

Relazione DSBD Samperi_Volpe

Abstract

Il progetto da noi presentato propone lo sviluppo di un'applicazione distribuita il cui scopo è quello di offrire agli utenti la possibilità di iscriversi e ricevere notifiche relative a offerte speciali sui prezzi dei voli. Gli utenti potranno indicare le proprie preferenze, specificando l'aeroporto di partenza, le eventuali destinazioni di interesse e il budget di spesa. In questo modo, saranno avvisati quando si presenteranno opportunità di volo in partenza dagli aeroporti da loro designati, adatte alle loro esigenze di budget.

Di seguito presentiamo il funzionamento:

-il **Client** si interfaccia con lo **User Controller** per registrarsi al servizio e per effettuare le richieste di sottoscrizione, una richiesta di iscrizione può essere del tipo iscrizione ad una tratta (in cui l'utente specifica l'aeroporto di destinazione, quello di origine, un budget e il numero di adulti), oppure può richiedere di iscriversi per ricevere delle offerte speciali riguardo voli che partono da un determinato aeroporto senza specifica destinazione (in questo caso l'utente inserisce solo l'aeroporto di origine e una soglia di prezzo).

Nel caso in cui esso si sia iscritto ad una tratta, esso verrà notificato ogni 24 ore nel caso in cui i prezzi dei voli relativi a quella tratta siano inferiori al budget inserito.

Nel caso in cui esso si sia iscritto ad un aeroporto, esso verrà notificato ogni 24 ore nel caso in cui ci siano voli in partenza dallo stesso aeroporto per una qualsiasi destinazione, con prezzo inferiore al budget inserito.

-**User Controller** svolge il ruolo dell'**APIGW**, esso infatti si interfaccia col client e si occupa quindi di registrare (o disiscrivere) l'utente e le sue "rules" nei relativi database; quindi, inoltrerà le richieste dell'utente rispettivamente ai microservizi: **UserInfo** nel caso di registrazione dell'utente al servizio, che ha accesso al database "users", e **Rules** nel caso in cui il client stia richiedendo di iscriversi/disiscrivere a delle specifiche tratte/aeroporti, il quale ha accesso al database "rules", al cui interno troviamo le tabelle tratte e aeroporti.

User Controller ha inoltre il ruolo di inoltrare le tratte e gli aeroporti di interesse dei clienti al microservizio **Controller Tratte**. Nel caso in cui l'inoltro delle informazioni a Controller Tratte non dovesse andare a buon fine, è stato implementato un meccanismo di rollback che eliminerà la rule appena inserita dal database Rules e che avviserà il cliente del fallimento dell'iscrizione.

-**Controller Tratte** ha quindi il ruolo, man mano che riceve tratte e aeroporti da **User Controller**, di inserire (o eliminare) questi dati nel proprio database "controllertratte", composto da due tabelle: tratte_salvate e aeroporti_salvati; ha inoltre il ruolo di inviare, quando richiesti, tali dati allo **Scraper** per permettergli di effettuare le richieste alle API esterne.

-Lo **Scraper** ogni 24h dovrà effettuare una richiesta a **Controller Tratte** per poter ricevere da esso le tratte e gli aeroporti aggiornati, così da poter effettuare le richieste alle API esterne offerte da Amadeus (<https://developers.amadeus.com>), in particolare utilizziamo Flight Offers Search per la ricerca delle tratte data origine, destinazione e numero di adulti, e Flight Inspiration Search per la ricerca delle offerte con sede di partenza l'aeroporto fornito dall'utente; quindi invierà i risultati ottenuti nei rispettivi topic Kafka: Tratte e Aeroporti.

In particolare, lo Scraper ha accesso al proprio database nel quale conserva anch'esso la lista delle tratte e degli aeroporti col quale farà richiesta alle API esterne, abbiamo scelto di implementare anche questo database (che si aggiorna ogni 24 ore tramite richiesta da parte dello Scraper a Controller Tratte) per poter permettere ugualmente al microservizio di effettuare le richieste all'API esterna anche nel caso in cui Controller Tratte per qualsiasi motivo dovesse andare in down.

Abbiamo infatti deciso di implementare un **Circuit Breaker** sulla funzione che da Scraper chiede i dati a Controller Tratte, così che, se la richiesta non dovesse andare a buon fine per 5 volte di seguito, allora lo Scraper chiamerà l'API esterna con i dati relativi alle 24 ore precedenti invece di non aver dati su cui effettuare la richiesta.

-l'**Elaboratore** è il consumer dei topic Kafka, quindi prende i dati da essi e si occuperà di effettuare una richiesta a **User Controller**, inviandogli i dati ottenuti, per ricevere le e-mail degli utenti interessati a quelle specifiche offerte; quindi, invierà al **Notifier** le e-mail e le informazioni da inviare agli utenti iscritti.

-il **Notifier** quindi, ottenute le informazioni dall'elaboratore, avviserà via e-mail il cliente.

Elenco microservizi e relative comunicazioni

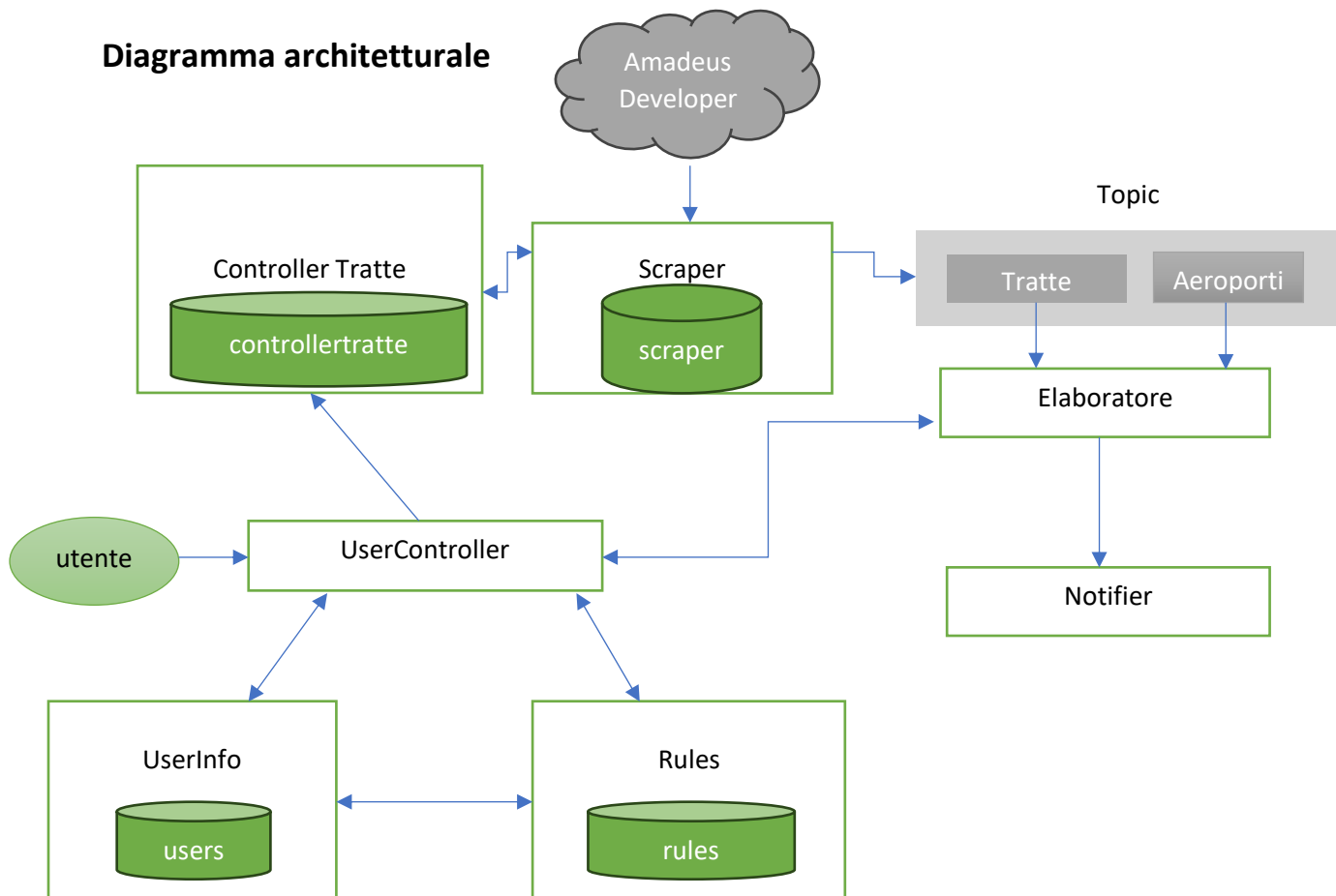
- UserController
- UserInfo
- Rules
- Controller_tratte
- Scraper
- Elaboratore
- Notifier

Nella nostra applicazione, il componente Scraper agisce come produttore dei topic **Kafka** "Tratte" e "Aeroporti", per ottenere i dati da inserire in questi topic, esso esegue richieste periodiche all'API esterna fornita da **amadeus developers** ogni 24 ore (in quanto tipicamente i prezzi dei voli cambiano ogni giorno e non con un periodo più breve), tali richieste consentono di raccogliere informazioni aggiornate sui prezzi delle tratte e sulle offerte disponibili. Dall'altro lato, il componente Elaboratore funge da consumatore, sottoscrivendosi ai medesimi topic.

L'utilizzo di Kafka come broker consente una comunicazione affidabile e asincrona tra i due componenti, garantendo una gestione efficiente dei dati.

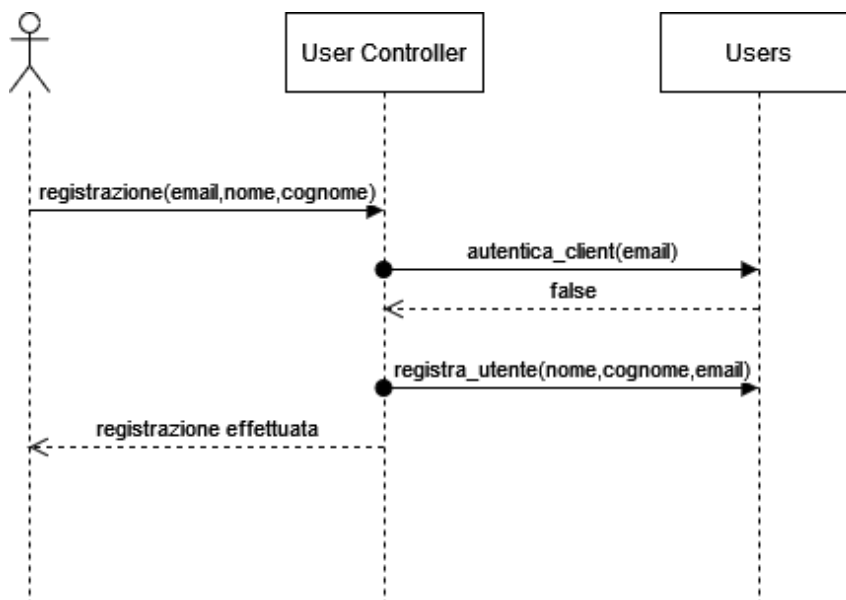
I restanti microservizi interagiscono tra loro attraverso il servizio **Flask**. Ciascun microservizio è implementato come un'applicazione Flask indipendente, e le comunicazioni avvengono mediante richieste HTTP POST e GET. Questa scelta di design offre una flessibilità significativa, permettendo ai microservizi di scambiare dati in modo rapido ed efficiente.

Diagramma architetturale



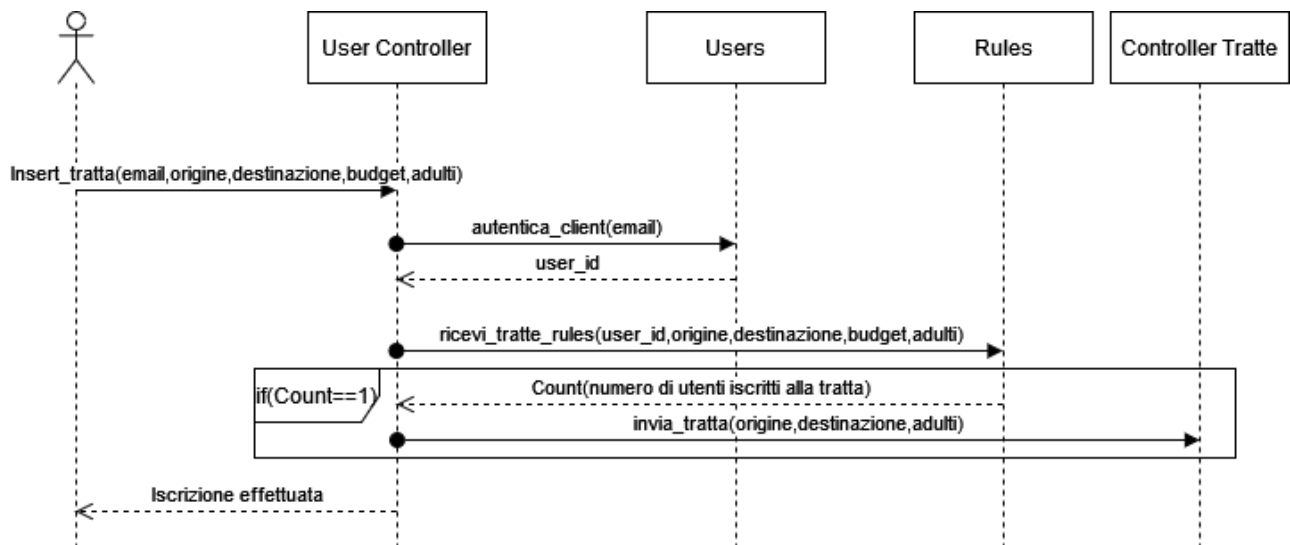
Diagrammi di interazione

Registrazione:



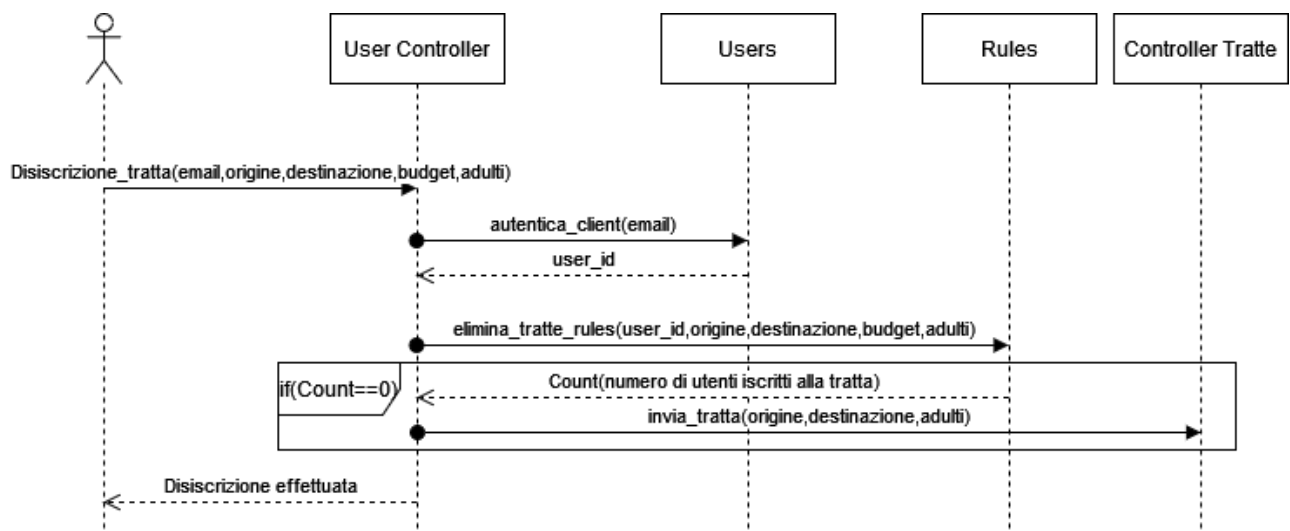
In questo caso Users, dopo aver verificato che l'utente non sia registrato, ritornando quindi un False, si occuperà di inserire i dati del nuovo utente nel proprio database.

Insert_tratta (simile per Insert_aeroporto):



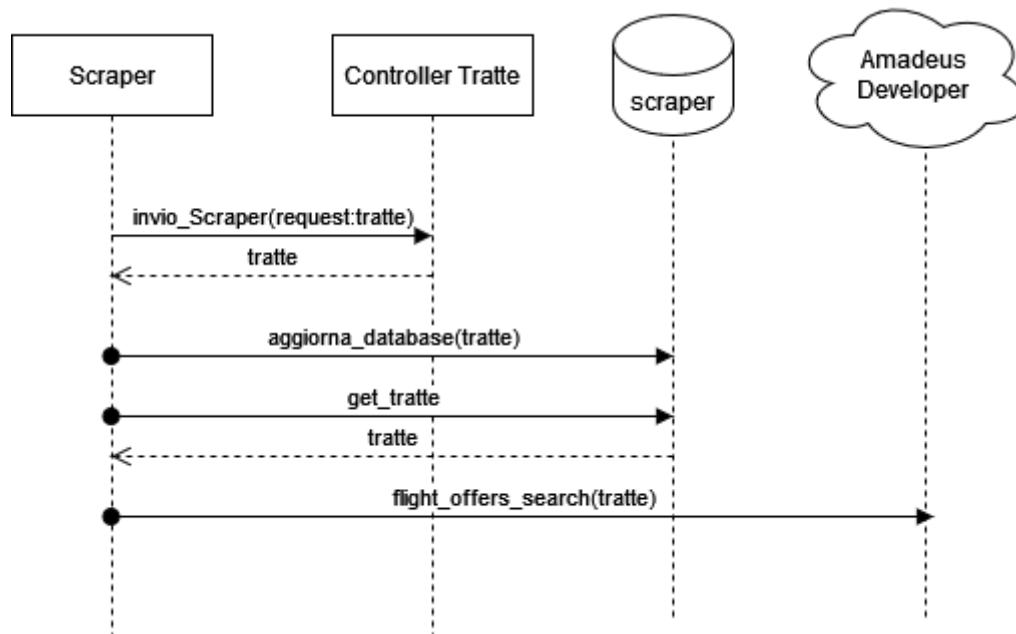
In questo caso, dopo aver autenticato l'utente e aver ottenuto il relativo user_id, Rules aggiunge al proprio database quella relativa tratta e ritorna il numero di persone, dopo l'insert, che hanno un riferimento a quella precisa tratta, nel caso in cui il numero sia uguale a 1 allora Controller_tratte, dopo aver ricevuto la tratta, la inserisce nella tabella "tratte_salvate" del proprio database.

Disiscrizione_tratta (simile per Disiscrizione_aeroporto):



In questo caso, dopo aver autenticato l'utente e aver ottenuto il relativo user_id, Rules elimina dal proprio database quella relativa tratta e ritorna il numero di persone, dopo l'insert, che hanno un riferimento a quella precisa tratta, nel caso in cui il numero sia uguale a 0 allora Controller_tratte, dopo aver ricevuto la tratta, la elimina dalla tabella "tratte_salvate" del proprio database.

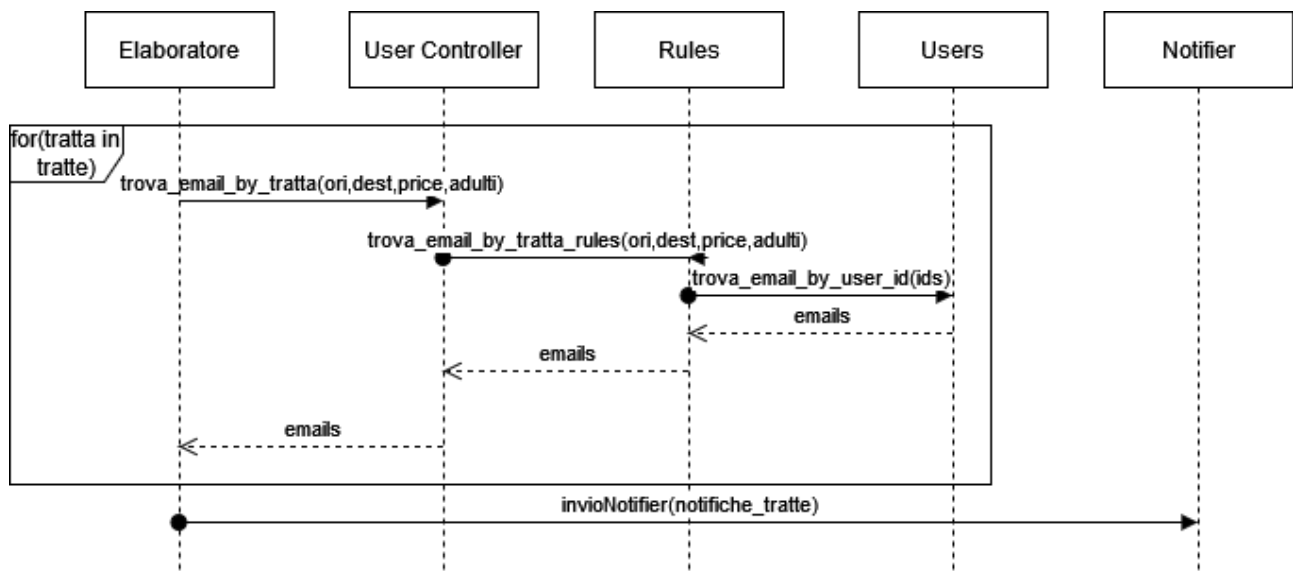
Scraping (delle tratte, simile per gli aeroporti):



In questo caso Scraper fa richiesta a Controller_tratte per ricevere le nuove tratte richieste dagli utenti e quindi aggiornare il proprio database, a questo punto legge il proprio database e si occuperà di effettuare la richiesta ad Amadeus Developer per ottenere i relativi prezzi, successivamente invierà al relativo topic tali offerte.

È stato implementato in particolare un Circuit Breaker sulla funzione “`invio_Scraper`”, esso permette, dopo cinque tentativi non andati a buon fine nella richiesta delle tratte/aeroporti aggiornate a Controller_tratte, di poter comunque effettuare la `flight_offers_search` con le tratte/aeroporti aggiornati al giorno precedente. È questo il motivo per cui abbiamo deciso di implementare un ulteriore database in Scraper.

Trova_emails (delle tratte, simile per aeroporti):



In questo caso Elaboratore, dopo aver letto le offerte presenti nei topic, si occuperà di fare richiesta a User_Controller per trovare le e-mail degli utenti effettivamente interessati a quelle offerte, il quale chiede a Rules che troverà gli id degli interessati, il quale fa richiesta a Users per ottenere le e-mail corrispondenti a questi ids. Ottenute le e-mail allora Elaboratore invierà a Notifier le e-mail con il relativo messaggio da inviare ai clienti.

Gestione della qualità del servizio

In aggiunta all'applicazione fino ad ora descritta, sono stati implementati i manifesti di ogni servizio per poter deployare l'applicazione in **kubernetes**, è stato quindi configurato un server **Prometheus**, il quale fa scraping delle metriche esposte da **node exporter**, **kafka**, **kube-state-metrics** e ulteriori, nel processo di monitoraggio esso fa inoltre scraping di tre metriche personalizzate da noi implementate, **scraping_time**, la quale misura il tempo che impiega il servizio scraper a richiedere la lista delle tratte/aeroporti e fare richiesta alle api esterne, **elaborating_tratte_time** e **elaborating_aeroporti_time**, le quali invece misurano il tempo che impiega il servizio elaboratore a ricevere i messaggi da kafka e trovare le email degli utenti interessati rispettivamente alle tratte/offerte su aeroporti. È inoltre presente un **alertmanager** che avviserà lo sviluppatore se il sistema rimane down per più di sei ore.

È stato quindi implementato un ulteriore microservizio, chiamato **SLAmanager**, il quale gestisce un database in cui memorizza le informazioni relative agli SLA (Service Level Agreement) in modo persistente. Tale microservizio espone le seguenti API all'utente:

- **Get_sla**

La quale ritorna la lista delle metriche attualmente presenti nel database e delle quali prometheus sta effettuando lo scraping.

- **Aggiungi_metrica**

La quale permette all'utente, dato il nome, una soglia e un valore desiderato, di poter inserire una nuova metrica nel database. Prima dell'inserimento nel database viene effettuato un controllo sulla validità o meno del nome della metrica, tale validità viene effettuata tramite query a Prometheus e verifica che la risposta dal server non sia vuota.

- **Elimina_metrica**

La quale permette all'utente di eliminare dal database una metrica dato il nome.

- **Get_valori_attuali**

La quale ritorna all'utente i valori attuali delle metriche nel SLA

- **Get_valori_desiderati**

La quale ritorna i valori desiderati delle metriche nel SLA

- **Get_stato_attuale**

La quale ritorna lo stato attuale delle metriche rispetto al SLA (violazione: true/false)

- **Get_violazioni_tempo**

La quale ritorna il numero di violazioni della soglia da parte di una metrica rispetto al SLA nelle precedenti “x” ore (valore inserito dall’utente)

- **Get_probabilità_violazioni**

La quale ritorna la probabilità di violazione della soglia da parte di una metrica rispetto al SLA nei prossimi “x” minuti (valore inserito dall’utente). Tramite **ARIMA** è stato creato un modello di previsione lavorando su dei dati di test contenenti i campioni degli ultimi 70 minuti, abbiamo allenato il modello sui campioni dei primi 60 minuti e confrontato il risultato con gli ultimi 10, abbiamo ripetuto l’operazione fino a trovare un modello adatto, a questo punto è possibile ottenere una previsione per i prossimi x minuti e ottenere quindi la probabilità di violazione della soglia negli stessi.

In base alla differenza tra la previsione e il valore reale è stato aggiustato il modello reale,

La parte del testing risulta commentata.



Come metrica è stata scelta: `node_memory_MemAvailable_bytes`, in quanto presenta un numero di campioni sufficienti per testare il modello (metriche come lo scraping time e il tempo di elaborazione vengono calcolate una volta ogni 24h), inoltre analizzando l’andamento temporale delle altre metriche presenti nel SLA, abbiamo notato che il loro andamento fosse pressoché costante.

L’SLA è inizializzato con le seguenti metriche:

- `node_network_receive_errs_total`

- node_network_transmit_errs_total
- node_memory_MemAvailable_bytes
- kafka_consumer_group_members
- count(kube_persistentvolume_created)
- count(kube_service_created)
- prometheus_sd_kubernetes_http_request_duration_seconds_count
- container_cpu_usage_seconds_total{pod=~"rules.*"}
- scraping_time
- elaborating_tratte_time
- elaborating_aeroporti_time

Build & Deploy

Dal repository Github è possibile effettuare il **docker-compose up** tramite il quale verranno caricate le immagini dei microservizi su docker, verranno inoltre avviati i container e i volumi.

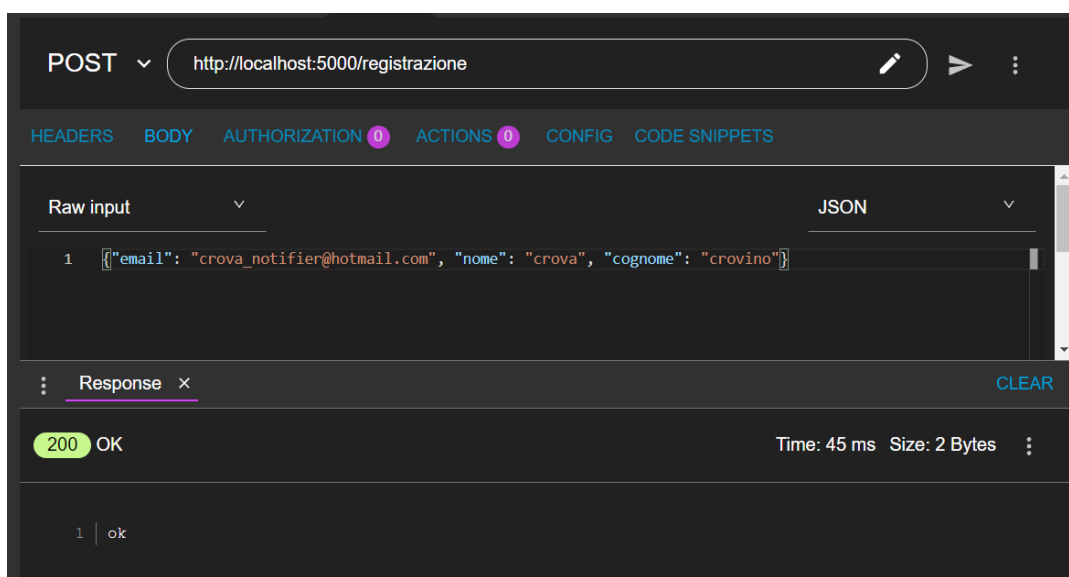
Per il deploy dell'applicazione in kubernetes basterà accedere alla directory Services/Kubernetes ed effettuare il comando **kubectl apply -f ./mysql** per applicare al cluster i file di configurazione.

Per interagire con l'applicazione è stato implementato un **Ingress**, grazie ad esso è possibile interagire col servizio user_controller tramite URI **localhost:80/user/***, mentre per interagire col servizio SLAmanager **localhost:80/slamanager/***.

Se si dovessero riscontrare problemi con l'Ingress, basterà effettuare il seguente comando: **kubectl port-forward service/user-controller-service 5000:5000** per poter interagire con user_controller tramite **localhost:5000/***; mentre **kubectl port-forward service/slamanager-service 5014:5014** per poter interagire con SLAmanager tramite **localhost:5014/***.

Per interagire con l'applicazione è sufficiente effettuare richieste tramite un'applicazione come **Postman**, seguendo gli esempi sotto.

Registrazione



Iscrizione a una tratta

POST http://localhost:5000/Insert_tratta

HEADERS BODY AUTHORIZATION ACTIONS CONFIG CODE SNIPPETS

Raw input JSON

```
1 { "email": "elesamati@gmail.com", "origine": "CTA", "destinazione": "FCO", "budget": 300, "adulti": 1 }
2
```

Response CLEAR

200 OK Time: 182 ms Size: 21 Bytes

1 | Iscrizione effettuata

Esempio e-mail ricevuta (esempio iscrizione alla tratta CTA-FCO)

notifier.dsbd@gmail.com 12:31 (0 minuti fa) ☆ 😊 ↩ ⋮

a me ▾

...

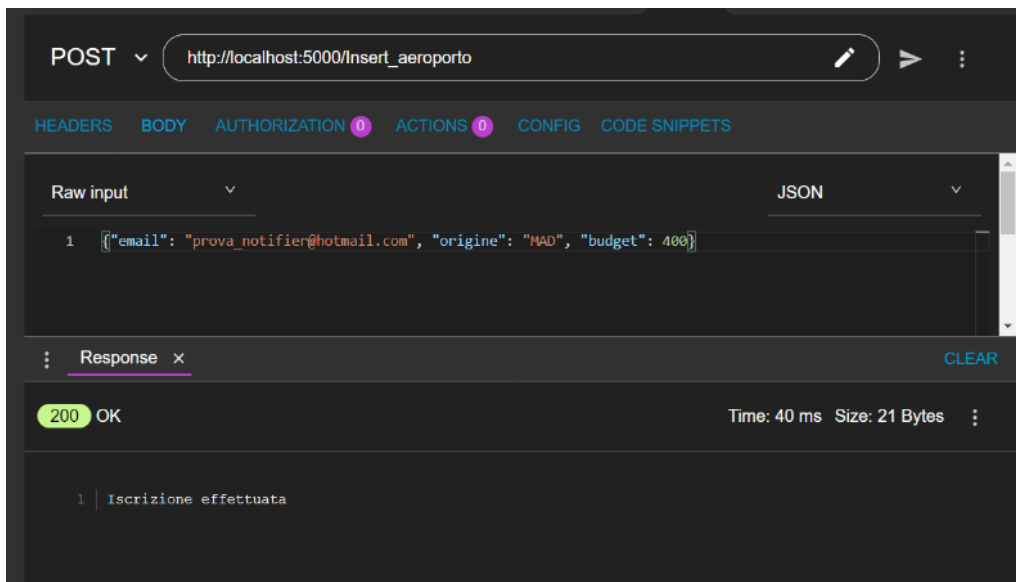
Caro elesamati@gmail.com,
questa e' l'offerta da te richiesta

```
{ "origin": "CTA", "destination": "FCO", "price": 43.89, "adulti": 1, "partenza": "2024-02-01T15:25:00" } { "origin": "CTA", "destination": "FCO", "price": 43.89, "adulti": 1, "partenza": "2024-02-01T06:00:00" }  
{ "origin": "CTA", "destination": "FCO", "price": 43.89, "adulti": 1, "partenza": "2024-02-01T10:15:00" }  
{ "origin": "CTA", "destination": "FCO", "price": 43.89, "adulti": 1, "partenza": "2024-02-01T19:10:00" }  
{ "origin": "CTA", "destination": "FCO", "price": 43.89, "adulti": 1, "partenza": "2024-02-01T12:00:00" }
```

Disiscrizione da una tratta

The screenshot shows the Swagger UI interface for a REST API. At the top, a POST request is defined for the endpoint `http://localhost:5000/Disiscrizione_tratta`. The request body is a JSON object with the following fields: `"origine": "CTA"`, `"destinazione": "FCO"`, `"email": "elesamati@gmail.com"`, and `"adulti": 1`. The response status is `200 OK`, and the response message is `Disiscrizione effettuata`.

Iscrizione a un aeroporto

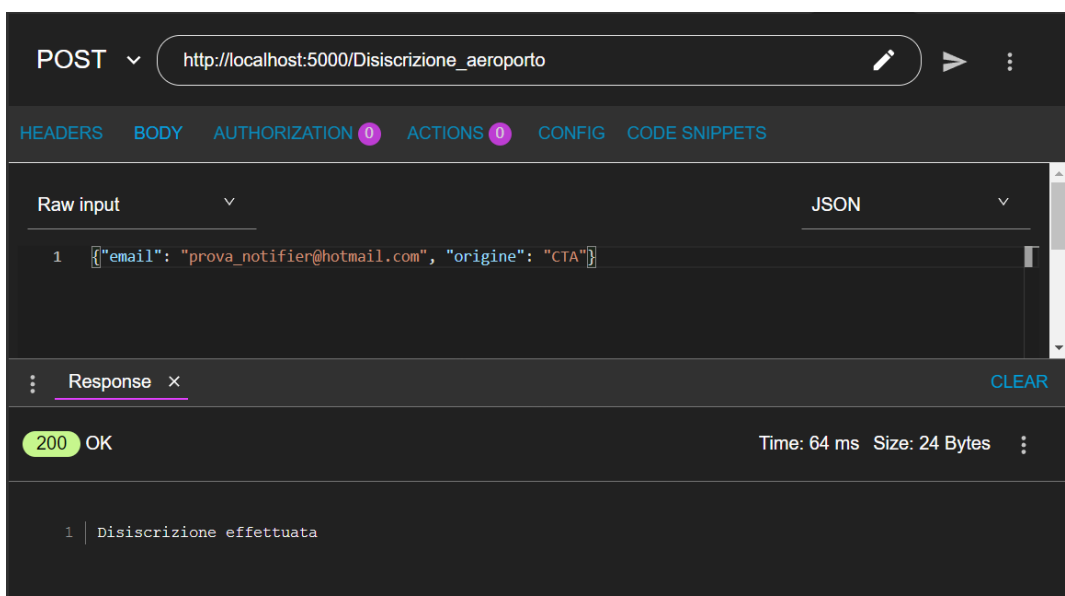


Esempio e-mail ricevuta (esempio iscrizione all'aeroporto di Madrid)

Caro elesamati@gmail.com ,
questa e' l'offerta da te richiesta
{ "origin": "MAD", "destination": "LPA", "price": 41.3, "partenza": "2024-02-01" } { "origin": "MAD", "destination": "OPO", "price": 48.58, "partenza": "2024-02-01" }
{ "origin": "MAD", "destination": "CDG", "price": 76.79, "partenza": "2024-02-01" }
{ "origin": "MAD", "destination": "PMI", "price": 77.54, "partenza": "2024-02-01" }
{ "origin": "MAD", "destination": "LIS", "price": 79.52, "partenza": "2024-02-01" }
{ "origin": "MAD", "destination": "RAK", "price": 82.9, "partenza": "2024-02-01" }
{ "origin": "MAD", "destination": "MXP", "price": 95.9, "partenza": "2024-02-01" }
{ "origin": "MAD", "destination": "LGW", "price": 99.54, "partenza": "2024-02-01" }
{ "origin": "MAD", "destination": "AMS", "price": 111.37, "partenza": "2024-02-01" }
{ "origin": "MAD", "destination": "FCO", "price": 134.12, "partenza": "2024-02-01" }

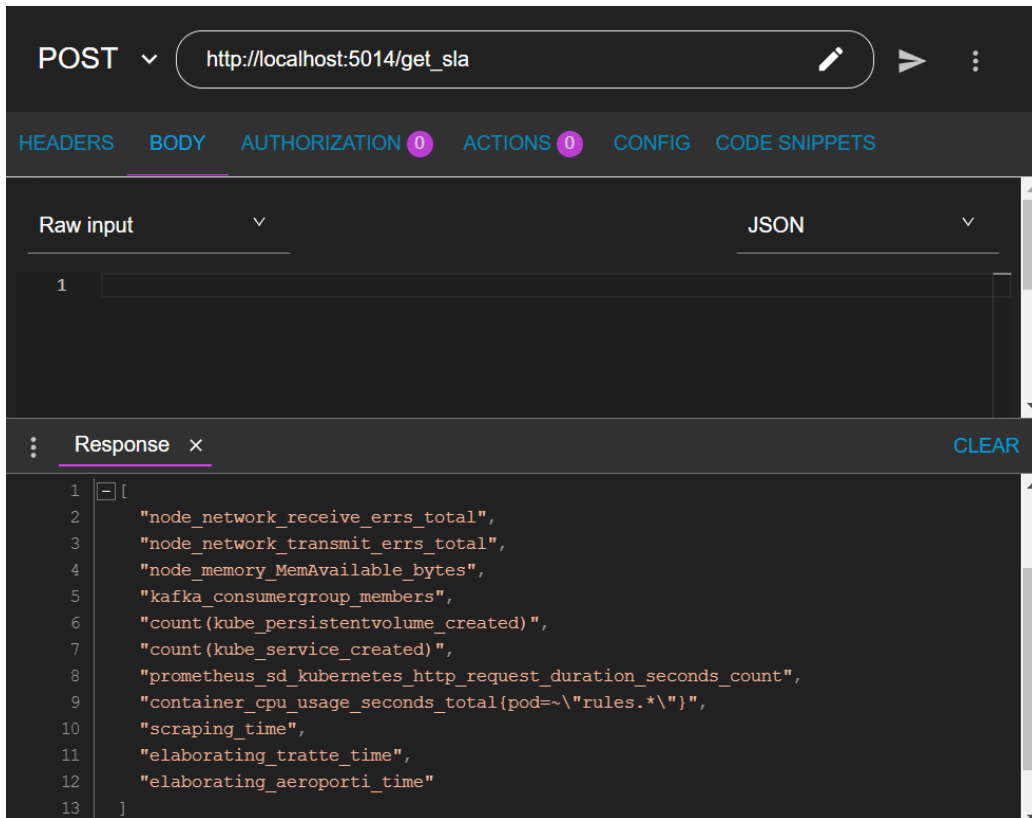
← Rispondi → Inoltra 😊

Disiscrizione da un aeroporto



In kubernetes vale lo stesso detto sopra ma è anche possibile comunicare con SLA manager:

Ottieni lista delle metriche presenti nell'SLA



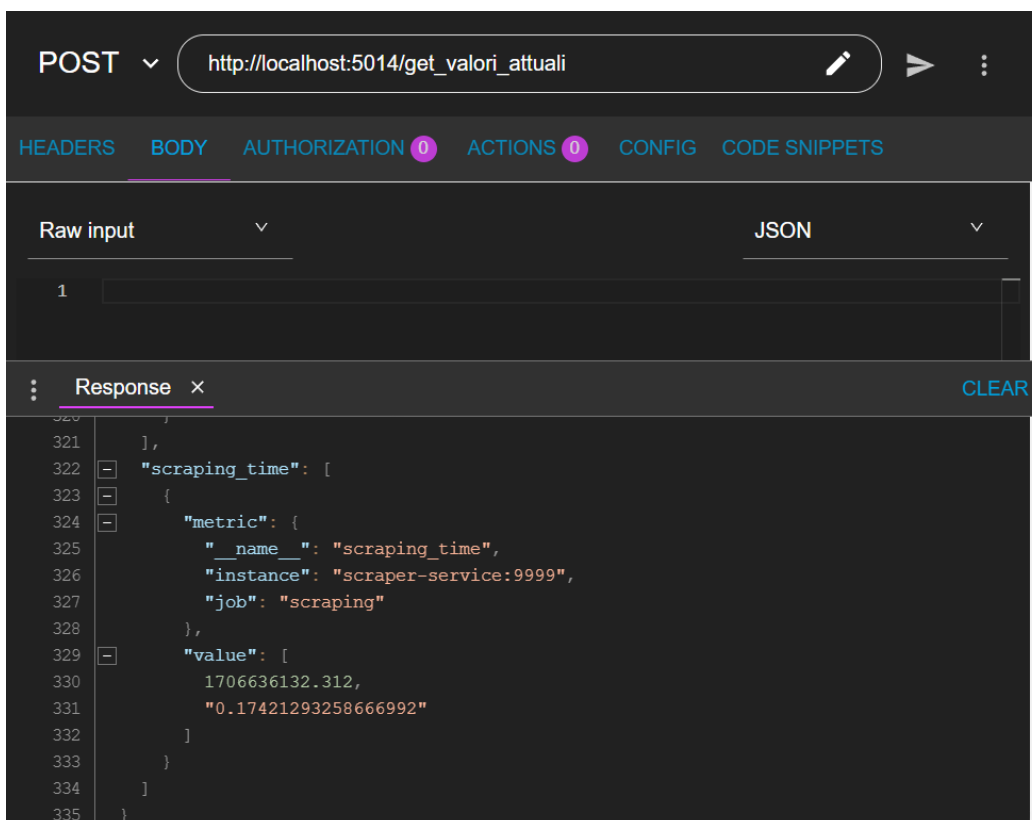
The screenshot shows a REST client interface with a POST request to `http://localhost:5014/get_sla`. The response is a JSON array of metric names.

```
POST http://localhost:5014/get_sla

Raw input
1

JSON
1
2
3
4
5
6
7
8
9
10
11
12
13
[
  "node_network_receive_errs_total",
  "node_network_transmit_errs_total",
  "node_memory_MemAvailable_bytes",
  "kafka_consumer_group_members",
  "count(kube_persistentvolume_created)",
  "count(kube_service_created)",
  "prometheus_sd_kubernetes_http_request_duration_seconds_count",
  "container_cpu_usage_seconds_total{pod=~\"rules.*\"}",
  "scraping_time",
  "elaborating_tratte_time",
  "elaborating_aeroporti_time"
]
```

Ottieni valori attuali delle metriche



The screenshot shows a REST client interface with a POST request to `http://localhost:5014/get_valori_attuali`. The response is a JSON object containing a list of metrics and their current values.

```
POST http://localhost:5014/get_valori_attuali

Raw input
1

JSON
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
{
  "scraping_time": [
    {
      "metric": {
        "name": "scraping_time",
        "instance": "scraper-service:9999",
        "job": "scraping"
      },
      "value": [
        1706636132.312,
        "0.17421293258666992"
      ]
    }
  ]
}
```

Eliminazione metrica da SLA

POST `http://localhost:5014/elimina_metrica`

HEADERS BODY AUTHORIZATION 0 ACTIONS 0 CONFIG CODE SNIPPETS

Raw input JSON

```
1 [{"nome": "kafka_consumergroup_members"}]
```

Response × CLEAR

200 OK Time: 37 ms Size: 17 Bytes

```
1 | metrica eliminata
```

Inserimento metrica nel SLA

POST `http://localhost:5014/aggiungi_metrica`

HEADERS BODY AUTHORIZATION 0 ACTIONS 0 CONFIG CODE SNIPPETS

Raw input JSON

```
1 [{"nome": "kafka_consumergroup_members", "soglia": 2, "desiderato": 2}]
```

Response × CLEAR

200 OK Time: 44 ms Size: 16 Bytes

```
1 | metrica aggiunta
```

Ottieni valori desiderati

POST ⌵ http://localhost:5014/get_valori_desiderati ✎ ➤ ⋮

HEADERS BODY AUTHORIZATION 0 ACTIONS 0 CONFIG CODE SNIPPETS

Raw input ⌵ JSON ⌵

1

⋮ Response × CLEAR

```
2  [
3    "node_network_receive_errs_total",
4    0
5  ],
6  [
7    "node_network_transmit_errs_total",
8    0
9  ],
10 [
11   "node_memory_MemAvailable_bytes",
12   884879000
13 ],
```

Ottieni stato attuale rispetto alle soglie (violazione:true/false)

POST ⌵ http://localhost:5014/get_stato_attuale ✎ ➤ ⋮

HEADERS BODY AUTHORIZATION 0 ACTIONS 0 CONFIG CODE SNIPPETS

Raw input ⌵ JSON ⌵

1

⋮ Response × CLEAR

200 OK Time: 93 ms Size: 455 Bytes ⋮

```
1  {
2    "container_cpu_usage_seconds_total{pod=~\"rules.*\"}": false,
3    "count(kube_persistentvolume_created)": false,
4    "count(kube_service_created)": false,
5    "elaborating_aeroporto_time": false,
6    "elaborating_tratte_time": false,
7    "kafka_consumergroup_members": false,
8    "node_memory_MemAvailable_bytes": true,
9    "node_network_receive_errs_total": false,
10   "node_network_transmit_errs_total": false,
11   "prometheus_sd_kubernetes_http_request_duration_seconds_count": true,
12   "scraping_time": false
13 }
```

Ritorna il numero di violazioni rispetto alle soglie avvenute nell'arco di "x" ore

```
POST http://localhost:5014/get_violazioni_tempo

Raw input: [{"ore":3}]

JSON: [{"ore":3}]

Response: {
  "container_cpu_usage_seconds_total{pod=~\"rules.*\"}": "0",
  "count(kube_persistentvolume_created)": "0",
  "count(kube_service_created)": "0",
  "elaborating_aeroporti_time": "0",
  "elaborating_tratte_time": "0",
  "kafka_consumergroup_members": "0",
  "node_memory_MemAvailable_bytes": "503",
  "node_network_receive_errs_total": "0",
  "node_network_transmit_errs_total": "0",
  "prometheus_sd_kubernetes_http_request_duration_seconds_count": "2015",
  "scraping_time": "0"
}
```

Calcolo probabilità di violazione di una soglia nei prossimi "x" minuti (metrica node_memory_MemAvailable_bytes)

```
POST http://localhost:5014/get_probabilità_violazioni

Raw input: [{"minuti":20}]

JSON: [{"minuti":20}]

Response: 200 OK
Time: 3100 ms Size: 53 Bytes

1 | probabilità violazione nei prossimi 20 minuti = 1.0%
```