PhotoHub

Progetto di Sistemi Distribuiti e Big Data

Gravagno - Di Mauro

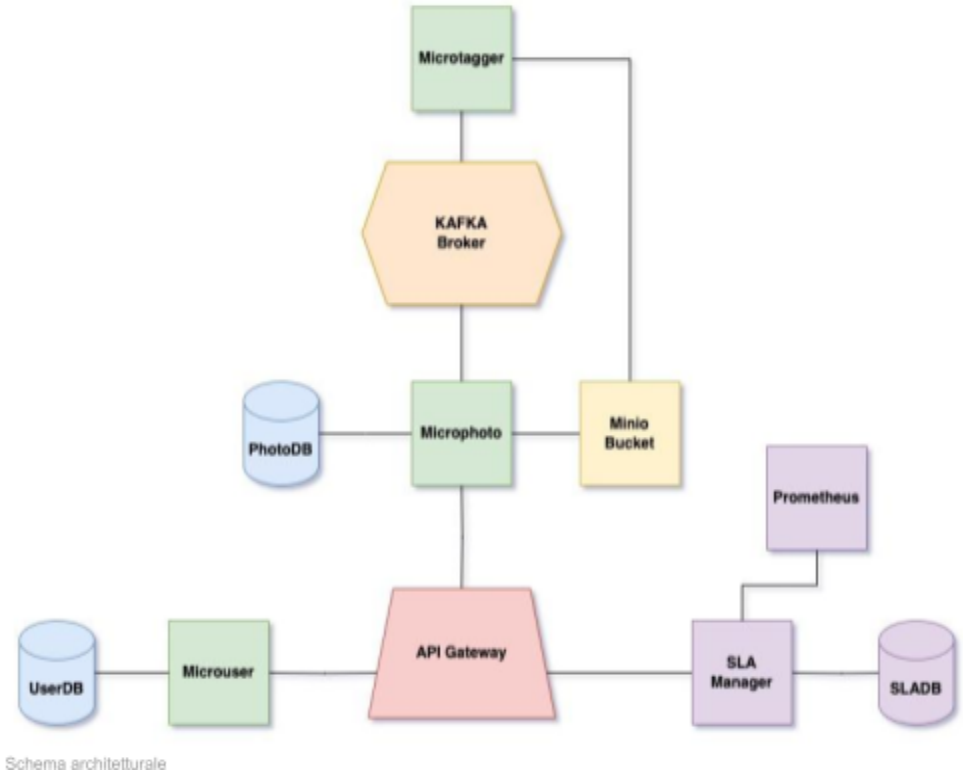
Descrizione applicazione:

L'idea alla base della nostra applicazione è quella di un social network in cui gli utenti, una volta effettuata la registrazione, possano caricare delle foto, potendone anche inserire una descrizione del contenuto. La novità nella nostra applicazione rispetto a quanto già presente nel mercato, sta nel fatto che a ciascuna foto caricata verrà associato un tag che ne riassume il contenuto oppure ne descrive una caratteristica. Nello specifico il tag viene calcolato sottoponendo la foto caricata ad una rete neurale: nel nostro caso, la scelta è ricaduta sulla rete AlexNET, in quanto facilmente utilizzabile e non eccessivamente esosa in termini di risorse richieste per svolgere l'inferenza.
Lato client, l'applicativo presenta inizialmente un'interfaccia d'accesso, dove l'utente potrà registrarsi o effettuare il login qualora si fosse già registrato in precedenza. Una volta effettuato l'accesso, l'utente avrà davanti un'interfaccia costituita da due tab: nel primo vedrà tutti i post che sono stati caricati nell'applicazione, mentre nel secondo tab potrà caricarne di nuovi.
Lato backend, l'applicativo è costituito da tre microservizi principali:

- Microuser: questo servizio è responsabile delle fasi di registrazione e accesso dell'utente all'applicazione;
- Microphoto: questo servizio è responsabile del caricamento di nuove foto e dell'l'ottenimento di quelle già caricate;
- Microtagger: questo servizio è responsabile dell'inferenza e del calcolo dei tag delle foto caricate.

In particolare l'utente registrato e autenticato dal Microuser utilizzerà un JWT token per richiedere l'autorizzazione nello svolgere le richieste successive negli altri microservizi. Per quanto concerne le comunicazioni Microphoto e Microtagger comunicano tra di loro sfruttando un broker Kafka: un'istanza di µPhoto pubblica sul broker la foto da taggare (il suo url) e un'istanza di µTagger, consumato il messaggio e calcolato il tag, lo pubblica a sua volta sul broker dove stavolta sarà un'istanza del servizio delle foto a consumare il messaggio e completare la transazione.
Vengono, inoltre, sfruttati servizi di storage, quali UserDB e PhotoDB (entrambi database MongoDB), sul quale salvare le informazioni sugli utenti e sulle foto: in

particolare, la foto in sé viene salvata all'interno del bucket S3 MinIO mentre, le informazioni sulle foto, come il nome utente di chi l'ha caricata, la sua descrizione, il tag e il suo url del bucket vengono salvate nel database PhotoDB.
Al fianco di questi microservizi vi è Prometheus che permette di monitorare dei SLI del sistema e collezionare serie temporali in modo da analizzare il comportamento del sistema in modo da identificare e prevedere eventuali violazioni del SLA.



Schema architetturale

L'immagine rappresenta uno schema della struttura a microservizi dell'applicazione evidenziando le comunicazioni tra essi e la loro natura (stateless, stateful, broker e gateway). Le comunicazioni tra Prometheus e i vari microservizi che pubblicano delle metriche non sono state evidenziate per non appesantire troppo il diagramma.

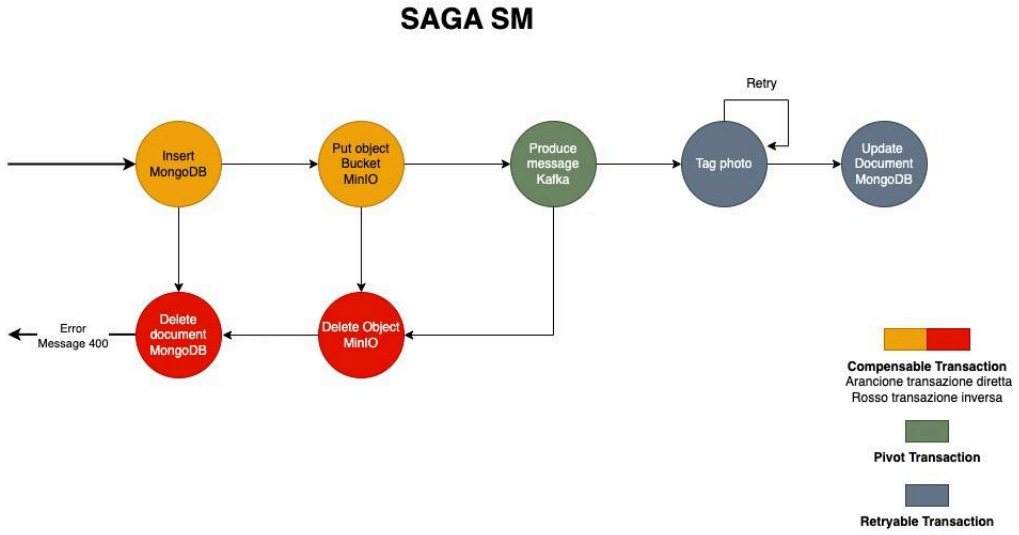
Scelte progettuali:

In questa sezione verranno elencate e spiegate le scelte progettuali più significative che abbiamo preso durante lo sviluppo dell'applicazione:

Comunicazione indiretta con Broker Kakfa

La comunicazione tra i microservizi Microphoto e Microtagger devono essere disaccoppiate temporalmente poiché l'inferenza sulla rete neurale potrebbe causare un rallentamento dei tempi di risposta per il caricamento della foto.
Per tale motivo abbiamo scelto di fare avvenire questa comunicazioni in modo indiretto con il broker Kafka e due topic fissi, ovvero il topic 'foto' e quello 'tag', che permettono la comunicazione indiretta bidirezionale: le istanze di Microphoto produrranno messaggi sul topic 'foto' dove sarà presente un gruppo di Microtagger come consumatori concorrenti che taggheranno le foto; le istanze di Microtagger risponderanno con i tag sul topic 'tag' dove i consumatori concorrenti saranno le istanze di Microphoto.

Pattern SAGA per gestire l'upload delle foto



Per gestire la transazione che concerne l'inserimento di una foto con i relativi tag all'interno del sistema si è pensato di utilizzare il pattern SAGA.
Le entità coinvolte nell'upload sono diverse: microphoto, photoDB, minIO e microtagger ognuna coinvolta in qualche tipo di elaborazione o di cambiamento di stato per raggiungere l'obiettivo finale.
Come rappresentato dalla figura vi è una divisione della transazione in due parti. La parte antecedente alla produzione del messaggio in Kafka (transazione pivot) è caratterizzata da una serie di transazioni di compensazioni per riportare lo stato del sistema in uno stato consistente.
La parte successiva alla transazione pivot eventualmente terminerà: ovvero, ci sarà un momento in cui se i microtagger riescono a taggare ed effettuare la

produzione del messaggio nel topic “tag”, committando il messaggio consumato nel topic “foto” così che la SAGA proceda in modo consistente eventualmente.
È importante osservare il fatto che nei consumer kafka, sia di microphoto che di microtagger, è disabilitata l’opzione “auto-commit” in modo che il commit di un messaggio consumato possa essere effettuato solamente dopo aver eseguito l’elaborazione del messaggio.
Questa gestione potrebbe causare la duplicazione delle richieste di tag e dunque la duplicazione delle relative risposte, ma essendo operazioni idempotenti ciò non costituisce un problema per la consistenza dell’applicazione.
Dunque sarà la gestione dei messaggi in Kafka e dei relativi commit a permettere il retry automatico della transazione che riguarda l’elaborazione dei tag della foto.
(Per una rappresentazione grafica dei messaggi scambiati tra le entità vedere la sezione “**Upload foto**”)

Assenza di thread nel microservizio Microtagger

La scelta progettuale deriva da un’altra scelta, ovvero quella di aumentare le partizioni del topic “foto” sulla base del numero di repliche del microservizio microtagger. Infatti, nel caso in cui vi fossero più partizioni che consumer microtagger, la ownership di più partizioni sarebbe affidata a un unico microtagger: in tal modo è necessario gestire un sistema che calcoli gli offset da committare per ogni partizione (essendo diversi), in modo da committare il più grande numero continuo dall’offset della partizione già elaborato per essa.
La gestione di queste problematiche è stata scritta, ma poi commentata per mancanza di risorse hardware con cui testare queste casistiche. Per tale motivo, si è preferito consegnare un codice più semplice ma meno efficiente con thread singolo.

Previsioni violazioni dello SLA Manager

Per quanto riguarda la parte relativa alla previsione delle violazioni ci siamo scontrati con una mancanza di dati a causa del poco utilizzo del sistema. Per tale motivo non abbiamo potuto eseguire delle analisi sulle serie temporali delle metriche raccolte dall’applicazione:

```
1. image_size: Number of bytes of the image
2. upload_image_time: Time elapsed for each upload
3. upload_and_tag_time:  Time  elapsed  to  complete  the  upload&tag
   procedure
4. inference_count: Number of inference performed
5. inference_time: Time elapsed for each inference
```

```
6. sessions_count: Number of users logged or registred
```

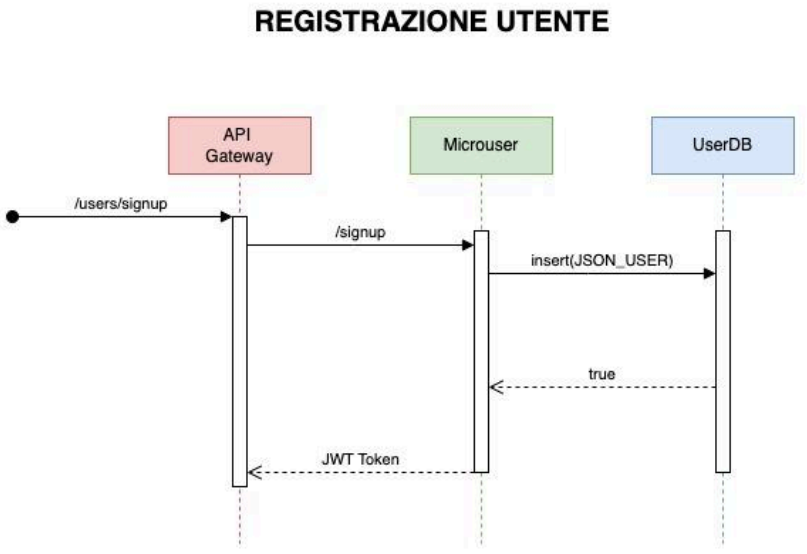
Nonostante tale deficit abbiamo stimato un utilizzo simile ad altre applicazioni di social networking come Instagram o TikTok e abbiamo pensato di applicare un modello di tipo SARIMAX poiché la seasonality delle applicazioni sopracitate sono dell'ordine delle 24 ore o dei 7 giorni.
Non avendo altre informazioni abbiamo effettuato un algoritmo di ricerca su tutta la griglia dei parametri considerando il prodotto cartesiano di tutti i possibili valori discreti sull'ordine di autoregressione, sull'ordine di differenziazione e così via.
La scelta dei parametri, e dunque del modello, avviene all'aggiunta della metrica con gli intervalli nella SLA: tale approccio è riduttivo perché necessiterebbe di una continua eliminazione e riaggiunta della metrica alla SLA. L'approccio potrebbe essere migliorato con una esecuzione periodica della scelta del modello per ogni metrica nello SLA o magari scatenata da un evento quale, ad esempio, l'aumento percentuale dei dati raccolti dal sistema.

JWT Token per autorizzazione

La scelta di utilizzare un JWT Token per autorizzare le richieste successive al login da parte dell'utente deriva dalla risoluzione, con un approccio più semplice della comunicazione diretta, dell'autenticazione dell'utente.
Il microphoto nel momento dell'upload necessita dell'autorizzazione dell'utente per accettare una foto nel sistema. Il JWT Token con la sua natura crittografica garantisce tale autorizzazione. Inoltre l'informazione aggiuntiva dell'username, inglobata dentro il token, permette di conoscere anche l'identità dell'utente che effettua le richieste.

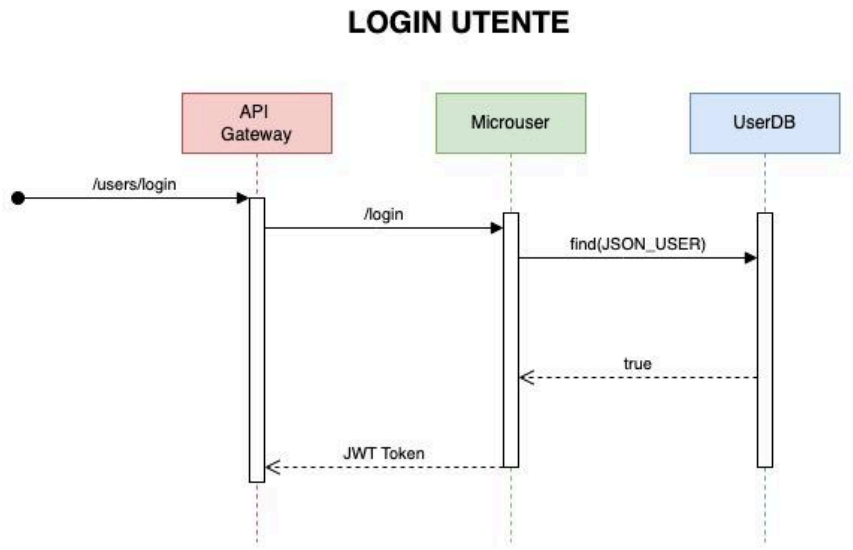
Endpoint e diagrammi di interazione:

- **Registrazione utente (/users/signup)**



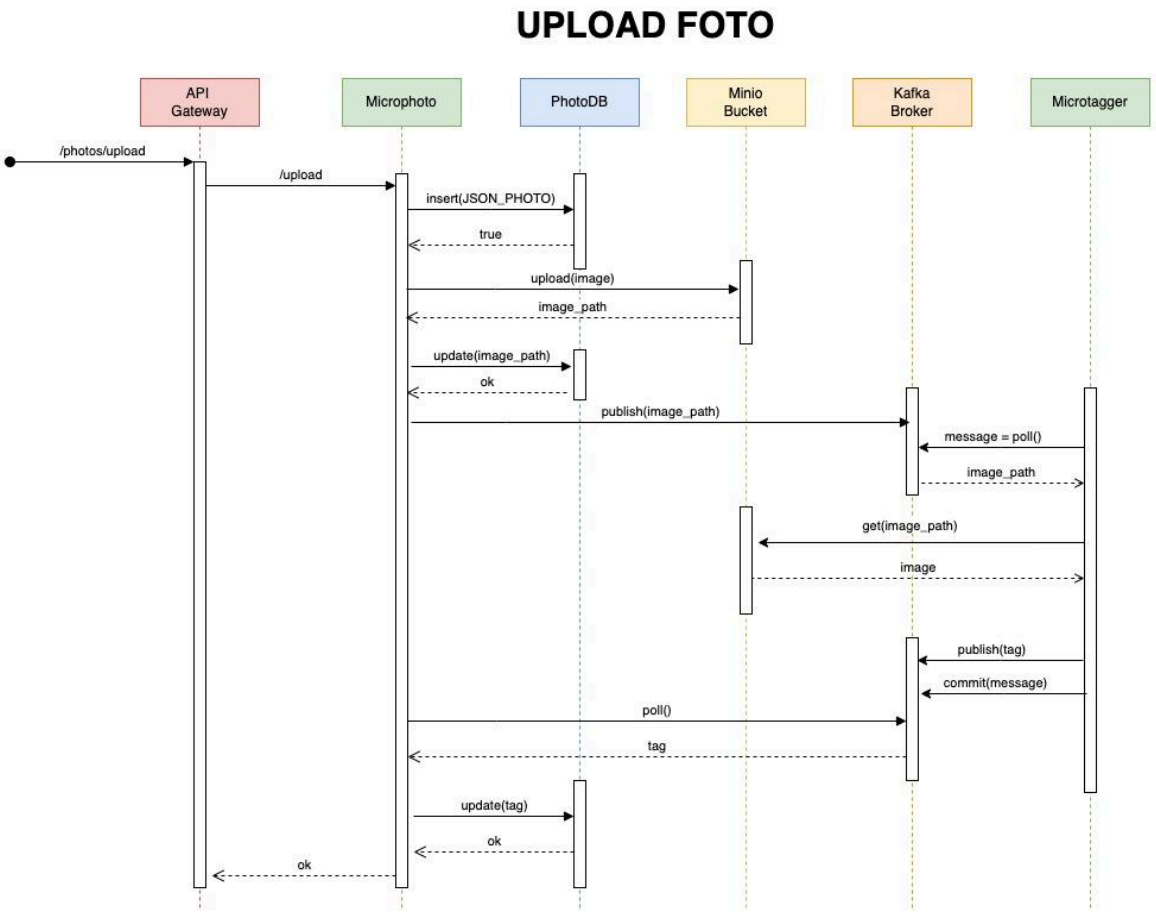
Quando l'utente vuole registrarsi all'interno dell'applicazione invia una richiesta POST contenente, all'interno del JSON, username e password con le quali si sta registrando. La richiesta viene inviata all'API Gateway sull'endpoint */users/signup* il quale, fungendo da proxy server, reindirizza la richiesta sull'endpoint */signup* del microservizio *Microuser*. Ricevuta questa richiesta, il servizio inserirà le informazioni sul database (viene controllato se esiste già un utente con quel username, restituendo un errore all'utente qualora fosse così) e, in caso di esito positivo nella registrazione, il microservizio restituirà un JWT Token all'utente, che verrà utilizzato per autorizzare tutte le richieste successive.

- **Login utente (/users/login)**



Quando l'utente vuole accedere all'applicazione invia una richiesta POST contenente, all'interno del JSON, l'username e la password con le quali sta provando ad accedere. La richiesta viene inviata all'API Gateway sull'endpoint */users/login* il quale, fungendo da proxy server, reindirizza la richiesta sull'endpoint */login* del microservizio *Microuser*. Ricevuta questa richiesta, il servizio cercherà l'utente nel database e, in caso di esito positivo, restituirà un JWT Token all'utente, che verrà utilizzato per autenticare tutte le richieste successive. Il funzionamento è identico alla registrazione dell'utente se non per il fatto che qualora non venisse trovato alcun utente con quelle credenziali, verrà restituito un errore.

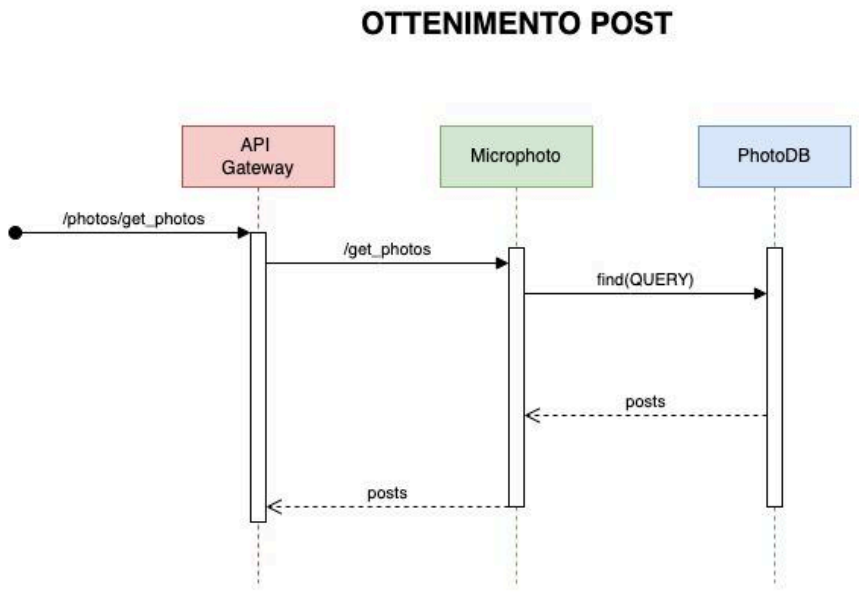
- **Upload foto (/photos/upload)**



Quando l'utente vuole caricare una foto nell'applicazione invia una richiesta POST contenente, all'interno del form (Content-Type: multipart/form-data), l'immagine che vuole inviare e una sua breve descrizione. La richiesta viene inviata al API Gateway sull'endpoint *<u>/photos/upload</u>* il quale, fungendo da proxy server, reindirizza la richiesta sull'endpoint *<u>/upload</u>* del microservizio *Microphoto*. Ricevuta la richiesta, innanzitutto il servizio ne verificherà la validità analizzando il token JWT passato nel campo del header Authorization: se la verifica va a buon fine si procederà con la richiesta altrimenti, si restituirà un messaggio di errore all'utente (STATUS CODE: 401 - Unauthorized).
Verificata la validità della richiesta, il microservizio caricherà prima il document con le informazioni della foto sul database (vengono salvati la

descrizione, l'username dell'utente che ha caricato la foto e l'istante di tempo in cui la foto è stata ricevuta) per poi salvare la foto all'interno del bucket S3 MinIO configurato con le policy in modo tale da permettere solamente la get dell'immagine senza autorizzazione alcuna. Dopodiché farà l'update del document precedentemente creato sul DB con il path della foto (utilizzato per poter scaricare la foto quando la si vorrà visualizzare nell'applicazione) e pubblicherà su Kafka, nel topic "foto", l'id della foto e il path dell'immagine sul bucket in maniera tale da poter permettere al servizio *Microtagger*, subscriber di questo topic, di scaricare la foto e di farne l'inferenza per svolgere il suo lavoro di individuazione del tag. Una volta calcolato, esso verrà pubblicato su Kafka, nel topic "tag", per poi essere trasmesso a Microphoto che è sottoscritto a questo topic. Infine, ottenuto il tag della foto, Microphoto ne aggiornerà le informazioni sul database, salvandone il tag.
Se tutto è andato bene, ovvero è stato creato il document sul DB e caricata la foto su MinIO, verrà restituito un messaggio di successo all'utente altrimenti, se si è verificata una qualche problematica verrà restituito un messaggio di errore. La seconda parte con la comunicazione su kafka è disaccoppiata temporalmente e non verrà aspettata la restituzione dei tag per rispondere con un messaggio HTTP all'utente.
(Vedere la sezione 'Scelte progettuali' per la gestione della transazione con il pattern SAGA)

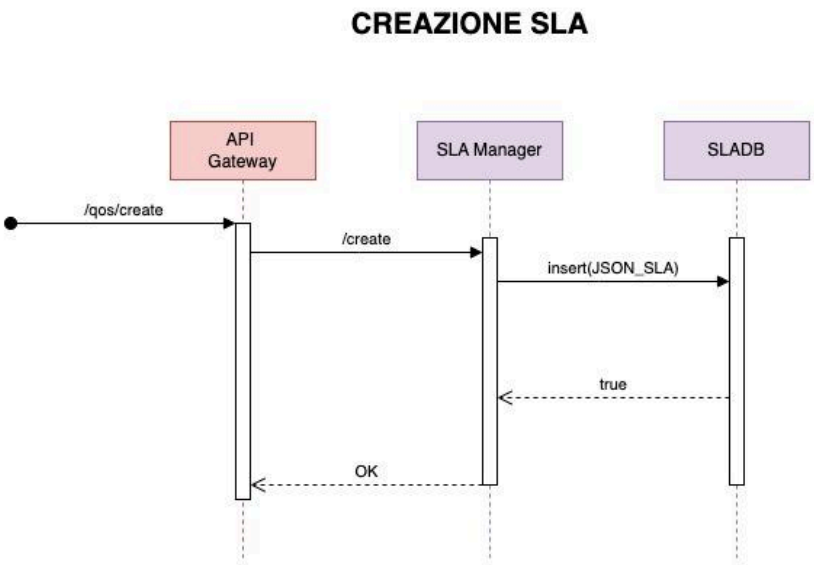
- **Ottenimento post (/photos/get_photos)**



Quando l'utente vuole ottenere uno o più post invia una richiesta GET. La richiesta viene inviata al API Gateway sull'endpoint */photos/get_photos* il quale, fungendo da proxy server, reindirizza la richiesta sull'endpoint */get_photos* del microservizio *Microphoto.* Il servizio, sulla base dei parametri passati nella richiesta, crea una query con la quale interrogherà il database, ottenendo in risposta gli ultimi 10 post che soddisfano le condizioni specificate, i quali verranno poi restituiti all'utente.
I parametri sono i seguenti:

- *username*: usato per indicare di voler visualizzare tutti i post ad eccezione di quelli caricati dall'utente specificato (opzionale);
- *tag*: usato per indicare la volontà di voler visualizzare tutti quei post che possiedo il tag specificato (opzionale);
- *skip*: usato per indicare da quale "pagina" del database leggere le informazioni. Il valore di default è 0.

- **Creazione SLA (/qos/create)**

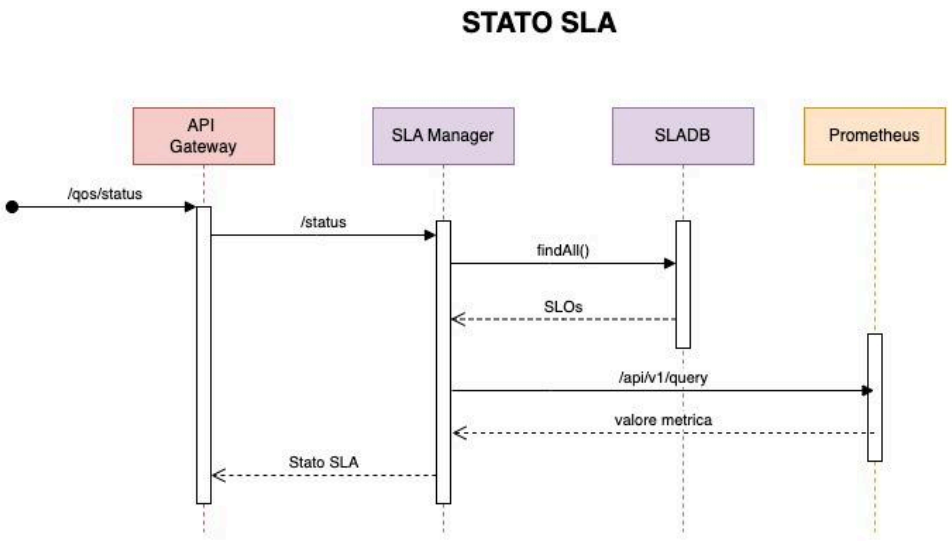


Quando l'admin dell'applicazione vuole creare/inserire una nuova metrica nella SLA, invierà una richiesta POST contenente, all'interno del JSON, le informazioni su tale metrica (nome della metrica, intervallo dei valori per soddisfare lo SLO e informazioni su una possibile aggregazione temporale* della metrica e tempo di aggregazione). La richiesta viene inviata all'API Gateway sull'endpoint */qos/create* il quale, fungendo da proxy server, reindirizza la richiesta sull'endpoint */create* del microservizio

SLA Manager. Il servizio, ricevuta tale richiesta, registrerà la metrica nel database e restituirà all'utente un messaggio di successo. Inoltre, ad ogni nuova metrica inserita, è fatto runnare un deamon thread che allena il modello di previsione in background in modo da continuare a rispondere alle nuove richieste dell'amministratore, ma ottenere i parametri migliori basandosi su una ricerca di tipo grid con il modello SARIMAX (Vedere la sezione 'Scelte progettuali' per la spiegazione della scelta del modello e dei range dei parametri).

*Le aggregazioni supportate sono: `median, increase, sum, avg.`

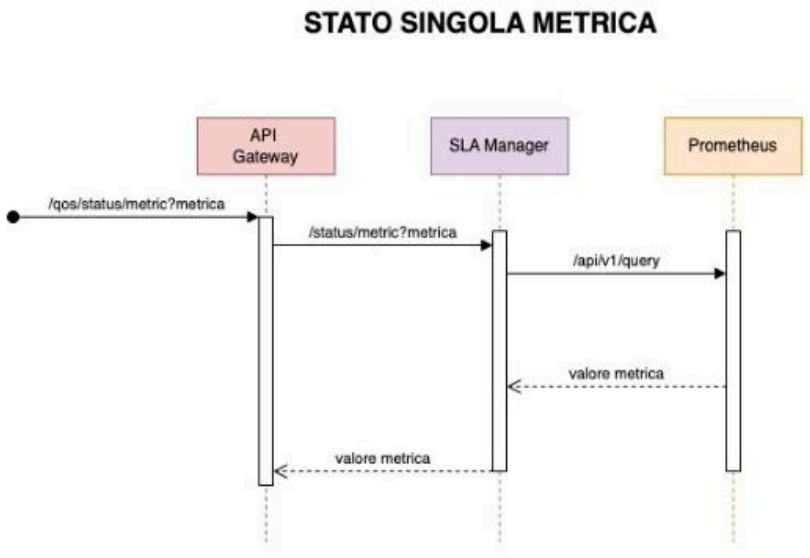
- **Stato SLA (/qos/status)**



Quando l'admin dell'applicazione vuole ottenere informazioni sullo stato della SLA, invierà una richiesta GET. La richiesta viene inviata all'API Gateway sull'endpoint */qos/status* il quale, fungendo da proxy server, reindirizza la richiesta sull'endpoint */status* del microservizio *SLA Manager*. Il servizio, ricevuta tale richiesta, otterrà dal database tutte le metriche registrate nella SLA e, per ciascuna di esse, interrogherà prometheus sull'endpoint */api/v1/query* per ottenere il valore attuale della metrica: tale valore verrà confrontato coi valori i valori max e min specificati nello SLO e dall'esito di questo confronto verrà registrato in un dictionary lo stato della SLA evidenziando lo stato di ogni SLO ("nome metrica"="false" c'è stata violazione, se invece il valore è "true" non c'è stata).

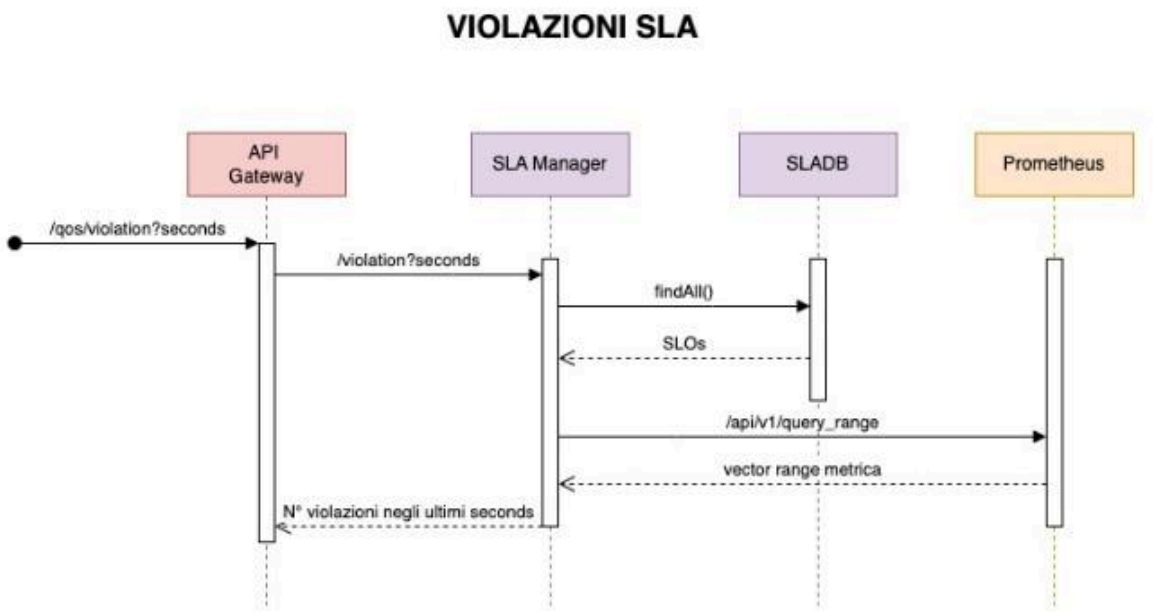
Per quanto riguarda le metriche create che riguardano un'aggregazione dei dati sono state utilizzate le funzioni di Prometheus come: `quantile_over_time, increase, sum, avg.`
L'output dell'endpoint interrogato dall'admin sarà il dictionary con le informazioni: metrica-stato (true: tutto bene, false: violazione).

- **Stato della singola metrica (/qos/status/metric)**



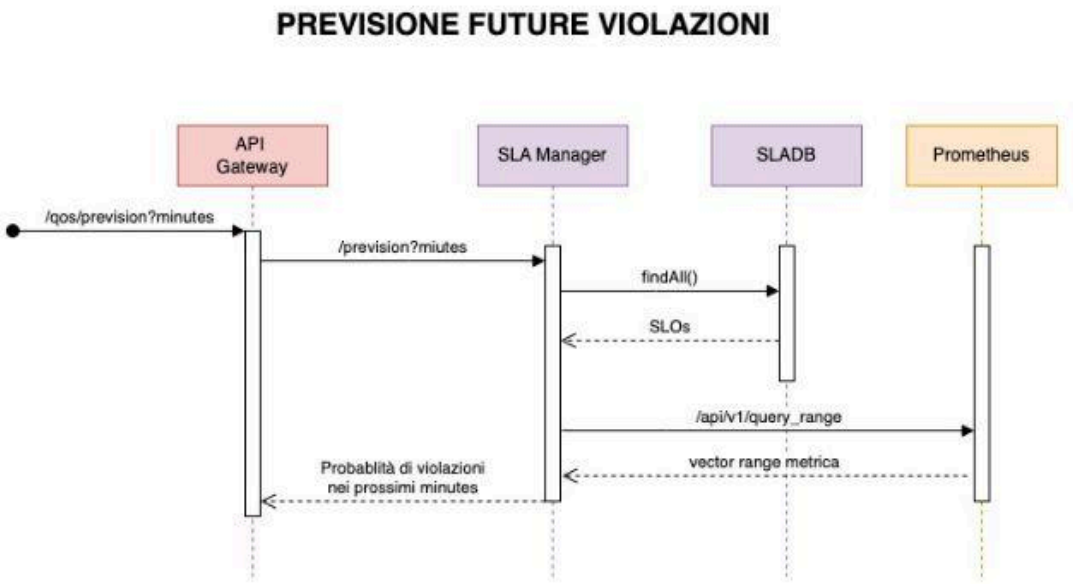
Quando l'admin dell'applicazione vuole ottenere informazioni sullo stato di una singola metrica, invierà una richiesta GET. La richiesta viene inviata al API Gateway sull'endpoint */qos/status/metric* il quale, fungendo da proxy server, reindirizza la richiesta sull'endpoint */status/metrica* del microservizio *SLA Manager*. Il servizio, ricevuta tale richiesta, prenderà la metrica passata al parametro metrics e invierà una richiesta a prometheus sull'endpoint */api/v1/query*, dal quale otterrà il valore corrente della metrica. Tale valore verrà restituito all'admin.
La richiesta può essere effettuata anche su un KPI che non è stato inserito nel SLA.

- **Violazioni SLA (/qos/violation)**



Quando l'admin dell'applicazione vuole sapere il numero di violazioni della SLA che sono accadute nelle ultime ore o minuti, invierà una richiesta GET specificando come parametro l'intervallo di tempo (in secondi) a cui è interessato. Il servizio, ricevuta tale richiesta, otterrà dal database tutte le metriche registrate nello SLA e, per ciascuna di esse, invierà una richiesta a prometheus sull'endpoint */api/v1/query_range* per ottenere il valore della metrica nell'intervallo specificato (richiede quindi un vector range), andando poi a confrontare i diversi campioni ottenuti con i valori di max e min specificati nell'agreement; in caso di violazione, incrementa il contatore delle violazioni per quella metrica.
In questo caso non sarà possibile applicare le funzioni di Prometheus elencate nella sezione "**Stato SLA**" poiché si lavora con dei range vector. La soluzione è stata quella di trasformare i dati JSON della richiesta HTTP fatta al server di Prometheus in Dataframes panda e applicare la rolling window sulla time serie per aggregare.
Alla fine verrà restituito all'admin un dictionary, in cui ogni voce specifica la metrica e il numero di violazioni che si sono verificate nell'intervallo specificato.

- **Previsione future violazioni (/qos/prevision)**



Quando l'admin dell'applicazione vuole sapere la probabilità di future violazioni della SLA nei prossimi minuti, invierà una richiesta GET specificando come parametro l'intervallo di tempo a cui è interessato. Il servizio, ricevuta tale richiesta, otterrà dal database tutte le metriche registrate nello SLA e, per ciascuna di esse, invierà una richiesta a prometheus sull'endpoint */api/v1/query_range* per ottenere il vector range della metrica nell'intervallo specificato; tale vettore verrà mandato in pasto al modello precedentemente allenato, in modo tale da ottenere la probabilità di future violazioni della metrica.
La probabilità è calcolata utilizzando un approccio casi favorevoli su casi totali.
Utilizzando l'intervallo di confidenza del modello si calcola la densità dei punti che non appartengono all'intervallo della metrica specificato nello SLO e rapportandolo sulla grandezza dell'intervallo si ottiene un rapporto che rappresenta la probabilità di violazione. Tale calcolo viene fatto sui valori estremi in modo da avere le probabilità maggiori di violazione.
Infine, all'admin dell'applicazione verrà restituito un dictionary contenente, per ogni metrica della SLA, la sua probabilità di violazione futura nei minuti indicati.