Un processo docker che raccoglie i dati del tempo (una volta al giorno, una volta per città) e mette i json di risposta dell’API in un database mongoDB.

**Punti cardine dell’applicazione:**

Un processo docker che raccoglie i dati del tempo (una volta al giorno, una volta per città, solo per le città per cui uno degli utenti ha chiesto una notifica) e mette i json di risposta dell’API in un database mongoDB. La scelta del database non relazionale ha ragion d’essere nella sua alta flessibilità intrinseca, migliore nel gestire un dato come un json ottenuto tramite una API esterna difficilmente rinchiudibile in schemi specifici, inoltre punto ulteriore è l’eccellente scalabilità orizzontale data da un database come mongoDB. Difatti per un sistema distribuito la scalabilità orizzontale è fondamentale in quanto seppur nel nostro piccolo applicativo di per sé sarebbe possibile scalare verticalmente, nel caso di un sistema distribuito di grandi dimensioni, come potrebbe ipoteticamente diventare il nostro, distribuire i dati su più server così da poter servire più ipotetici clienti e il costo minore dell’aggiungere nodi rispetto a irrobustire quelli già presenti sono virtù fondamentali.

Nel nostro caso specifico va però detto che usiamo un singolo database mongoDB che immagazzina un json contente tutte le previsioni del giorno per una data città per la quale è richiesta almeno una notifica da un utente e una volta al giorno viene sostituito con il giorno successivo. Ciò è fatto per garantire la maggior affidabilità possibile senza dover fare continue chiamate all’API che possono risultare molto lente e ridurre la disponibilità e velocità del servizio, al tempo stesso però riteniamo sia importante non sfruttare il servizio di previsione dei giorni successivi vista l’alta percentuale di errore rispetto ad una previsione riguardante il giorno stesso.

Le richieste dati dall’api (per città) vengono fatte se sono presenti delle città nel database mysql nella tabella (città utilizzate). Se viene inserita un constraint ad una nuova città dopo che il programma ha già fatto le richieste, questo farà la call all’api solo delle città mancanti.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tabelle mysql | | | | |
| Nome tabella | Campi | | | |
| Città | ID | Città | Latitudine | Longitudine |
| Utenti | ID | Nome | Cognome | @telegram o email (char 128) |
| Sottoscrizioni | ID | Cod\_utente | Cod\_città | cod\_constraint |
| Constraint | ID | Valore | Cod\_legenda |  |
| Legenda | ID | Nome (text) | Descrizione (text) |  |

* Città ha i propri valori di longitudine e latitudine richiesti dalla api
* Utenti contiene tutti gli utenti registrati al servizio, con il loro metodo di recapito
* Le sottoscrizioni corrispondono a cosa vuole sapere ogni utente in funzione di una determinata città
* Constraint contiene il valore puro da controllare
* Legenda descrive ogni constraint cosa fa, dando un significato umano all’ID della tabella Contraint. 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 è maggiore di una soglia
* Temperatura minima, se la temperatura registrata è minore di una soglia
* Se si vuole sapere il valore di temperatura massima a prescindere
* Se si vuole sapere il valore di temperatura minima a prescindere
* Umidità (stesso ragionamento della temperatura)
* Quantità di pioggia (se supera un valore in mm/h indicato)
* Presenza di neve, o c’è o no
* Velocità del vento, se superiore a un valore indicato in m/s
* Fase lunare:

Immagine che contiene schermata, testo, cerchio, diagramma

Descrizione generata automaticamente

* + Luna nuova
  + Luna crescente
  + Luna piena
  + Luna calante
* Percentuale di nuvolosità
* Indice raggi ultravioletti
* Quando ci sarà l’alba solare
* Quando ci sarà il tramonto solare
* Quando ci sarà l’alba lunare
* Quando ci sarà il tramonto lunare

Un processo si occuperà della verifica delle constraint. Se alcune di queste sono soddisfatte, il processo manderà una notifica (solo una volta al giorno) all’utente interessato.  
Qualora un utente modifichi il valore di un suo constraint, o ne aggiunga un altro, il processo (o un altro che se ne occupa) invierà una nuova notifica se il constraint modificato/aggiunto è soddisfatto.

P.S.: il lavoro di richiesta dati alla api è scalabile ma per farlo si deve pagare. Con la sottoscrizione gratuita abbiamo un limite di 60 chiamate al minuto, 1.000 chiamate al giorno, 1.000.000 di chiamate al mese.

***Funzionamento***

**Caso 1: Aggiornamento giornaliero dei dati**

1)Si fa richiesta all’Api weatherApi dei dati delle previsioni relative alle città per cui sono attivi constraint dagli utenti e si inseriscono nel database Mongo

2)Dal database MySql si verifica se i vari constraint degli utenti sono rispettati e si recuperano i dati nel database Mongo cosi da verificare quali di essi siano verificati

3)Se vi sono dei constraint verificati si inviano agli utenti le notifiche

**Caso 2: Un utente richiede un nuovo constraint relativo a una città già richiesta**

* 1. Viene richiesto un nuovo constraint da un utente

2.1) Si salva nel database MySql il nuovo constraint

3.1)Si verifica se il constraint è verificato nel database MongoDB

4.1)Se verificato si invia la notifica all’utente su telegram

**Caso 3: Un utente richiede un nuovo constraint relativo a una città non richiesta finora**

* 1. Viene richiesto un nuovo constraint da un utente

2.2) Si salva nel database MySql il nuovo constraint

3.2) Se il constraint è su una città non richiesta in precedenza da un utente si fa una richiesta all’api e si salva sul database MongoDb il json relativo alle previsioni della città

4.2) Si verifica se il constraint è verificato

5.2) Se verificato si invia la notifica all’utente su telegram

***Kafka***

Per quanto riguarda Kafka si è deciso di avere i seguenti topic kafka:

Topics:

1 topic dove si inviano tutti i dati raw

N topic dove si filtrano i raw e rimangono le varie città , N in quanto uno per città

1 topic che ha le preferenze prese dal database

1 che verifica le preferenze e invia notifiche

***Produttori (Producer):***

I produttori sono responsabili di inviare dati a specifici topic di Kafka. Nel contesto dell'applicazione:

**Raw Weather Data Producer:**

Questo produttore invia dati meteorologici grezzi a un topic chiamato 'raw-weather-data'.

Ad esempio temperatura, quantità di pioggia, presenza di neve per diverse città. Vista la struttura della Api di fatto 1 Json per città

**Filtered Data Producers:**

Dopo la ricezione dei dati grezzi, vengono eseguite operazioni di filtraggio per le città di cui qualcuno è interessato.

I dati filtrati vengono quindi inviati a topic dedicati per ciascuna città (es. 'filtered-weather-data-cittàX').

**User Preferences Producer:**

Invia le preferenze degli utenti (es. temperatura massima, quantità di pioggia minima) al topic 'user-preferences'.

***Consumatori***

I consumatori si iscrivono a specifici topic e ricevono i dati pubblicati dai produttori. Nel contesto dell'applicazione:

**Consumatore filtered Data:**

Consumano i dati filtrati per città dai topic dedicati.

Esempio: un consumatore iscritto al topic 'filtered-weather-data-city1' riceverà dati filtrati solo per la città 1.

**Consumatore user Preferences:**

Consuma le preferenze degli utenti dal topic 'user-preferences'.

**Notification Consumer:**

Si iscrive al topic 'user-notifications', che riceve notifiche generate quando le condizioni meteorologiche soddisfano le preferenze degli utenti.

Quando riceve una notifica, il consumatore può gestire l'invio effettivo delle notifiche agli utenti (ad esempio, invio di notifiche push o e-mail).

***Interazioni***

Le interazioni nel sistema sono basate sulla pubblicazione e il consumo di messaggi nei topic di Kafka:

**Produzione di Dati:**

I produttori inviano dati ai topic corrispondenti (es. raw-weather-data, filtered-weather-data-cityX, user-preferences).

**Consumo di Dati:**

I consumatori si iscrivono ai topic di loro interesse.

Consumano i messaggi dai topic per ottenere dati grezzi, dati filtrati per città, e preferenze degli utenti.

**Elaborazione:**

I consumatori possono elaborare i dati in base alle logiche di filtraggio, aggregazione o verifica delle condizioni.

**Notifiche:**

Il consumatore responsabile delle notifiche riceve i messaggi da 'user-notifications'.

Quando riceve una notifica, può inviare effettivamente un avviso all'utente (ad esempio, notifica push, e-mail).