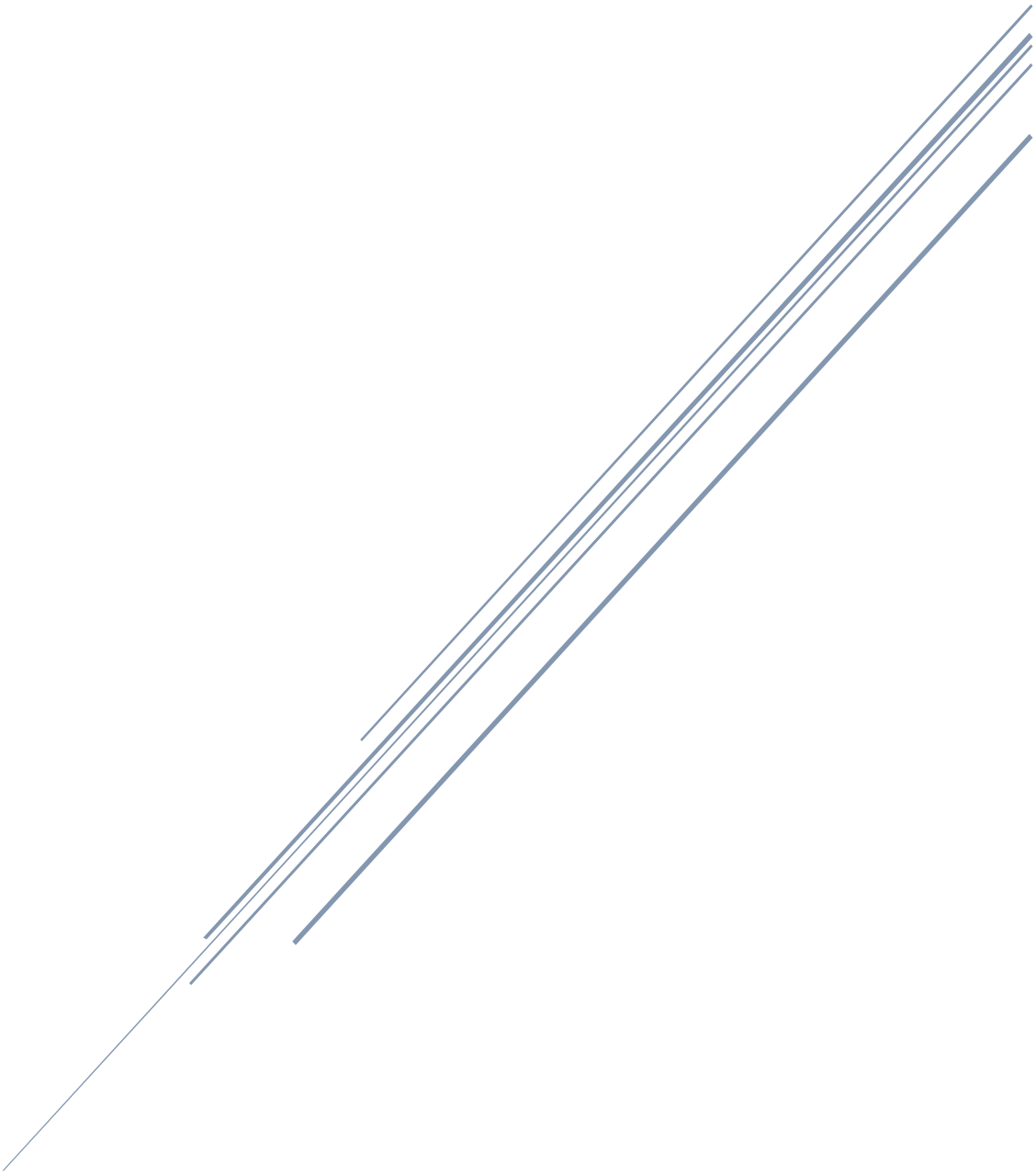
PROGETTO DSBD: MYTRAFFIC

A.A 2023/2024



Fabio Castiglione, Giovanni Caschetto

Sommario

1 ABSTRACT

Il progetto si propone come obiettivo la realizzazione di un'applicazione a microservizi distribuiti che permetta ad un utente di tracciare i tempi di percorrenza dei propri percorsi preferiti. L'utente, dopo essersi registrato alla piattaforma, può inserire nella sua area privata informazioni inerenti le tratte da effettuare in auto, in modo da ottenere informazioni relative ai tempi di percorrenza in base al traffico presente. Le informazioni richieste per ogni tratta sono:

- **Città, CAP** e **Indirizzo di partenza**;
- **Città, CAP** e **Indirizzo di arrivo**;
- **Orario di partenza desiderato**;
- **Soglia di notifica desiderata**;
- **Preferenza ricezione notifiche sugli anticipi**.

L'insieme di queste informazioni viene definito **alert**. Dopo l'aggiunta dell'alert, il sistema periodicamente effettuerà delle verifiche tramite le **Bing maps API** sulla situazione del traffico dei percorsi scelti da parte degli utenti: se il tempo di percorrenza risulta essere superiore del tempo nominale di percorso di una quantità soglia definita dall'utente, il sistema procederà con la notifica all'utente via l'email fornita in fase di registrazione (con tempo nominale si intende un tempo medio restituito dalla API con orario di partenza scelto dall'utente). È possibile essere anche notificati sugli anticipi della stessa quantità soglia.

Esempio**:** il tempo di percorrenza medio tra Catania e Messina è 60 minuti e l'utente imposta soglia di notifica 10 minuti, il sistema invierà la notifica se il tempo di viaggio è superiore o uguale a 70 minuti. In caso l'utente volesse essere notificato sugli anticipi, il sistema manderà la notifica se il tempo di viaggio previsto è inferiore a 50 minuti.

Sono state implementate altresì le funzionalità di unscribe da un'alert oppure la modifica di un alert, dove l'utente rispettivamente potrà cancellare la propria sottoscrizione a quella tratta oppure modificare la soglia o preferenza nella ricezione delle notifiche.

2 STRUTTURA DELL'APPLICAZIONE

2.1 MICROSERVIZI IMPLEMENTATI

La seguente figura illustra i microservizi applicativi implementati e le modalità di comunicazione

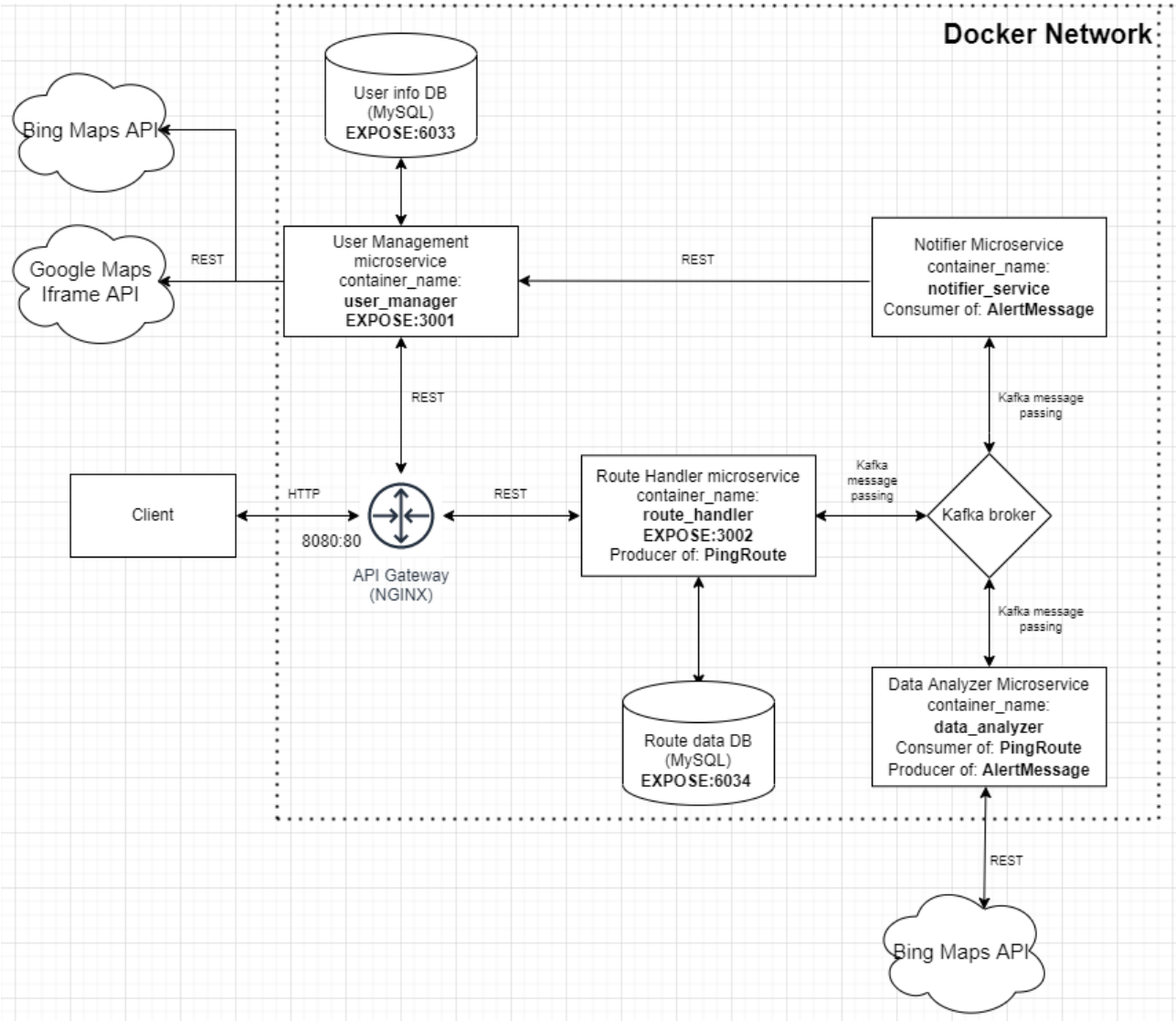


Figura 2.1 Struttura microservizi di business logic

2.1.1 MICROSERVIZIO 1: NGINX (API GATEWAY)

NGINX è un webserver leggero e ad alte prestazioni in grado di svolgere operazioni di load balancing e di reverse proxing. In questo contesto NGINX ha il ruolo di **reverse proxing**: esso rappresenta il punto di ingresso del sistema da parte degli utenti in quanto quest'ultimi si interfacceranno solamente con questo elemento, rendendo trasparente quindi l'architettura a microservizi. Esso espone la porta 8080 mappata sulla porta 80 interna al container. Digitando quindi sul browser l'indirizzo 127.0.0.1:8080 ci si interfaccerà con questo microservizio.

Nota: per evitare di usare localhost nella barra degli indirizzi e quindi fraintendere l'indirizzo locale del container con quello della macchina dell'utente, è stata effettuata una mappatura sul file etc/hosts dell'indirizzo 127.0.0.1 con **my.traffic.com**. Quindi per accedere al servizio è sufficiente digitare my.traffic.com/...

Lo User Manager Microservice è il microservizio che si occupa di mostrare le varie pagine all'utente: se quest'ultimo, ad esempio, volesse accedere alla pagina di login, egli digiterebbe il seguente URL:

http://my.traffic.com:8080/login

nel file .conf di NGINX è presente una direttiva di questo tipo:

```
server {
    listen 80;
    server_name my.traffic.com;

    location = /login {

        proxy_pass http://user_manager:3001/login;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        sub_filter 'href="/static/' 'href="http://my.traffic.com:3001/static/';
        sub_filter 'src="/static/' 'src="http://my.traffic.com:3001/static/';
        sub_filter_once off;

    }
…
```

Snippet 2.1 Esempio di direttiva NGINX

Digitando quindi l'URL sopra indicato, NGINX eseguirà il redirect presso la location /login: la direttiva **proxy_pass** indica l'URL verso il quale reindirizzare la richiesta http. Nel caso in cui la route punta a mostrare pagine web, allora anche le richieste http per il recupero delle risorse js e css deve essere indirizzato al microservizio corretto: per fare ciò si utilizza la direttiva **sub_filter** che, ad esempio, indirizza tutte le richieste del tipo 'href="/static/' verso 'href="http://my.traffic.com:3001/static/' in modo tale che gli stylesheet abbiano l'attributo src impostato correttamente. La direttiva **sub_filter_once off** indica infine che sono ammesse più operazioni di filtering durante il redirect presso questa location. Se quindi si devono recuperare più file css, si avrà che NGINX effettuerà più operazioni di filtering su quell'indirizzo.

Nota: la direttiva proxy pass contiene come URL il **nome del container** che soddisfa la richiesta. Quest'operazione è ammessa poiché tutti i vari microservizi fanno parte della **rete interna docker**, denominata *my_traffic_network,* ove al suo interno è presente un DNS che è in grado di associare la corrispondenza nome-indirizzo_ip del container. Non è possibile, tuttavia, usare nella direttiva di sub_filter il nome del container in quanto esso non è risolvibile al di fuori della rete docker.

2.1.2 MICROSERVIZIO 2: USER MANAGER

Il secondo microservizio è responsabile di gestire le interazioni con l'utente. Si occupa di restituire all'utente le pagine html da mostrare all'utente, nonché effettuare le operazioni di registrazione e di autenticazioni. Come visibile dalla Figura 2.1, il suddetto microservizio è connesso ad un database MySQL dove vengono conservate tutti i dati di registrazione dell'utente. Il microservizio è stato implementato come un'applicazione Flask che espone un insieme di **route** ognuna appartenente ad una seguente tipologia:

- **Route accessibili all'utente**: che restituiscono pagine html oppure gestiscono i form inviati dall'utente;
- **Route di servizio**: sono route esposte ad altri microservizi interni alla rete docker. Rientrano in questa categoria, ad esempio, la route di fetch delle metriche del container

Parametri del container

Nome del container	user_manager
Variabili di ambiente	– Stringa di connessione al database
Dipendenze	– UserDB
Porte esposte	3001
API Implementate	– **/login:** o **GET:** ritorna la pagina di login all'utente o **POST:** invia il form di login e verifica le credenziali – **/register:** o **GET:** ritorna la pagina di registrazione all'utente o **POST:** invio del form di registrazione. Verifica e effettua

	il salvataggio dei dati utente nel database.
	- **/privateArea**:
	  - **GET & POST:** reindirizzamento all'area privata utente dopo l'invio del form di login (POST) oppure a seguito di click sul rispettivo link se la sessione non risulta scaduta (GET)
	- **/editAlert/<alert_id>:**
	  - **GET:** reindirizzamento verso la pagina "privateArea.html" con i dati precompilati del rispettivo alert
	- **/getEmail/<user_id>:**
	  - **GET:** ritorna l'indirizzo email dato l'id di un utente.

2.1.3 MICROSERVIZIO 3: ROUTE HANDLER

Il terzo microservizio è responsabile del salvataggio e dell'invio dei dati delle tratte e delle sottoscrizioni dell'utente. Come il precedente, anche questo microservizio è stato implementato come un'applicazione Flask anche se quest'ultimo possiede una seconda modalità di comunicazione con gli altri microservizi: la comunicazione tra User manager e Route handler avviene tramite REST: un esempio di comunicazione è l'invio del form presente nella privateArea contenenti i dati di sottoscrizione ad una tratta: il route handler esponendo l'endpoint */sendData*, all'invio del form, i dati verranno ricevuti dal microservizio e salvati nel database. Altri endpoint esposti da parte del microservizio sono */deleteAlert/<subscription_id>* e */editAlert/<subscription_id>* che rispettivamente permettono la cancellazione e la modifica di una sottoscrizione. Il microservizio, inoltre, ad intervalli regolari (per scopi di sviluppo impostato ad 1 minuto, poi impostata a 15 minuti), effettua una query al proprio database recuperando tutte le sottoscrizioni registrate e per ognuna viene preparato un messaggio kafka con topic *PingRoute* che verrà inviato al microservizio successivo

che si occuperà di elaborarlo. L'attività di query del database è effettuata da un thread in background eseguito ogni 15 minuti. Il database a cui fa affidamento il Route Handler è di tipo SQL contenente al suo interno due tabelle: una denominata **routes** e una **subscriptions**. Nella prima vengono contenuti tutti i parametri della tratta:

- **Dati della partenza**: departureCity, departureCAP, departureAddress, departureLatitude, departureLongitude;
- **Dati di arrivo**: arrivalCity, arrivalCAP, arrivalAddress, arrivalLatitude, arrivalLongitude.

Nella seconda tabella sono contenuti i dati di sottoscrizione dell'utente:

- **ID della tratta sottoscritta;**
- **ID dell'utente sottoscrivente**;
- **Orario di partenza desiderato**;
- **Soglia di notifica**;
- **Preferenza sugli anticipi**.

Questa struttura dati possiede i seguenti vantaggi:

- Nel caso in cui un utente si voglia sottoscrivere ad una tratta già presente nel database, si andrà ad inserire solo la sottoscrizione associata alla tratta. Si andrà a ridurre quindi il numero di entry all'interno della tabella *routes* del database;
- Riduzione del numero di query alle API esterne: nel caso in cui due o più utenti selezionano lo stesso orario di partenza su una stessa tratta, la query alle API esterne sarà sempre unica e i dati restituiti saranno confrontati successivamente con i rispettivi parametri di notifica.

Di seguito la struttura di un messaggio kafka con topic *PingRoute*:

```
class message:
   route_id = 0
   departureLatitude = 0,
   departureLongitude = 0,
   arrivalLatitude = 0,
   arrivalLongitude = 0,
   subscriptionsList = ()
```

Snippet 2.2 Struttura di un messaggio kafka con topic "PingRoute"

NOTA: Per ridurre il numero di imprecisioni di posizione da parte delle API di Bing Maps si è deciso di richiedere all'utente, oltre a città e indirizzo di partenza e destinazione anche il CAP. Tutti i dati di partenza e destinazione sono stati convertiti in coppie di latitudine / longitudine.

Parametri del container

Nome del container	route_handler
Variabili di ambiente	- Stringa di connessione al database
Dipendenze	- RoutesDB - Kafka broker
Porte esposte	3002

API Implementate	- **/getData/<user_id>:**
	  - **GET:** ritorna tutte le sottoscrizioni di un utente dato user_id
	- **/deleteAlert/alert_id/route_id:**
	  - **GET:** cancella la sottoscrizione di identificativo alert_id e nel caso non ci siano più sottoscrizioni per la tratta con route_id, viene cancellata anche la tratta.
	- **/getSubscriptionData/alert_id:**
	  - **GET:** ritorna i dati di una sottoscrizione e della relativa tratta
	- **/changeAlert**
	  - **POST:** modifica l'alert con i dati inseriti dall'utente
	- **/sendData**
	  - **POST:** verifica i dati ricevuti per la sottoscrizione ad una tratta. Salva nel database la tratta e la relativa sottoscrizione

Topic Kafka coinvolti	- **PingRoute (Producer):** topic contenente la tratta e le sottoscrizioni da ispezionare.

2.1.4 MICROSERVIZIO 4: DATA PROCESSER

Il quarto microservizio è responsabile di elaborare i messaggi ricevuti dal Route Handler. Questo microservizio si tratta di un consumer Kafka dei messaggi inviati dal Route Handler e contemporaneamente un producer Kafka dei messaggi destinati al Notifier Microservice. Per ogni messaggio ricevuto, il microservizio recupera gli orari sottoscritti dai vari utenti ed effettua per ognuno di quest'ultimi una richiesta alle API di Bing Maps. Ricevuti i tempi di percorrenza, si effettuano i vari calcoli di verifica in base alla soglia di notifica scelta dall'utente e nel caso di riscontro positivo, viene creato un messaggio kafka con topic *AlertMessage* includendo lo *user_id* dell'utente sottoscrivente, la *route_id* sottoscritta e un breve messaggio informativo da inviare via mail.

Parametri del container

Nome del container	data_analyzer
Variabili di ambiente	– Bing Maps API Key
Dipendenze	– Kafka broker
Porte esposte	N/A
Topic coinvolti	– **PingRoute** (Consumer): topic contenente la tratta e le sottoscrizioni da ispezionare. – **AlertMessage** (Producer): topic contenente l'utente e la tratta da notificare. Include anche il messaggio da inviare via email.

2.1.5 MICROSERVIZIO 5. NOTIFIER MICROSERVICE

L'ultimo microservizio è responsabile della notifica dell'utente sottoscritto. Il microservizio è un kafka consumer di messaggi con topic *AlertMessage*. Alla ricezione di ogni messaggio, il microservizio si occupa di ottenere la mail dell'utente a partire dallo user_id ricevuto nel messaggio tramite una richiesta REST allo User Manager. Ricevuto l'indirizzo email, il microservizio compila i campi della mail e la invia all'utente.

Parametri del container

Nome del container	notifier_service
Variabili di ambiente	– Username di accesso al mailing server (email mittente) – Password di accesso al mailing server – SMTP server port – SMTP server host
Dipendenze	– Kafka broker
Porte esposte	N/A
Topic coinvolti	– **AlertMessage** (Consumer): topic contenente l'utente e la tratta da notificare. Include anche il messaggio da inviare via email.

2.2 MODALITÀ DI COMUNICAZIONE

Si analizzino più nel dettaglio le modalità di comunicazione adoperate nel sistema.

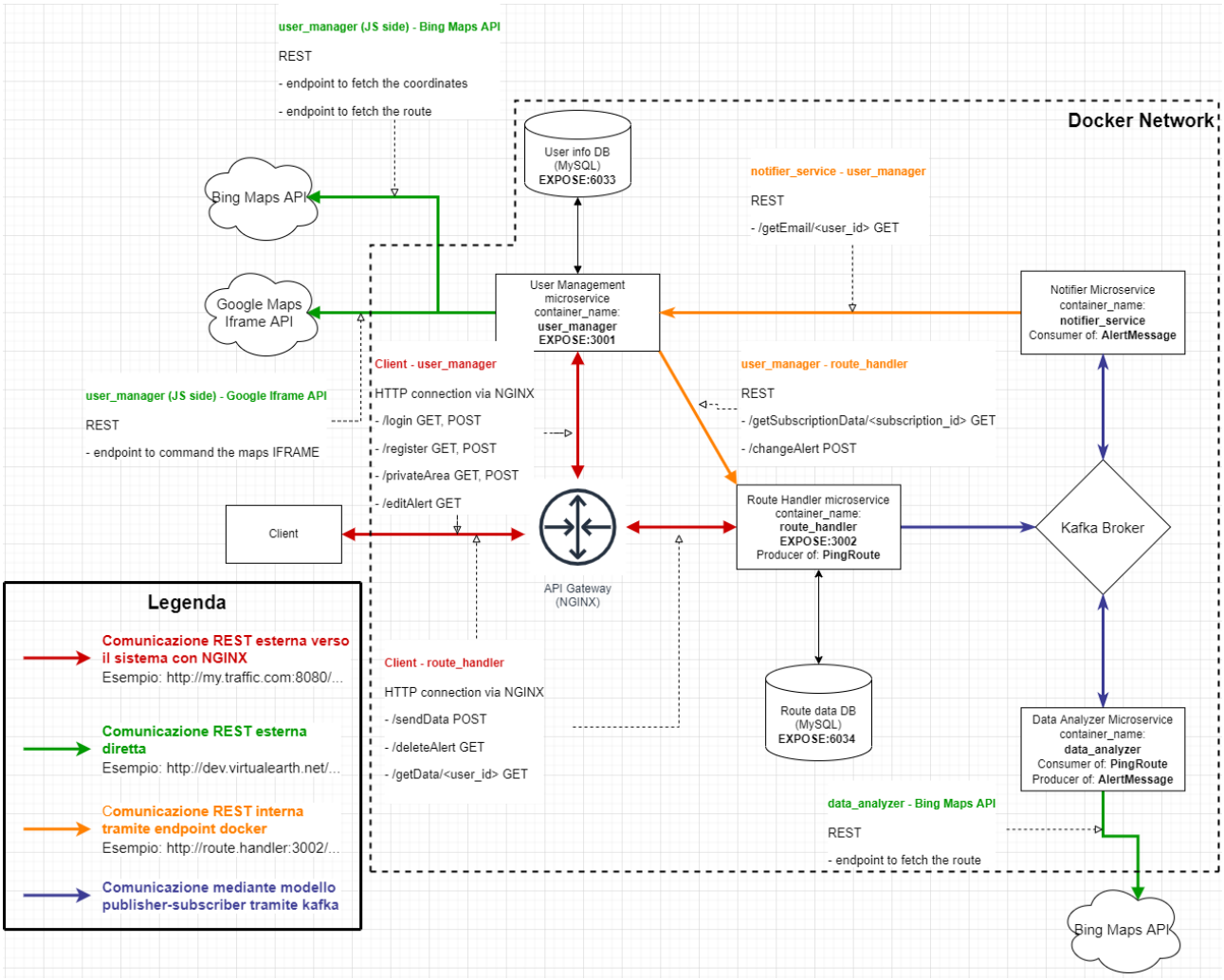


Figura 2.2 Schema di comunicazione tra i vari microservizi

Le modalità di comunicazione utilizzate sono di tipo **REST** e **Publishing subscriber**. Le comunicazioni di tipo REST sono di diverse categorie indicate dalle varie colorazioni:

- **Comunicazione REST esterna tramite NGINX** si intendono tutte le richieste che provengono dai vari client utente verso il sistema. Come esposto nel paragrafo precedente, NGINX ha il ruolo di inoltrare la richiesta HTTP dall'utente verso un particolare microservizio. Fanno parte di questo tipo di comunicazioni le richieste delle pagine html, inoltrate poi allo User Manager, oppure richieste di cancellazioni di alert (/deleteAlert), invio del form di sottoscrizione (/sendData) oppure ottenimento degli alert sottoscritti da parte dell'utente da visualizzare nella pagina delle sottoscrizioni (/getData). Quest'ultime tre richieste vengono inoltrate al Route Handler.

- **Comunicazione REST esterna diretta**: si intendono richieste HTTP dirette verso endpoint esterni al sistema quali ad esempio Bing Maps API o Google Maps Iframe API. Queste richieste possono essere effettuate direttamente da un microservizio, come ad esempio il Data Analyzer, oppure da script JavaScript integrati nella pagina html.
- **Comunicazione REST interna tramite endpoint docker**: si intendono richieste effettuate per richiedere un servizio / dato da un microservizio all'interno della rete docker. I container all'interno della rete docker sono identificabili tramite il loro nome; pertanto, sono effettuabili richieste HTTP del tipo *http://user_manager:3001/getEmail/...* Nel sistema in questione, ad esempio, il Notifier Service avendo a disposizione l'user_id dell'utente sottoscritto, può richiedere allo User Manager la corrispondente email effettuando una richiesta a tale endpoint.
- **Comunicazione publisher-subscriber** con broker kafka: si tratta di un tipo di comunicazione che permette uno scambio di messaggi affidabile suddiviso in topic: ogni nodo riceverà messaggi solo del topic a cui risulta essere sottoscritto. I topic adoperati nella prima fase di sviluppo sono:
  - **PingRoute**: messaggio inviato dal route handler al data analyzer contenente la tratta da controllare e le relative sottoscrizioni;
  - **AlertMessage**: messaggio inviato dal data analyzer al notifier service per segnalare una violazione della sottoscrizione di un determinato utente.

Sebbene anche in questi due casi la comunicazione avviene sempre tra due entità, la scelta di usare una comunicazione indiretta è stata supportata dalle seguenti osservazioni:

- I messaggi inviati in questi contesti sono monodirezionali, cioè dopo che vengono inviati, a parte la ricezione di un ack, non richiedono una risposta che deve essere elaborata;
- I microservizi connessi al broker sono servizi che all'aumentare delle dimensioni del sistema necessitano di essere scalati. Questo porta ad avere un'architettura dove si possono avere gruppi di produttori e gruppi di consumatori dove il singolo topic verrà opportunamente suddiviso (sharding del topic). Ovviamente anche il broker sarà un servizio da scalare opportunatamente, in modo da non rappresentare un single point of failure.

I messaggi inviati tramite kafka non richiedono alcun tipo di persistenza; pertanto, il fattore di replicazione è stato impostato ad 1. Anche il numero di ack per messaggio è stato impostato ad 1 in quanto non sono messaggi critici che richiedono diversi tentativi di consegna.

3 GESTIONE DELLA QUALITY OF SERVICE (QOS)

Lo sviluppo della seconda fase dell'elaborato prevede l'implementazione di un sistema di monitoraggio di tipo whitebox dell'applicazione. Il monitoraggio whitebox prevede la cattura di metriche che permettono di verificare l'efficienza e lo stato del sistema. A tal proposito è stato integrato nel sistema **Prometheus**, un sistema di monitoraggio open-source usato per catturare delle metriche di sistema. Obiettivo di questa seconda fase è anche definire un Service Level Agreement (SLA), ossia un determinato livello di qualità, sulla base delle metriche esposte.

3.1 EVOLUZIONE DELL'ARCHITETTURA

L'architettura esposta in Figura 2.1 viene arricchita di altri nuovi microservizi per la gestione della QoS:

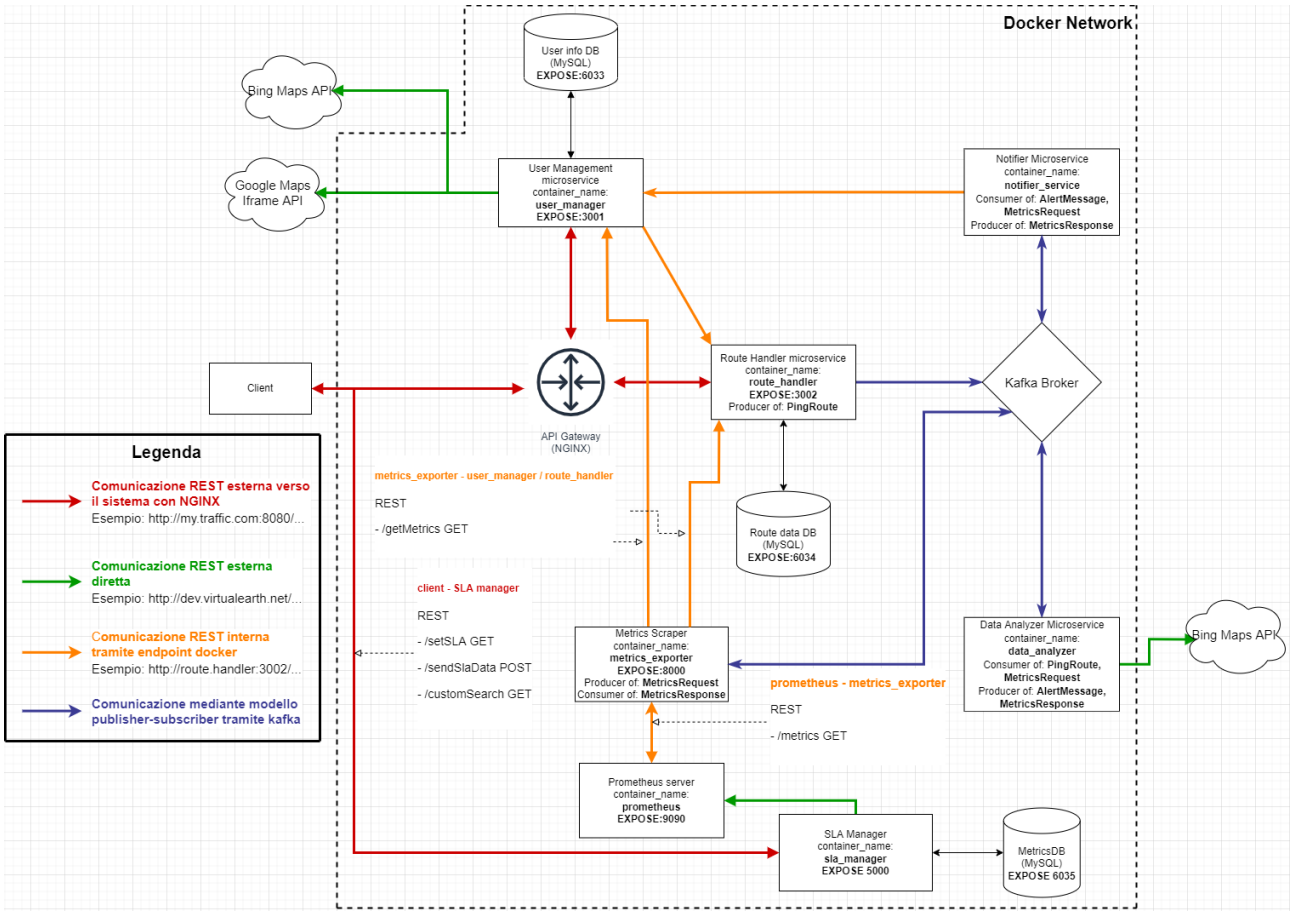


Figura 3.1 Architettura con integrazione di microservizi per la gestione della QoS

3.1.1 SCELTA DELLE METRICHE

Il primo passo per implementare un sistema di monitoraggio è la scelta delle metriche da monitorare: esse devono rappresentare un indice della salute e delle performance del sistema. Le metriche raccolte in questo contesto si suddividono nelle seguenti categorie:

- **Metriche del nodo:** rientrano in questa categoria le metriche estratte dall'host su cui il container è deployato. Sono state scelte le seguenti metriche per questa categoria:
    - **CPU_load:** Indica il tasso di utilizzo in % della CPU;
    - **RAM_load:** indica il tasso di utilizzo in % della RAM;
- **Metriche applicative**: rientrano in questa categoria metriche estratte da uno specifico microservizio per monitorare le performance di una determinata parte / funzione di un microservizio. Sono state scelte in questa categoria:
    - **API_response_time:** è il tempo di risposta medio da parte di un external service;
    - **Query_db_time:** è il tempo necessario per effettuare un'interrogazione completa del database (misurato solo nel route handler, in quanto è l'unico microservizio che effettua una query di tutti i dati).
- **Metriche di container**: sono metriche intrinseche del container. Rientrano in questa categoria:
    - **Restart_count:** numero di restart effettuati dal container (estratta ma non inseribile nella SLA dell'applicazione). Utile per capire il possibile numero di errori fatali che si possono verificare nei container e che prevedono il riavvio di quest'ultimo;
    - **Availabilty_index**: rapporto tra container_uptime e tempo totale espresso in %.

Le metriche raccolte permettono di avere un andamento generale delle caratteristiche del sistema: le metriche di nodo e le metriche applicative permettono di valutare il carico sui singoli nodi e quindi eventualmente intraprendere azioni di replicazione. Le metriche di container invece sono un'ottima misura della availability del sistema. In uno scenario applicativo reale, ogni determinato container deve avere le rispettive soglie di lavoro: ad esempio, container che effettuano calcoli continui, come ad esempio il route handler, presenteranno tasso d'uso della cpu più elevati, a differenza di servizi quali il notifier che eseguono sporadicamente e che in linea di massima restano sempre in ascolto di messaggi. In via esemplificativa, si è scelto che la metrica viene considerata in range se il valore massimo restituito tra i vari container risulta compresa tra i valori di massimo e minimo scelti. La seguente tabella mostra in quali microservizi vengono raccolte queste metriche.

Container	Metriche di nodo	Metriche applicative	Metriche di container
User Manager	– CPU_load – RAM_load	N/A	– Restart_count – Availabilty_index
Route Handler	– CPU_load – RAM_load	– Query_db_time	– Restart_count – Availabilty_index
Data Analyzer	– CPU_load – RAM_load	– API_response_time	– Restart_count – Availabilty_index
Notifier Service	– CPU_load – RAM_load	N/A	– Restart_count – Availabilty_index

Tabella 1 Metriche raccolte dai vari microservizi

3.1.2 RACCOLTA DELLE METRICHE

In riferimento alla Figura 3.1, i microservizi aggiunti per permettere un'operazione di monitoraggio del sistema sono i seguenti:

- **Metrics Exporter**: è il microservizio responsabile di raccogliere ad intervalli regolari le metriche dai vari microservizi. La raccolta di quest'ultime avviene secondo due modalità differenti:
  - **REST:** l'exporter effettua una richiesta HTTP diretta al microservizio, che espone un endpoint per il fetch delle metriche. Tutti i microservizi implementati come applicazioni Flask utilizzano questa modalità di raccolta metriche;
  - **Topic Kafka**: l'exporter è sia un produttore / consumatore kafka. Egli produce messaggi di topic *MetricRequest* che verrà ricevuto da tutti i microservizi adoperanti kafka, ad eccezione del route_handler che non risulta sottoscritto al topic in quanto per questo microservizio le metriche vengono recuperate tramite la modalità REST. Alla ricezione del messaggio *MetricRequest*, i microservizi effettueranno la misurazione delle proprie metriche e invieranno un messaggio kafka con *MetricResponse* a cui l'exporter è sottoscritto.

Parametri del container

Nome del container	metrics_exporter
Variabili di ambiente	N/A
Dipendenze	– Prometheus – Kafka broker
Porte esposte	8000
API Implementate	N/A
Topic Kafka coinvolti	– **MetricRequest (Producer):** topic che richiede le metriche ai microservizi adoperanti kafka; – **MetricResponse (Consumer):** topic che contiene le metriche ottenute in risposta al messaggio di MetricRequest

- **Prometheus server**: è un servizio che espone sulla porta 9090 una dashboard che mostra le metriche raccolte. Prometheus è configurato tramite un file noto come *prometheus.yml* contenente tutte le impostazioni di scraping e di elaborazione dati, nonché i *target* da cui recuperare le metriche.

```yaml
global:
  scrape_interval: 20s
  evaluation_interval: 30s

scrape_configs:

- job_name: 'my-exporter'
  static_configs:
  - targets: ['metrics_exporter:8000']
```

Snippet 3.1 File di configurazione di prometheus (prometheus.yml)

Lo Snippet 3.1 mostra la configurazione del server prometheus dell'applicazione. È stato impostato un tempo di scrape di 20 secondi: si avrà quindi che il server recupererà i dati dall'exporter ogni 20 secondi, mentre la valutazione (e aggiornamento) delle metriche avverrà

ogni 30 secondi. Tramite le direttive *job_name* è possibile specificare i vari exporter di metriche, in questo caso il metrics_exporter emetterà le metriche sulla porta 8000 al path default */metrics*.

- **SLA Manager:** è il microservizio che permette di fissare un certo Service Level Agreement come composizione di vari Service Level Objective. Un SLO consiste nel definire un intervallo di due valori a cui una specifica metrica deve appartenere. L'SLO è stato soddisfatto se il valore della metrica risulta in range. La SLA è soddisfatta quindi se tutti gli SLO sono soddisfatti. L'utente, grazie agli endpoint esposti, è in grado di interagire direttamente con il microservizio ed è in grado di visualizzare lo stato della SLA tramite una dashboard interattiva.

Parametri del container

Nome del container	sla_manager
Variabili di ambiente	N/A
Dipendenze	- Prometheus - MetricsDB
Porte esposte	5000

API Implementate	- **/setSLA**
	  - **GET:** ritorna la pagina di modifica della SLA, mostrando le metriche attualmente inserite e un riepilogo delle violazioni nelle ultime 1, 3 e 6 ore
	- **/sendSlaData:**
	  - **POST:** aggiorna i dati della SLA con le nuove preferenze
	- **/customSearch:**
	  - **POST:** mostra i dati delle violazioni della SLA nell'intervallo scelto dall'utente

Nella tabella seguente sono mostrati, per ogni metrica, i valori indicativi rilevati nel corso dello sviluppo:

Metrica	Valore di riferimento	Note
cpu_load	15-30%	Influenzate dalle performance dell'host che esegue l'applicativo
ram_load	50-60%	Influenzate dalle performance dell'host che esegue l'applicativo
api_response_time	~=0.25s	
query_db_time	~=0.0044s	
container_restart_count	11	Alcuni container effettuano una serie di restart in fase di startup causati dal broker che non si è avviato. Si intendono i restart totali.
availability_rate	> 98%	

4 DEPLOYMENT

Il deployment dell'applicazione è stato pensato per garantire un avvio pressoché automatizzato. Sono state definite all'interno dei vari dockerfile e nei docker-compose un set di costanti e vari caricamenti di dati prova in modo da rendere agevole il test dell'applicazione.

4.1 DATI DI PROVA

Ogni database possiede un set di dati di prova che ad ogni avvio del container vengono caricati nel database. Grazie, inoltre, all'utilizzo dei **volumi** docker, è possibile salvare tutti i dati che vengono creati nel corso del test, per poi essere ricaricati successivamente. Ogni database, quindi, possiede i seguenti elementi di persistenza:

- **Database_init.sql:** contiene un set di dati minimale sotto forma di query SQL che permette di avere appunto dei dati di test pronti anche al primo avvio del container. All'interno del dockerfile di un database viene specificato un comando di COPY del file di init nella directory del container */docker-entrypoint-initdb.d* dove verrà letto ed eseguito da MySQL ;

- **Volume docker:** viene eseguito un mapping tra una specifica directory del container con una porzione di spazio sul file system dell'host. Quindi se durante un test vengono inseriti dati nel database, questi persisteranno anche dopo che il container verrà cancellato.

4.2 VARIABILI DI AMBIENTE

Le variabili di ambiente sono dei dati costanti che vengono definite per ciascun microservizio e consentono di configurare parti dell'applicazione. Esempi di variabili di ambiente sono:

- **Stringhe di connessione ai database**;
- **Credenziali di accesso** a risorse esterne quali API KEY o al server di mailing
- **Endpoint di accesso** specifici (es. URL del server prometheus)

4.3 BUILD & DEPLOY

1. Impostare nel file etc/hosts alla voce dell'indirizzo 127.0.0.1 il dominio my.traffic.com come segue:

```
127.0.0.1   localhost my.traffic.com
127.0.1.1   your-hostname
```

Snippet 4.1 Configurazione del file etc/hosts

2. Effettuare la build tramite *sudo docker-compose up -d --build*
3. Tutti gli endpoint esposti nelle pagine precedenti sono accessibili digitando URL *http://my.traffic.com:8080/*... Oppure in alternativa usando il numero di porta del corrispondente microservizio.