Gruppo: Barbagallo Maria Lucia 1000027119, Caneva Gianfranco 1000025054

HOSPITAL PLATFORM

1. Introduzione

Hospital Platform è un'applicazione web che permette la gestione di pazienti da parte dei medici curanti e dal personale della reception, offrendo servizi quali:

- Per i medici è possibile effettuare la gestione di cartelle cliniche e il ricovero
- Per la reception è possibile inserire nuovi pazienti per ciascun medico.

Tale applicazione è stata sviluppata in java Spring e consiste in un insieme di microservizi indipendenti e cooperanti, deployati all'interno di container Docker che offrono a ciascun microservizio un ambiente completo per l'esecuzione del software, includendo librerie ed eventuali tool necessari per l'esecuzione.

Le immagini da cui i container vanno in run sono descritte nei corrispettivi Dockerfile associati a ciascun microservizio.

Per non dover avviare manualmente ciascun container, è stato utilizzato il tool docker-compose che permette di definire es eseguire un'applicazione multicontainer, a partire dalle direttive fornite nel file di configurazione Docker-Compose.yaml. Grazie a questo tool, i container verranno eseguiti tutti assieme, ciascuno nel proprio ambiente isolato e potranno comunicare con gli altri container attraverso una rete interna offerta da Docker, detta Docker Bridge.

Tuttavia, l'utilizzo di Docker compose comporta la necessità di eseguire manualmente molte azioni di gestione dei container. Dunque, per migliorare ulteriormente la gestione dell'applicazione e automatizzare molti processi di management dei container, Hospital Platform è stata sviluppata anche per poter essere deployata in un cluster Kubernetes.

Kubernetes è una piattaforma open source di orchestrazione e gestione di container che vengono raggruppati in un cluster che si compone di diverse tipologie di nodi, quali:

- POD: contiene uno più container dell'applicazione, i quali condividono l'indirizzo IP, il nome dell'host e altre risorse che li rendono visibili come un tutt'uno dall'esterno, seppur continuino ad essere container separati e indipendenti. Nella nostra applicazione, a ciascun POD è associato un solo container (n° POD= n° container).
- Service Nodes: Nodo con indirizzo IP sempre uguale e intermedia affinché dall'esterno si possa contattare il POD ad esso associato, anche nel caso in cui questo venga redeployato e quindi abbia cambiato indirizzo IP.
  Ci sono diversi tipi di Service Node. Nel progetto Hospital Platform, sono stati utilizzati, in particolari, Service Node del tipo:
  - Node Port: service che espone l'indirizzo IP anche all'esterno del cluster.
  - Cluster IP: service che espone l'indirizzo IP solo all'interno del cluster.
- Deployment: Nodo di più alto livello che fornisce le direttive per la creazione dei POD e dei relativi ReplicaSet al fine di mantenere il sistema nello stato desiderato dal programmatore. In particolare, crea nuovi replicaSet, rimuove quelli già esistenti e cambia lo stato attuale del cluster così da raggiungere lo stato target desiderato dal programmatore.
  Nell'applicazione Hospital Platform, vi è un deployment per ogni microservizio e per i database ad essi collegati.
- ReplicaSet: gestisce le repliche del POD corrispondente così da garantire che un certo numero di repliche siano sempre attive e funzionanti. Vengono creati automaticamente dai Deployment.

- StatefulSet: è un Deployment Statefull ovvero un Deployment che gestisce i POD che devono essere persistenti e che devono quindi mantenere uno stato.
  Nell'applicazione Hospital Platform implementata su Kuberanetes, Kafka è deployato in uno StatefulSet.
- DaemonSet: è controller che si assicura che i nodi POD siano in run e li monitora. Questi vengono usati anche come exporter di monitoraggio ed infatti nel progetto Hospital Platform questo tipo di nodo è stato usato proprio a tal fine.

I microservizi comunicano tra loro tramite il servizio di messaggistica Broker Based "Kafka" ed espongono all'esterno delle API REST basate sui metodi HTTP (Get,Post,Put,Delete).

Kafka è stato usato anche per implementare la transazione SAGA relativa all'assegnazione dei posti letto.

Il client interagisce con l'applicazione attraverso un microservizio Frontend deployato in un container nginx, sviluppato in HTML e JavaScript.

Infine l'applicazione si occupa anche misurare ed esporre delle metriche, attraverso un endpoint, così da poter permettere un monitoring whiteBox. Grazie a Kubernetes è anche stato implementato un monitoring blackbox.

Tutte le metriche sono state acquisite da un nodo Prometheus, anch'esso deployato all'interno del cluster Kubernetes ma in un namespace diverso.

Sulle metriche misurate è stato quindi fatto un esempio di analisi predittiva, attraverso uno script python.

2. Architettura

Hospital Platform è composto da quattro microservizi:

- Frontend: si occupa di gestire l'interazione con il client esponendo un'interfaccia Web basata su HTML, CSS e JavaScript. Il microservizio è posto in run all'interno di un container nginx, ovvero un web server che permette di esporre pagine Web navigabili tramite browser.
  Ciascuna pagina Web ha un proprio codice javascript che permette al Client di inviare richieste HTTP al backend gestendone l'invio tramite la funzione fetch e le AJAX della libreria JQuery.
  In particolare, le pagine web esposte sono:
  - Index: Pagina di default mostrata all'utente. Permette la registrazione di medici e receptionist.
  - PageLogin: Pagina di login che permette all'utente, sia esso medico o receptionist, di accedere alla propria pagina.
  - PageReceptionist: Pagina mostrata solo dopo il login da parte di un receptionist. Contiene un button che reindirizza alla pagina di Inserimento di un paziente.
  - InserisciPaziente: Pagina che permette ad un receptionist di inserire i dati anagrafici di un paziente, selezionare il medico curante per tale paziente e poi inviare la richiesta di inserimento nel Database.
  - PageDoctor: Pagina mostrata dopo il login da parte di un dottore. Permette la ricerca di un paziente e la gestione della cartella clinica del paziente selezionato.
    NOTA: è possibile avere pazienti omonimi ed è per questo motivo che la ricerca di un nome e di un cognome non porta direttamente alla cartella clinica del paziente ma ad una lista dei pazienti aventi il nome cercato.

NOTA: I prossimi tre microservizi, sono stati sviluppati usando il framework Java Spring e il pattern MVC (model view controller) ed espongono e gestiscono le API REST all'interno delle classi Java

Controller. Inoltre, la comunicazione tra ciascun microservizio e il proprio database Mysql è stata gestita attraverso l'utilizzo di JPA.

- Reglog Service: microservizio che si compone di due controller per la gestione del login e della registrazione degli utenti. Esso espone un totale di tre API REST che sono:
    - VerificaUsername (GET): che, fornito uno username in fase di registrazione, verifica se esso è presente o meno nel database.
    - Register (POST): riceve in input un JSON contenente i dati di un utente che si vuole registrare e una volta convertito il JSON in un oggetto java, lo inserisce nel Database solo se l'utente non è già presente, usando un CRUD Repository.
      Inoltre, dopo il save nel Database viene inviato un messaggio, su un topic kafka ascoltato dagli altri due microservizi, contenente l'id dell'utente che ha fatto il login e il tipo (doctor o receptionist), così da poter gestire la sessione.
      Nel caso in cui sia stato registrato un medico, il messaggio conterrà anche nome e cognome e ID del medico stesso, così che quest'ultimo possa essere inserito anche nel database della Reception.
    - Login (GET): verifica se la coppia username-password ricevuta in input è presente nel Database ed in tal caso effettua il login inviando un messaggio su un topic kafka per la gestione della sessione.
- Reception Service: microservizio per la gestione dell'inserimento dei pazienti da parte di un receptionist che ha effettuato il login. Questo microservizio espone tre API:
    - InsertPaziente (POST): riceve in input un JSON contenente i dati di un paziente che si vuole inserire e una volta convertito il JSON in un oggetto java, lo inserisce nel Database solo se il paziente non è già presente, usando un CRUD Repository.
      Inoltre, sia nel caso di save che nel caso in cui il paziente sia già presente nel database della reception, viene inviato un messaggio, su un topic kafka ascoltato dal microservizio Doc Service, così da comunicare a quest'ultimo i dati anagrafici del paziente da inserire nel database del dottore associato.
    - ListaMedici (GET): restituisce la lista dei medici registrati nel database della reception.
    - NameOfLogged (GET): restituisce il nome del receptionist che ha effettuato il login, così da mostrarlo lato frontend.
- Doc Service: microservizio per la ricerca dei pazienti e la gestione della cartella clinica e del ricovero del paziente cercato. Questo microservizio espone diverse API:
    - Cerca (GET): riceve in input nome e cognome di un paziente e restituisce una lista di tutti i pazienti omonimi associati al medico che ha effettuato il login aventi quel nome e quel cognome.
    - Insert/Edit/Delete Diagnosis: sono tre API che permettono rispettivamente l'inserimento(POST), la modifica(PUT) e l'eliminazione(DELETE) della diagnosi per il paziente selezionato. Tale API riceve in input l'id del paziente a cui è associata la diagnosi e nel caso di POST e PUT riceve anche un body JSON contenente il testo della diagnosi che verrà poi inserita nel database.
    - Insert/Edit/Delete Cure: sono tre API che permettono rispettivamente l'inserimento(POST), la modifica(PUT) e l'eliminazione(DELETE) della cura per il paziente selezionato. Tale API riceve in input l'id del paziente a cui è associata la cura e nel caso di POST e PUT riceve anche un body JSON contenente il testo della cura che verrà poi inserita nel database.
      NOTA: è impossibile inserire una cura se non c'è una diagnosi associata a quel paziente.
    - Ricovera (PUT): prende in input l'id del paziente da ricoverare e trigghera l'inizio della transazione SAGA (vedi sotto) alla fine della quale restituisce una risposta http.

L’attesa della terminazione della SAGA è stata gestita attraverso un meccanismo event-driven (notifica la fine della transazione).

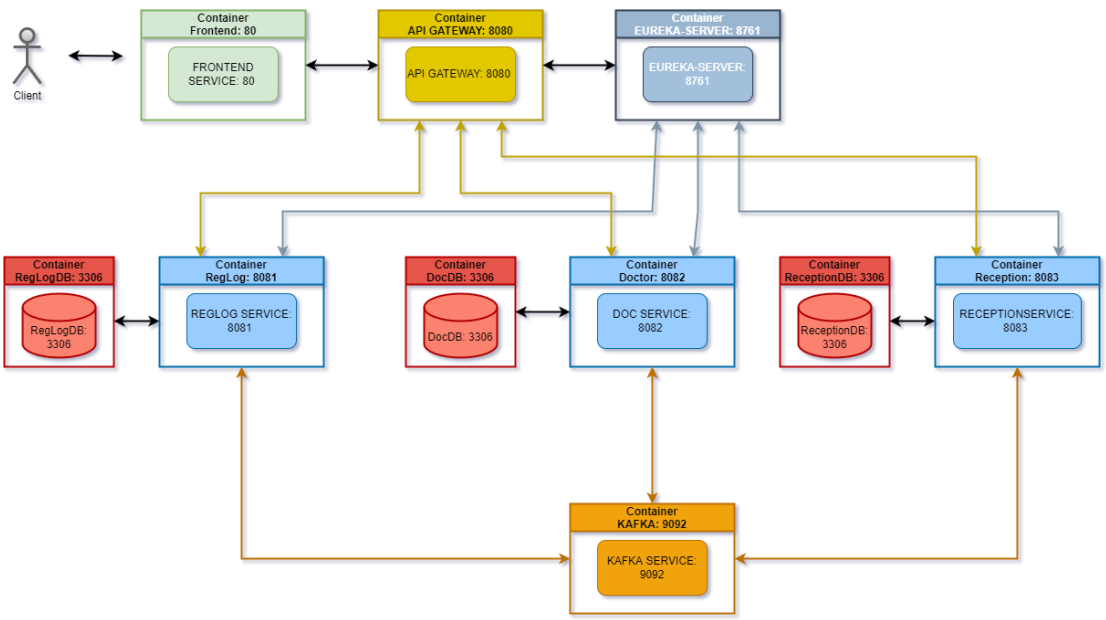
- Dimetti (PUT): prende in input l’id del paziente da dimettere, cambia lo stato del paziente nel database (ricovero=false), restituisce una risposta http e invia un messaggio su un topic kafka ascoltato dal Reception Service al fine di mantenere la consistenza del paziente e dei posti letto.

NOTA: i tre microservizi Spring hanno anche un controller dedicato al controllo delle richieste di ping.

ARCHITETTURA IN DOCKER COMPOSE:

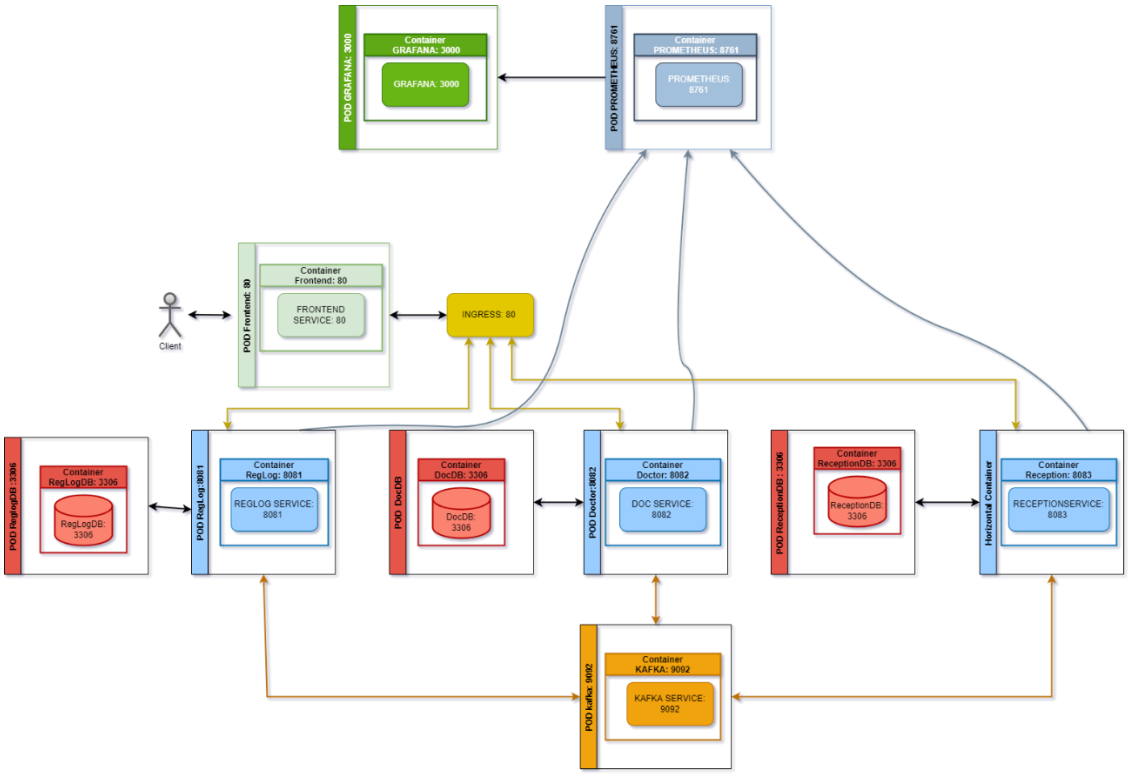
Oltre i microservizi sopra citati, per permettere eventuali scaling e una migliore gestione e pulizia dell’applicazione sono stati aggiunti:

- API GATEWAY: microservizio sviluppato in Java Spring utilizzando il tool Spring Cloud Gateway che permette di implementare un nodo che si occupa di intercettare le richieste da parte del client e inoltrarle ai microservizi interni all’applicazione così da permettere la Location Trasparency e l’introduzione di un eventuale algoritmo di Load Balancing.
  In particolare, l’API gateway è stato configurato per esporre la porta 8080 ed essere contattato all’indirizzo IP “localhost”.
- SERVER DI DISCOVERY: microservizio sviluppato in Java Spring utilizzando il tool Eureka Netflix. Questo server permette agli altri microservizi di registrarsi così che l’API GATEWAY possa ottenere per ciascuno l’indirizzo IP e la porta che espongono, senza doverli conoscere lui stesso. Così, se i microservizi vengono replicati, viene automatizzato il processo di load balancing.



ARCHITETTURA IN KUBERNETES:

A differenza dell'architettura Docker, l'architettura Kubernetes non presenta né l'API Gateway né il Discovery server in quanto entrambi questi servizi sono offerti nativamente da kubernetes stesso. In particolare, l'API gateway viene sostituito da un nodo di tipo ingress.



TRANSAZIONE SAGA:

Come già accennato, è stata implementata una transazione SAGA per la gestione dei posti letto:

Il numero di posti letto massimo è un parametro fisso, per semplicità posto a 5, definito nel main del Reception service. La transazione è qui necessaria in quanto, essedo i posti letto una risorsa condivisa e limitata, bisogna evitare inconsistenze nel caso in cui due medici facciano una richiesta di ricovero concorrente in corrispondenza dell'ultimo posto letto disponibile: in tal caso, essendo i database del medico e della reception separati e indipendenti, senza una transazione potrebbe accadere che un medico memorizzi nel database che il paziente è stato ricoverato ma il controllo dui posti letto è effettuato dal microservizio della reception per cui, l'inserimento potrebbe non andare a buon fine e quindi sarebbe necessario un rollback lato Doctor Service. La SAGA garantisce quindi l'eventual consistency e l'atomicità di questa operazione.

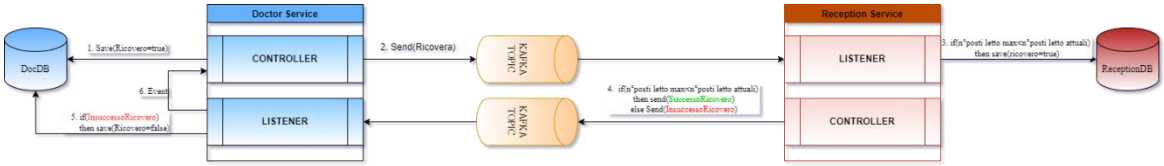
Da un punto di vista implementativo si è scelto l'approccio Coreography, cioè il controllo della transazione è distribuito nei due microservizi Doc e Reception Service, attraverso l'invio di messaggi asincroni in un topic kafka a cui entrambi sono sottoscritti. I passi della transazione sono i seguenti:

a. Il controller del Doc Service riceve una richiesta API di "Ricovero" ed esegue una sua transazione locale in cui setta a true, nel database associato allo stesso microservizio, il campo Ricovero del paziente associato a quella richiesta.
   Dopo di che, invia un messaggio nel topic kafka con scritto "Ricovero".
   Nota: il messaggio inviato e la gestione della SAGA sono entrambi asincroni, quindi il microservizio continuerà ad operare (non bloccante). Tuttavia, il controller verrà messo

comunque in attesa della terminazione della saga, notificata tramite un meccanismo a eventi, così da dare la risposta consistente al client.

b. Il Listener del reception Service riceve il messaggio sul topic kafka, verifica se vi sono ancora posti letto disponibili ed in caso positivo esegue il ricovero del paziente anche nel suo database e invia un messaggio di successo al microservizio Doc Service.
   Nel caso in cui, invece, non vi siano più posti letto disponibili, non esegue alcuna transazione locale ma invia un messaggio di "Insuccesso Ricovero" nel topic kafka.

c. Il Listener del Doc Service riceve il messaggio da parte del receptionist e :
   - Se il messaggio è "Successo Ricovero" allora, trigghera l'evento che sblocca il controller in attesa della terminazione della SAGA il quale risponderà al client indicando il successo dell'operazione.
   - Se il messaggio è "Insuccesso Ricovero" allora esegue la transazione di comensazione (mette a false il campo ricovera nell'entry del database relativa al paziente) e trigghera l'evento che sblocca il controller in attesa della terminazione della saga il quale risponderà al client indicando il fallimento dell'operazione.



3. Monitoraggio

Per poter monitorare tutte le metriche dei POD è stato necessario introdurre, all'interno del cluster Kubernates:

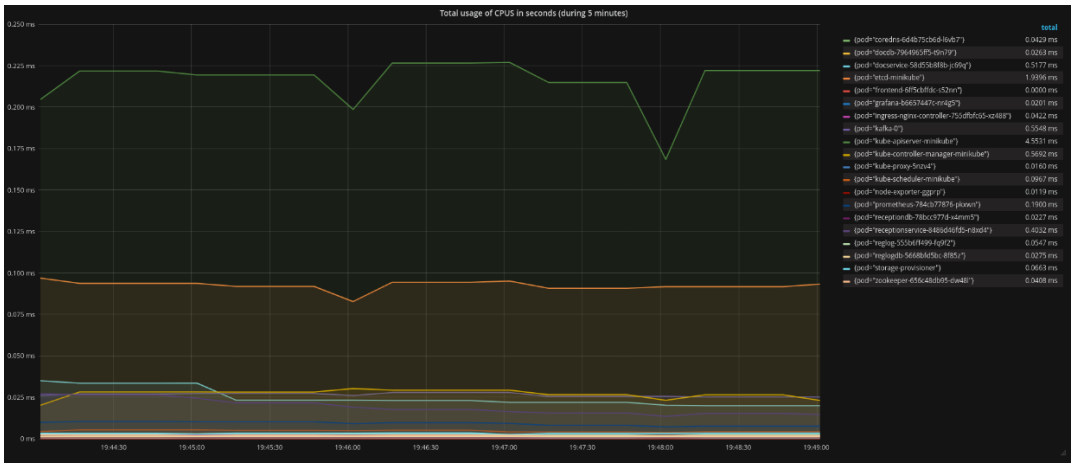
- Un deployment per Prometheus con il relativo service (di tipo NodePoirt, così da permettere l'accesso alla sua User Interface)
- Un nodo di tipo DaemonSet, descritto nel paragrafo 1, su cui gira l'applicazione NodeExporter. Questa applicazione si occupa di monitorare i POD e collezionare le metriche così che poi Prometheus possa farne il pull.

Le metriche sono state ottenute tramite query PromQL e poi graficate attraverso delle dashboard Grafana ed è per questo che nel cluster Kubernetes sono stati anche inseriti un deployment e un service per Grafana.

Il sistema Hospital Platform è stato progettato per il supporto di monitoraggio sia BlackBox che WhiteBox. In particolare:

- Monitoraggio BlackBox: il sistema è visto come una scatola chiusa e sarà Kubernetes a misurare alcune metriche dei POD e rendere disponibili a Prometheus che ne farà poi il pull.
  Nel progetto, sono state osservate 4 metriche BlackBox:
  - Uso totale della CPU di ciascun POD, calcolato ad ogni secondo e monitorato per un intervallo di 5 minuti.

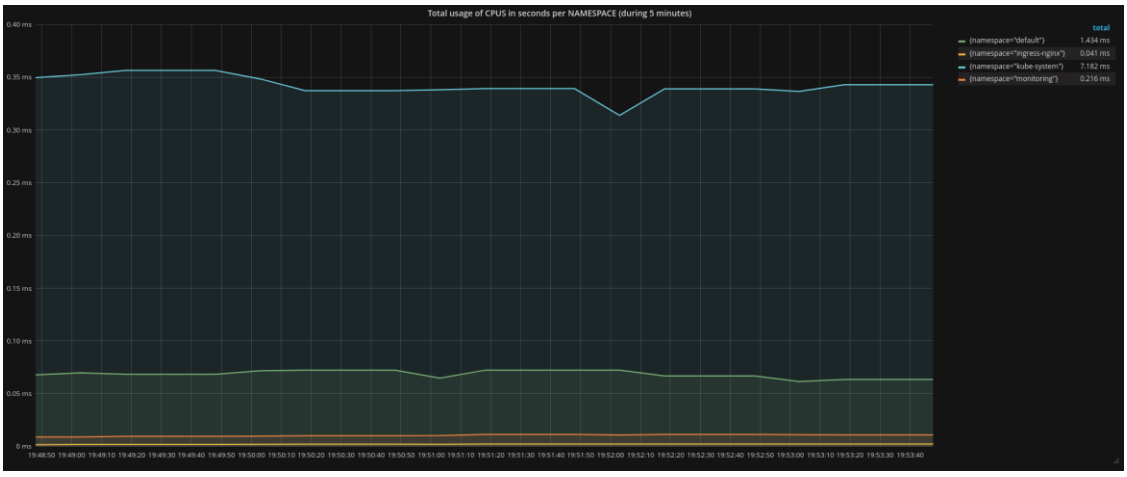
```
sum(rate(container_cpu_usage_seconds_total{container!="POD",pod!=""}[5m])) by (pod)
```



Dal grafico si evince che i microservizi propri della logica applicativa non consumano molta risorsa Cpu in quanto non sono stati sottoposti ad un traffico intenso. Invece,a consumare più CPU rispetto agli altri microservizi sono i POD relativi alla gestione di kubernetes stesso come kube-api o minikube stesso: questi due microservizi sono infatti sempre attivi perché si occupano della gestione del cluster.

- Uso totale della CPU di ciascun NAMESPACE, calcolato ad ogni secondo e monitorato per un intervallo di 5 minuti.

```
sum(rate(container_cpu_usage_seconds_total{container!="POD", namespace!=""}[5m])) by (namespace)
```

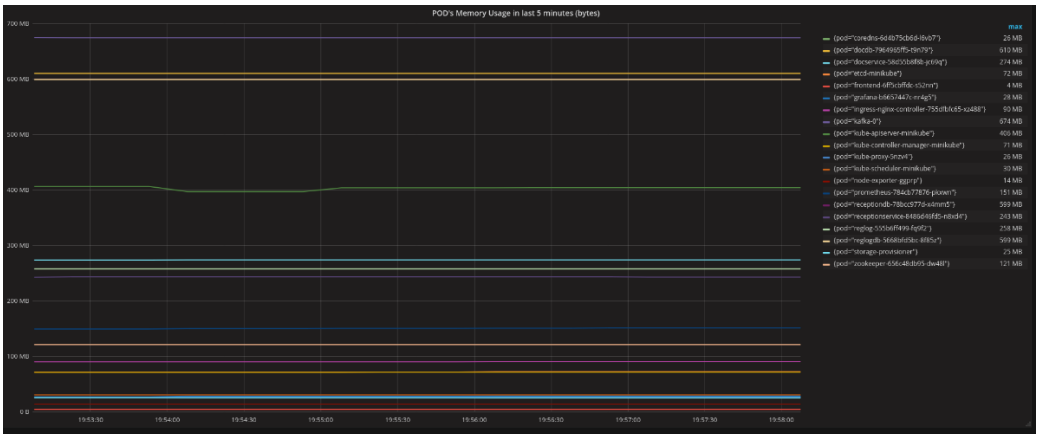


Anche in tal caso viene confemrato che a consumare più risorse CPU sono i sistemi di gestione di kubernates. Ed infatti, il namespace che consuma di più è kube-system, seguito dal namespace default in cui sono deployati i POD dell'applicazione.

Gli altri duenamespace sono, in ordine: monitoring che contiene i POD di Prometheus e Grafana ed infine il namespace ingress-nginx che non ricevendo tante richieste in quell'intervallo, non è soggetto a sforzi.

- Uso della memoria totale, espresso in byte occupati da ciascun POD e monitorato per 5 minuti:

```
sum(container_memory_usage_bytes{container!="POD",pod!=""}) by (pod)
```



Ciascun POD utilizza approssimativamente sempre la stessa quantità di memoria nel tempo in quanto il sistema non è stato sottoposto ad importanti carichi di lavoro.

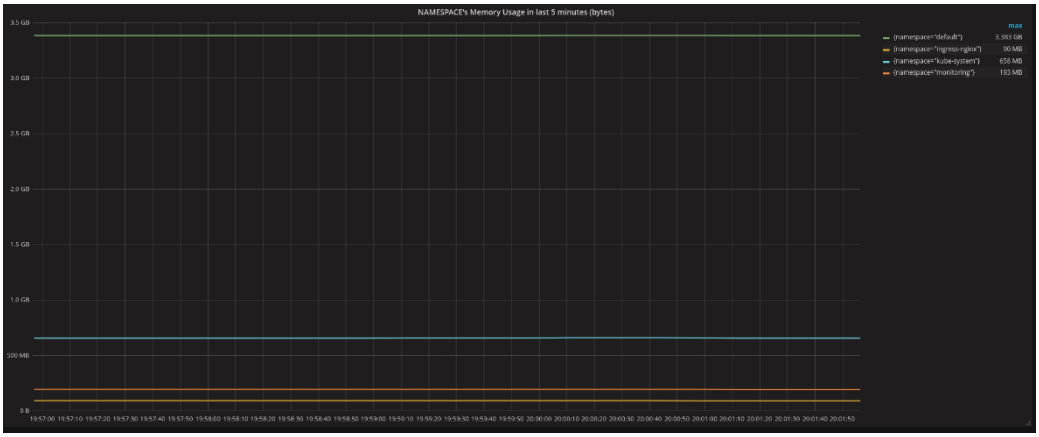
Il POD che consuma più memoria in assoluto è kafka in quanto contiene tutti i messaggi scambiati tra i container ed essendo uno statefull set, mantiene i dati di ogni esecuzione, accumulandoli nel tempo.

A seguire kafka, per uso di memoria, sono il database del dottore nel quale sono stati inseriti diversi pazienti con maggiori informazioni rispetto agli stessi pazienti inseriti nel databse della reception (es. cura, diagnosi ecc…).

Quindi in generale, ad occupare più spazio sono i microservizi che detengono informazioni di sistema e messaggistica e i vari database.

- Uso della memoria totale, espresso in byte occupati da ciascun NAMESPACE e monitorato per 5 minuti:

```
sum(container_memory_usage_bytes{container!="POD",namespace!=""}) by (namespace)
```

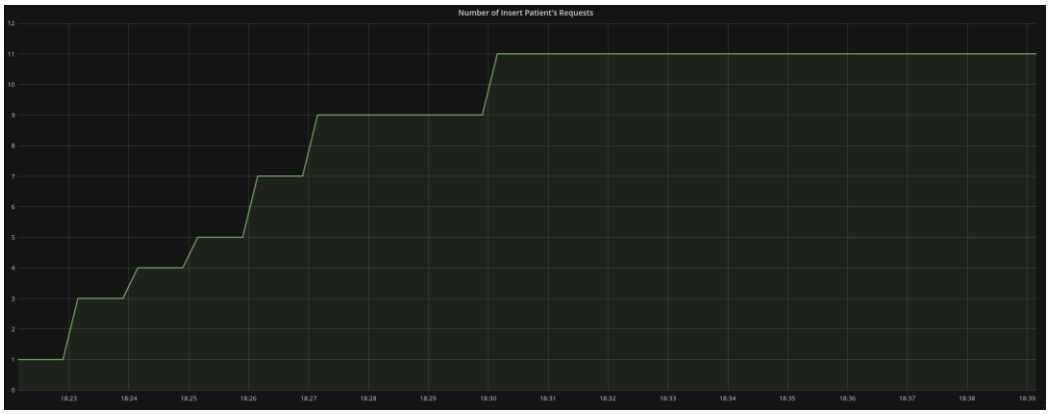


Il namespace a occupare più memoria è “default” il quale infatti contiene il POD di kafka e i databases che, come visto nell’analisi precedente riferita a ciascun pod, occupano più spazio.

- Monitoraggio WhiteBox: Il sistema misura le metriche definite dal programmatore e le espone esplicitamente.
  All’interno di ogni microservizio spring sono state inserita le dipendenze “micrometer” , “actuator” e “AOP”. Grazie a queste dipendenze è stato possibile creare delle metriche custom che vengono misurate dallo stesso microservizio e poi esposte, attraverso l’endpoint “/actuator/prometheus” dal quale Prometheus potrà effettuare il pull.

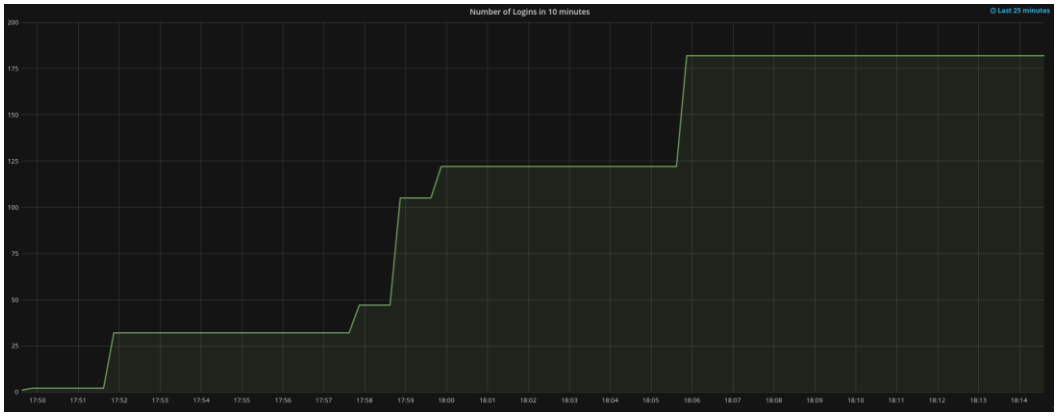
La query PromQL si ridurrà al solo nome dato alla metrica richiesta, definito nel controller del microservizio in cui monitorare la metrica.
In particolare, le metriche scelte sono:

- Numero di richieste totali, occorse in ciascun istante, di Inserimento Pazienti e monitorate in un intervallo di 20 minuti :



Considerando che è stato inserito manualmente ciascun paziente, per un totale di 11 pazienti inseriti relativi allo stesso medico, allora si può dire che il monitoraggio sia corretto: infatti è normale che tra un inserimento e l'altro, essendo essi manuali, ci siano tratti stazionari.

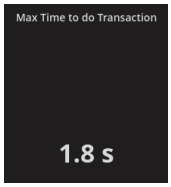
- Numero di richieste totali di Login eseguite in 10 minuti e monitorate in un intervallo di 25 minuti:



È bene notare che, al fine di rendere più realistico questo monitoraggio, è stato utilizzato uno script shell (.sh) che implementa le curl di una registrazione e 30 login sequenziali (ciclo for) e tale script è stato avviato in parallelo su più terminali, così da simulare richieste parallele.

Infatti, dall'avvio del monitoraggio alla fine, vi sono dei tratti molto ripidi che corrispondono agli istanti in cui lo script esegue anche più volte in parallelo.

- Tempo massimo impiegato per eseguire la transazione SAGA:
è una metrica scalare, misurata calcolando il tempo intercorso tra la chiamata da parte del frontend all'API "Ricovero" e la ricezione della risposta.



È un tempo alto, considerando che l'ideale sarebbe avere tempi nell'ordine dei ms.

FORECASTING:

Il progetto contiene anche uno script python che permette di eseguire la previsione dell'andamento della metrica riferita al numero di richieste di Login.

Le previsioni sulle metriche sono importanti ai fini della gestione del sistema in quanto permettono di poter attuare delle azioni di scaling in modo predittivo ovvero in anticipo rispetto all'effettiva variazione di carico.

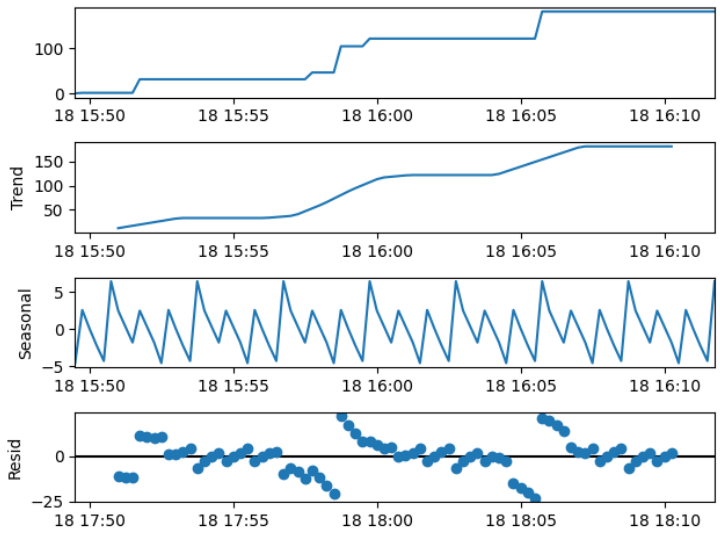
In particolare, lo script utilizza la libreria "pythonStatsModel" che contiene delle classi e dei metodi per poter eseguire queste predizioni.

È bene notare che la serie temporale contiene originariamente un totale 101 campioni in 25 minuti, che corrisponde a circa 4 campioni al minuto. È stato quindi eseguito un campionamento di 0.25T,con T=1minuto, ottenendo così 4 campioni al minuto.

Per prima cosa viene utilizzato la funzione seasonal_decompose che scompone la serie temporale relativa al numero di login, nelle tre componenti:

- Trend: rappresenta l'andamento crescente o decrescente dei valori della serie temporale al variare del tempo.
- Seasonality: rappresenta l'andamento ripetitivo dei valori nel breve periodo.
- Residual: insieme di punti in cui ciascun punto è la differenza tra il valore della metrica nella serie temporale originale in quell'istante e il valore predetto dal modello per quell'istante di tempo. In altre parole è una curva degli errori tra originale e predizione, durante il fitting del modello.

I risultati ottenuti sono:



Come ci si aspetta, il trend segue correttamente l'andamento della serie temporale.

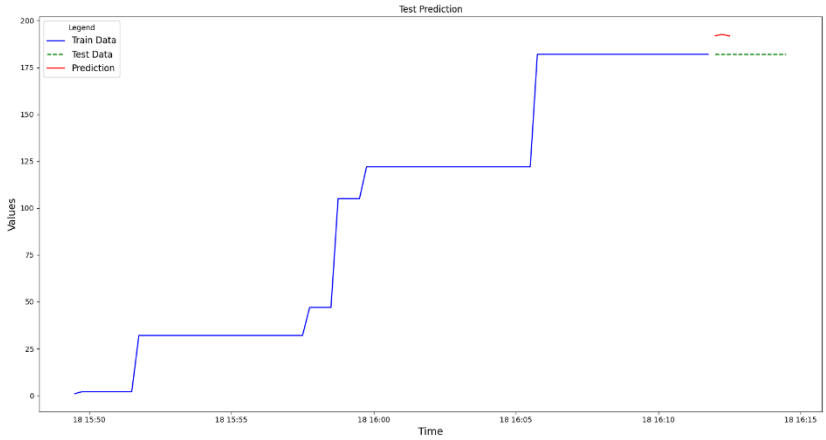
La seasonal ci mostra come la serie abbia questo andamento periodico in cui il numero di richieste eseguite dapprima aumenta e poi si riduce in quanto ci sono periodi di inattività del login.

Il residual invece evidenzia la presenza frequente di errori elevati dovuti al basso numero di campioni con cui il modello si è alleanto.

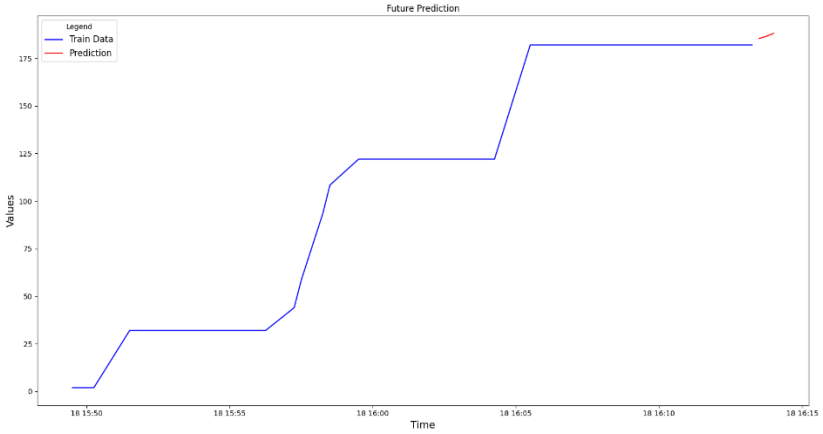
È stato poi utilizzato il modello ExponentialSmoothing che prende in input Trend e Seasonality e a partire da queste predice come evolverà il numero di richieste di login nel tempo.

Il modello è stato applicato:

- Sia ai dati originali:



  La predizione, anche se non segue esattamente i dati di test, sembra comunque rispettare l'andamento dapprima crescente della serie temporale.
- Sia ai dati modificati tramite roll (ogni campione è stato sostituito con la media dei 5 campioni ad esso precedenti) e shift (di ciascun campione a 5 posizione precedenti).
  Queste modifiche sono utili per mettere in evidenza il comportamento del modello e quindi per capire se il modello basa la sua predizione sulla base dei valori o sulla base delle caratteristiche dell'andamento nel tempo della serie temporale stessa.



  In tal caso, la predizione sembra essere corretta in quanto rappresenta esattamente l'andamento crescente della serie temporale.