

**Distributed Systems and Big Data 2023-2024**

Weather Event Notifier

Alessandro Genovese 1000002043

Francesco Pennisi 1000055702

INDICE

[INTRODUZIONE 5](#_Toc114680447)

[1. Abstract 8](#_Toc114680448)

[2. Architettura del sistema 14](#_Toc114680455)

[3. Scelte Progettuali 21](#_Toc114680462)

[3.1. Comunicazioni 21](#_Toc114680463)

[3.2. Gestione repliche 22](#_Toc114680464)

[3.3. Sicurezza Secrets 22](#_Toc114680465)

[3.4. 24](#_Toc114680466)

[3.5. 24](#_Toc114680467)

[3.6. Serie temporali analizzate 25](#_Toc114680468)

[3.7. Acquisizione ed elaborazione dei dati 26](#_Toc114680469)

[3.8. Memorizzazione 29](#_Toc114680470)

[4. Monitoraggio QoS 32](#_Toc114680471)

[4.1. Prometheus 32](#_Toc114680472)

[4.2. Cadvisor 32](#_Toc114680473)

[4.3. SLA\_manager 32](#_Toc114680474)

[4.4. Forecasting 33](#_Toc114680475)

[4.5. Serie temporali analizzate 25](#_Toc114680468)

5. BUILD E DEPLOY SU README.md

[INDICE DELLE FIGURE 34](#_Toc114680476)

[RINGRAZIAMENTI 35](#_Toc114680477)

[BIBLIOGRAFIA 36](#_Toc114680478)

1. **Abstract**

Lo scopo di questo progetto è la progettazione e lo sviluppo di un sistema distribuito che possa essere eseguito su piattaforme architecture-independent.

Il sistema in oggetto è realizzato seguendo un pattern architetturale a microservizi, i quali vengono impacchettati in appositi container, facendo uso della tecnologia di containerizzazione Docker, che permette di raccogliere e isolare i microservizi in ambienti runtime completi, corredati di tutti i file necessari per l’esecuzione in modo da garantire la portabilità su qualunque infrastruttura (hardware e software) che sia docker-enabled.

Il sistema in esame ha l’obiettivo di permettere agli utenti registrati di indicare, per ogni località di interesse, dei parametri meteorologici, ad esempio la massima temperatura o l’eventuale presenza di pioggia. Tali parametri saranno monitorati dal sistema stesso. In caso di violazione delle condizioni meteo specificate, gli utenti verranno notificati dal sistema tramite e-mail.

A tale scopo, ad intervalli regolari, anch’essi a discrezione dell’utente, l’applicativo recupera le informazioni meteorologiche appoggiandosi al servizio terzo OpenWeather (<https://openweathermap.org/api>) mediante richieste REST API. Tali informazioni vengono filtrate e opportunamente elaborate sulla base delle condizioni sottomesse dagli utenti.

1. **Architettura del sistema**

Il sistema, dal punto di vista funzionale, è composto dai seguenti microservizi: Weather Management Service (abbreviato in WMS), User Manager Service (UM), Worker Service e Notifier Service.

Inoltre, è stata prevista l’introduzione di un’attività di QoS management che prevede l’inserimento nell’architettura del sistema di tre ulteriori microservizi: Prometheus, Cadvisor e Service Level Agreement (SLA) Manager, i quali sono trattati in seguito (#TODO: indicare nome paragrafo).

Immagine che contiene testo, diagramma, schermata, Piano

Descrizione generata automaticamente

Figura Architettura del sistema

In breve, sono spiegati i microservizi utilizzati:

* **User Manager Service (UM):** garantisce la registrazione e la successiva autenticazione degli utenti nel sistema. L’autenticazione sfrutta la tecnologia JWT token, dunque l’utente attraverso l’utilizzo di un client (e.g. *Postman*) effettua il login tramite richiesta REST API e ottiene un token JWT, il quale viene memorizzato e inserito nell’header *HTTP Authorization* in tutte le successive richieste effettuate dall’utente.

Nel momento in cui l’utente volesse inserire/modificare dei parametri meteorologici per una determinata località contatterà il microservizio WMS inviando il token di autorizzazione.

Quest’ultimo microservizio autenticherà l’utente passando il token allo User Manager, il quale dopo aver validato il token restituirà al WMS le informazioni necessarie affinché la richiesta dell’utente possa essere soddisfatta con successo.

L’UM è collegato a un suo Database MySQL che include la tabella Users. Tale tabella contiene le informazioni relative all’ utente: indirizzo e-mail, password e codice identificativo univoco dell’utente.

* **Weather Management Service (WMS):** permette all’utente l’inserimento, la modifica e l’eliminazione dei parametri meteorologici relativi a una o più località a scelta dell’utente.

Il WMS è collegato a un suo Database MySQL che include la tabella *Locations*, in cui vengono memorizzate le informazioni relative alle località di interesse degli utenti, e la tabella *UserConstraints*.

Quest’ultima contiene le regole specificate dagli utenti, un *trigger\_period* che indica ogni quanti minuti l’utente desidera che il sistema monitori le regole sottoscritte, infine un timestamp che viene aggiornato ogni volta che il WMS prende in considerazione la specifica entry.

Il WMS ciclicamente, attraverso l’utilizzo di un timer, controlla la tabella UserConstraints e di volta in volta prende in considerazione solo le entries per cui è trascorso il *trigger\_period*.

Il compito del WMS in sostanza è quello di costruire dei messaggi Kafka da pubblicare sul topic *event\_update*, del quale il microservizio Worker è sottoscrittore.

Tali messaggi includono le regole degli utenti di cui il Worker dovrà controllare le eventuali violazioni. La costruzione del messaggio è approfondita al paragrafo ()

* **Worker Service:** ha la responsabilità di contattare il servizio OpenWeather per ottenere i dati metereologici attuali e verificare l’eventuale violazione dei parametri specificati dagli utenti.

Ciò avviene seguendo questo ordine procedurale: il worker preleva dal topic Kafka *event\_update* i messaggi pubblicati dal WMS, estrarre le regole da controllare, contatta il servizio OpenWeather, confronta i risultati ottenuti con quelli ricevuti dal WMS.

Nel caso in cui si presentino delle violazioni il worker le pubblicherà in un messaggio sul topic Kafka *event\_to\_be\_notified*. All’interno del messaggio ogni parametro violato è associato all’utente che ne ha richiesto il monitoraggio tramite il suo codice identificativo univoco.

* **Notifier Service:** il suo compito è quello di prelevare eventuali messaggi Kafka pubblicati sul topic *event\_to\_be\_notified* dal worker.

Una volta estratto il messaggio si occupa di avvisare gli utenti delle avvenute violazioni mediante l’invio di una mail.

Il notifier è in grado di recuperare l’e-mail dell’utente contattando lo User Manager tramite il meccanismo di comunicazione gRPC, è possibile identificare univocamente l’utente grazie al suo codice identificativo contenuto nel messaggio Kafka.

* **Prometheus:** è un sistema open source utilizzato come strumento di monitoraggio che offre la possibilità di monitorare le proprie applicazioni conservando le metriche monitorate.

Ogni microservizio, precedentemente trattato, espone delle metriche al proprio endpoint *metrics*, Prometheus effettua, ad intervalli regolari, uno scraping di tali metriche attraverso una richiesta REST API all’endpointsopra menzionato.

Le metriche collezionate sono reperibili, da Prometheus stesso, tramite richieste REST API.

* **SLA Manager:** permette di definire, modificare il *Service Level Agreement* dell’applicazione come composizione di un set di metriche specificate dall’admin del sistema. Permette di aggiungere, modificare, rimuovere le metriche del *Service Level Agreement*. L’admin per ogni metrica specifica i valori target che costituisco il *Service Level Objectives* (SLO).

Il microservizio in questione ottiene il *Service Level Indicator* (SLI) attraverso delle query in *PROMQL* al microservizio Prometheus.

In particolare, permette di controllare lo stato di ogni metrica e confrontare i valori attuali con quelli target, segnalando il numero di violazioni avvenute.

Infine, è stato implementato un meccanismo di previsione dei valori futuri delle metriche sottoscritte dall’admin, in tal modo è possibile calcolare la probabilità che ci siano delle violazioni in un futuro prossimo.

* **Kafka:** è un sistema di messaggistica distribuita di tipo publish-subscribe, progettato per essere veloce, scalabile e consentire la persistenza dei messaggi. Tutti gli attori che interagiscono con Kafka possono essere considerati client, publisher e/o subscriber.

I publisher pubblicano dei messaggi su un topic, mentre i subscriber reperiscono i messaggi pubblicati sui topic a cui si sono sottoscritti.

La comunicazione è pensata per essere asincrona e indiretta, in modo che non ci sia il bisogno che i processi comunicanti conoscono i rispettivi indirizzi IP. Ogni client comunica in maniera diretta esclusivamente con il broker Kafka.

Un messaggio è costituito da dati di qualsiasi natura. Spetta al client definire il formato del messaggio.

* **Nginx:** è un web-server open-source ad alte prestazioni, utilizzato anche come reverse proxy. Garantisce un basso consumo di memoria e un’elevata concorrenza. Anziché creare nuovi processi per ogni richiesta web, Nginx utilizza un approccio asincrono, basato sugli eventi, in cui le richieste vengono gestite in un singolo thread. Nel progetto in esame è stato utilizzato con la funzione di API gateway, in modo da assicurare la *Location transparency* e la *Partition transparency*.

Nginx riceve le richieste dell’utente tramite richieste REST API e le inoltra all’appropriato microservizio tra User Manager, WMS ed SLA Manager.

* CADVISOR???

1. **Scelte Progettuali**

Nel seguente capitolo sono illustrate le scelte progettuali in merito alla gestione della Fault Tolerance, delle repliche e della comunicazione.

**3.1) Fault Tolerance**

Ogni microservizio è autonomo e indipendente dagli altri, questo significa che ognuno di essi può subire un proprio guasto, creando un effetto domino che può far crollare l'intero sistema. Implementare dei meccanismi di *fault tolerance* assicura che un sistema può continuare a funzionare e fornire servizi anche se qualcosa va storto.

Nel sistema in questione la fault tolerance è stata implementata grazie all’utilizzo di Database MySQL agganciati ai relativi microservizi. Nello specifico i database associati al worker e il notifier hanno lo scopo di memorizzare il lavoro corrente.

Sia il worker che il notifier sono dei client subscriber di Kafka. Prendendo in considerazione il worker, esso una volta prelevati i dati ed eseguito il commit su Kafka è l’unico nodo che possiede i dati; dunque, se dovesse andare in down perderebbe le informazioni acquisite e si avrebbe un fallimento del sistema. Per risolvere questo problema è stato implementato un Database MySQL in modo tale che il worker prima di fare il commit a Kafka conservi le informazioni nel suo DB.

In tal modo, se il nodo dovesse cadere dopo aver fatto il commit a Kafka, al suo riavvio sarebbe in grado di svolgere correttamente il suo lavoro poiché ha i dati memorizzati nel suo DB. Mentre, se dovesse cadere prima di aver memorizzato i dati nel suo DB, il problema verrebbe risolto da Kafka stesso, poiché non avendo ricevuto il commit, al riavvio del worker invierebbe nuovamente lo stesso messaggio.

Il caso più delicato è quando il worker dovesse cadere dopo aver memorizzati i dati, ma prima di eseguire il commit. In questo caso specifico, il worker al suo riavvio prima gestisce il lavoro presente nel suo DB, che rappresenta il lavoro non portato a termine, dopodiché riceverà da Kafka per la seconda volta lo stesso messaggio e dunque lo elaborerà due volte.

Nel sistema in esame questo non rappresenta un problema, poiché la doppia rielaborazione di uno stesso messaggio potrebbe al più comportare l’invio di due mail uguali all’utente. Ciò non rappresenta un fallimento del sistema, difatti si è preferito implementare Kafka seguendo l’approccio *at least once*, ovvero vogliamo assicurare che ogni messaggio arrivi a destinazione almeno una volta. Sarebbe molto più grave per il sistema perdere del tutto un messaggio e mancare l’invio della notifica all’utente.

Per il notifier il discorso è analogo, poiché il notifier semplicemente estrae i dati pubblicati dal worker e si occupa dell’effettivo invio della notifica. È bene sottolineare che il worker pubblicherà un messaggio solo se sono avvenute delle violazioni dei parametri indicati dall’utente. Questo significa che qualora il worker dovesse rientrare nel caso precedentemente esposto e dunque ricevere due messaggi uguali, se tali messaggi non presentano delle violazioni non avverrà la doppia pubblicazione su Kafka.

Nel database del notifier è stato inoltre previsto un campo *sent*, il quale è indica se una data notifica è stata già inviata o meno. Ciò consente di mantenere uno storico delle notifiche inviate, per ognuna di esse vi è associato anche un timestamp per indicare il momento dell’invio. In più in caso di riavvio del microservizio permette di recuperare eventuali notifiche non ancora inviate; difatti il notifier appena avviato come prima azione controlla se ha nel suo DB delle notifiche da inviare.

**3.2) Gestione delle repliche**

Il sistema in esame è adatto alla replicazione dei suoi microservizi. Ciò consente di distribuire il carico di lavoro di ogni microservizio alle rispettive repliche, mantenendo la consistenza dei dati e permettendo una maggiore scalabilità del sistema stesso.

Più nel dettaglio, ogni istanza del worker e del notifier è caratterizzata da un proprio codice identificativo univoco (worker\_id, notifier\_id). Tale codice costituisce uno dei campi dei loro rispettivi DB. In tal modo ogni replica andrà ad agire solo sulle entries di sua pertinenza, velocizzando l’esecuzione delle query pur mantenendo un unico database condiviso tra le varie repliche. Questo è possibile poiché i precedenti microservizi non hanno bisogno di mantenere uno stato, semplicemente l’uso dei loro database è pensato solo per fornire un meccanismo di fault tolerance.

Per quanto riguarda il WMS, anch’esso può essere replicato. A differenza dei microservizi precedenti non è stato previsto l’utilizzo di un codice identificativo univo nel database, poiché il WMS ha il compito di mantenere l’elenco aggiornato delle regole di ciascun utente. Dato che un utente può modificare le proprie regole allora l’uso di un codice identificato, della specifica istanza del WMS, avrebbe ostacolato la modifica dei parametri metereologici degli utenti, perché sarebbe stato necessario indirizzare le richieste di ogni utente alla specifica istanza che ne detenesse le regole. Ciò avrebbe comportato la perdita della *Location transparency* e una diminuzione della scalabilità.

Tuttavia, il WMS è comunque replicabile essendo che le diverse repliche sfruttano un *timestamp* presente nel database che indica il momento una specifica regola è stata verificata, ovvero utilizzata per la costruzione di un messaggio Kafka, da una qualsiasi istanza del WMS.

Dunque, ogni istanza del WMS a intervalli regolari, controlla se esistono delle entries per cui sia trascorso il *trigger\_period* a partire dal valore presente nel campo *timestamp*. Se ciò si dovesse verificare allora l’istanza provvederebbe alla costruzione del messaggio Kafka inserendo quelle specifiche entries. Il campo *timestamp* verrà aggiornato solo quando l’istanza del WMS riceverà da Kafka l’ack della corretta consegna.

Durante l’attesa potrebbe capitare che un’altra istanza acceda al database e prenda in considerazione le stesse entries considerate nel messaggio Kafka di cui si attende l’ack, ciò comporterebbe la costruzione di due messaggi Kafka identici in un tempo minore rispetto a quello del *trigger\_period.*

Questo avrebbe come conseguenza la possibilità dell’invio di due notifiche esattamente identiche per lo stesso utente. Per risolvere tale problema, è stato aggiunto un campo booleano *checked*,nella tabella del database del WMS, che permette di segnalare che una certa istanza ha già preso in considerazione quella specifica entry, ha costruito un messaggio Kafka e ne attende l’ack.

Infine, anche lo User Manager è stato implementato in modo tale da essere replicabile, poiché vi è un database condiviso fra tutti gli UM service. Inoltre, avendo implementato un meccanismo di autenticazione basato sull’utilizzo di un token JWT, il quale viene incluso nell’header http di ogni richiesta proveniente dal client, ciascuna replica potrà autenticare l’utente senza che venga instaurata una sessione.

**3.3) Comunicazioni**

Le tecnologie di comunicazione che sono state sfruttate sono:

* **REST API**
* **gRPC**
* **Apache Kafka Message**

La comunicazione REST API viene utilizzata dal client (e.g. *Postman)* cosicché le richieste dell’utente possano arrivare all’api gateway *nginx*, a sua volta nginx sfruttare REST API per comunicare con i microservizi WMS, UM ed SLA manager, infine, anche Prometheus sfrutta tale tecnologia per effettuare lo scraping delle metriche. È stata scelta tale tecnologia per via della sua semplicità e per l’alta disponibilità di client già esistenti che eseguono richieste REST API. Nei cinque microservizi sviluppati nel sistema in esame (visibili in giallo nella figura 1 *Architettura del sistema*) è stato implementato un server Flask per esporre gli endpoint dei rispettivi microservizi.

La tecnologia gRPC viene utilizzata come comunicazione interna del sistema, ovvero per creare un canale da WMS a User Manager per verificare l’utente tramite l’utilizzo del token JWT, passato come parametro della richiesta gRPC. Inoltre, è utilizzata per far comunicare il Notifier con l’UM al fine di ottenere l’indirizzo e-mail dell’utente. È stata scelta questa tecnologia perché permette di avere alte prestazioni e non ha la necessità di esporre degli endpoint. Inoltre, il programmatore non si occupa degli aspetti espliciti della comunicazione, ma solo di definire le interfacce delle procedure remote, richiamandole come delle funzioni locali. Ciò rende più snella e chiara l’interazione tra i microservizi.

La tecnologia Apache Kafka Message è stata già discussa, in questo paragrafo è spiegato come vengono costruiti i messaggi Kafka da pubblicare sui topic *event\_update* e *event\_to\_be\_notified*.

Il WMS invece di pubblicare un messaggio per ogni entry nel suo database costruisce un messaggio Kafka per città di interesse, ovvero raggruppa tutti gli utenti interessati alla specifica località e aggiunge le regole di ciascun utente. Quindi verranno pubblicati tanti messaggi quante sono le località differenti, anziché il numero di entries. Questo evita di appesantire la comunicazione con il broker Kafka.

Supponendo ci siano dieci utenti interessati alla città di Catania e ognuno specifica dieci parametri da controllare, allora nel database del WMS ci saranno dieci entries, una per utente. Dunque, il WMS non costruirà dieci messaggi, bensì essendo tutti gli utenti interessati alla città di Catania costruirà un singolo messaggio contente tutte le informazioni riguardanti gli utenti e le rispettive regole.

Inoltre, il messaggio così costruito permette al worker, che è sottoscrittore al topic, di effettuare ad OpenWeather una sola richiesta per città. In questo modo può controllare tutti i parametri specificati da tutti gli utenti interessati a quella città con un’unica richiesta, riducendone al minimo la dipendenza dal servizio esterno.

Riprendendo l’esempio precedente, se ognuno dei dieci utenti ha indicato dieci parametri metereologici da controllare, quindi per un totale di cento parametri, il worker potrà verificarli tutti attraverso una singola richiesta.

Tale strategia favorisce la scalabilità dei microservizi, poiché riduce enormemente il carico di lavoro, sia del WMS, sia del broker Kafka, sia del worker.