Weatherfy

L'obiettivo della nostra applicazione è la verifica di vincoli (constraint) prefissati dagli utenti sulle condizioni metereologiche di una o più città, con successiva notifica agli utenti in caso di vincolo rispettato.

L'utente sceglierà quali vincoli fanno al caso suo e ogni giorno gli verrà inviata una notifica se tale vincolo trova riscontro rispetto alle previsioni metereologiche del giorno (si usa un servizio di API esterne per le previsioni metereologiche). Nel caso di inserimento di un nuovo vincolo verrà subito fatta la verifica e in caso di esito positivo sarà inviata all'utente una notifica sul vincolo appena inserito.

Gli utenti possono richiedere di visualizzare la lista dei propri vincoli attualmente attivi, in modo da avere una visione completa su cosa hanno fatto una sottoscrizione.

L'applicazione permette agli utenti di cancellare vincoli precedentemente sottoscritti, qualora questi non siano più interessati a quella particolare notifica.

L'interfaccia usata dall'utente è un chatbot Telegram. La prima volta che ci si collega si dovrà digitare "*/start*" così da registrarsi e poter utilizzare l'applicativo e registrare le proprie richieste di notifica.
Le notifiche vengono ricevute dagli utenti in 2 possibili occasioni: una volta al giorno alle ore 8:00 per tutti i vincoli inseriti in precedenza; se in seguito alla sottoscrizione di un nuovo vincolo questo è immediatamente verificato.

L'applicativo usa per la comunicazione fra microservizi Apache Kafka e dunque un modello publish subscriber basato su topic che separano logicamente i flussi dati.

Obiettivo della nostra applicazione è quello di essere un sistema distribuito altamente scalabile sia orizzontalmente che verticalmente, efficiente, adattabile ad eventuali modifiche e trasparente agli utenti che non devono essere consapevoli della distribuzione fisica dei componenti.

Scelta Databases

La scelta dei database è stata presa tenendo in considerazione dei punti chiave:

- Non tutti i microservizi accedono al database, quindi avere un “micro database” per servizio non sarebbe ottimale e introdurrebbe overhead superfluo
- Database unico e non distribuito, in modo da avere una maggiore flessibilità nella gestione degli accessi, delle autorizzazioni e un maggiore isolamento dei dati
- Le tabelle dei database possono essere di sola lettura o di sola scrittura, e che queste fossero accessibili solo dal microservizio interessato. Con questo isolamento tra operazioni è possibile distribuire carichi di lavoro diversi in modo più efficiente (traffico dati di lettura per utenti separato da traffico di lettura per microservizi, con conseguente eliminazione di interferenze con operazioni di scrittura) e avere una semplificazione per la gestione e la manutenzione del sistema
- Consistenza dati: Le tabelle read-only garantiscono che i ati letti siano sempre consistenti e stabili grazie all’assenza di interferenze di operazioni di scrittura concorrenti.

Si è optato per utilizzare un database relazionale, mySQL, e un database non relazionale, MongoDB.

- Il database non relazionale MongoDB è stato scelto per la sua alta flessibilità intrinseca, ideale per gestire oggetti con strutture variabili come i dati json. Queste tipologie di dati vengono ampiamente utilizzati nel mondo delle API REST e in Weatherfy vengono memorizzati temporaneamente le previsioni fornite dalla API OpenWeather.
  Un database di questo tipo offre eccellenti caratteristiche di scalabilità orizzontali, particolarmente utile, se non fondamentale, in un sistema distribuito. Le previsioni meteo vengono effettuate mediamente una volta al giorno (una per città), e questi risultati vengono immagazzinati in questo database (il suo compito è quello di essere una sorta di cache). La maggior parte del carico computazionale sta nell’utilizzare questi dati di previsione metereologica, e qualora questo sistema debba essere ingrandito, la possibilità di scalabilità orizzontale è ottima.
  La scelta di immagazzinare temporaneamente i risultati della API porta ad una diminuzione delle chiamate verso il servizio esterno (riduzione dei costi di utilizzo) e una maggiore velocità del servizio poiché la maggior parte delle richieste non hanno bisogno di dati esterna.
  In futuro è possibile aumentare la frequenza di chiamate al servizio, basta passare da un servizio gratuito ad un a pagamento

- Il database relazionale viene utilizzato per la memorizzazione dei dati che hanno bisogno di persistenza quali:
  - Dati utente
  - Vincoli richiesti dagli utenti
  - Città per le quali bisogna richiedere le previsioni

  Si è deciso di optare per mySQL a causa della forte complessità delle relazioni tra le varie tabelle e della presenza di dati ben strutturati. Questo database gestisce efficacemente la concorrenza in ambienti ad alto traffico (il motore InnoDB supporta il locking delle righe e la gestione transazionale per garantire coerenza dei dati, fondamentale per la gestione dei constraint).

Struttura tabelle MySql

Le tabelle database del database sono strutturate in questo modo:

citta	
Nome campo	Tipo Campo
id	int (11)
nome	varchar (255)
latitudine	double
longitudine	double
last_updated	datetime

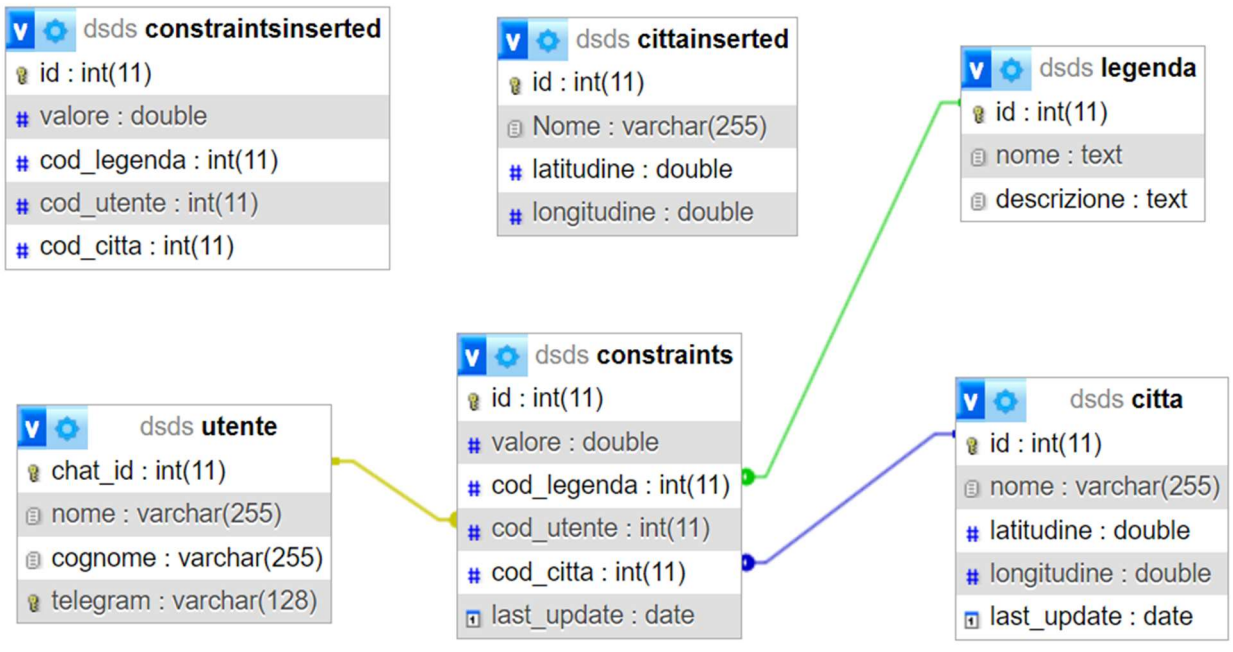
constraintinserted	
Nome campo	Tipo Campo
id	int (11)
valore	double
cod_legenda	int (11)
cod_utente	int (11)
cod_citta	int (11)

cittainserted	
Nome campo	Tipo Campo
id	int (11)
nome	varchar (255)
latitudine	double
longitudine	double

utente	
Nome campo	Tipo Campo
chat_id	int (11)
nome	varchar (255)
cognome	varchar (255)
telegram	varchar (255)

constraint	
Nome campo	Tipo Campo
id	int (11)
valore	double
cod_legenda	int (11)
cod_utente	int (11)
cod_citta	int (11)
last_updated	datetime

legenda	
Nome campo	Tipo Campo
id	int (11)
nome	text
descrizione	text



- **cittainserted:** tabella in cui si inseriscono i dati delle città.
- **citta:** versione read-only della tabella ***cittainserted***, mantenuta coerente tramite trigger SQL.
- **utenti:** contiene tutti gli utenti registrati al servizio, con il loro metodo di recapito.
- **constraintsinserted:** la tabella in cui vengono inseriti i constraint utente, ovvero le preferenze sulle notifiche di ogni utente in funzione di una determinata città.
- **constraints:** versione read-only della tabella ***constraintinserted***, mantenuta coerente tramite trigger SQL.
- **legenda:** descrive ogni constraint cosa fa, dando un significato umano all'ID della tabella ***contraints*** / ***constraintsinserted***. Non ci sono duplicati e il cod_*legenda* della tabella ***constraint*** indica un solo campo della tabella ***legenda***.

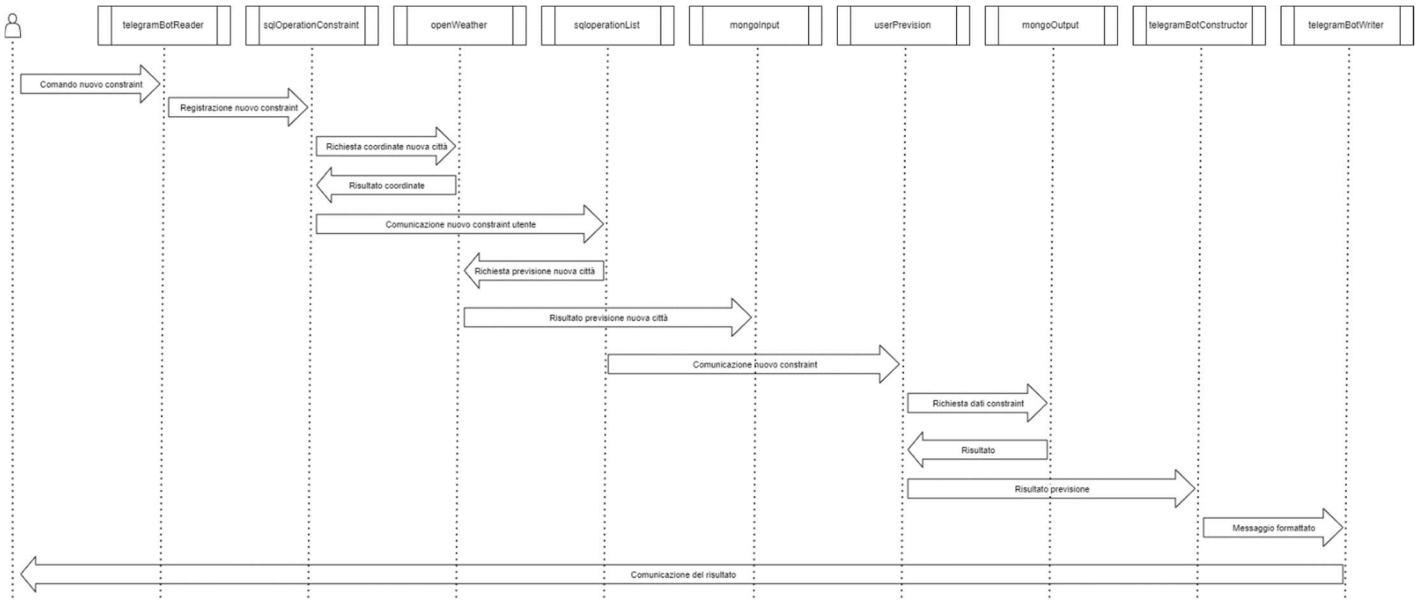
I constraint che possono essere fatti sono:

- **Temperatura massima:** Se la temperatura registrata (in gradi celsius) è maggiore di una soglia
- **Temperatura minima:** Se la temperatura registrata (in gradi celsius) è minore di una soglia
- **Temperatura massima:** Valore di temperatura massima prevista (in gradi celsius)
- **Temperatura minima:** Valore di temperatura minima prevista (in gradi celsius)
- **Umidità:** Se l umidità registrata è minore di una soglia
- **Quantità di pioggia:** Se piove più di un valore di soglia (in mm/h)
- **Presenza di neve:** Restituirà "si" o "no" se ci sarà o meno neve
- **Velocità del vento:** Se la velocità del vento è superiore a un valore indicato
- **Percentuale di nuvolosita:** Se il cielo è coperto per più di un valore di soglia)
- **Indice di raggi ultravioletti:** Restituisce il valore di raggi ultravioletti
- **Alba solare:** Quando accadrà l alba solare
- **Tramonto solare:** Quando accadrà il tramonto solare
- **Alba lunare:** Quando accadrà l alba lunare
- **Tramonto lunare:** Quando accadrà il tramonto lunare

Le richieste alla API possono essere fatte per una città alla volta. L'applicazione richiederà i dati metereologici delle città memorizzate nel database solo se queste sono collegate ad almeno un constraint nella tabella ***contraints*** / ***constraintsinserted***.

Funzionamento

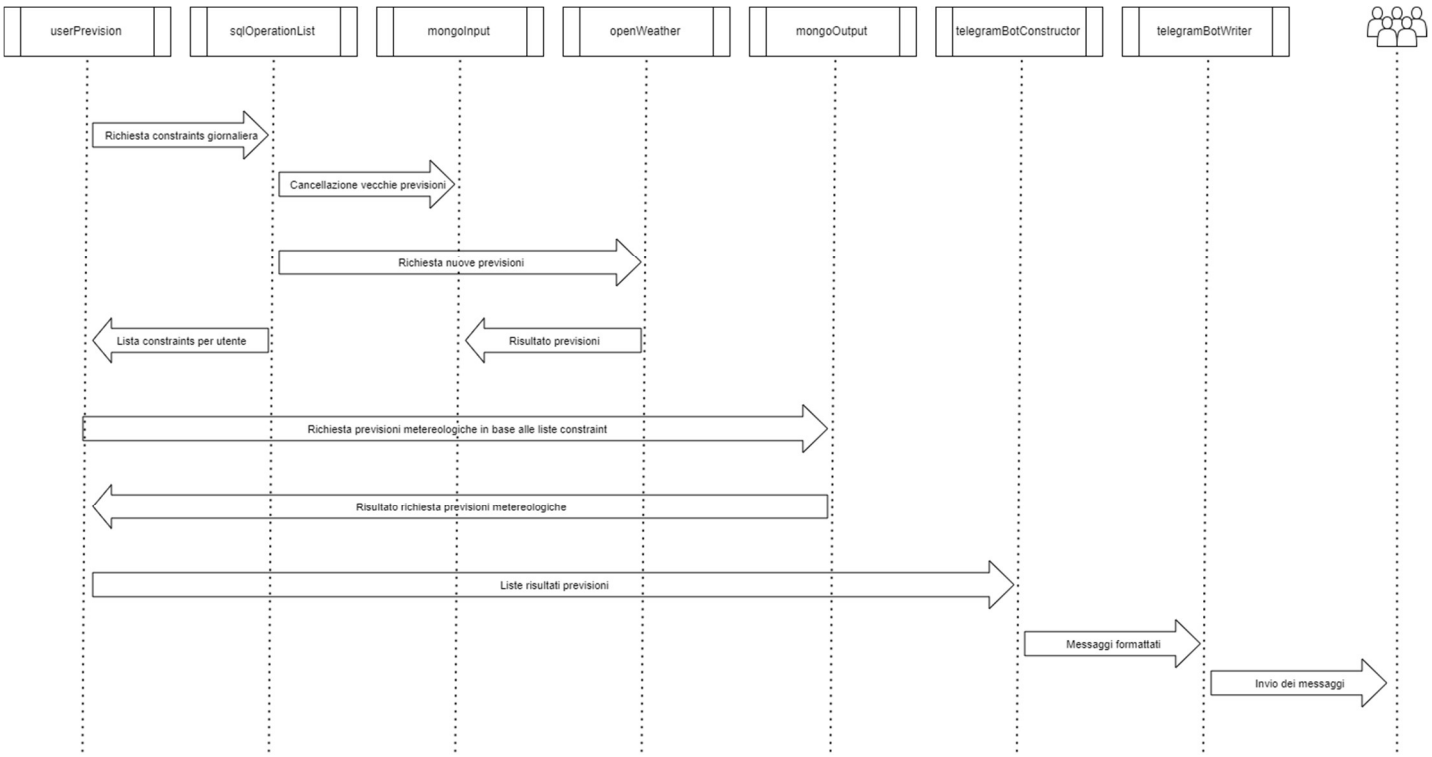
Caso d'uso 1: Aggiornamento giornaliero dei dati.



Giornalmente alle ore 8:00, vengono aggiornati i dati delle previsioni:

1. Le previsioni effettuate il giorno precedente, se presenti, vengono cancellate dal database mongo.
2. Vengono effettuate le previsioni delle città che sono presenti almeno in un constraint utente
3. Le previsioni ottenute vengono memorizzate nella collection "previsioni" del database mongo. Questi verranno copiati in un'altra collection chiamata "previsioni_only_read", dove è possibile leggere solamente.
4. Vengono create tante liste di constraint quanti sono gli utenti (ogni lista contiene solamente i constraint di un determinato utente)
5. Vengono recuperati i dati di previsione delle città dalla collection "previsioni_only_read" e vengono effettuati tutti i controlli dei constraint
6. Vengono inviate le notifiche agli utenti se almeno uno dei suoi constraint è verificato

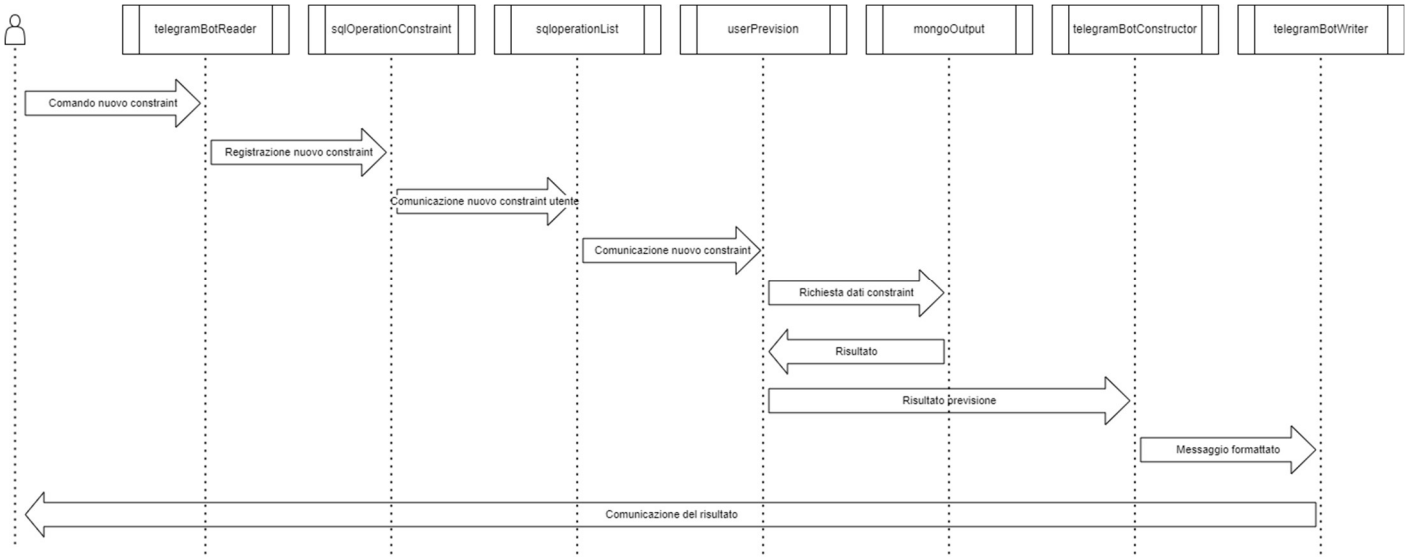
Caso d'uso 2: Un utente richiede un nuovo constraint relativo a una città non richiesta finora.



Viene richiesto un nuovo constraint da un utente, di una città non presente nel sistema:

1. Viene salvato nel database MySql il nuovo constraint nella tabella constraint_inserted write only.
2. Se il constraint è su una città non richiesta in precedenza da un utente, questa verrà aggiunta alla tabella **citta** del database relazionale.
3. Viene inoltrata una richiesta all'API esterna di previsione meteorologica e il risultato di previsione viene memorizzato sul database MongoDb alla collection previsioni.
4. Si verifica che il constraint è soddisfatto positivamente e qualora lo sia viene inviata la notifica all'utente.

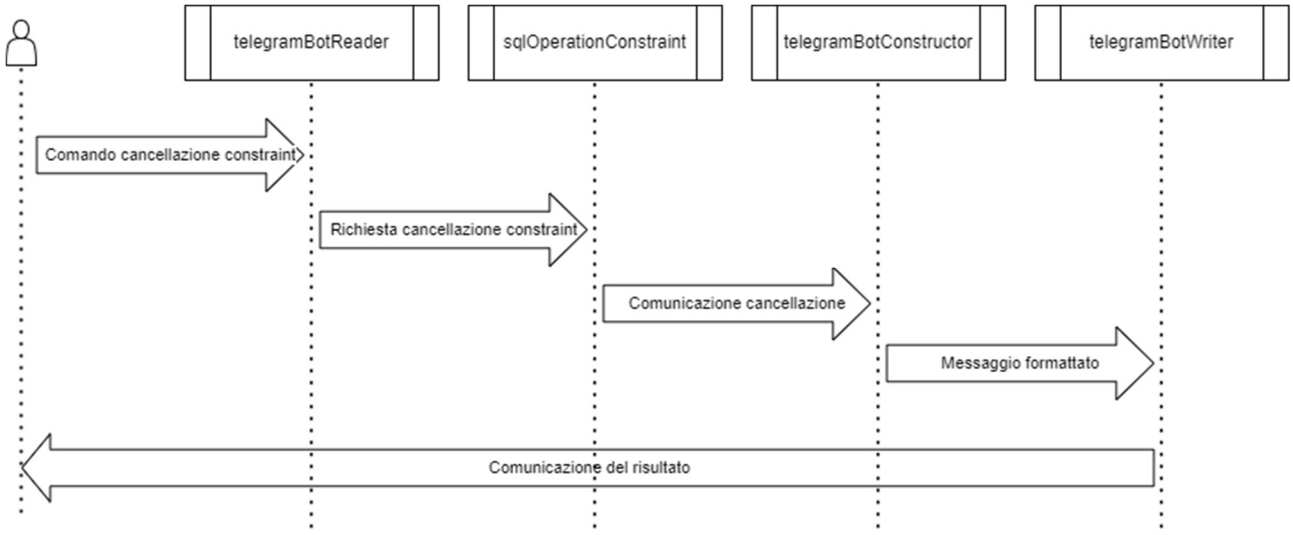
Caso d'uso 3: Un utente richiede un nuovo constraint relativo a una città già richiesta.



Viene richiesto un nuovo constraint da un utente:

1. Viene salvato nel database MySql il nuovo constraint nella tabella constraint_inserted write only.
2. Si verifica che il constraint è soddisfatto positivamente e qualora lo sia viene inviata la notifica all'utente.

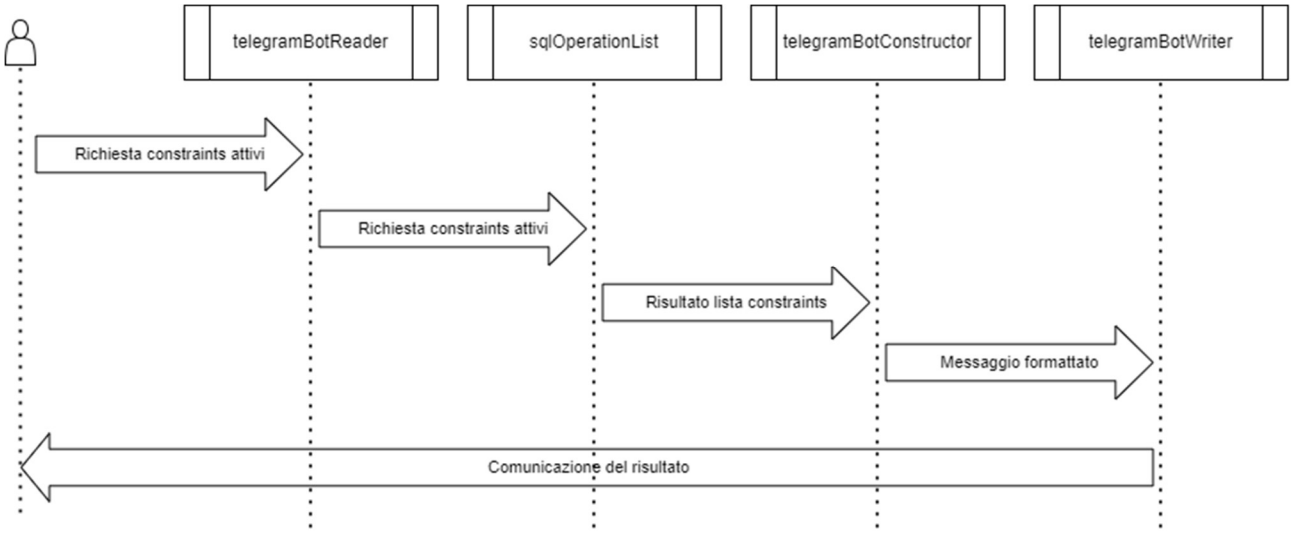
Caso d'uso 4: Un utente richiede la cancellazione di un suo constraint di cui non vuole più ricevere notifiche.



Viene richiesta la cancellazione tramite il bot telegram di un constraint:

1. Si prende dal bot telegram i dati del constraint da eliminare
2. Si elimina tale constraint dalla tabella constraintinserted, cosi che non sia più soggetto alla verifica giornaliera.
3. Si invia all'utente una notifica di successo della rimozione.

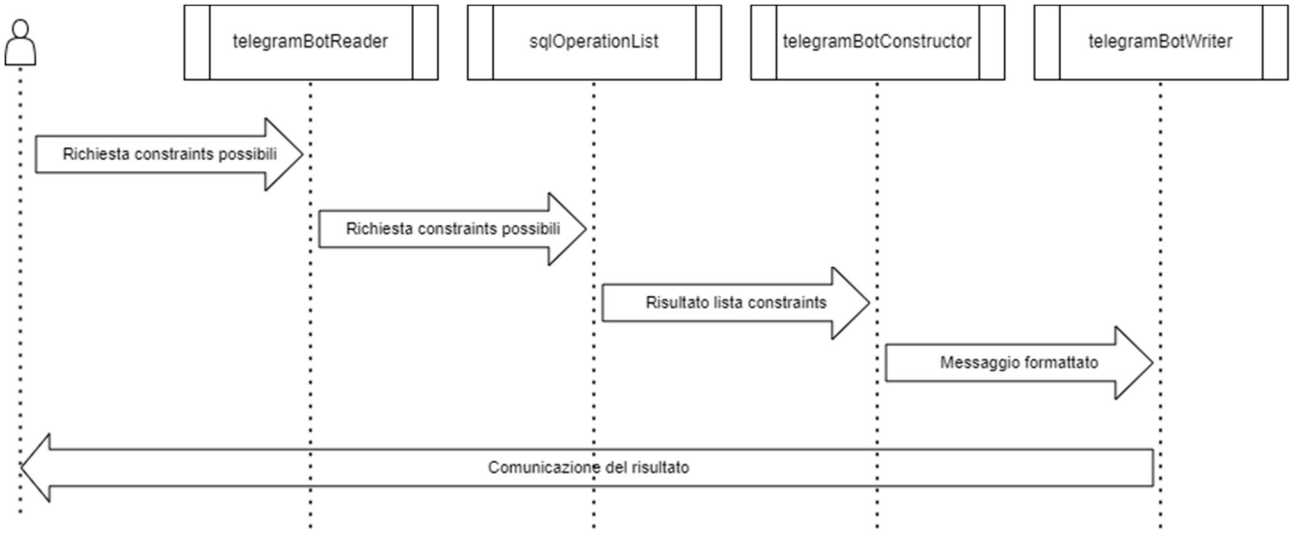
Caso d’uso 5: Un utente richiede la visualizzazione dei constraint attualmente attivi.



Viene richiesta la lista constraint dell’utente attivi in quel momento:

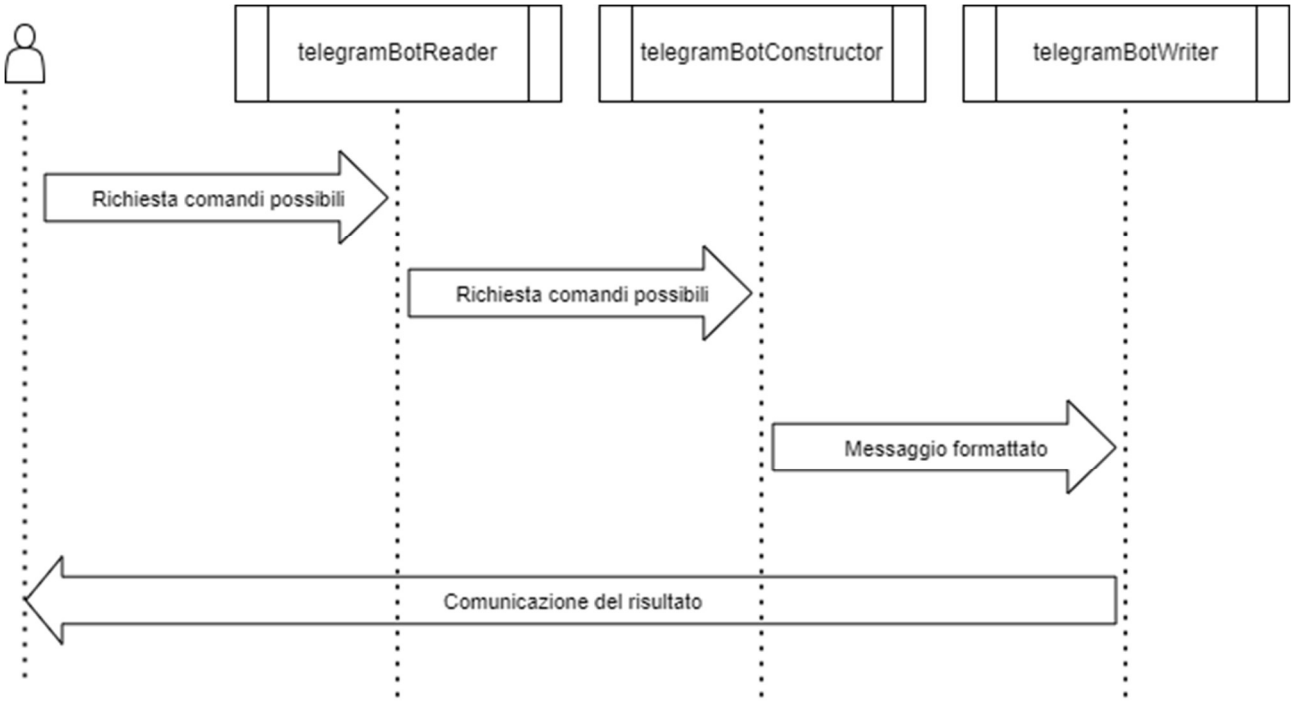
1. Viene richiesta al database sql di creare una lista di constraint per quell’utente.
2. La lista viene formattata in messaggio di testo.
3. Viene inoltrata la lista all’utente telegram.

Caso d’uso 6: Un utente richiede la visualizzazione dei constraint che è possibile fare.



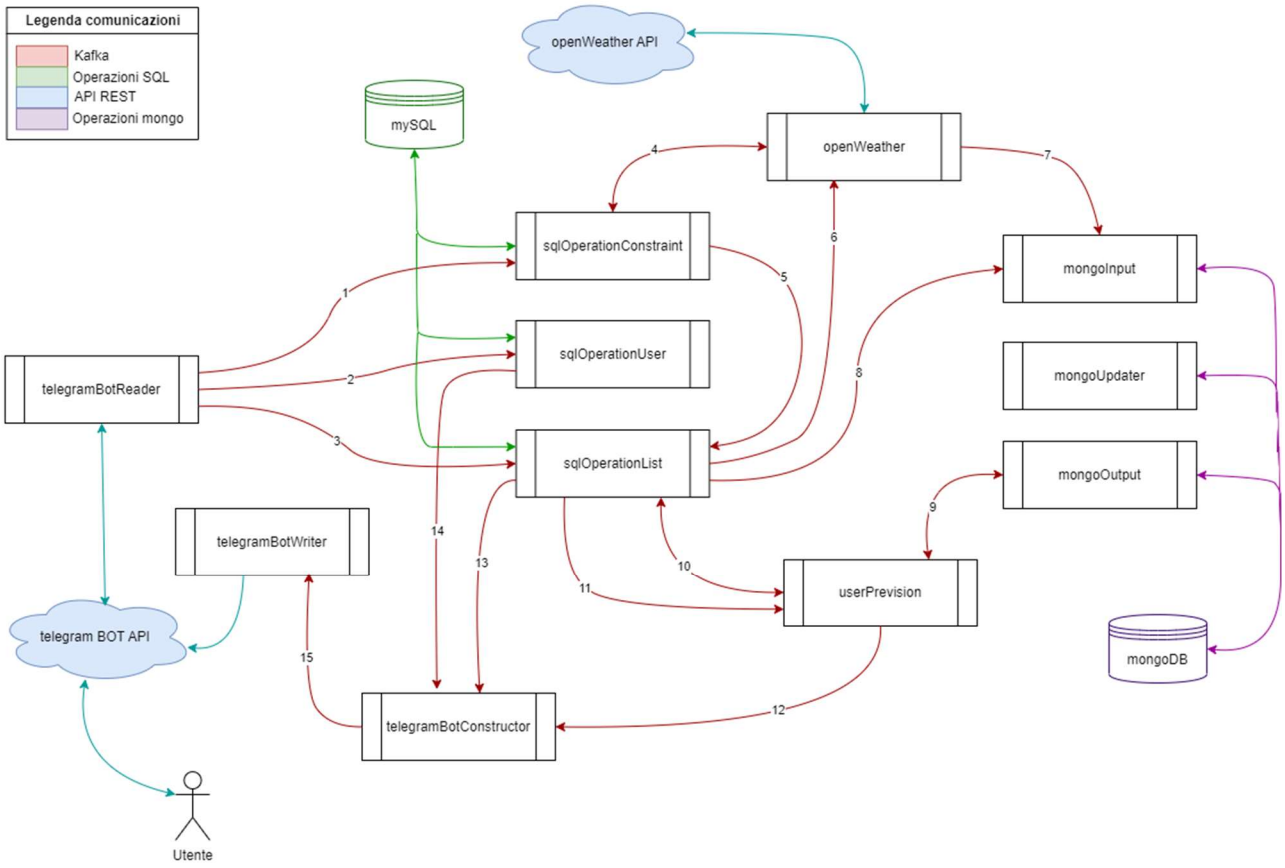
1. Viene richiesta al database sql di creare una lista dei constraint possibili.
2. La lista viene formattata in messaggio di testo.
3. Viene inoltrata la lista all’utente telegram.

Caso d’uso 7: Un utente richiede la visualizzazione dei comandi utilizzabili sul bot.



1. La richiesta per visualizzare i comandi utilizzabili viene inoltrata a telegramBotConstructor
2. La lista viene formattata in messaggio di testo.
3. Viene inoltrata la lista all’utente telegram.

Descrizione Diagramma Architetturale



1. **Utente**:
    a. Tramite il telegram ChatBot invia messaggi di comando al server telegram (non nostro, proprietario di telegram). La comunicazione è bidirezionale, quindi l'utentepotrà ricevere risposte(messaggi) dal server telegram se dovesse essere necessaria una notifica (riguardante o la corretta registrazione o un constraint inserito/verificato).
2. **Telegram BOT API (Server Telegram)**:
    a. Server proprietario di telegram che tramite Api permette di comandare un BOT e la chat che questo ha con gli utenti.
    b. La comunicazione tra il sistema e il server telegram è basato sulle API proprietarie di telegram.
3. **TelegramBotReader**:
    a. Il microservizio comunica tramite Rest Api al server telegram per verificare se vi sono nuovi messaggi dall'utente. A seconda del tipo di comando inserito comunicherà tramite Kafka al microservizio corrispondente alla richiesta utente.
    b. Comunica con il microservizio sqlOperationConstraint se il comando richiesto dall'utente è di inserimento o rimozione di un constraint.
    c. Comunica con il microservizio sqlOperationList se il comando richiesto dall'utente è di mostrare i propri constraint già inseriti oppure di mostrare i possibili constraint eseguibili dall'utente.
    d. Comunica con il microservizio sqlUser se il comando è /start cosi da eseguire la registrazione dell'utente.
4. **TelegramBotWriter**:
    a. Tramite Kafka prende i messaggi inviati dal microservizio telegramBotConstructor .
    b. Tramite Api Rest inoltra tali messaggi (notifiche se si parla di constraint) al server telegram che poi li invierà all'utente.
5. **TelegramBotConstructor**:

a. Riceve tramite Kafka dai vari microservizi (sqlOperationList, sqlUser e User prevision) i risultati delle richieste/previsioni effettuate e li formatta correttamente per il successivo invio dei messaggi su telegram
b. Invia (tramite Kafka) al microservizio TelegramBotWriter i messaggi opportunamente formattati.

6. **SqlOperationConstraint**:
    a. Riceve da telegramBotReader tramite Kafka richieste di inserimento o rimozione di un constraint
    b. Aggiunge/rimuove dal database relazionale MySql il constraint richiesto dall'utente.
    c. In caso di inserimento di un nuovo constraint:
        i. Se la città non è presente nel database, comunica con il microservizio openweather per ottenere le coordinate geografiche
        ii. In seguito all'inserimento del constraint invia a sqlOperationList, tramite kafka, una notifica di richiesta di verifica del nuovo constraint
7. **SqlOperationList**:
    a. Questo microservizio è consumer di due microservizi distinti:
        telegramBotReader:
        i. Riceve da telegramBotReader le richieste dell'utente di visualizzare la propria lista di constraint e la lista dei possibili constraint selezionabili.
        ii. Fa una query al database relazionale riguardante la richiesta specifica data dall'utente e riceve i dati richiesti
        iii. Invia al microservizio TelegramBotConstructor tramite kafka i risultati delle operazioni effettuate
        UserPrevision:
        i. In seguito alla richiesta di UserPrevision, comunica con il microservizio OpenWeather, tramite Kafka, la lista città di cui devono essere effettuate le previsioni
        ii. Comunica con il microservizio UserPrevision se riceve, tramite Kafka, dal microservizio sqlOperationConstraint un nuovo constraint di cui deve essere effettuata la verifica (con eventuale notifica all'utente).
        iii. Comunica con il microservizio MongoInput una volta al giorno (frequenza variabile in base alla richiesta e necessità degli utenti) e prima di comunicare con userPrevision (bloccante) cosi da richiedere la pulizia dei dati obsoleti.
8. **SqlUser**:
    a. Riceve messaggi, tramite Kafka, di richiesta di inserimento utente
    b. Salva un nuovo utente nel database MySql
    c. Dopo aver inserito il nuovo utente, o in caso di errore, comunica con il microservizio telegramBotConstructor, tramite Kafka, per dare agli utenti un messaggio di benvenuto o di errore (nel caso l'inserimento non fosse riuscito).
9. **UserPrevision**:
    a. Invia, tramite Kafka, una richiesta al microservizio SqlOperationList per avere la lista dei constraint per ogni utente (operazione automatica con frequenza giornaliera).
    b. Riceve la risposta alla richiesta precedente da SqlOperationList
    c. Può ricevere una lista di constraint, tramite Kafka, dal microservizio SqlOperationList per utente, qualora l'utente aggiunga un nuovo constraint dopo l'aggiornamento giornaliero dei constraint (descritto al punto precedente).
    d. Invia al microservizio MongoOutput, tramite Kafka, una richiesta a quest'ultimo di fare una query con dei vincoli prefissati (dovuti ai constraint utente ottenuti al punto b & c).
    e. Riceve dal microservizio MongoOutput, tramite Kafka, i risultati delle query.
    f. Invia, tramite Kafka, al microservizio telegramBotConstructor i risultati delle query che poi preparerà dei messaggi prefissati in base al constraint.

10. **OpenWeather**:
    a. Riceve dal microservizio sqlOperationList, tramite Kafka, una richiesta di ricerca previsioni per la città.
    b. Riceve dal microservizio sqlOperationConstraint, tramite Kafka, una richiesta di ricerca coordinate per una data città
    c. Effettua la richiesta per le coordinate e invia, tramite Kafka, il risultato al microservizio sqlOperationConstraint.
    d. Invia al microservizio MongoInput, tramite Kafka, il json contenente le previsioni per la città.
11. **MongoInput**:
    a. Riceve dal microservizio OpenWeather, tramite Kafka, i json delle previsioni richieste.
    b. Inserisce tali dati nel database non relazionale MongoDb
12. **MongoOutput**:
    a. Riceve dal microservizio UserPrevision i dati con cui effettuare il filtraggio dati e la verifica dei constraint.
    b. Effettua la query al database MongoDb e riceve i risultati.
    c. Invia il risultato della richiesta di verifica constraint, punto a, tramite kafka al microservizio UserPrevision
13. **MongoUpdater**:
    a. Microservizio che permette la replicazione dei dati Mongo al momento di inserimento, update o delete. Ha come ragion d'essere la divisione in più collection separate, una di sola lettura e una di sola scrittura.

Nota: Le entità descritte sono tutte microservizi da noi implementati ad eccezione di: utente, Telegram BOT API, OpenWeather API.

Kafka

La scelta dei topic kafka è stata basata sulla divisione logica dei flussi dati in modo che tipologie simili di messaggi o messaggi con valore logico simile, vengano inviati sullo stesso topic.
Non tutti i flussi dati con significato logico "simile" sono stati uniti su uno stesso topic (ad esempio request constraint e e meteo commands sono entrambi flussi di richieste/comandi), ma sono stati divisi per evitare di avere troppi topic con funzionalità simili ma distinte. In questo modo si evita di aumentare la complessità di gestione e aumentare il consumo di risorse, che portano ad aumentare le risorse necessarie in termini di memoria e CPU.

Segue la lista dei topic Kafka utilizzati:

1. *Bot_Command*: Topic dove verranno inviati messaggi dal microservizio che si occupa di leggere i messaggi inviati dall'utente. In base al tipo di messaggio, corrisponde un'operazione specifica inviata ad una partizione del topic. Successivamente gli altri microservizi che svolgono una delle funzionalità legate al topic, leggeranno il messaggio riservato alla propria partizione.
2. *Bot_Message*: Topic dove verranno inviati messaggi dal costruttore che verranno successivamente inviati all'utente. Tendenzialmente si tratta di notifiche di vincolo rispettato, ritorno della lista constraint attualmente attivi o semplicemente una notifica di successo di qualche operazione.
3. *Bot_Notifier*: Topic dove vengono inviati messaggi di notifica con i risultati delle varie richieste fatte dall'utente. Il microservizio TelegramBotConstructor consumatore di questo topic in base al valore della key crea messaggi diversi da inviare all'utente.
4. *Api_Params*: Topic dove vengono inviati i parametri di cui il microservizio openweather farà una richiesta API esterna. Essendovi 2 tipi di servizi differenti fatti con questi parametri si usa una chiave per distinguere l'operazione.
5. *Meteo_Prevision*: Topic dove vengono inviati i dati ottenuti dalle previsioni meteo per le varie città. Tali messaggi verranno poi consumati da un microservizio dedito all'inserimento su Mongo degli stessi.
6. *Coords_result*: Topic dove vengono inviati i dati ottenuti dal servizio di API esterno che si occupa di ritornare le coordinate data una città. Tali dati verranno poi usati dal microservizio SqlOperationConstraint che immagazzinerà nel database le coordinate associate ad una data città.
7. *Meteo_result*: Topic dove vengono inviati i risultati dei vincoli da verificare per i vari utenti. Tali dati verranno poi elaborati da un altro microservizio che li consuma dal suddetto topic.
8. *Meteo_commands*: Topic dove vengono inviati comandi di operazioni sui dati. In particolar modo vi sono 2 flussi, uno che richiede il clear di tutti i dati obsoleti e un altro che richiede la verifica di dati vincoli sui dati presenti sul database Mongo.
9. *Request_Constraint*: Topic dove si invia una richiesta di invio della lista dei constraint attivi per i vari utenti
10. *Response_Constraint*:Topic dove la lista dei constraint presa nel database relazionale viene inviata sotto forma di messaggi Kafka. Tale lista è separata in due flussi separati tramite partizione. Un flusso creato da una richiesta giornaliera di verifica e conseguente notifica degli utenti e un altro dato dall'aggiunta e conseguente verifica di un nuovo constraint da un dato utente.
11. *New_Constraint*: Topic dove vengono inviati i nuovi constraint verso il microservizio user prevision cosi che vengano verificati.

Per quanto riguarda la scelta del numero di partizioni dei vari topic, che ricordiamo influenza il parallelismo del sistema, si è scelto di adottare un modello dinamico basato sul numero di broker kafka calcolato a run time.

Da un punto di vista pratico è buona prassi scegliere il numero di partizioni in funzione della dimensione del cluster di broker. Di conseguenza qualora si abbia un piccolo cluster contente:

- un numero inferiore a 6 broker, il numero di partizioni sarebbero uguali a num_broker*3
- un numero compreso tra 6 e 12 broker, il numero di partizioni corrisponderebbero a (num_broker *2) +2
- un numero superiore a 12 broker, il numero di partizioni corrisponderebbero a 2*num_broker

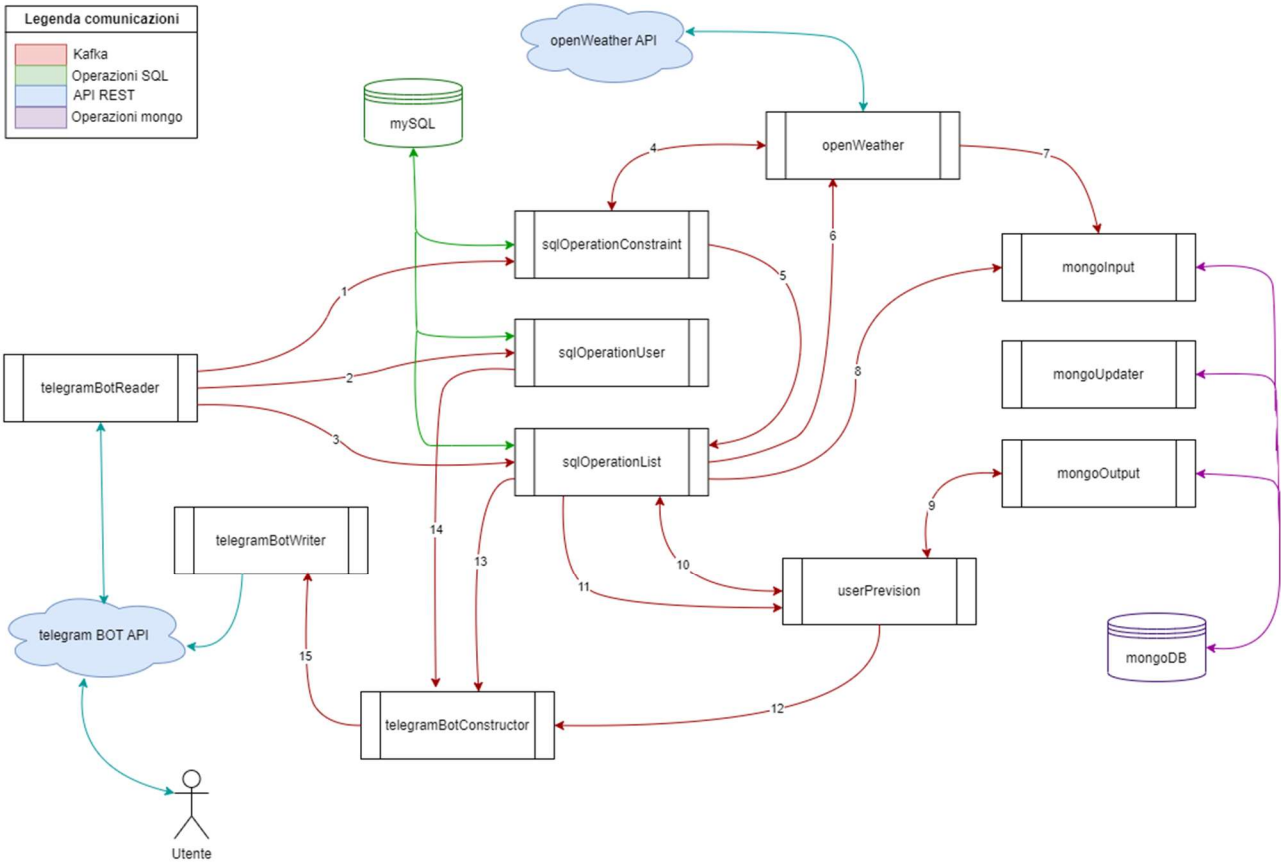
Le affermazioni precedenti sono state fatte tenendo in considerazione che il numero di partizioni deve sempre essere almeno uguale al numero di consumatori che consumano dal topic in questione.

Il fattore di replicazione aumenta la resilienza del sistema. Se il fattore di replicazione è pari a ***x***, sarà possibile avere fino a ***x-1*** broker che falliscono, senza che questi influenzino troppo l'availability del sistema. Al tempo stesso, all'aumentare del numero di nodi replica, si rischia un degrado consistente delle prestazioni, incrementando la latenza.
La configurazione attuale dell'applicazione prevede un replication factor pari a 3, uguale al numero di broker attualmente utilizzati. Qualora si volesse aumentare la dimensione dell'applicazione, aumentando il numero di broker, il fattore di replicazione rimarrebbe invariato a 3. Nel caso opposto, il numero di broker e il fattore di replicazione avranno numero uguale.

Per quanto riguarda i meccanismi di distinzione flusso si è scelto di usare, ogni qualvolta fosse possibile e necessario, diverse partizioni di un topic kafka.
Grazie a questa distinzione logica, qualora ci siano più consumer sottoscritti allo stesso topic, i flussi vengono distinti dalle partizioni, impedendo che i diversi microservizi consumino dei messaggi che non erano destinati a loro.
Un altro beneficio delle partizioni è la loro scalabilità, in quanto ogni partizione di un topic può essere gestita da un broker separato, consentendo la distribuzione del carico tra i nodi del cluster Kafka. Nel caso in cui in un topic, un flusso dati fosse molto più utilizzato di un altro, è possibile aumentare le partizioni dedicate alla partizione "più gettonata" garantendo una migliore ottimizzazione.

Nel caso in cui ci sia un solo consumer ma diversi producer, la distinzione dei flussi viene fatta tramite key poiché tutti i messaggi sono indirizzati allo stesso destinatario. In base alla chiave, questo smista alle diverse operazioni che il microservizio può svolgere.

Flussi dati fra i microservizi usando Kafka



La parte sottostante elenca i topic utilizzati e le eventuali partizioni. Viene inoltre indicato un identificativo numerico, corrispondente ad un'unica freccia rossa dello schema architetturale.
ESEMPIO:

<nome topic>: *<come viene distinto il flusso (assente se non viene distinto)>*

- *<identificativo numerico freccia>*: *<mittente flusso>* => *<destinatario flusso>*

Bot_Command: distinzione flusso di dati tramite partizioni

- 3: telegramBotReader => sqlOperationList
- 1: telegramBotReader => sqlOperationConstraint
- 2: telegramBotReader => sqlUser

Bot_Message:

- 15: TelegramBotConstructor => TelegramBotWriter

Bot_Notifier: distinzione flusso di dati tramite key

- 14: sqlUser => telegramBotConstructor
- 13: sqlOperationList => telegramBotConstructor
- 12: userPrevision => telegramBotConstructor

Api_Params: distinzione flusso di dati tramite key

- 6: sqlOperationList => openweather
- 4: sqlOperationConstraint => openWeather

Meteo_Prevision:

- 7: openWeather => mongoInput

Coords_Result:

- 4: openWeather => sqlOperationConstraint

Meteo_Results:

- 9: mongoOutput => userPrevision

Meteo_Commands: distinzione flusso di dati tramite partizioni

- 8: sqlOperationList => mongoInput
- 10: userPrevision => mongoOutput

Request_Constraint:

- 10: userPrevision => sqlOperationList

Response_Constraint: distinzione flusso di dati tramite partizioni

- 10: sqlOperationList => userPrevision
- 5: sqlOperationConstraint => sqlOperationList

New_Constraints:

- 11: sqlOperationList => userPrevision

Problemi deployment

In fase di deployment abbiamo riscontrato un grave problema, che ci ha impedito di testare l'applicativo su Docker come avremmo voluto.
Ci siamo trovati impossibilitati ad eseguire l'applicazione intera a causa delle scarse risorse hardware dei nostri computer, costringendoci al test di non più di 2 immagini per volta.

Ci riserviamo la facoltà del remoto dubbio, di aver impostato erroneamente dei parametri di configurazione di WSL2 e/o HyperV, portandoci ad avere un risultato fallimentare nel deployment.

Kubernetes

Il problema della scarsità delle risorse hardware può essere risolto utilizzando Kubernetes (k8s). Docker compose permette l'esecuzione multicontainer su uno stesso dispositivo, con l'utilizzo di k8s, è possibile migliorare la scalabilità e la distribuzione orizzontale dei container (tramite i pod, unità minime di deployment per k8s contenenti 1 o più containers).
È possibile eseguire alcuni pod sui nostri portatili (nodi in kubernetes) e sfruttare altri nodi esterni per ovviare ai gravi deficit hardware. Tramite k8s si può attuare un vero ed effettivo approccio al deployment di un applicativo con migliaia di container necessari alla sua esecuzione, magari anche sfruttando risorse esterne a pagamento in cloud (secondo un modello pay as you use e Platform as a Service, Paas abbreviato).

L'utilizzo di K8s permette, qualora vi fossero esigenze specifiche, di aggiungere nodi nei periodi temporali ove è previsto un incremento consistente del traffico (approccio orizzontale), permettendo così il funzionamento regolare e corretto del sistema.

Kubernetes ha la peculiarità di orchestrare automaticamente i container, mascherando gli eventuali fault. Qualora questi vadano in uno stato di fault, k8s è in grado di sostituirli istantaneamente, evitando di intaccare la availability del sistema.