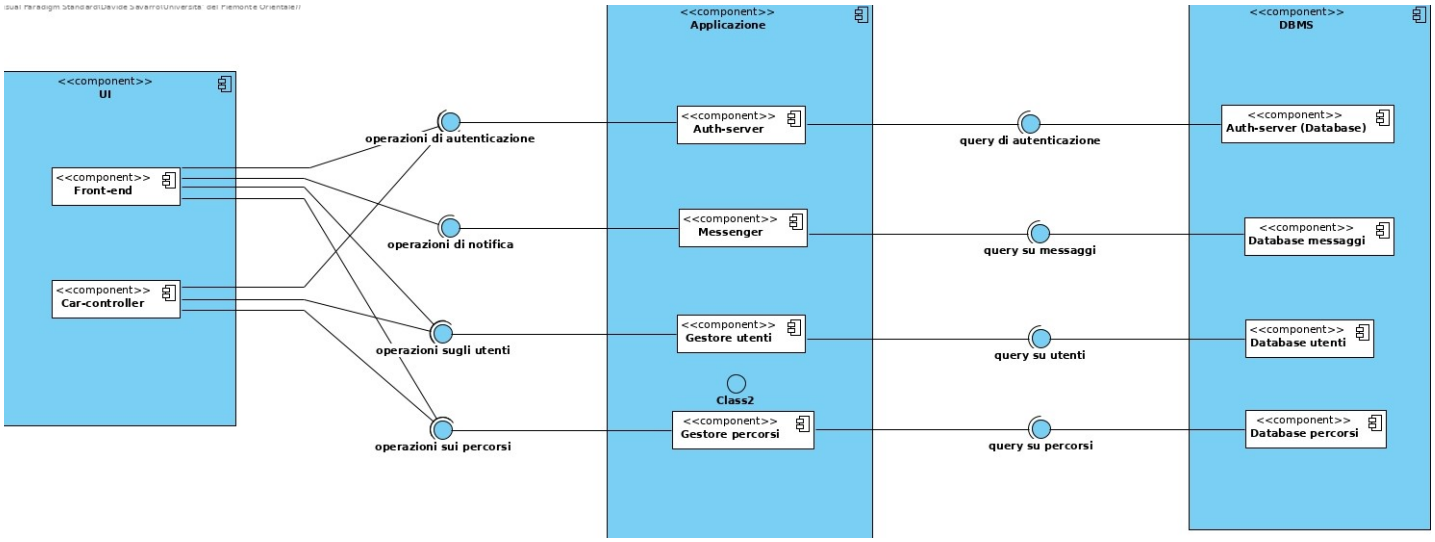


Dove:

- **<<Extend>>:** indica un' azione ulteriore che PUO' essere fatta a partire da un' altra
- **<<Include>>:** indica un' azione che DEVE essere fatta a partire da un' altra

Diagramma delle componenti

Una volta avuta una visione più generale dei requisiti ad alto livello che doveva rispecchiare l' architettura, ci siamo focalizzati sullo stile che doveva presentare tale architettura.
Per rendere più chiaro questo concetto abbiamo utilizzato un *diagramma delle componenti*, il quale ha evidenziato piuttosto bene che nonostante l' architettura fosse a microservizi, rispecchiasse molto il concetto dell' architettura 3-tier (*Presentazione-Elaborazione-Gestione*).



Dal diagramma risulta essere piuttosto evidente la somiglianza con l' architettura classica.
Abbiamo i due attori principali (Front-End e Car-controller) che comunicano con i vari gestori e l' auth-server, questi a loro si appoggiano su dei propri database (ogni gestore ha il proprio database e lo gestisce in maniera autonoma).
Ogni gestore quindi dovrà occupare di verificare la validità del token passatogli, e se questa è garantita, restituire i dati che vengono richiesti.

Diagramma di deployment

Seguendo il processo di ingegnerizzazione del software, ultimato il diagramma dei componenti, è stato realizzato il diagramma di deployments, utile per definire meglio la posizione di tali attori e gestori e come questi comunicassero fra di loro.

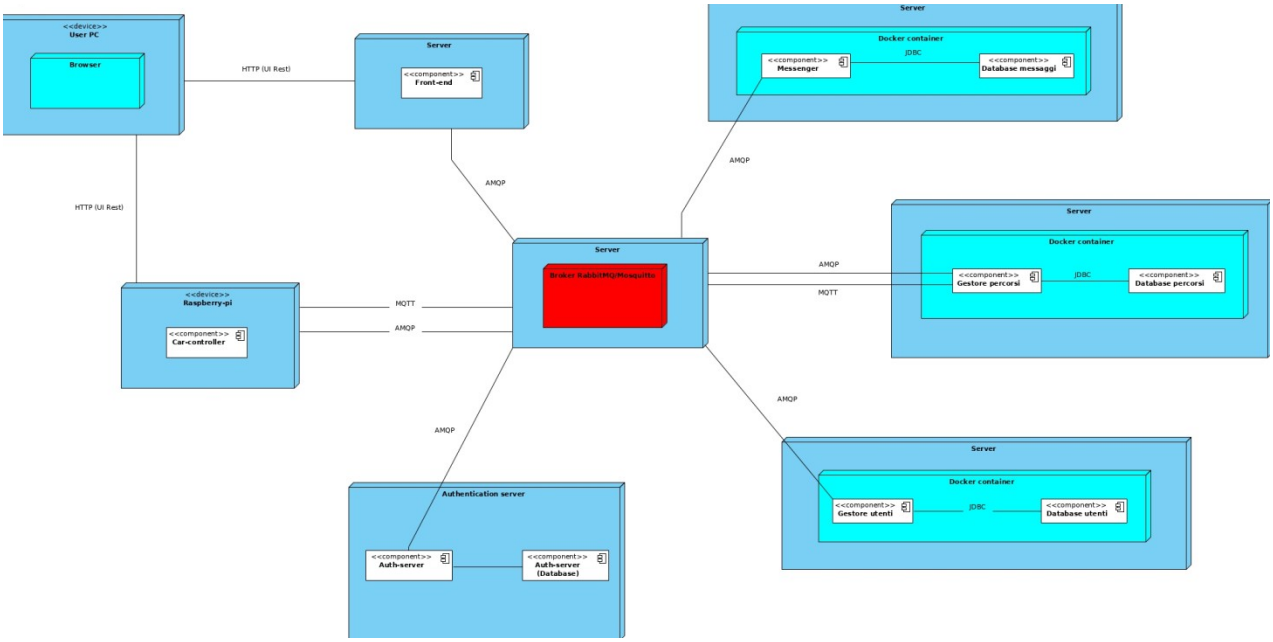
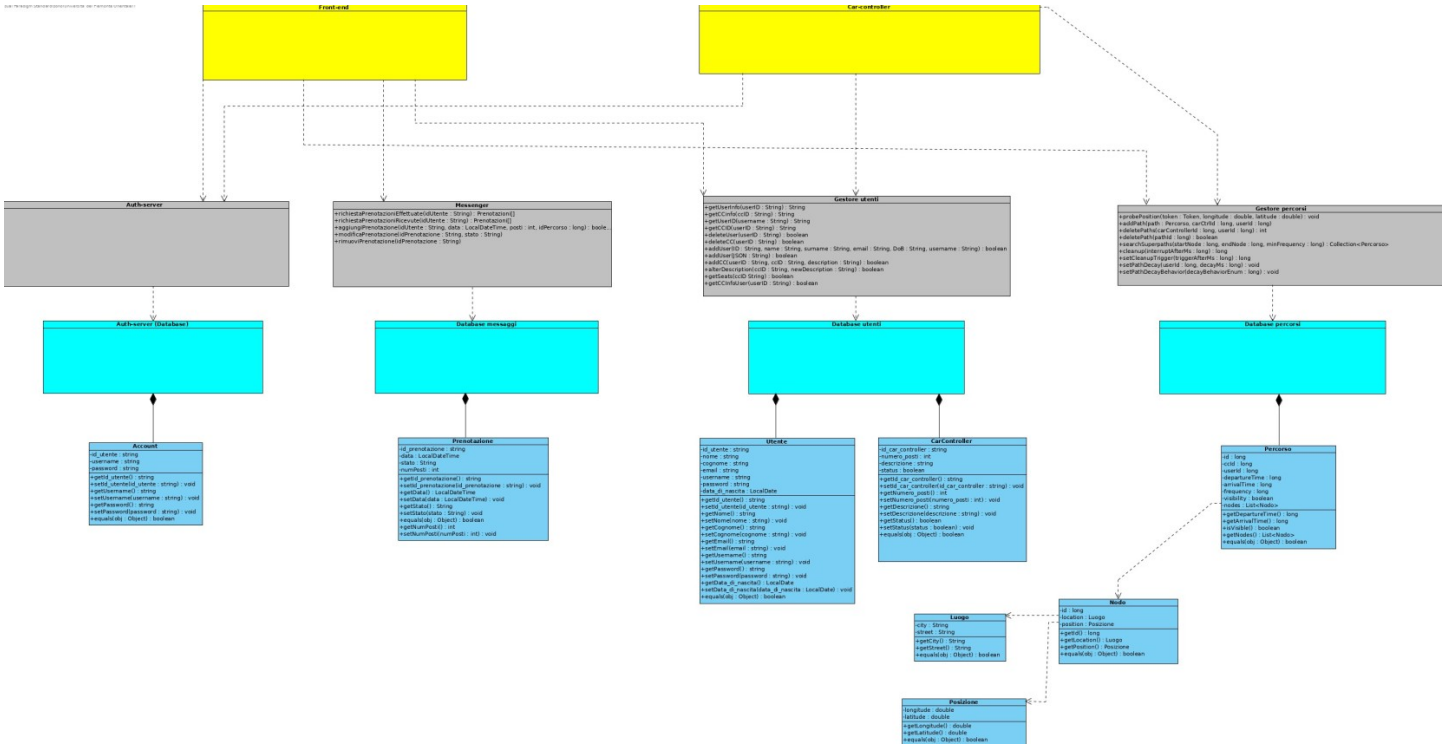


Diagramma delle classi (Architettura)



Questo diagramma, in collaborazione con gli altri, ci ha finalmente dato la possibilità di poter finalmente passare alla progettazione dei *diagrammi di sequenza*.
Infatti è risultato importante per prima cosa poter definire i metodi che ciascun componente offre in modo tale da aver le API corrette da inserire poi nei diagrammi di sequenza.
Per la realizzazione di questo ultimo tipo di diagrammi, si è deciso che ciascun sviluppatore provvedesse a realizzare i diagrammi delle proprie API, cioè quelle esposte dal proprio microservizio.

Gestore Utenti

Le API offerte da tale microservizio sono:

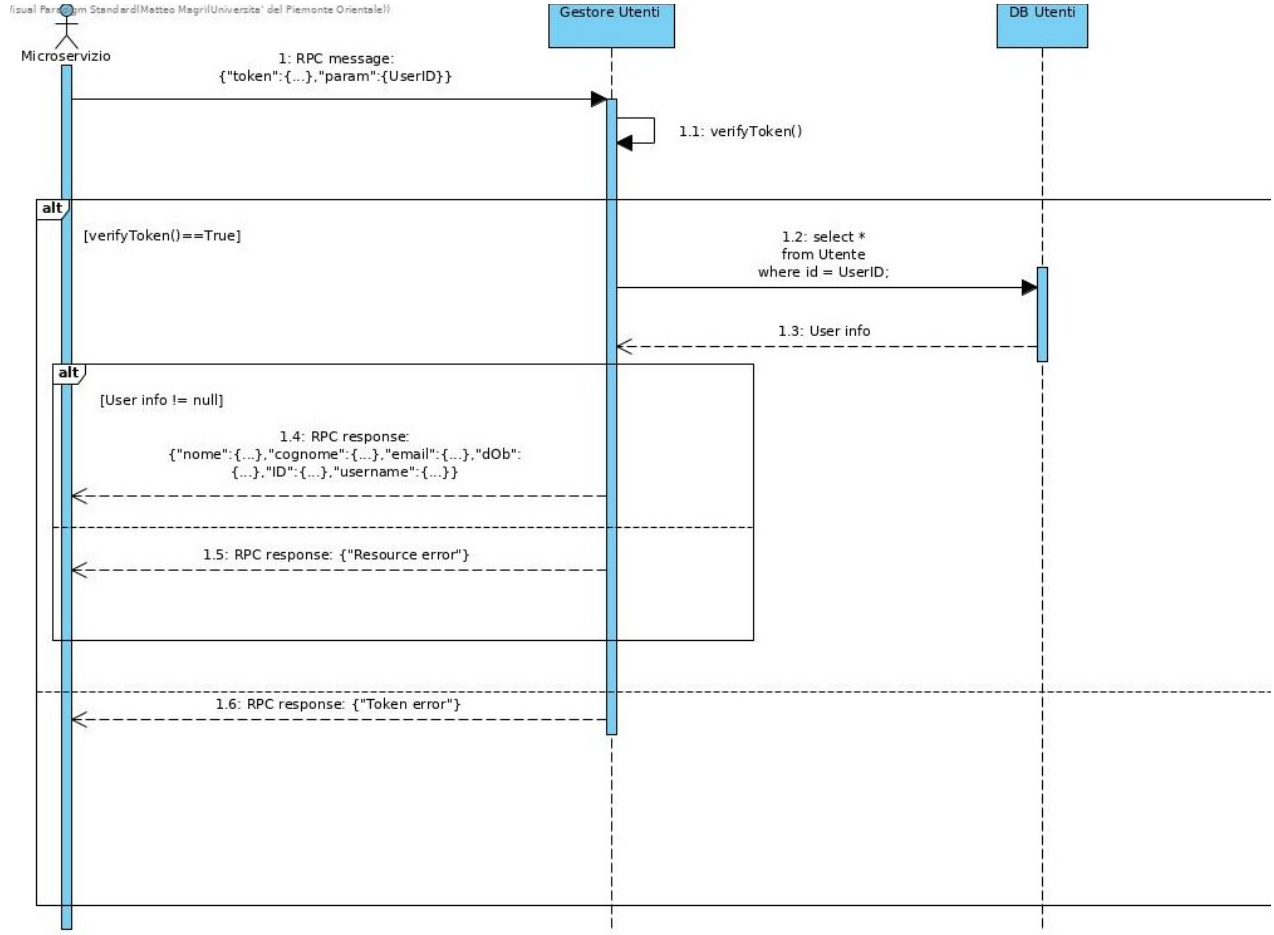
- `getUserInfo:` funzione che restituisce le informazioni relative ad un utente dato il suo ID
- `addUser:` funzione che dati i valore degli attributi di un utente provvede a memorizzarlo all' interno del database
- `addCC:` funzione che dati i valore degli attributi di un car controller provvede a memorizzarlo all' interno del database
- `alterDescription:` funzione che permette di modificare la descrizione di un car controller presente all' interno del database
- `deleteCC:` funzione che permette di eliminare un car controller dal DB, provvedendo anche a disassociarlo dal corrispondente proprietario
- `deleteUser:` funzione che permette di eliminare un utente dal DB
- `getCCInfo:` funzione che restituisce le informazioni relative ad un car controller (CC)
- `getCCID:` funzione che restituisce l' ID di un car controller
- `getCCInfoUser:` funzione che restituisce le informazioni di un car controller associato ad uno specifico utente
- `getSeats:` funzione che restituisce il numero di posti associato ad un car controller (che ricordo essere a sua volta associato ad una macchina)
- `getStatus:` funzione che restituisce lo stato corrente di un car controller (ON=invia posizioni, OFF=non invia posizioni)
- `setStatus:` funzione che permette di modificare lo stato corrente di un car controller
- `getUserID`: funzione che permette di recuperare l' ID di un utente dato il suo username
- `getUserInfoFromCC:` funzione che permette di recuperare le informazioni di un utente dato l' ID del suo car controller

E' doveroso specificare che queste API sono state implementate in software sottoforma di **classi** (sottoclassi della classe Thread) e non di metodi, in modo tale da poter aumentare le performance del backend, il quale si occupa unicamente di prelevare i messaggi dalle code e delega per ciascuna richiesta un thread per svolgerla in maniera autonoma.

Per ogni API che sarà descritta qui di seguito si inizierà con una analisi tecnica focalizzandosi sulla progettazione, infine si passerà ad una analisi dedicata all' implementazione.

`GetUserInfo`

E' la classe che restituisce le informazioni relative ad un utente dato il suo ID.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente due chiavi:

- **token:** alla quale viene associata la stringa del token
- **param:** alla quale vengono associati i parametri di input che vengono usati dalla classe in questione per poter effettuare le dovute elaborazioni. In questo caso come unico parametro abbiamo l' ID dell' utente (UserID) sottoforma di stringa

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà a verificare il token, a seguito di questa azione gli scenari possibili sono due: la verifica avviene correttamente oppure no.
Nel primo caso, il gestore una volta aver verificato la correttezza e la validità del token, provvederà a contattare il proprio DB, chiedendo a questo di restituire tutti gli attributi associati allo specifico utente. Anche in questo caso gli scenari sono due:

1. Vengono restituite le informazioni richieste: il gestore provvede quindi ad "impacchettarle" in un messaggio JSON pronte ad essere spedite al microservizio client
2. La risorsa non c'è o vi è stato qualche problema durante la comunicazione con il DB: in questo caso il gestore restituirà una stringa "`Resource error`" per notificare al client che vi è stato un errore e non può essere accontentato

Nel caso in cui invece il token non dovesse essere verificato correttamente, il gestore ovviamente non provvederà ad eseguire alcun tipo di azione, e notificherà il client con un messaggio di errore "`Token error`".

Output

Il messaggio di output consiste di una stringa JSON formata dai seguenti parametri:

- **nome:** contiene il nome dell' utente in formato `String`
- **cognome:** contiene il cognome dell' utente in formato `String`
- **email:** contiene l' email dell' utente in formato `String`
- **dOB:** acronimo di Date of Birthday, contiene l' anno di nascita dell' utente in `String` nel formato *yyyy-mm-dd*
- **ID:** contiene l' ID dell' utente in formato `String`
- **username:** contiene l' username dell' utente in formato `String`

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_get_user_info`.
Come annunciato nei paragrafi precedenti non sarà lui a farsi carico direttamente di risolvere la richiesta ma delegherà una classe specifica (`GetUserInfo.java`), la quale sostanzialmente è un thread indipendente.

```
/**************************************
 * CALLBACK GET_USER_INFO
 *************************************/
DeliverCallback callbackGetUserInfo = (consumerTag, delivery) -> {

    GetUserInfo gui = new GetUserInfo(delivery, channel);
    gui.run();
};
```

Il costruttore di questa classe riceverà come parametri di input la `delivery` e il `channel`, necessari per leggere il messaggio dalla coda e poi spedire la risposta al client.
Da notare il metodo `.run()` tipico della classe Thread.
Ovviamente il backend non rimarrà in attesa del completamento di questo, ma potrà procedere ad analizzare e servire altre richieste pendenti.

```
public class GetUserInfo extends Thread
```

Nel momento in cui viene eseguito il metodo `.run()` e quindi in thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Se questa ha successo provvede a contattare il DB locale

```java
@Override
public void run() {
    AMQP.BasicProperties replyProps = new AMQP.BasicProperties.Builder()
            .correlationId(delivery.getProperties().getCorrelationId()).build();

    String response = ""; // stringa di risposta da inviare indietro

    try {
        // Leggo la richiesta
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println("Operazione richiesta: getUserInfo");
        // elaboro message
        JSONObject obj = new JSONObject(message);
        String token = obj.getString("token");
        String userID = obj.getString("param");
        // Verificare il token
        PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
        TokenVerifier tokenVer = new TokenVerifier(key);
        if (tokenVer.verify(token)) {
            // Se la verifica va a termine, contatto il DB
            Connection conn = null;
            Statement stmt = null;
            Class.forName(JDBC_DRIVER);
            System.out.println("Connessione al database...");
            conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
            System.out.println("Connessione eseguita con successo!!!");
            stmt = conn.createStatement();
            System.out.println(userID);
            String sql = "SELECT * FROM UTENTE WHERE ID = " + "\'" + userID + "\'";
            ResultSet rs = stmt.executeQuery(sql);
            JSONObject messageJSON = null;
```

Come avviene la verifica del token?
Per verificare il token vengono usate due classi la **`PublicKeyRetriever`** e la **`TokenVerifier`**.

```java
public class PublicKeyRetriever {

    private final String reqUrl;
    private final RestTemplate restTemplate;

    public PublicKeyRetriever(String host, String realm, int port) {
        reqUrl = "http://" + host + ":" + port + "/auth/realms/" + realm;
        restTemplate = new RestTemplateBuilder().build();
    }

    //Formato JSON
    public String getPlainJson() {
        return restTemplate.getForObject(reqUrl, String.class);
    }

    //Recupero chiave pubblica del server in formato stringa
    public String getPublicKey() throws CommunicationException {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            JsonNode obj = objectMapper.readTree(getPlainJson());
            return obj.get("public_key").asText();
        } catch (JsonProcessingException e) {
            throw new CommunicationException();
        }
    }
}
```

La prima classe ha il compito, come suggerisce il nome, di recuperare la chiave pubblica dall' auth server per poter verificare la firma del token (che ricordo essere stato firmato con la chiave privata dell' auth server se è autentico).
Il metodo `getPlainJson()` restituisce il body della risposta HTTP come stringa in formato JSON, mentre il metodo `getPublicKey()` parsifica il JSON restituito dal metodo precedente, per ottenere la chiave pubblica sottoforma di stringa.

```
public class TokenVerifier {

    private String realmPublicKey;

    //Costruttore
    public TokenVerifier(PublicKeyRetriever pubRetr) {
        try {
            realmPublicKey = pubRetr.getPublicKey();
        } catch (CommunicationException e) {
            realmPublicKey = null;
        }
    }

    public TokenVerifier() {
        realmPublicKey = null;
    }

    //Metodo per verificare il token
    public boolean verify(String jwt) {
        if(realmPublicKey == null) return false;
        try {
            Jwts.parser()
                    .setSigningKey(getKey(realmPublicKey))
                    .parseClaimsJws(jwt);
            return true;
        } catch (ExpiredJwtException | MalformedJwtException | SignatureException |
                UnsupportedJwtException | IllegalArgumentException e) {
            System.out.println("Token error");
        }
        return false;
    }

    //Metodo che recupera ID dell' utente dal token
    public String getUserId(String jwt) {
        if(realmPublicKey == null) return null;
        Claims claims = Jwts.parser()
                .setSigningKey(getKey(realmPublicKey))
                .parseClaimsJws(jwt).getBody();
        return claims.getSubject();
    }
  //Restituisce chiave pubblica in formato PublicKey, usato poi dal parser
  public static PublicKey getKey(String key){
      try{
          byte[] byteKey = Base64.getDecoder().decode(key.getBytes());
          X509EncodedKeySpec X509publicKey = new X509EncodedKeySpec(byteKey);
          KeyFactory kf = KeyFactory.getInstance("RSA");

          return kf.generatePublic(X509publicKey);
      }
      catch(Exception e){
          e.printStackTrace();
      }

      return null;
  }
```

La classe `TokenVerifier` contiene un' unica variaibile che è `realmPublicKey`, questa variabile conterrà la chiave pubblica dell' auth server prelevata con il metodo descritto precedentemente.
Il metodo principale di questa classe è il metodo `getKey()`, che data la chiave pubblica del server sottoforma di stringa, ha il compito di convertirla in un oggetto `PublicKey`, che verrà poi usato dagli altri metodi per verificare la firma del token.
Il metodo verify è un booleano, di conseguenza restituirà *true* se la verifica è andata a buon fine, nel caso in cui ci siano problemi restituirà *false*.

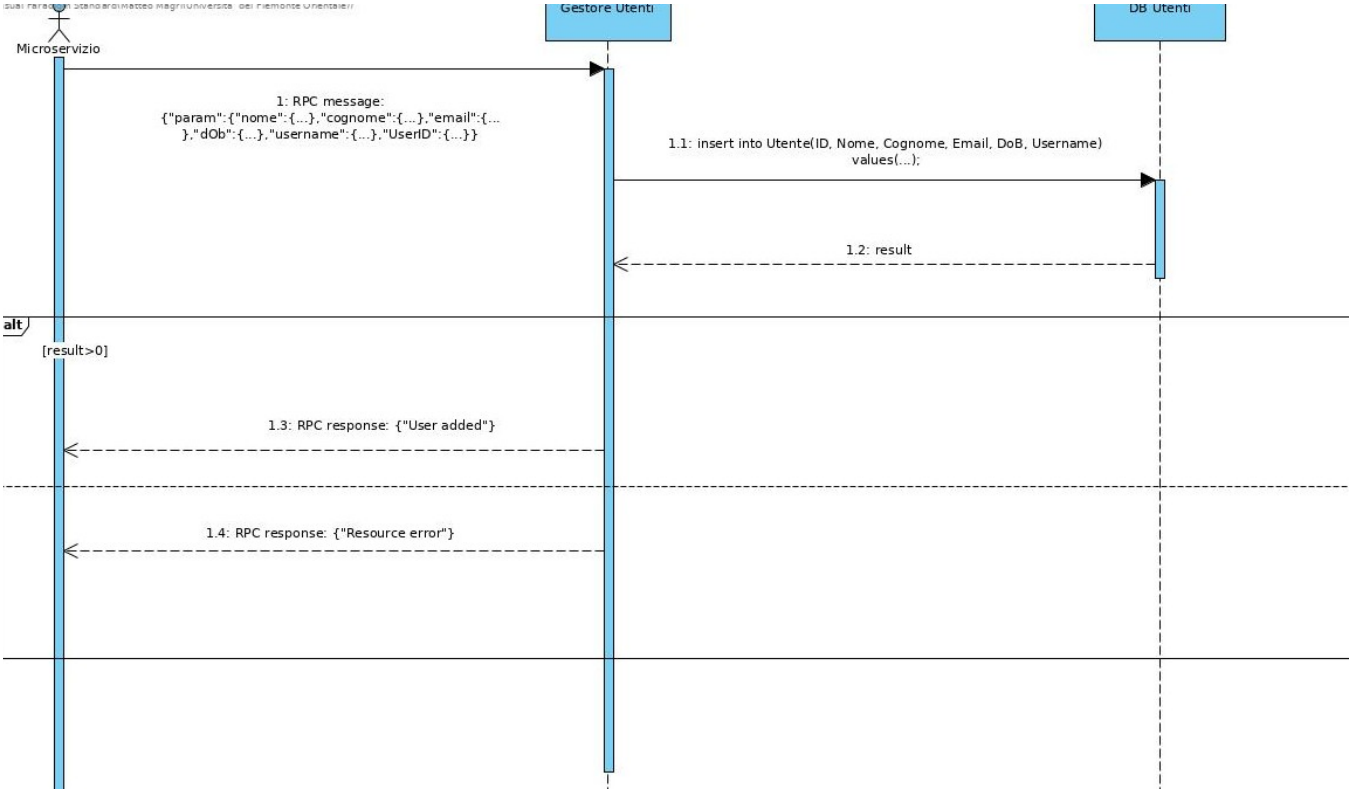
Una volta che la verifica del token è andata a termine, il thread contatterà il DB ed eseguirà la query SQL necessaria per il recupero dei dati, questi verranno salvati in un recordo `ResultSet` che dovrà poi essere parsificato come mostrato dall' immagine seguente.

```
            // Estrazione dei dati
            while (rs.next()) {
                String id = rs.getString("id");
                String nome = rs.getString("nome");
                String cognome = rs.getString("cognome");
                String email = rs.getString("email");
                String dob = rs.getString("dob");
                String username = rs.getString("username");
                // Costruzione oggetto JSON
                messageJSON = new JSONObject();
                messageJSON.put("nome", nome);
                messageJSON.put("cognome", cognome);
                messageJSON.put("email", email);
                messageJSON.put("dOb", dob);
                messageJSON.put("ID", id);
                messageJSON.put("username", username);
            }
            // Trasformazione in stringa dell' oggetto
            response = messageJSON.toString();
            // Chiudo la connessione
            rs.close();
        } else {
            // Altrimenti restituisco un token error
            response = "Token error";
        }
    } catch (RuntimeException | JSONException e) {
        response = "Resource error";
    } catch (ClassNotFoundException | UnsupportedEncodingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
```

I parametri vengono prelevati e salvati in opportune variabili, successivamente viene costruito il messaggio di risposta rispettando il formato descritto nei diagrammi di sequenza.
Nel caso in cui la risorsa non sia presente o il formato JSON non sia stato "impachettato" correttamente, verrà restituita come preannunciato la stringa "Resource error".
Stesso discorso vale se la verifica del token dovesse non andare a buon fine, in questo caso verrà restituita la stringa "Token error".

`AddUser`

E' la classe che dati i valore degli attributi di un utente provvede a memorizzarlo all' interno del database.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente la seguente chiave:

- **param:** questa a sua volta conterrà all' interno un altro JSON Object, contenente tutti i parametri necessari:
  - nome
  - cognome
  - email
  - username
  - UserID

Aventi lo stesso significato di quelli descritti precedentemente.

Funzionamento

Il funzionamento è analogo per quanto avveniva nella precedente funzione e per quanto avverrà anche nelle altre (dato che il backend deve svolgere più o meno sempre le stesse funzioni).
Una volta ricevuta la chiamata, il gestore provvederà ad aggiungere il nuovo utente, chiedendo al proprio DB di memorizzarlo. Anche in questo caso gli scenari sono due:

1. L' aggiunta va a termine
2. Si verificano degli errori, in questo caso il gestore restituirà una stringa "`Resource error`" per notificare al client che vi è stato un errore e non può essere accontentato

Output

Il backend restituirà "User added" nel caso in cui l' aggiunta vada a buon fine, "Resource error" altrimenti.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome **`rpc_add_user`**.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (**`AddUser.java`**), la quale sostanzialmente è un thread indipendente (così come per ogni classe).

```
/**************************************
 * CALLBACK ADDUSER
 **************************************/
DeliverCallback callbackAddUser = (consumerTag, delivery) -> {
    AddUser au = new AddUser(delivery, channel);
    au.run();
};
```

Nel momento in cui viene eseguito il metodo **`.run()`** e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

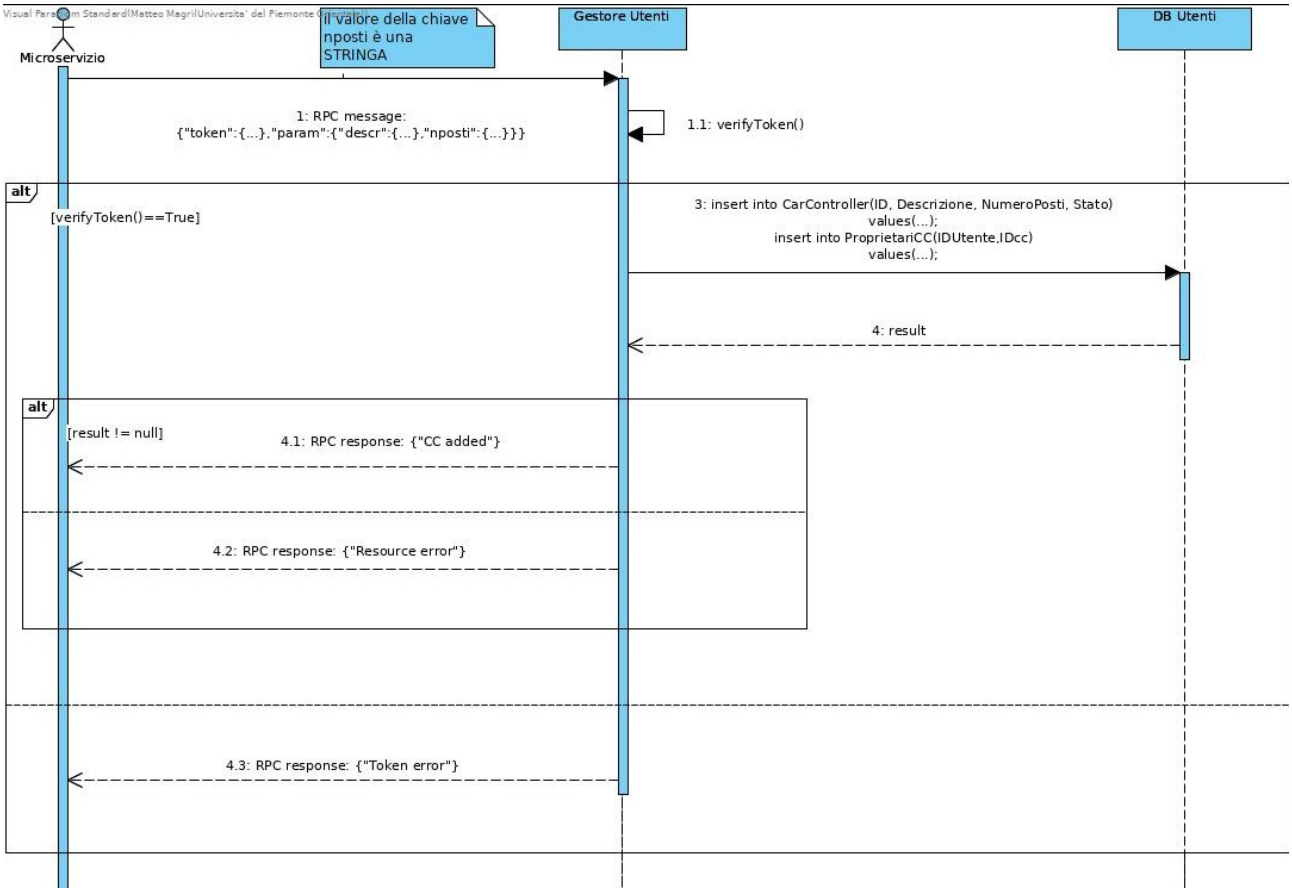
- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Effettua l' inserimento dell' utente all' interno del DB
- Se questo ha successo restituisce "User added", "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: addUser");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    JSONObject param = obj.getJSONObject("param");
    String userID = param.getString("UserID");
    String nome = param.getString("nome");
    String cognome = param.getString("cognome");
    String email = param.getString("email");
    String dob = param.getString("dOb");
    String username = param.getString("username");
    Connection conn = null;
    Statement stmt = null;
    Class.forName(JDBC_DRIVER);
    System.out.println("Connessione al database...");
    conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
    System.out.println("Connessione eseguita con successo!!!");
    stmt = conn.createStatement();
    System.out.println(userID);
    String sql = "INSERT INTO UTENTE (ID, Nome, Cognome, Email, DoB, Username) VALUES (\'" + userID + "\',"
            + "\'" + nome + "\'," + "\'" + cognome + "\'," + "\'" + email + "\'," + "\'" + dob + "\'," + "\'"
            + username + "\')";
    int rs = stmt.executeUpdate(sql);
    stmt.close();
    conn.close();
    if (rs > 0) {
        response = "User added";
    } else {
        response = "Resource error";
    }

} catch (RuntimeException | JSONException e) {
    response = "Resource error";
} catch (ClassNotFoundException | UnsupportedEncodingException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
```

AddCC

E' la classe che dati i valore degli attributi di un car controller provvede a memorizzarlo all' interno del database ed associarlo all' utente che ne ha fatto richiesta.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** questa a sua volta conterrà all' interno un altro JSON Object, contenente tutti i parametri necessari:
  - descr: campo che contiene la descrizione del car controller
  - nposti: campo che contiente il numero di posti associati al car controller

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token, dove gli scenari sono sempre i soliti due.
In caso non ci siano problemi, provvede ad aggiungere il nuovo car controller ed associarlo all' utente interessato.

Output

Il backend restituirà "User added" nel caso in cui l' aggiunta vada a buon fine, "Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_add_cc`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`AddCC.java`).

```
/**************************************
 * CALLBACK ADDUSER
 *************************************/
DeliverCallback callbackAddUser = (consumerTag, delivery) -> {
    AddUser au = new AddUser(delivery, channel);
    au.run();
};
```

Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

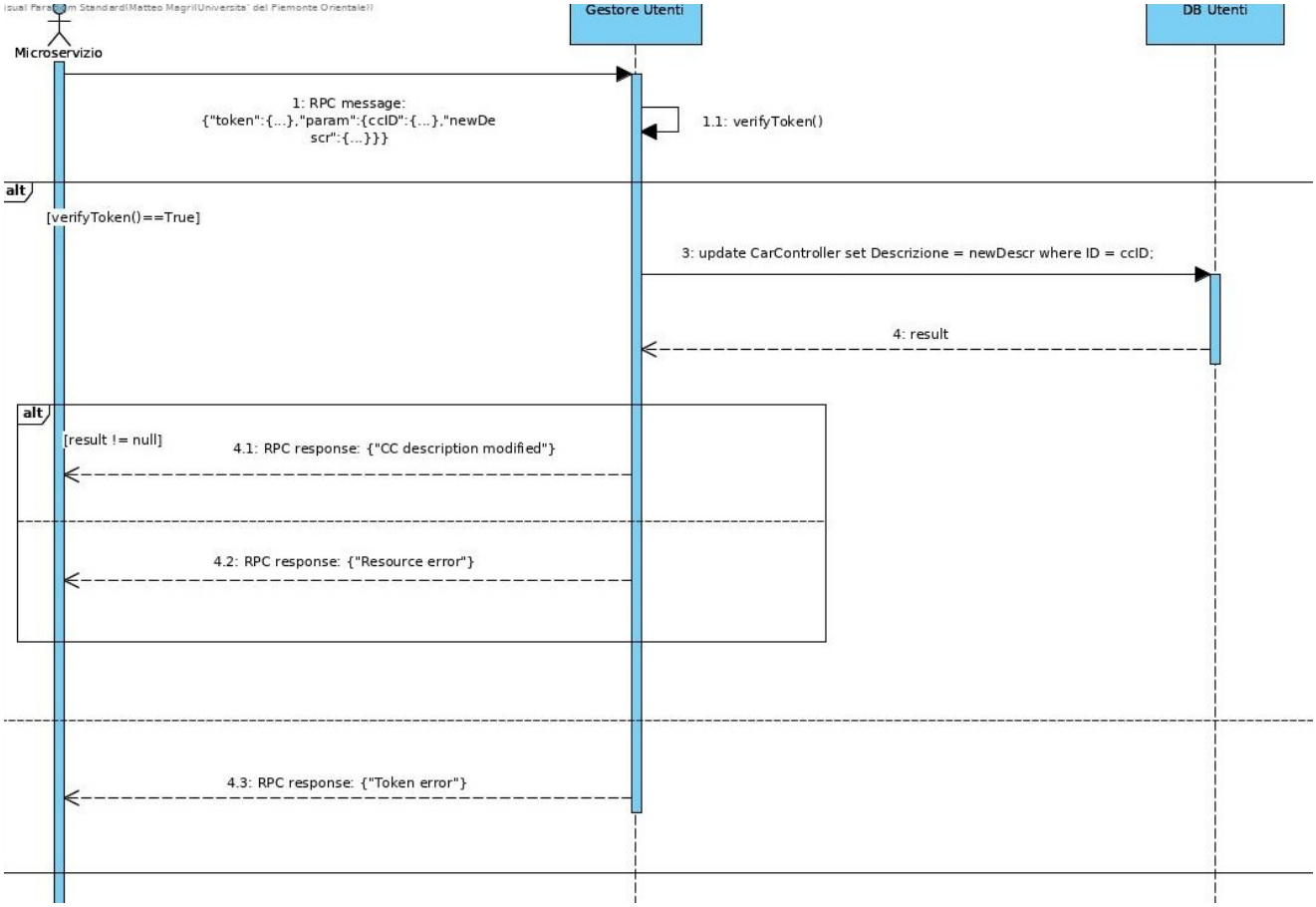
- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Effettua l' inserimento del car controller all' interno del DB e lo associa all' utente
- Se questo ha successo restituisce "CC added", "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: addCC");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    JSONObject param = obj.getJSONObject("param");
    String descr = param.getString("descr");
    String nposti = param.getString("nposti");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    String userId = tokenVer.getUserId(token);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String ccID = UUID.randomUUID().toString();
        String sql = "INSERT INTO CARCONTROLLER (ID, Descrizione, NumeroPosti, Stato) VALUES (\'" + ccID + "\',"
                + "\'" + descr + "\'," + "\'" + nposti + "\'," + "\'" + "ON" + "\')";
        int rs = stmt.executeUpdate(sql);
        if (rs > 0) {
            sql = "INSERT INTO PROPRIETARICC (IDUtente, IDcc) VALUES (\'" + userId + "\'," + "\'" + ccID
                    + "\')";
            rs = stmt.executeUpdate(sql);
            if (rs > 0) {
                response = "CC added";
            } else {
                response = "Resource error";
            }
        } else
            response = "Resource error";

    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
```

AlterDescription

E' la classe che permette di modificare la descrizione di un car controller presente all' interno del database.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** questa a sua volta conterrà all' interno un altro JSON Object, contenente tutti i parametri necessari:
  - newDescr: campo che contiene la nuova descrizione del car controller
  - ccID: campo che contiente l' ID del car controller da modificare

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a modificare la descrizione del car controller.

Output

Il backend restituirà "CC description modified" nel caso in cui la modifica vada a buon fine, "Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_alter_descr`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`AlterDescription.java`).

```
/**************************************
 * CALLBACK ALTER_DESCRIPTION
 **************************************/
DeliverCallback callbackAlterDescr = (consumerTag, delivery) -> {
    AlterDescription ad = new AlterDescription(delivery, channel);
    ad.run();
};
```

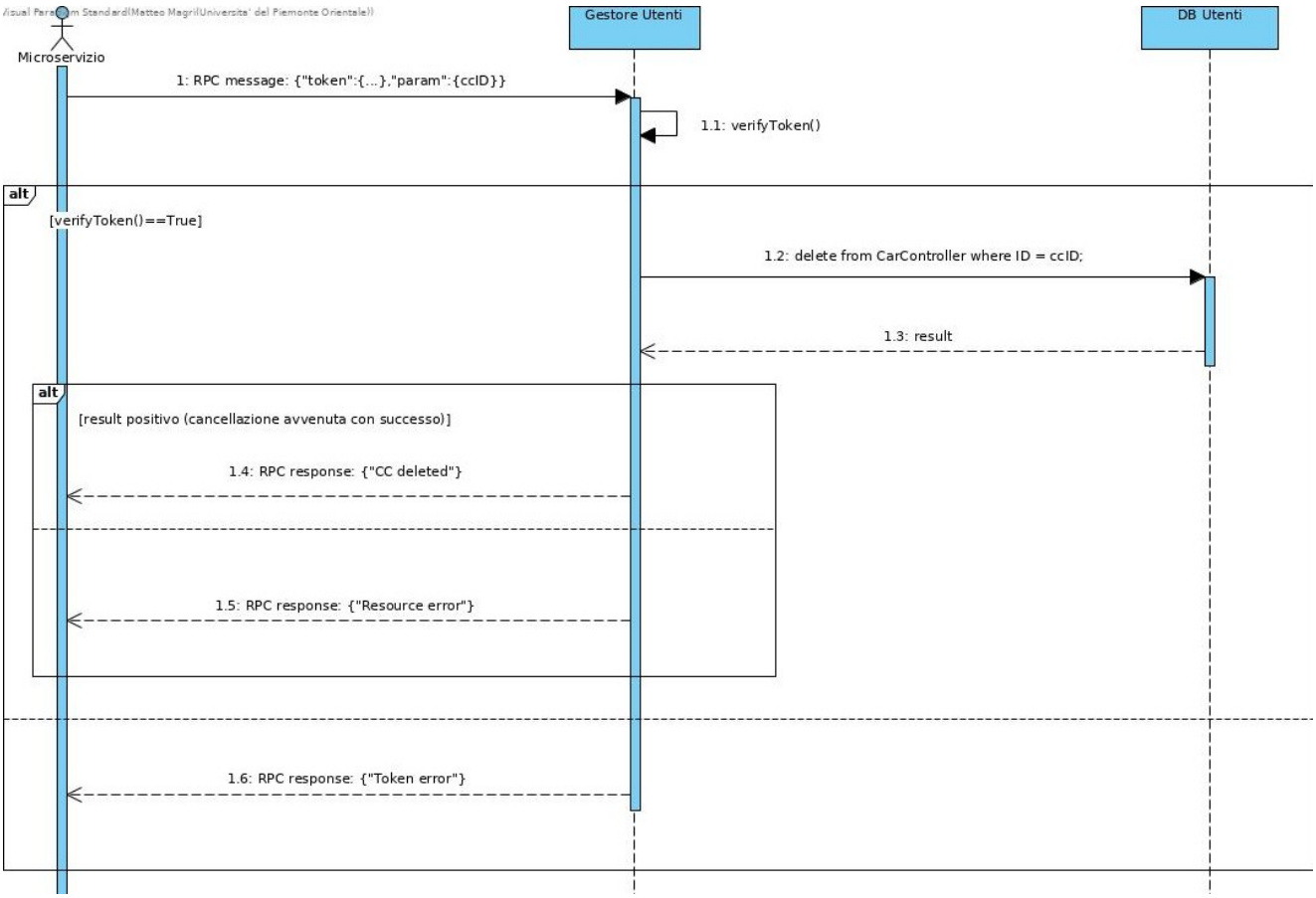
Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Effettua la modifica della descrizione del car controller all' interno del DB
- Se questo ha successo restituisce "CC description modified", "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message;
    message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: alterDescription");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    JSONObject param = obj.getJSONObject("param");
    String newDescr = param.getString("newDescr");
    String ccID = param.getString("ccID");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "UPDATE CarController SET Descrizione = " + "\'" + newDescr + "\'" + " WHERE ID = " + "\'"
                + ccID + "\'";
        int rs = stmt.executeUpdate(sql);
        if (rs > 0) {
            response = "CC description modified";
        } else {
            response = "Resource error";
        }
    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
} catch (ClassNotFoundException | UnsupportedEncodingException e) {
    e.printStackTrace();
```

DeleteCC

E' la classe che permette di eliminare un car controller dal DB, provvedendo anche a disassociarlo dal corrispondente proprietario.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' ID del car controller da eliminare

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede ad eliminare il CC ed a disassociarlo dall' utente.

Output

Il backend restituirà "CC deleted" nel caso in cui la modifica vada a buon fine, "Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_delete_cc`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`DeleteCC.java`).

```
/**************************************
 * CALLBACK DELETECC
 **************************************/
DeliverCallback callbackDeleteCC = (consumerTag, delivery) -> {
    DeleteCC dcc = new DeleteCC(delivery, channel);
    dcc.run();
};
```

Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

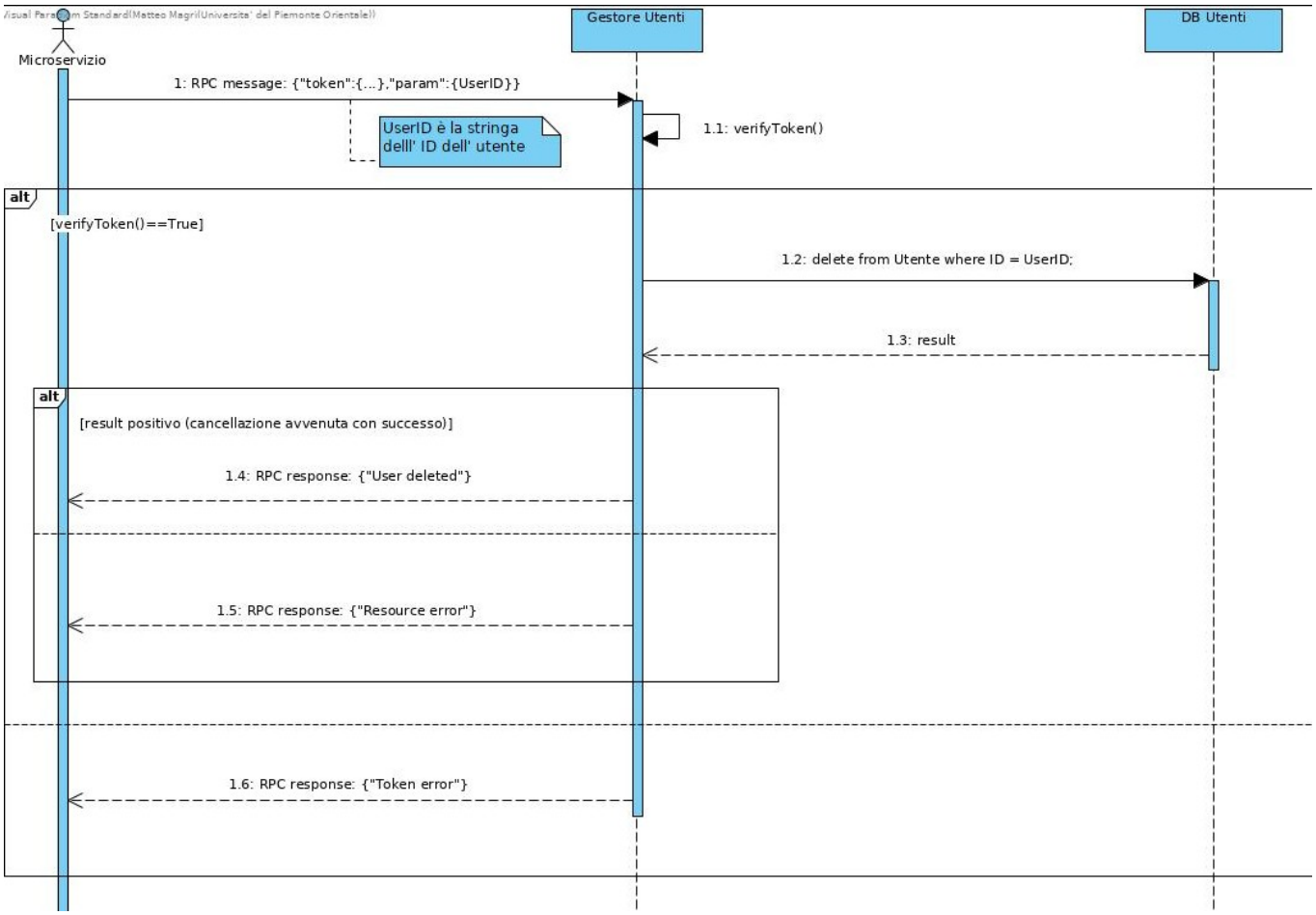
- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Effettua la cancellazione
- Se questa ha successo restituisce "CC deleted", "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: deleteCC");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    String ccID = obj.getString("param");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "DELETE FROM CarController WHERE ID = " + "\'" + ccID + "\'";
        int rs = stmt.executeUpdate(sql);
        if (rs > 0) {
            response = "CC deleted";
        } else {
            response = "Resource error";
        }

    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
} catch (ClassNotFoundException | UnsupportedEncodingException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
```

DeleteUser

E' la classe che permette di eliminare un utente dal DB.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' ID dell' utente

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede ad eliminare l'utente.

Output

Il backend restituirà "User deleted" nel caso in cui la modifica vada a buon fine, "Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_delete_user`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`DeleteUser.java`).

```
/**************************************
 * CALLBACK DELETEUSER
 **************************************/
DeliverCallback callbackDeleteUser = (consumerTag, delivery) -> {
    DeleteUser du = new DeleteUser(delivery, channel);
    du.run();
};
```

Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

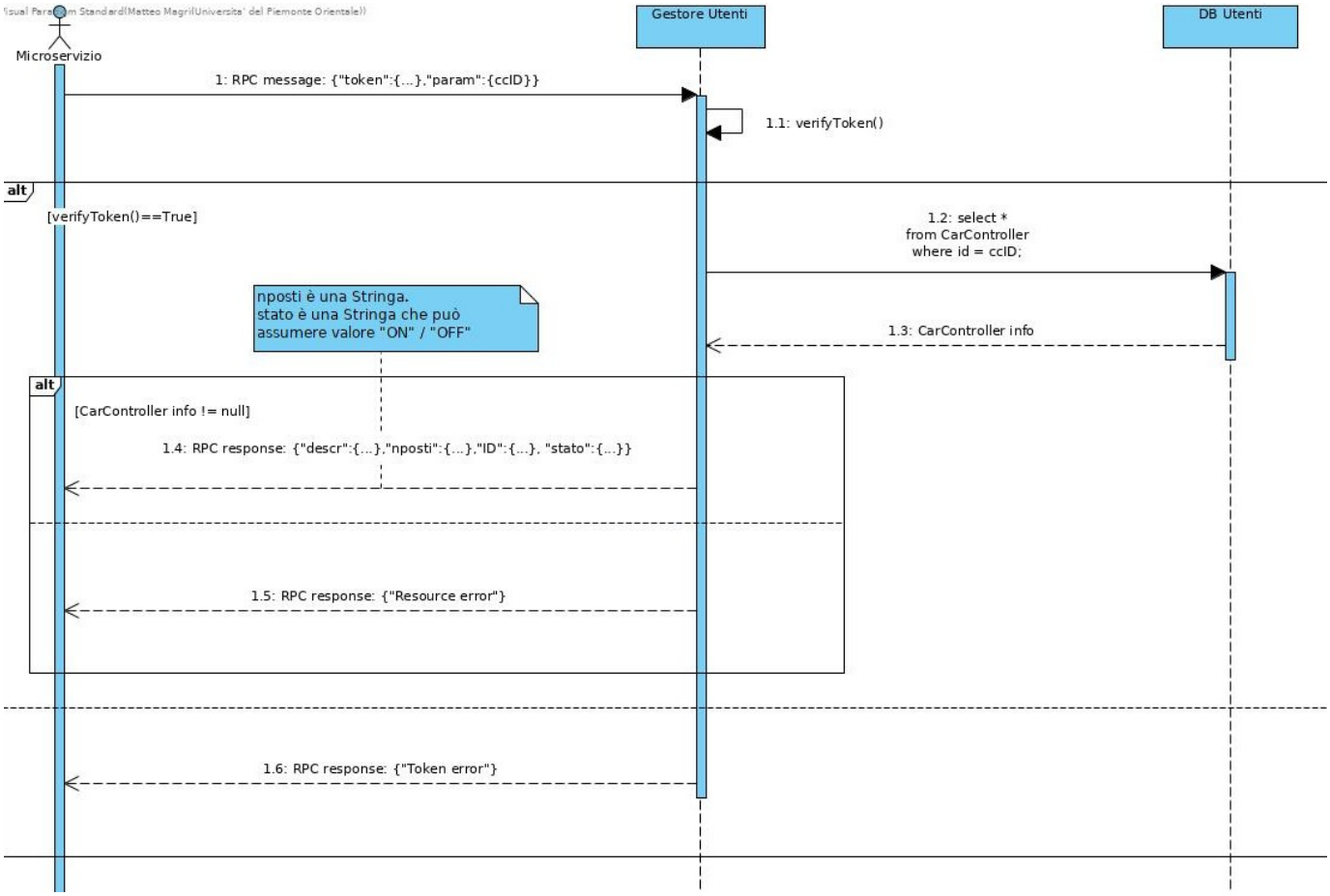
- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Effettua la cancellazione
- Se questa ha successo restituisce "User deleted", "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: deleteUser");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    String userID = obj.getString("param");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "DELETE FROM Utente WHERE ID = " + "\'" + userID + "\'";
        int rs = stmt.executeUpdate(sql);
        if (rs > 0) {
            response = "User deleted";
        } else {
            response = "Resource error";
        }

    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
} catch (ClassNotFoundException | UnsupportedEncodingException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
```

`GetCCInfo`

E' la classe che restituisce le informazioni relative ad un car controller (CC).



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' ID del car controller

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a recuperare le informazioni del car controller.

Output

Il messaggio di output consiste di una stringa JSON formata dai seguenti parametri:

- **descr:** contiene la descrizione del car controller in formato `String`
- **nposti:** contiene il numero di posti in formato `String`
- **ID:** contiene l' ID del car controller in formato `String`
- **stato:** contiene lo stato del controller in `String`, che può essere "ON" oppure "OFF", dove ON indica che è abilitato a mandare le posizioni e registrare il percorso, OFF che è disabilitato

"Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_get_cc_info`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`GetCCInfo.java`).

```
/**************************************
 * CALLBACK GET_CC_INFO
 *************************************/
DeliverCallback callbackGetCCInfo = (consumerTag, delivery) -> {
    GetCCInfo gcci = new GetCCInfo(delivery, channel);
    gcci.run();
};
```

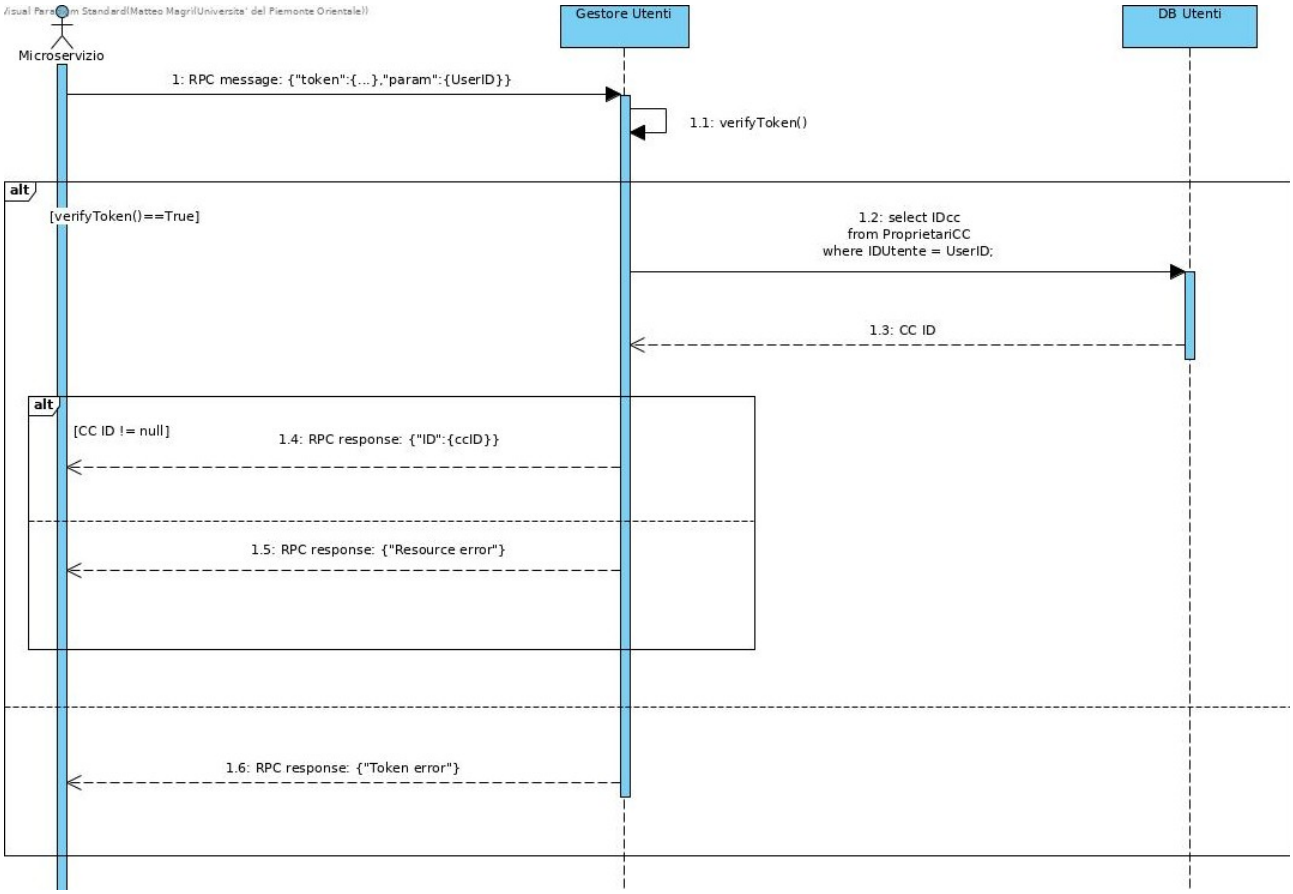
Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Contatta il DB per recuperare i dati associati al car controller
- Se questa ha successo restituisce la stringa JSON descritta precedentemente, "Resource error" altrimenti

```
        String token = obj.getString("token");
        String ccID = obj.getString("param");
        // Verificare il token
        PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
        TokenVerifier tokenVer = new TokenVerifier(key);
        if (tokenVer.verify(token)) {
            // Se la verifica va a termine, contatto il DB
            Connection conn = null;
            Statement stmt = null;
            Class.forName(JDBC_DRIVER);
            System.out.println("Connessione al database...");
            conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
            System.out.println("Connessione eseguita con successo!!!");
            stmt = conn.createStatement();
            String sql = "SELECT * FROM CarController WHERE ID = " + "\'" + ccID + "\'";
            ResultSet rs = stmt.executeQuery(sql);
            JSONObject messageJSON = null;
            // Estrazione dei dati
            while (rs.next()) {
                // Retrieve by column name
                String id = rs.getString("id");
                String descrizione = rs.getString("descrizione");
                String nposti = rs.getString("numeroposti");
                String stato = rs.getString("stato");
                // Costruzione oggetto JSON
                messageJSON = new JSONObject();
                messageJSON.put("ID", id);
                messageJSON.put("descr", descrizione);
                messageJSON.put("nposti", nposti);
                messageJSON.put("stato", stato);
            }
            // Trasformazione in stringa dell' oggetto
            response = messageJSON.toString();
            // Chiudo la connessione
            rs.close();
        } else {
            // Altrimenti restituisco un token error
            response = "Token error";
        }
    } catch (RuntimeException | JSONException e) {
        response = "Resource error";
    } catch (ClassNotFoundException | UnsupportedEncodingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
```

`GetCCID`

E' la classe che restituisce unicamente l' ID di un car controller dato l'ID di un utente.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' ID dell' utente

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a recuperare l' ID del car controller.

Output

Il messaggio di output consiste di una stringa JSON formata dal seguente parametro:

- **ID:** contiene l' ID del car controller in formato `String`

"Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_get_cc_id`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`GetCCID.java`).

```
/**************************************
 * CALLBACK GET_CC_ID
 **************************************/
DeliverCallback callbackGetCCID = (consumerTag, delivery) -> {
    GetCCID gccid = new GetCCID(delivery, channel);
    gccid.run();
};
```

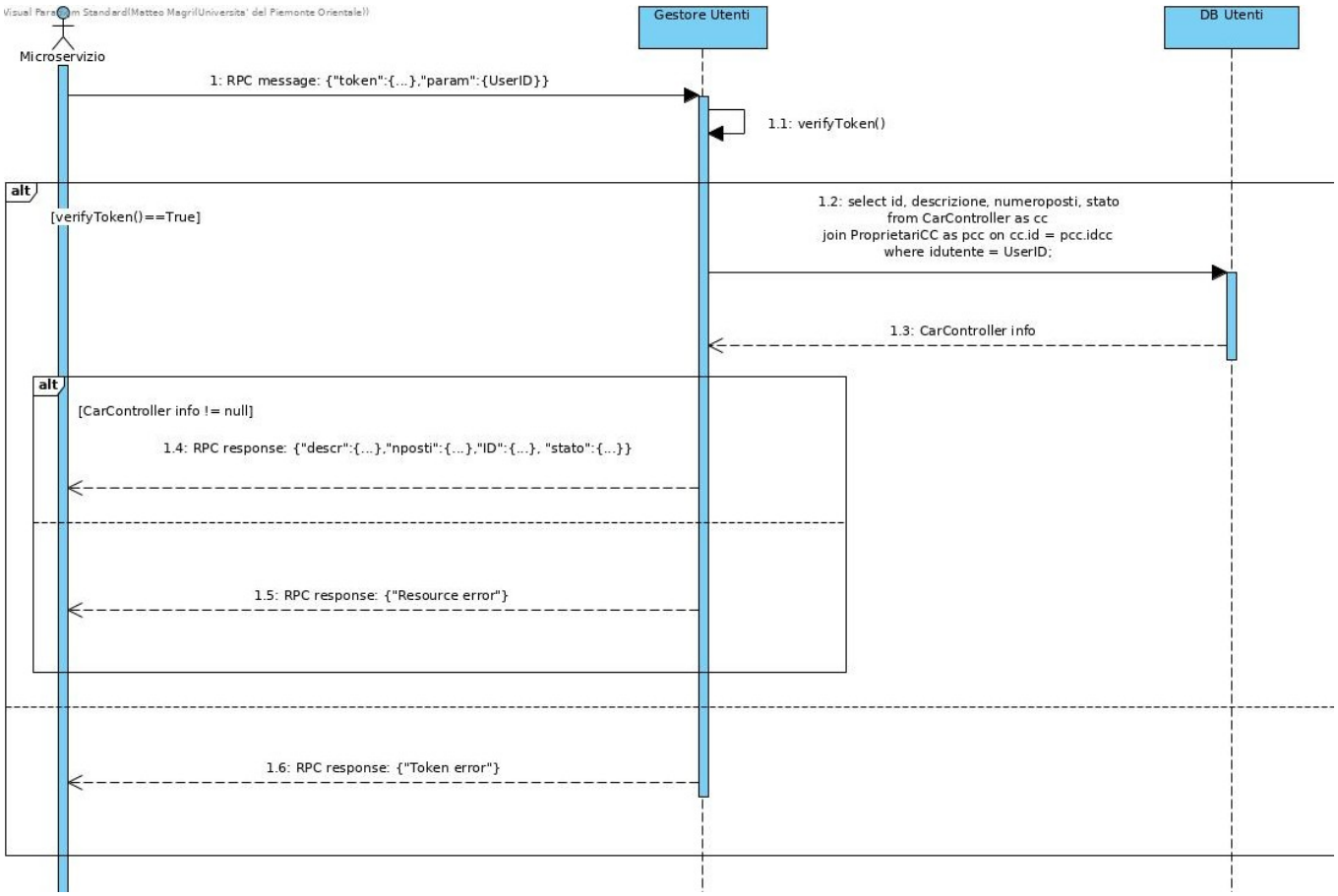
Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Contatta il DB per recuperare l' ID del car controller
- Se questa ha successo restituisce la stringa JSON descritta precedentemente, "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: getCCID");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    String userID = obj.getString("param");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "SELECT IDcc FROM ProprietariCC WHERE IDUtente = " + "\'" + userID + "\'";
        ResultSet rs = stmt.executeQuery(sql);
        JSONObject messageJSON = null;
        // Estrazione dei dati
        while (rs.next()) {
            // Retrieve by column name
            String id = rs.getString("idcc");
            // Costruzione oggetto JSON
            messageJSON = new JSONObject();
            messageJSON.put("ID", id);
        }
        // Trasformazione in stringa dell' oggetto
        response = messageJSON.toString();
        // Chiudo la connessione
        rs.close();
    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
```

`GetCCInfoUser`

E' la classe che restituisce le informazioni di un car controller associato ad uno specifico utente.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' ID dell' utente

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a recuperare le informazioni del car controller associato.

Output

Il messaggio di output consiste di una stringa JSON formata dai seguenti parametri:

- **ID:** contiene l' ID del car controller in formato `String`
- **descr:** contenente la descrizione del car controller
- **nposti:** contenente il numero di posti associato al car controller
- **stato:** contenente lo stato corrente del car controller

"Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_get_cc_info_user`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`GetCCInfoUser.java`).

```
/**************************************
 * CALLBACK GET_CC_INFO_USER
 **************************************/
DeliverCallback callbackGetCCInfoUser = (consumerTag, delivery) -> {
    GetCCInfoUser gcciu = new GetCCInfoUser(delivery, channel);
    gcciu.run();
};
```

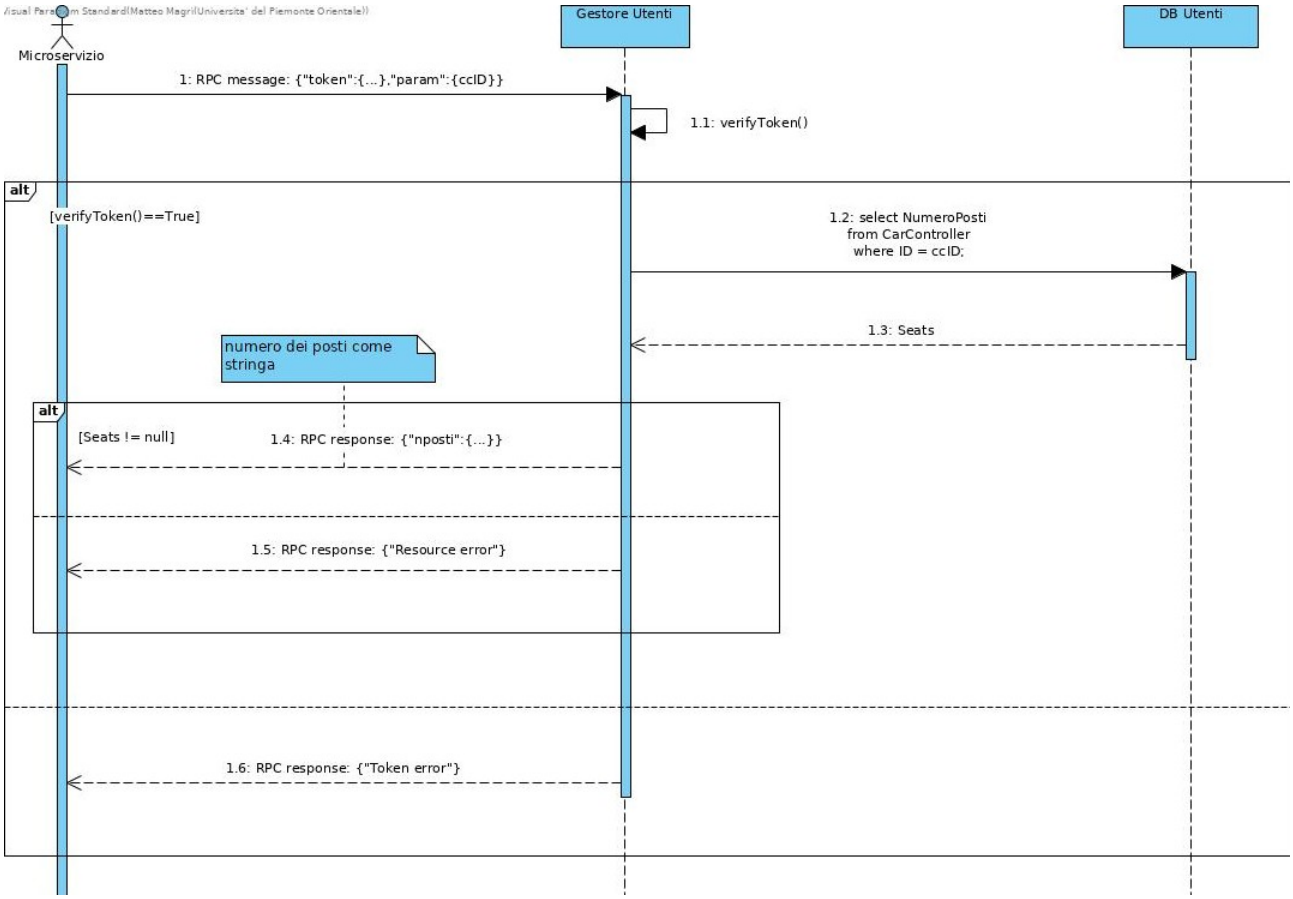
Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Contatta il DB per recuperare le informazioni del car controller
- Se questa ha successo restituisce la stringa JSON descritta precedentemente, "Resource error" altrimenti

```
// Leggo la richiesta
String message = new String(delivery.getBody(), "UTF-8");
System.out.println("Operazione richiesta: getCCInfoUser");
// elaboro message
JSONObject obj = new JSONObject(message);
String token = obj.getString("token");
String userID = obj.getString("param");
// Verificare il token
PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
TokenVerifier tokenVer = new TokenVerifier(key);
if (tokenVer.verify(token)) {
    // Se la verifica va a termine, contatto il DB
    Connection conn = null;
    Statement stmt = null;
    Class.forName(JDBC_DRIVER);
    System.out.println("Connessione al database...");
    conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
    System.out.println("Connessione eseguita con successo!!!");
    stmt = conn.createStatement();
    String sql = "SELECT id,descrizione,numeroposti,stato FROM CarController AS cc"
            + " JOIN  ProprietariCC AS pcc ON cc.id = pcc.idcc WHERE idutente = " + "\'" + userID + "\'";
    ResultSet rs = stmt.executeQuery(sql);
    JSONObject messageJSON = null;
    // Estrazione dei dati
    while (rs.next()) {
        // Retrieve by column name
        String id = rs.getString("id");
        String descrizione = rs.getString("descrizione");
        String nposti = rs.getString("numeroposti");
        String stato = rs.getString("stato");
        // Costruzione oggetto JSON
        messageJSON = new JSONObject();
        messageJSON.put("ID", id);
        messageJSON.put("descr", descrizione);
        messageJSON.put("nposti", nposti);
        messageJSON.put("stato", stato);
    }
    // Trasformazione in stringa dell' oggetto
    response = messageJSON.toString();
    // Chiudo la connessione
    rs.close();
} else {
    // Altrimenti restituisco un token error
    response = "Token error";
```

GetSeats

E' la classe che restituisce il numero di posti associato ad un car controller.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' ID del car controller

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a recuperare il numero dei posti del car controller.

Output

Il messaggio di output consiste di una stringa JSON formata unicamente dal seguente parametro:

- **nposti:** contenente il numero di posti associato al car controller

"Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_get_seats`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`GetSeats.java`).

```
/**************************************
 * CALLBACK GET_SEATS
 **************************************/
DeliverCallback callbackGetSeats = (consumerTag, delivery) -> {
    GetSeats gs = new GetSeats(delivery, channel);
    gs.run();
};
```

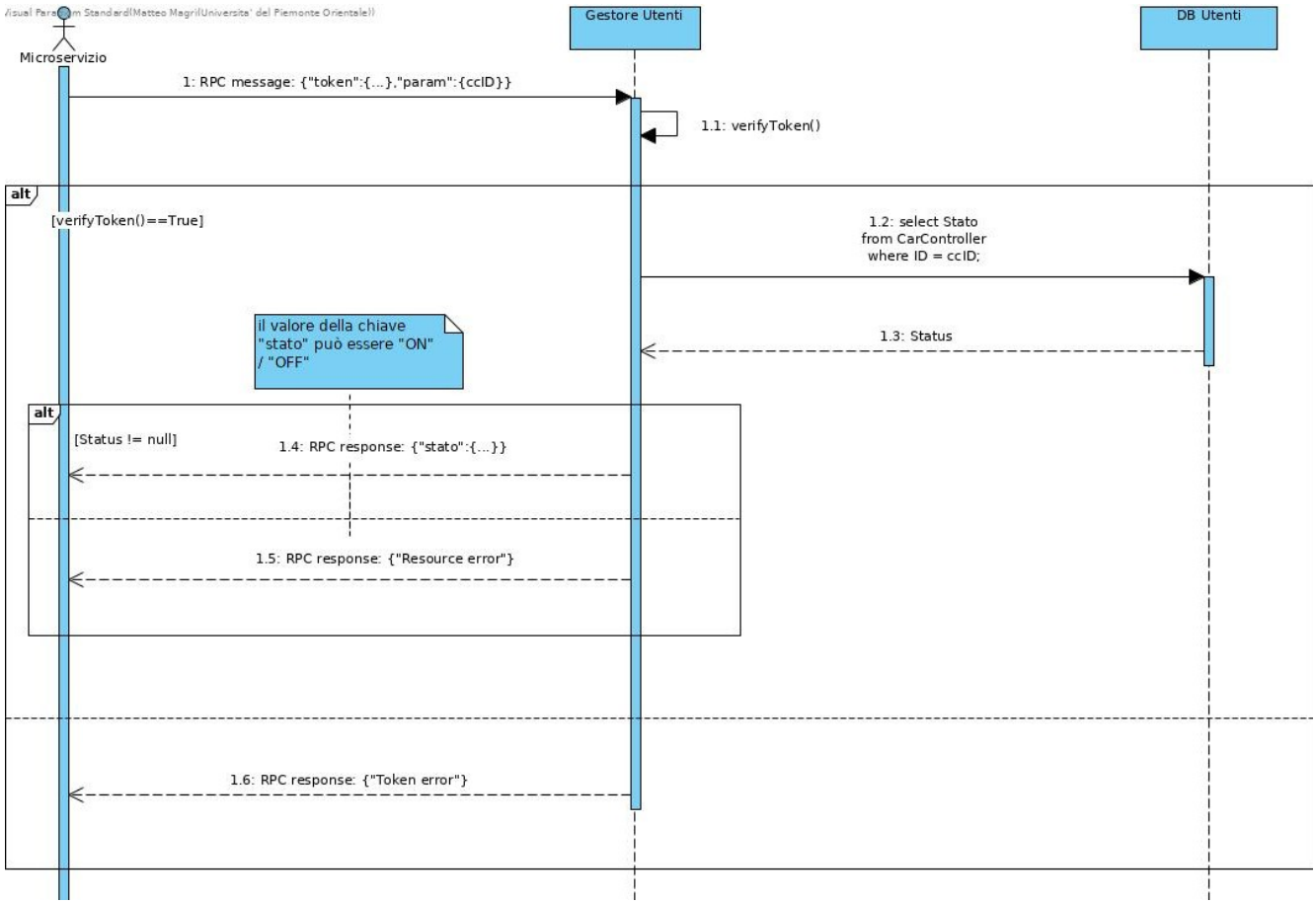
Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Contatta il DB per recuperare il numero di posti
- Se questa ha successo restituisce la stringa JSON descritta precedentemente, "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: getSeats");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    String ccID = obj.getString("param");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "SELECT numeroposti FROM CarController WHERE ID = " + "\'" + ccID + "\'";
        ResultSet rs = stmt.executeQuery(sql);
        JSONObject messageJSON = null;
        // Estrazione dei dati
        while (rs.next()) {
            // Retrieve by column name
            String nposti = rs.getString("numeroposti");
            // Costruzione oggetto JSON
            messageJSON = new JSONObject();
            messageJSON.put("nposti", nposti);
        }
        // Trasformazione in stringa dell' oggetto
        response = messageJSON.toString();
        // Chiudo la connessione
        rs.close();
    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
} catch (ClassNotFoundException | UnsupportedEncodingException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
```

GetStatus

E' la classe che restituisce lo stato corrente di un car controller (ON=invia posizioni, OFF=non invia posizioni).



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' ID del car controller

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a recuperare lo stato del car controller.

Output

Il messaggio di output consiste di una stringa JSON formata unicamente dal seguente parametro:

- **stato:** contenente lo stato corrente del car controller

"Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_get_status`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`GetStatus.java`).

```
/**************************************
 * CALLBACK GET_STATUS
 **************************************/
DeliverCallback callbackGetStatus = (consumerTag, delivery) -> {
    GetStatus gs = new GetStatus(delivery, channel);
    gs.run();
};
```

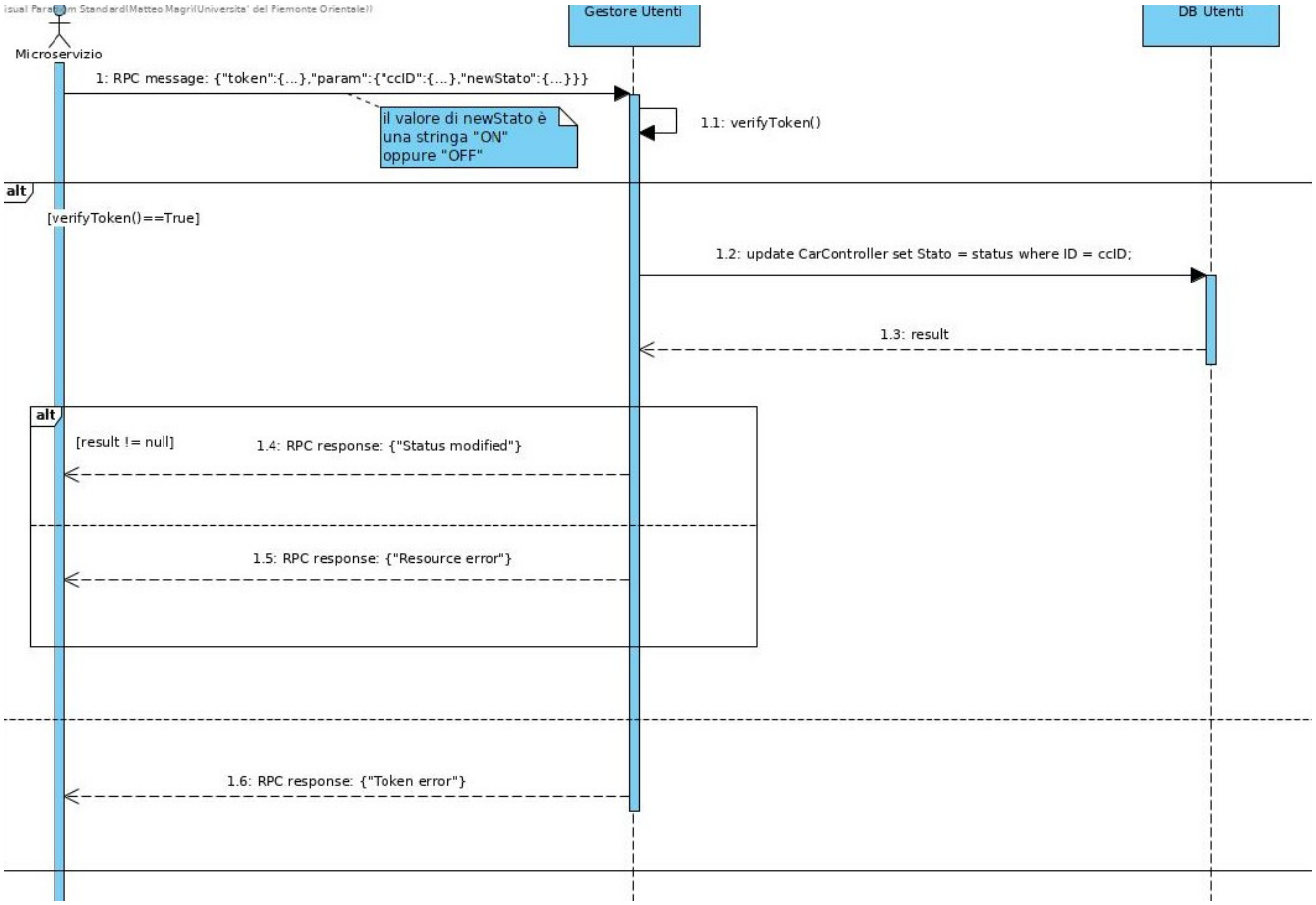
Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Contatta il DB per recuperare lo stato
- Se questa ha successo restituisce la stringa JSON descritta precedentemente, "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: getStatus");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    String ccID = obj.getString("param");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "SELECT stato FROM CarController WHERE ID = " + "\'" + ccID + "\'";
        ResultSet rs = stmt.executeQuery(sql);
        JSONObject messageJSON = null;
        // Estrazione dei dati
        while (rs.next()) {
            // Retrieve by column name
            String status = rs.getString("stato");
            // Costruzione oggetto JSON
            messageJSON = new JSONObject();
            messageJSON.put("stato", status);
        }
        // Trasformazione in stringa dell' oggetto
        response = messageJSON.toString();
        // Chiudo la connessione
        rs.close();
    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
```

SetStatus

E' la classe che permette di modificare lo stato corrente di un car controller.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente a sua volta due parametri:
  - **ccID:** ID del car controller
  - **newStato:** nuovo valore da impostare come stato al car controller

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a cambiare lo stato del car controller.

Output

Il backend restituirà "Status modified" nel caso in cui la modifica vada a buon fine, "Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_set_status`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`SetStatus.java`).

```
/**************************************
 * CALLBACK SET_STATUS
 **************************************/
DeliverCallback callbackSetStatus = (consumerTag, delivery) -> {
    SetStatus ss = new SetStatus(delivery, channel);
    ss.run();
};
```

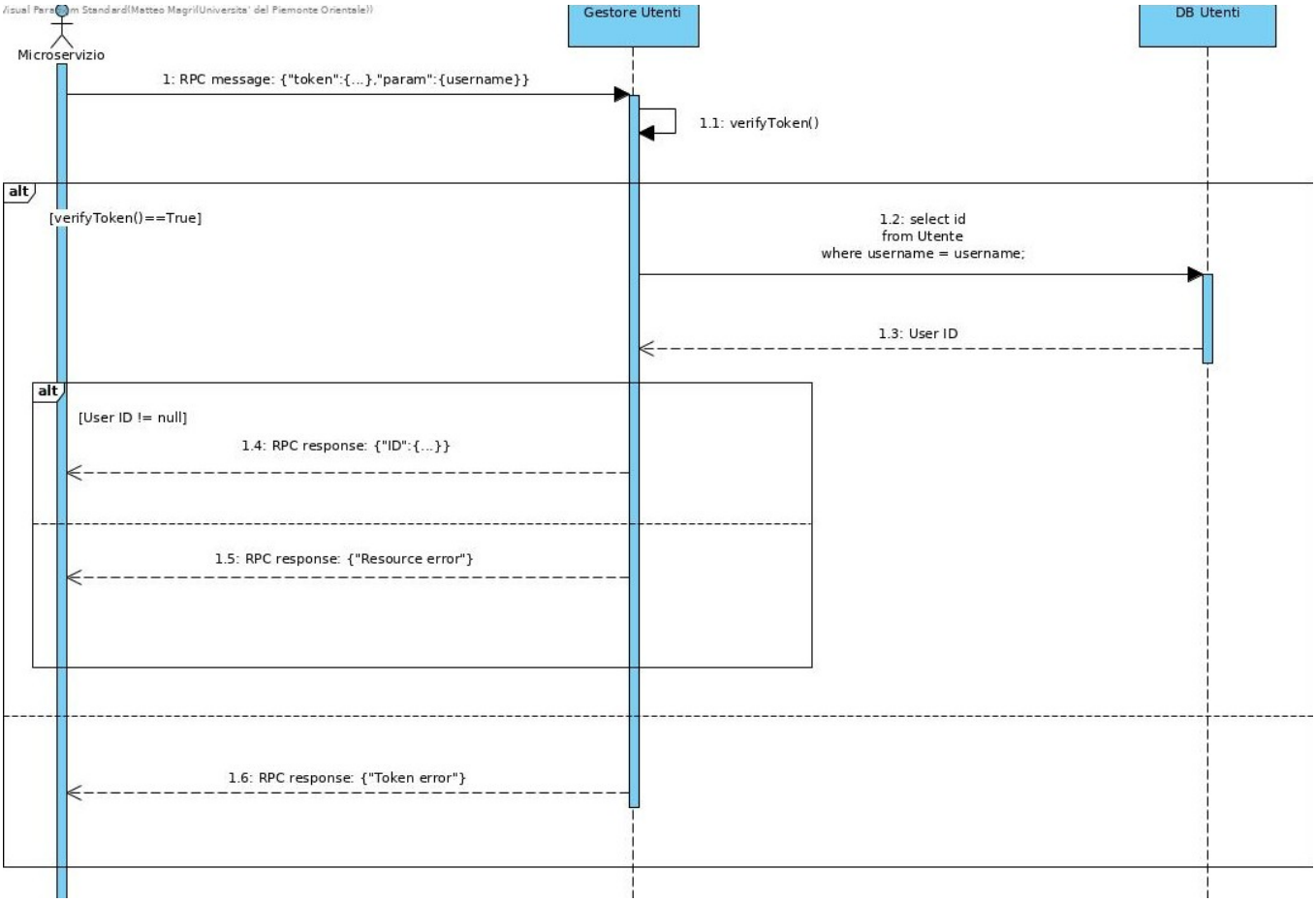
Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Contatta il DB per modificare lo stato
- Se questa ha successo restituisce "Status modified" , "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message;
    message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: setStatus");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    JSONObject param = obj.getJSONObject("param");
    String newStato = param.getString("newStato");
    String ccID = param.getString("ccID");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "UPDATE CarController SET Stato = " + "\'" + newStato + "\'" + " WHERE ID = " + "\'"
                + ccID + "\'";
        int rs = stmt.executeUpdate(sql);
        if (rs > 0) {
            response = "Status modified";
        } else {
            response = "Resource error";
        }
    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
} catch (ClassNotFoundException | UnsupportedEncodingException e) {
    e.printStackTrace();
```

GetUserID

E' la classe che permette di recuperare l' ID di un utente dato il suo username.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' username dell' utente

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a recuperare l' ID dell' utente.

Output

Il messaggio di output consiste di una stringa JSON formata unicamente dal seguente parametro:

- **ID:** contenente l' ID dell' utente

"Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_get_user_id`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`GetUserID.java`).

```
/**************************************
 * CALLBACK GET_USER_ID
 **************************************/
DeliverCallback callbackGetUserID = (consumerTag, delivery) -> {
    GetUserID gui = new GetUserID(delivery, channel);
    gui.run();
};
```

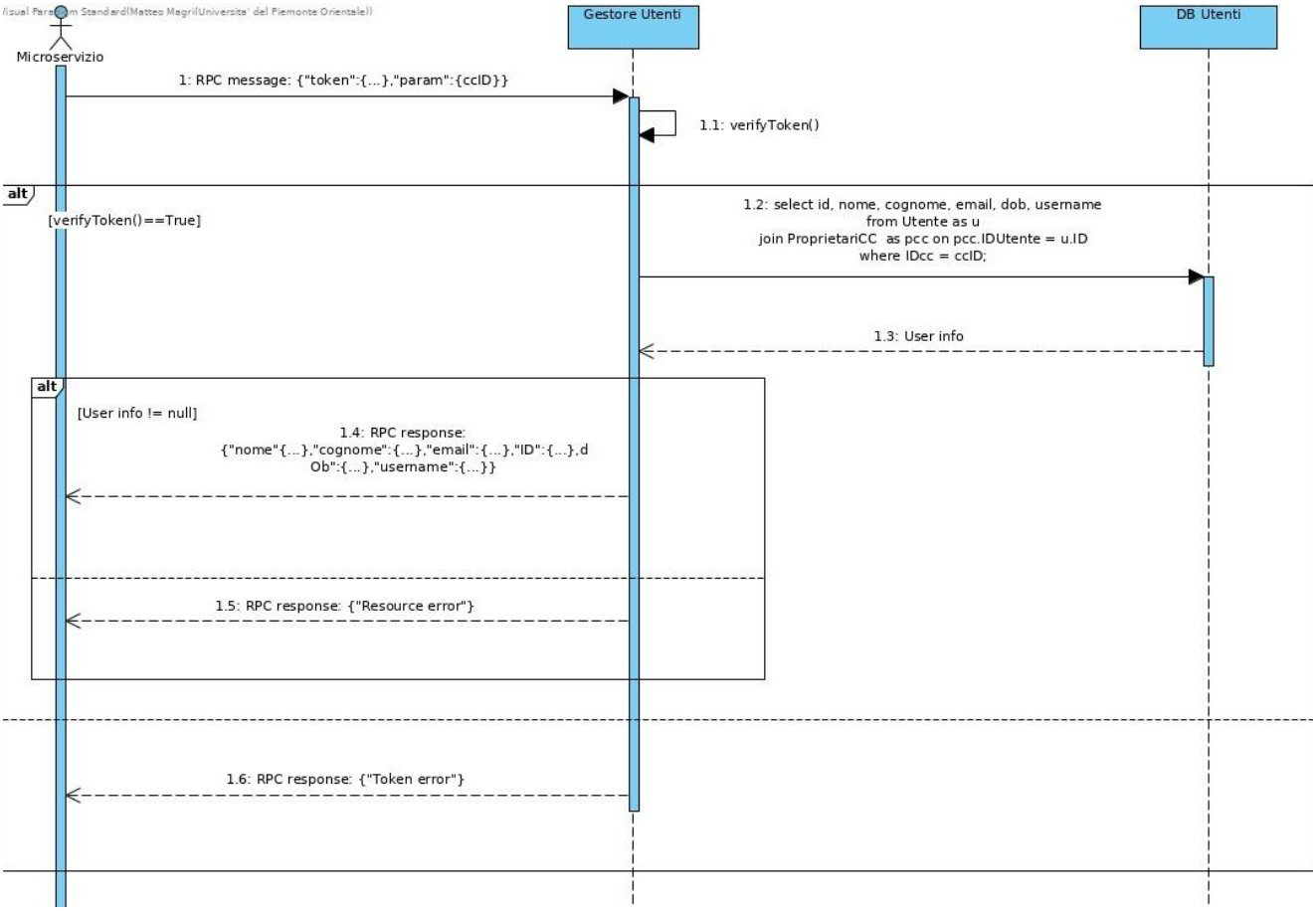
Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Contatta il DB per recuperare l'ID dell' utente
- Se questa ha successo restituisce la stringa JSON descritta precedentemente, "Resource error" altrimenti

```
try {
    // Leggo la richiesta
    String message = new String(delivery.getBody(), "UTF-8");
    System.out.println("Operazione richiesta: getUserInfo");
    // elaboro message
    JSONObject obj = new JSONObject(message);
    String token = obj.getString("token");
    String username = obj.getString("param");
    // Verificare il token
    PublicKeyRetriever key = new PublicKeyRetriever("localhost", "Demo", 8080);
    TokenVerifier tokenVer = new TokenVerifier(key);
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "SELECT id FROM UTENTE WHERE username = " + "\'" + username + "\'";
        ResultSet rs = stmt.executeQuery(sql);
        JSONObject messageJSON = null;
        // Estrazione dei dati
        while (rs.next()) {
            // Retrieve by column name
            String id = rs.getString("id");
            // Costruzione oggetto JSON
            messageJSON = new JSONObject();
            messageJSON.put("ID", id);
        }
        // Trasformazione in stringa dell' oggetto
        response = messageJSON.toString();
        // Chiudo la connessione
        rs.close();
    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
```

`GetUserInfoFromCC`

E' la classe che permette di recuperare le informazioni di un utente dato l' ID del suo car controller.



Parametri di input

Il Gestore Utenti si aspetta di ricevere dal microservizio client un messaggio JSON (stringa) contenente le seguenti chiavi:

- **token:** contenente il token
- **param:** contenente unicamente l' ID del car controller

Funzionamento

Una volta ricevuta la chiamata, il gestore provvederà per prima cosa a verificare il token.
In caso non ci siano problemi, provvede a recuperare le informazioni relative all' utente associato al car controller.

Output

Il messaggio di output consiste di una stringa JSON formata dai seguenti parametri:

- **nome:** contiene il nome dell' utente in formato `String`
- **cognome:** contiene il cognome dell' utente in formato `String`
- **email:** contiene l' email dell' utente in formato `String`
- **dOB:** acronimo di Date of Birthday, contiene l' anno di nascita dell' utente in `String` nel formato *yyyy-mm-dd*
- **ID:** contiene l' ID dell' utente in formato `String`
- **username:** contiene l' username dell' utente in formato `String`

Verrà restituito "Resource error" in caso di errore, "Token error" nel caso in cui il token non venga verificato correttamente.

Implementazione

Il backend riceve i messaggi e le richieste per questo metodo in una specifica coda con nome `rpc_get_user_info_from_cc`.
Anche qua non sarà lui a farsi carico direttamente di risolvere la richiesta, ma delegherà una classe specifica (`GetUserInfoFromCC.java`).

```
/**************************************
 * CALLBACK GET_USER_INFO_FROM_CC
 **************************************/
DeliverCallback callbackGetUserInfoFromCC = (consumerTag, delivery) -> {
    GetUserInfoFromCC gui = new GetUserInfoFromCC(delivery, channel);
    gui.run();
};
```

Nel momento in cui viene eseguito il metodo `.run()` e quindi il thread viene avviato, questo svolge in ordine i seguenti compiti:

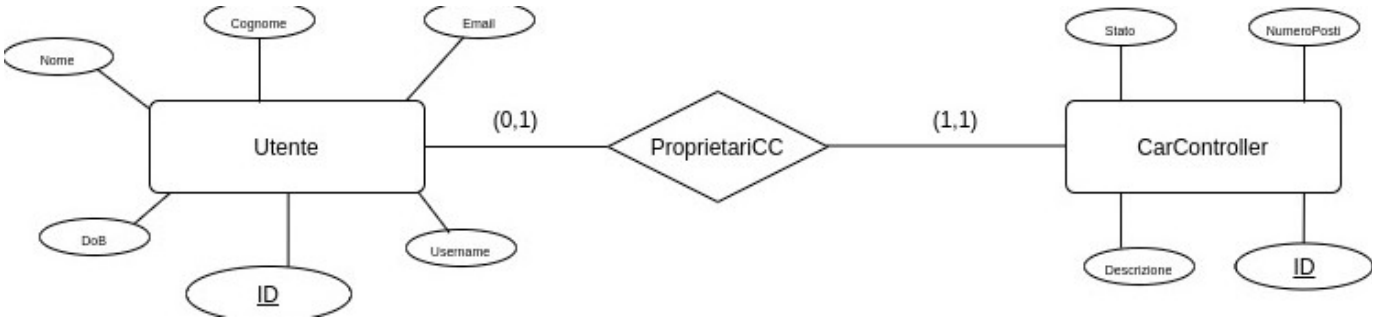
- Preleva il messaggio dalla coda
- Parsifica la stringa JSON per ottenere i parametri
- Verifica il token
- Contatta il DB per recuperare le informazioni dell' utente
- Se questa ha successo restituisce la stringa JSON descritta precedentemente, "Resource error" altrimenti

```
    if (tokenVer.verify(token)) {
        // Se la verifica va a termine, contatto il DB
        Connection conn = null;
        Statement stmt = null;
        Class.forName(JDBC_DRIVER);
        System.out.println("Connessione al database...");
        conn = DriverManager.getConnection(DB_URL, USERDB, PASSDB);
        System.out.println("Connessione eseguita con successo!!!");
        stmt = conn.createStatement();
        String sql = "SELECT id,nome,cognome,email,dob,username FROM UTENTE AS u "
                + "JOIN ProprietariCC AS pcc ON pcc.idutente = u.id WHERE IDcc = " + "\'" + ccID + "\'";
        ResultSet rs = stmt.executeQuery(sql);
        JSONObject messageJSON = null;
        // Estrazione dei dati
        while (rs.next()) {
            // Retrieve by column name
            String id = rs.getString("id");
            String nome = rs.getString("nome");
            String cognome = rs.getString("cognome");
            String email = rs.getString("email");
            String dob = rs.getString("dob");
            String username = rs.getString("username");
            // Costruzione oggetto JSON
            messageJSON = new JSONObject();
            messageJSON.put("nome", nome);
            messageJSON.put("cognome", cognome);
            messageJSON.put("email", email);
            messageJSON.put("dOb", dob);
            messageJSON.put("ID", id);
            messageJSON.put("username", username);
        }
        // Trasformazione in stringa dell' oggetto
        response = messageJSON.toString();
        // Chiudo la connessione
        rs.close();
    } else {
        // Altrimenti restituisco un token error
        response = "Token error";
    }
} catch (RuntimeException | JSONException e) {
    response = "Resource error";
```

Database

Per realizzare il database è stato utilizzato H2, un database SQL gratuito scritto in Java ed utilizzabile per mezzo delle giuste dipendenze Maven, supportato molto bene dal framework Spring.

Schema-ER



Entità

Utente:

- Nome: `varchar`
- Cognome: `varchar`
- Email: `varchar`
- DoB: `date`
- ID: `varchar` (*primary key*)
- Username: `varchar`

CarController:

- Stato: `varchar`
- NumeroPosti: `smallint`
- Descrizione: `text`
- ID: `varchar` (*primary key*)

Relazioni

ProprietariCC:

- IDUtente: `varchar REFERENCES Utente(ID)`
- IDcc: `varchar REFERENCES CarController(ID)`

Front-End

```java
// Accedo per recuperare le informazioni degli utenti
@GetMapping(value = "/getUserInfo")
public void getUserInfo(HttpServletRequest request)
        throws IOException, TimeoutException, InterruptedException, JSONException {
    String token = getKeycloakSecurityContext(request).getTokenString();
    String userId = getKeycloakSecurityContext(request).getToken().getSubject();
    //String userId = "";
    // chiamata al gestore utente
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(HOST);
    factory.setConnectionTimeout(300000);
    factory.setUsername(USER);
    factory.setPassword(PASSWD);
    connection = factory.newConnection();
    channel = connection.createChannel();
    // messaggio RPC
    JSONObject messageJSON = new JSONObject();
    messageJSON.put("token", token);
    messageJSON.put("param", userId);
    String query = messageJSON.toString();
    String requestQueueName = "rpc_get_user_info"; // coda dove invia il client ed il server rimane in attesa
    String response = call(query, requestQueueName);
    // stampa completa dell' utente
    System.out.println("\n\n********************");
    System.out.println("Richiesta recupero info dell' utente con ID: " + userId);
    System.out.println("Ecco la risposta dal server:\n");
    System.out.println(response);
}
```

Utilizzando il framework Spring ed il suo server interno Tomcat, in combinazione con le ottime API offerte da Keycloak, è stato reso possibile realizzare un front-end abbastanza semplice.

Con l' annotazione **`@GetMapping`** seguito da: (**`value = "/getUserInfo"`**), s' intende dire a Spring di mappare, e quindi eseguire la seguente funzione, nel momento in cui viene contattato su:

`localhost:numeroPortaSpring/getUserInfo`

Ovviamente questo procedimento è stato coniato per ogni API descritta precedentemente per poterle testare singolarmente.

Implementazione

Le prime due righe di codice servono rispettivamente per prelevare il **`token`** e l' **`userId`**.
Per fare questo nel momento in cui viene visitata la pagina */getUserInfo*, Spring riderige l' utente sull' auth server per l' autenticazione, eseguita questa sarà possibile ottenere token e ID dell' utente.

Successivamente vengono predisposti i fattori necessari per poter cominciare una comunicazione via AMQP con il backend, ovvero vengono inizializzati connessione e canale.

Prima di spedire il messaggio nella queue giusta, che ricordo essere in questo caso **`rpc_get_user_info`**, viene inizializzato il messaggio JSON che il backend si aspetta di ricevere, secondo lo schema descritto dai diagrammi di sequenza.

Infine il front-end rimane in attesa della risposta da parte del backend, ed una volta ricevuta la stamperà.

Matteo Magrì 20025172