Relazione DSBD Finocchiaro_Rovito

Sommario

Indice Tabelle

I. Abstract

Il progetto consiste nello sviluppo di un sistema distribuito basato su microservizi per gestire utenti e dati finanziari. L'applicazione consente agli utenti di registrarsi, aggiornare dati e associare uno o più ticker finanziari, tracciandone lo storico grazie all'integrazione con Yahoo Finance.
La comunicazione tra server e il client si basa su una comunicazione grpc, che consente di trasmettere richieste e risposte in modo rapido ed efficiente. Il sistema implementa la politica at-most-once, che assicura che alcune richieste vengano processate al massimo una sola volta, evitando esecuzioni multiple della stessa richiesta; quali: RegisterUser e UpdateUser. Per migliorare l'efficienza nella gestione del database, è stato adottato il pattern CQRS, che introduce una chiara separazione tra le operazioni di lettura e di scrittura.
Gli utenti potranno configurare soglie personalizzate (high-value e low-value) per i loro ticker preferiti che verranno monitorate,notificate attraverso Apache Kafka, che garantisce un'elaborazione scalabile e asincrona dei dati e delle notifiche. Quando i valori dei ticker vengono aggiornati il DataCollector invierà un messaggio a un topic Kafka chiamato to-alert-system per notificare l'aggiornamento dei dati. L'AlertSystem successivamente analizzerà il database e invierà notifiche tramite Kafka su un altro topic, to-notifier, per gli utenti i cui ticker hanno oltrepassato le soglie stabilite. Infine, l'AlertNotifierSystem invierà le notifiche agli utenti tramite e-mail, e se configurato, anche tramite Telegram.
Viene deciso rispetto all'implementazione precedente di distribuire i microservizi utilizzando Kubernetes, creando un cluster locale tramite lo strumento Kind. Kubernetes permetterà di orchestrare i vari componenti del sistema. Ogni microservizio sarà eseguito in un container isolato, garantendo resilienza e la capacità di gestire eventuali guasti senza compromettere l'intero sistema.
Introdotto nella nuova versione il monitoraggio delle performance dei microservizi,tramite il white-box monitoring utilizzando Prometheus. Ogni microservizio esporrà due tipi di metriche attraverso gli exporter Prometheus: una metrica di tipo GAUGE, e una metrica di tipo COUNTER, che terrà traccia di eventi incrementali come il numero di richieste ricevute o il numero di errori.
La combinazione dei microservizi, Kafka, Kubernetes e Prometheus, offre un sistema robusto, scalabile e performante, in grado di gestire notifiche personalizzate e carichi variabili con efficienza e affidabilità.

II. Descrizione delle principali funzionalità implementate

– At Most-Once

Viene implementato il meccanismo at most once su alcune operazioni ritenute più critiche per il sistema, garantendo che un'operazione venga eseguita al massimo una volta, evitando che lo stesso comando venga elaborato più volte, causando quindi inconsistenze o errori nel sistema.

Questo principio viene implementato tramite l'uso di identificativi univoci (come requestid e userid) e due principali meccanismi:

1. **Cache**: Ogni volta che un'operazione viene eseguita, come la registrazione di un utente o l'aggiornamento di un ticker, il risultato dell'operazione viene memorizzato in una cache. La cache funge da "memoria temporanea" per i risultati delle operazioni recenti. Se una nuova richiesta arriva con gli stessi dati, come l'ID utente o l'ID richiesta, il sistema verifica la cache. Se trova già una risposta associata, restituisce quella risposta senza eseguire nuovamente l'operazione. Questo evita che lo stesso lavoro venga ripetuto e permette di risparmiare tempo e risorse.
2. **Lock (Blocco)**: Per evitare che più processi accedano e modifichino la cache contemporaneamente, causando errori o conflitti, viene utilizzato un meccanismo di lock. Questo garantisce che solo una richiesta alla volta possa accedere e modificare la cache, evitando problemi di concorrenza. In pratica, quando una richiesta è in fase di elaborazione, altre richieste simili devono aspettare che la prima sia completata prima di essere elaborate.

Quindi quando arriva una nuova richiesta con gli stessi dati, il sistema controlla prima la cache e, se viene trovata una risposta esistente, restituisce quella risposta senza rieseguire l'operazione. Questo approccio riduce il carico sul sistema e garantisce che le operazioni siano gestite in modo coerente e sicuro.

– Implementazione del Pattern CQRS

Per migliorare la gestione delle operazioni di lettura e scrittura sul database, è stato introdotto il pattern **CQRS** (Command Query Responsibility Segregation), separando le operazioni in tre flussi distinti:

- **Command**: gestisce le operazioni che modificano lo stato del sistema (registrazione di nuovi utenti, aggiornamento dei profili).
- **Lecture**: si occupa delle operazioni di lettura, come la consultazione dei valori dei ticker e delle soglie associate agli utenti.
- **Connection:** gestisce la connessione del database evitando di ripeterlo in ogni file.

L'introduzione di CQRS migliora le prestazioni del sistema e ottimizza le operazioni di lettura, separando le preoccupazioni e facilitando la gestione dei dati in modo scalabile.

– Deployment su Kubernetes:

Rispetto alla precedente implementazione, l'applicazione è distribuita su piattaforma **Kubernetes**, utilizzando **Kind** (Kubernetes in Docker) per il cluster locale. I seguenti oggetti Kubernetes sono stati utilizzati per il deployment:

- **Deployment dei microservizi**: Ogni microservizio (DataCollector, AlertSystem, AlertNotifierSystem, ecc) è stato containerizzato e gestito tramite Kubernetes. Le immagini Docker vengono caricate nel cluster con i comandi kind load docker-image.
- **Service dei microservizi**: Ad ogni pod configurato tramite il file Deployment è associato un Service che rappresenta un’astrazione dell’indirizzo ip del pod, questo permette la comunicazione tra i vari servizi dentro k8s (ClusterIp) fornendo un indirizzo ip persistente e

nel caso si voglia far comunicare verso l'esterno per esporre il microservizio viene configurato come **NodePort.**

– Sistema di Notifica Asincrono con Kafka

Una delle richieste principali del progetto è stata l'implementazione di un sistema di notifiche che avvisa l'utente quando il valore di un ticker scende sotto o supera la soglia definita. Questo sistema è stato realizzato tramite l'uso di Kafka, un sistema di messaggistica asincrona. Grazie alla sua architettura, ci permette di trasferire i messaggi tra i vari componenti del sistema in modo rapido e affidabile. Inoltre, l'uso di Kafka ci consente di separare logicamente i produttori e i consumatori di dati, senza preoccuparsi direttamente di chi li utilizzerà o di come saranno elaborati. Questo approccio migliora la modularità del sistema e lo rende più flessibile. Un altro vantaggio importante è la persistenza dei messaggi. Anche in caso di guasti o interruzioni, Kafka garantisce che i messaggi non vadano persi e possano essere recuperati in un secondo momento. Questo aumenta l'affidabilità complessiva del sistema. Inoltre, scegliendo Kafka, ci prepariamo a gestire scenari futuri in cui il carico di lavoro o le necessità del sistema potrebbero aumentare. Kafka, infatti, è altamente scalabile e può crescere con il sistema, aggiungendo nuovi broker, partizioni e funzionalità senza richiedere modifiche radicali.

Viene deciso di utilizzare un solo broker Kafka per diverse ragioni. La prima è legata alla semplicità: implementare un singolo broker è più facile e richiede meno configurazioni. Infatti, il carico di lavoro è basso e non giustifica la complessità di un cluster con più broker. Inoltre, con un solo broker, possiamo ridurre l'uso di risorse hardware e semplificare la manutenzione. Questo ci consente di bilanciare la necessità di semplicità operativa con la possibilità di espansione futura, costruendo una base solida per il sistema senza complicare eccessivamente l'implementazione iniziale. Essendo anche tutto il sistema mantenuto in locale.

Vediamo come funziona Kafka:

- DataCollector: Il DataCollector è responsabile del recupero dei valori aggiornati dei ticker e dell'aggiornamento nel database. Dopo l'aggiornamento, invia un messaggio (producer) a un topic Kafka chiamato to-alert-system, notificando che l'aggiornamento è stato completato.
- AlertSystem: L'AlertSystem è un servizio che ascolta il topic to-alert-system, quindi consumatore e produttore del topic to-notifier. Una volta ricevuto un messaggio, il sistema controlla se i valori dei ticker sono al di sopra del high-value o al di sotto del low-value. Se una soglia è superata, l'AlertSystem invia un messaggio al topic to-notifier con le informazioni necessarie per inviare la notifica (e-mail, chat-id, ticker, condizione di superamento della soglia).
- AlertNotifierSystem: L'AlertNotifierSystem è un consumatore del topic to-notifier. Una volta ricevuto il messaggio, invia un'e-mail all'utente, utilizzando le informazioni fornite (e-mail, ticker, soglia superata). Se il valore del ticker supera la soglia superiore, l'e-mail avvisa che il ticker ha superato la soglia superiore , mentre se scende sotto la soglia inferiore, l'e-mail avvisa che il ticker è sceso sotto la soglia inferiore.

Oltre all'invio dell'e-mail se correttamente configurato è possibile inviare una notifica su Telegram con le stesse informazioni.

– White-box Monitoring con Prometheus:

È stato introdotto nell'implementazione seguente un sistema di monitoraggio delle performance con Prometheus.

I microservizi forniscono almeno due di queste metriche:

- Una metrica di tipo **GAUGE** per misurare variabili dinamiche come il tempo di risposta o il tempo di aggiornamento.
- Una metrica di tipo **COUNTER** per monitorare eventi incrementali come il numero di richieste ricevute, il numero di errori o il numero di invocazioni di una funzione.
- Una metrica di tipo **HISTOGRAM**, che consente di raccogliere e analizzare la distribuzione di un particolare valore, utile per analizzare la distribuzione di valori in un intervallo, come il tempo di latenza, e fornisce una panoramica completa dei valori estremi e delle tendenze, invece di concentrarsi solo sulla media.

Le metriche sono etichettate con il nome del servizio e il nodo che ospita l'esecuzione, consentendo un monitoraggio dettagliato e la possibilità di analizzare le performance per singolo microservizio.

➢ Integrazione con Alertmanager:

Il sistema utilizza Alertmanager per la gestione degli allarmi, garantendo una risposta tempestiva a eventi critici. La configurazione del sistema prevede:

- **Rilevamento automatico delle anomalie** nelle metriche raccolte da Prometheus, come servizi inattivi o un aumento anomalo degli errori.
- **Routing personalizzato** degli allarmi che inviano, come configurato all'indirizzo predefinito utilizzando Gmail come smarthost.
- **Raggruppamento e gestione avanzata**:
    - Le notifiche sono raggruppate per ridurre notifiche superflue, con un'attesa iniziale di 30 secondi.
    - Gli intervalli di notifica sono configurati a 5 minuti per i nuovi allarmi e a 1 ora per gli allarmi ripetuti, assicurando un equilibrio tra tempestività e gestione efficiente degli incidenti.
- **Persistenza dei dati degli allarmi**: la configurazione utilizza un PersistentVolumeClaim (PVC) da 2 GiB per mantenere lo stato degli allarmi, garantendo continuità anche in caso di riavvio o interruzione del sistema.

L'integrazione con Alertmanager offre numerosi vantaggi per la gestione degli avvisi e il monitoraggio del sistema. Garantisce che vengano inviate notifiche affidabili a canali predefiniti (ad esempio e-mail) riducendo i falsi positivi grazie ad un sistema avanzato di configurazione delle regole di routing. Inoltre, raggruppa avvisi simili e applica filtri per evitare notifiche irrilevanti o ridondanti permettendo la gestione proattiva degli incidenti e consentendo ai team di identificare, risolvere i problemi prima che abbiano un impatto grave sul servizio. Ciò non solo migliora

l’affidabilità complessiva del sistema, ma garantisce anche risposte più rapide ed efficaci ai problemi critici.

➢ Integrazione con l'esportatore del nodo:

Il sistema include Node Exporter, uno strumento che raccoglie informazioni sulle risorse di sistema come CPU, memoria, rete e utilizzo dello spazio di archiviazione da ciascun nodo del cluster. Ciò ti consente di monitorare non solo le prestazioni dei tuoi microservizi, ma anche le risorse della tua infrastruttura, offrendoti un quadro completo dello stato di salute del tuo sistema.
Node Exporter garantisce che tutte le risorse siano monitorate in modo uniforme. Le informazioni raccolte vengono inviate a Prometheus ed elaborate per fornire un quadro accurato delle prestazioni delle risorse di sistema.

III. Principali microservizi:

– Server:

- **Funzione:** Componente fondamentale del sistema, progettata per gestire le richieste e garantire l'elaborazione dei dati in modo scalabile ed efficiente.
- **Ruolo nell'architettura**:
  - **Gestione delle richieste del client:** Le richieste vengono ricevute tramite il protocollo grpc permettendo una comunicazione bidirezionale.
  - **Interazione con il database:** Il sever interagisce direttamente con il database per archiviare o modificare informazioni, così da fornire i dati richiesti al client.
  - **Gestione delle API:** Il server gestisce le varie operazione con la chiamata di diverse API che consento ai client di interagire con il sistema.

– Data Collector:

- **Funzione:** Si occupa di recuperare periodicamente i dati associati ai titoli azionari, utilizzando la libreria yfinance.
- **Ruolo nell'architettura**:
  - **Recupero dati**: Recupero elenco dei tickers per aggiornare successivamente i dati
  - **Inserimento dati**: Inserisce i dati aggiornati periodicamente all’interno del database per garantire la persistenza dei dati e una successiva consultazione.

– Database:

- **Funzione**: Fondamentale per rendere i dati persistenti, utilizzando la politica per la gestione e archiviazione delle informazioni basate su MySQL.
- **Ruolo nell'architettura**:
  - **Archiviazione:** Permette di archiviare utenti, ticker e dati storici relativi ai titoli azionari includendo sia i valori che i timestamp associati.
  - **Relazione**: Permette di stabilire la relazione tra i diversi dati come tra utenti e i tickers, consentendo di associare ad ogni utente il ticker d’interesse.

- ## Zookeeper:
    - **Funzione**: Zookeeper è utilizzato per la gestione dei broker Kafka. Agisce come sistema di coordinamento e gestione dei metadati per il Kafka cluster, assicurando che i broker Kafka possano operare correttamente in un ambiente distribuito.
    - **Ruolo nell'architettura**: Gestisce la configurazione di Kafka e coordina la comunicazione tra i broker per garantire l'affidabilità e la coerenza del sistema.
- ## Schema Registry:
    - **Funzione**: Lo Schema Registry gestisce gli schemi dei dati scambiati tra i microservizi, in particolare tra il sistema Kafka e gli altri componenti. Fornisce una gestione centralizzata degli schemi per i messaggi Kafka, garantendo che i dati inviati e ricevuti siano conformi agli schemi definiti.
    - **Ruolo nell'architettura**: Assicura che i dati scambiati tra i microservizi siano validi, prevenendo errori di serializzazione e garantendo la compatibilità tra le diverse versioni degli schemi.
- ## Kafka Broker :
    - **Funzione**: Kafka funge da sistema di messaggistica asincrona. I broker Kafka sono responsabili della gestione dei topic e dei messaggi, facilitando la comunicazione tra i microservizi in modo altamente scalabile e resiliente.
    - **Ruolo nell'architettura**: Kafka è utilizzato per orchestrare la comunicazione asincrona tra i vari microservizi, consentendo l'aggiornamento e la notifica degli utenti tramite flussi di dati asincroni (event-driven).
- ## Telegram Bot Service:
    - **Funzione**: Questo servizio implementa un bot Telegram che interagisce con gli utenti per associare un’e-mail alla loro chat_id, salvando queste informazioni nel database.
    - **Ruolo nell'architettura**:
        - **Gestione delle richieste utenti**: Riceve messaggi dagli utenti tramite l'API di Telegram e li elabora per raccogliere dati.
        - **Validazione dei dati**: Controlla se il chat_id di un utente è già associato a un'email nel database.
        - **Persistenza dati**: Salva le associazioni dell’email con il chat_id nel database.
        - **Comunicazione**: Invia messaggi agli utenti per guidarli nel completamento della configurazione.
- ## Alert System Microservice:
    - **Funzione**: Questo microservizio si occupa di monitorare i dati aggiornati dei tickers e di determinare se un'azione deve essere intrapresa in base alle soglie configurate per ogni utente.
    - **Ruolo nell'architettura**:

- **Monitoraggio delle soglie**: Verifica se il valore di un ticker supera la soglia massima o scende sotto la soglia minima.
- **Generazione di eventi di alert**: Quando una soglia viene superata, invia un messaggio al sistema di notifica per informare gli utenti.

Alert Notifier System Microservice:

- **Funzione**: Il sistema di notifiche gestisce l'invio delle notifiche via e-mail o su altre piattaforme agli utenti quando le condizioni di alert sono soddisfatte.
- **Ruolo nell'architettura**: Riceve eventi dal sistema di alert e invia notifiche agli utenti per informarli dei cambiamenti nel valore dei loro tickers preferiti.

Prometheus:

- **Funzione:** Il sistema di monitoraggio delle performance con Prometheus raccoglie metriche dai microservizi per monitorare variabili dinamiche.
- **Ruolo nell'architettura:**
  - **Raccolta delle metriche:** Raccoglie metriche di tipo GAUGE (per misurare variabili dinamiche come il tempo di risposta) e COUNTER (per monitorare eventi incrementali come richieste, errori e invocazioni).
  - **Monitoraggio delle performance:** Le metriche sono etichettate con il nome del servizio e il nodo che ospita l'esecuzione, permettendo un monitoraggio dettagliato per ciascun microservizio e nodo, offrendo così una visibilità completa sulle performance del sistema.

Di seguito verranno elencate le metriche implementate per ogni microservizio:

- **Server**
  - **REQUEST_COUNT:**
    - **Tipo di metrica**: Counter**.**
    - **Descrizione:** Conta il numero totale di richieste gRPC ricevute dal server, classificandole per metodo (method), esito (success), servizio (service) e nodo (node). Questa metrica consente di monitorare il volume e l'esito delle richieste elaborate dal server.
  - **ERROR_COUNT:**
    - **Tipo di metrica**: Counter**.**
    - **Descrizione:** Conta il numero totale di errori verificatisi durante l'elaborazione delle richieste gRPC, suddivisi per metodo (method), servizio (service) e nodo (node). Aiuta a individuare problemi specifici nel sistema e a migliorare la stabilità.
  - **USER_SESSION_DURATION:**
    - **Tipo di metrica**: Gauge**.**
    - **Descrizione:** Misura la durata della sessione utente in secondi, con suddivisione per servizio (service) e nodo (node). Questa metrica consente di analizzare il comportamento degli utenti e il tempo che trascorrono interagendo con il sistema.
  - **REQUEST_DURATION:**
    - **Tipo di metrica**: Histogram.
    - **Descrizione:**

        - Monitora la durata delle richieste gRPC in secondi, classificandole per metodo (method), esito (success), servizio (service) e nodo (node). Fornisce informazioni dettagliate sulle prestazioni del server e sulla latenza degli endpoint gRPC.
- **Alert Notifier:**
    - **EMAIL_SENT:**
        - **Tipo di metrica**: Counter
        - **Descrizione**: Conta il numero totale di e-mail inviate con successo, suddivise per tipo di messaggio (message_type), servizio (service) e nodo (node). Questa metrica consente di monitorare il volume di e-mail inviate correttamente dal sistema.
    - **TELEGRAM_MESSAGE_SENT:**
        - **Tipo di metrica**: Counter
        - **Descrizione:** Conta il numero totale di messaggi Telegram inviati con successo, distinguendoli per tipo di messaggio (message_type), servizio (service) e nodo (node). È utile per analizzare il traffico dei messaggi Telegram gestiti dal sistema.
    - **EMAIL_SEND_ERRORS:**
        - **Tipo di metrica**: Counter
        - **Descrizione:** Registra il numero totale di errori durante l'invio di e-mail, suddivisi per servizio (service) e nodo (node). Aiuta a identificare eventuali problemi o instabilità nel processo di invio delle e-mail.
    - **TELEGRAM_SEND_ERRORS:**
        - **Tipo di metrica**: Counter
        - **Descrizione:** Conta il numero totale di errori verificatisi durante l'invio di messaggi Telegram, distinguendoli per servizio (service) e nodo (node). È utile per monitorare la stabilità del sistema e rilevare eventuali malfunzionamenti.
    - **EMAIL_SEND_LATENCY:**
        - **Tipo di metrica**: Gauge
        - **Descrizione:** Misura la latenza dell'invio delle e-mail (tempo impiegato per completare l'invio), classificandola per servizio (service), nodo (node) e tipo di messaggio (message_type). Fornisce un’indicazione delle prestazioni del sistema per le e-mail.
    - **TELEGRAM_SEND_LATENCY:**
        - **Tipo di metrica**: Gauge
        - **Descrizione:** Monitora la latenza dell'invio di messaggi Telegram (tempo necessario per inviarli), suddivisa per servizio (service), nodo (node) e tipo di messaggio (message_type). Questa metrica aiuta a ottimizzare le prestazioni del sistema per l'invio di messaggi Telegram.
- **Alert System:**
    - **MESSAGES_CONSUMED:**
        - **Tipo di metrica**: Counter
        - **Descrizione:** Conta il numero totale di messaggi consumati dal topic to-alert-system, suddivisi per servizio (service) e nodo (node). Questa metrica aiuta a monitorare il flusso di messaggi in ingresso al sistema.
    - **MESSAGES_PRODUCED:**
        - **Tipo di metrica**: Gauge

        - **Descrizione:** Registra il numero totale di messaggi prodotti sul topic to-notifier, classificandoli per servizio (service) e nodo (node). Fornisce una visione del traffico in uscita dal sistema.
    - **MESSAGES_PENDING:**
        - **Tipo di metrica**: Gauge
        - **Descrizione:** Monitora il numero di messaggi attualmente in attesa di essere prodotti, distinguendoli per servizio (service) e nodo (node). Questa metrica è utile per individuare eventuali colli di bottiglia o problemi nella gestione della coda.
    - **PROCESSING_TIME:**
        - **Tipo di metrica**: Histogram
        - **Descrizione:** Misura il tempo necessario per elaborare un singolo messaggio, registrandolo come un Histogram. Suddivide i dati per servizio (service) e nodo (node) e fornisce informazioni dettagliate sulle prestazioni dell'elaborazione dei messaggi.
    - **DB_QUERY_LATENCY:**
        - **Tipo di metrica**: Histogram
        - **Descrizione:** Registra la latenza delle query al database (tempo impiegato per completare una query), classificandola per servizio (service) e nodo (node). Questa metrica è fondamentale per identificare problemi di prestazioni legati al database.
- **Data Collector:**
    - **PROCESSED_TICKERS:**
        - **Tipo di metrica**: Counter
        - **Descrizione:** Conta il numero totale di ticker processati dal data-collector, suddivisi per servizio (service) e nodo (node). Questa metrica aiuta a monitorare l'efficienza del sistema nel gestire i ticker.
    - **TICKER_PROCESSING_ERRORS:**
        - **Tipo di metrica**: Counter
        - **Descrizione:** Registra il numero totale di errori nell'elaborazione dei ticker, classificati per servizio (service) e nodo (node). Aiuta a monitorare i problemi nel processo di elaborazione dei ticker e a garantire l'affidabilità del sistema.
    - **LAST_RUN_DURATION:**
        - **Tipo di metrica**: Gauge
        - **Descrizione:** Misura la durata (in secondi) dell'ultima esecuzione completa della funzione run(), distinguendo tra servizio (service) e nodo (node). Questa metrica consente di monitorare quanto tempo impiega il sistema a completare un ciclo di esecuzione della funzione.
    - **TICKER_PROCESSING_DURATION:**
        - **Tipo di metrica**: Histogram
        - **Descrizione:** Monitora la durata in secondi dell'elaborazione di un singolo ticker, suddivisa per servizio (service) e nodo (node). Fornisce informazioni dettagliate sulla velocità e sull'efficienza del processo di elaborazione dei ticker.
- **Telegram Bot:**
    - **MESSAGE_SENT_COUNT:**
        - **Tipo di metrica**: Counter

        - **Descrizione:** Conta il numero totale di messaggi inviati tramite il bot Telegram, suddivisi per tipo di messaggio (message_type), servizio (service) e nodo (node). Questa metrica aiuta a monitorare l'attività del bot, distinguendo tra i diversi tipi di messaggi inviati.
    - **ERROR_COUNT:**
        - **Tipo di metrica**: Counter
        - **Descrizione:** Registra il numero totale di errori nelle richieste API Telegram, classificati per tipo di errore (error_type), servizio (service) e nodo (node). Questa metrica è utile per rilevare e diagnosticare eventuali problemi con le comunicazioni API di Telegram.
    - **TELEGRAM_API_RESPONSE_LATENCY:**
        - **Tipo di metrica**: Gauge
        - **Descrizione:** Misura la latenza delle risposte provenienti dall'API Telegram in secondi, classificando i dati per servizio (service) e nodo (node). Aiuta a monitorare la velocità e l'efficienza delle comunicazioni tra il bot e l'API di Telegram.

Alertmanager:

- **Funzione:** Il sistema utilizza Alertmanager per la gestione automatica degli allarmi, garantendo una risposta tempestiva a eventi critici come anomalie nel sistema.
- **Ruolo nell'architettura:**
    - **Gestione degli allarmi:** Riceve gli allarmi da Prometheus e li instrada verso canali predefiniti (ad esempio, e-mail) e li raggruppa per evitare notifiche superflue.
    - **Notifiche intelligenti:** Le notifiche vengono inviate dopo un’attesa iniziale e a intervalli configurabili.

Node Exporter:

- **Funzione:** Node Exporter raccoglie informazioni dettagliate sulle risorse di sistema completando il monitoraggio delle performance dei microservizi.
- **Ruolo nell'architettura:**
    - **Monitoraggio delle risorse:** Raccoglie metriche sui componenti infrastrutturali e invia i dati a Prometheus, che li elabora per fornire una visibilità accurata delle risorse del sistema.
    - **Sicurezza e distribuzione:** Node Exporter è configurato per essere eseguito su tutti i nodi del cluster, raccogliendo dati in modo uniforme e sicuro.

IV. Interazioni tra i Componenti

Client ↔Server:

- Il client comunica mandando richieste gRPC al Server. Nello specifico il client prepara la richiesta inviando i parametri necessari e sulla base delle operazioni che vuole effettuare, stabilendo un canale grpc. Inoltre, il client permette di visualizzare i dati nel corretto formato

che saranno restituiti dalla richiesta effettuata precedentemente, permettendo successive operazioni tramite interfaccia sui dati in possesso.

- Il server appena riceve e verifica la richiesta, elabora e genera i dati necessari. Successivamente il server restituisce al client una risposta contenente i dati elaborati e un messaggio che indica il corretto funzionamento dell'operazione. La risposta viene mandata sempre via canale gRPC precedentemente stabilito.

– Server ↔Database:

- Il server comunica direttamente con il database effettuando operazioni CRUD.
  - **Create**: effettuando la chiamata dell'API RegisterUser il server va ad inserire i nuovi utenti nella tabella con rispettivo Ticker con AddTicker va ad associare un nuovo Ticker ad un utente ed eventualmente aggiungerlo se non presente.
  - **Read**: Attraverso le chiamate Login, ShowTickersUser, GetLatestValue e GetAverageValue, va ad eseguire delle azioni di recupero dei dati dal database
  - **Update**: Nell'API di UpdateUser, permette di modificare i ticker associati ad un determinato utente.
  - **Delete**: tramite DeleteUser, DeleteTickerUser permettono al server di eliminare nelle rispettive tabelle e grazie alla relazione in cascata anche nelle altre associate.

Inoltre, svolge operazioni di aggregazione utilizzato nel calcolo della media e di controllo come verifica la presenza di dati all'interno del databse

- Il database restituisce al server i dati richiesti tramite le query e permettere così poi al server di rispondere all'utente.

– Database ↔ Data Collector:

- Il Database fornisce i ticker presenti che vengono utilizzati successivamente dal data collector per capire quali dati aggiornare e recuperare.
- Il Data Collector responsabile di mantenere aggiornati i dati associati ai tickers comunica direttamente con il database. In specifico:
- Recupera l'elenco dei ticker dal database.
- Aggiorna valori associati ai ticker in maniera periodica.

– Data Collector ↔ Circuit Breaker

- Il DataCollector per controllare e gestire le chiamate verso le risorse utilizza il circuit breaker, consentendo al sistema di rilevare errori, timeout e permettendo il funzionamento anche quando le risorse non sono disponibili
- Il circuit breaker si comporta come una sorta di controllore; infatti, caso di errore o stato aperto il circuit breaker interrompe il flusso.

– Circuit Breaker ↔ Yahoo Finance

- Il circuit breaker nei confronti di yahoo finance si comporta come una sorta di meccanismo di protezione per le richieste e il recupero dei dati. Permette:

    - **Protezione**: Infatti quando viene invocata la richiesta per accedere a yahoo finance il circuit breaker monitora le richieste rilevando errori timeout.
    - **Valutazione flusso**: Viene eseguita una valutazione del flusso nei seguenti stati:
        - **Stato Closed:** se non sono stati rilevati errori il circuit breaker consente al Data Collector di inoltrare le richieste e il risultato torna indietro al Data Collector.
        - **Stato Open:** se vengono rilevati un numero elevato di errori consecutivi, apre il circuito e blocca le richieste del Data Collector evitando richieste verso la risorsa e viene segnalato tramite un messaggio di errore, permettendo di evitare ulteriori fallimenti o timeout su servizi esterni.
        - **Stato Half-Open:** Dopo un periodo di timeout, il circuit breaker entra in questo stato che permette di testare la risorsa ed effettuare un numero limitato di richieste. Nel caso in cui le richieste abbiano successo il circuit breaker torna nello stato "Closed" altrimenti torna nello stato "Open".
- Yahoo Finance, quindi, fornisce le informazioni e i dati necessari. In caso di errori vengono gestiti dal circuit breaker.

– DataCollector ↔ Kafka:

- **DataCollector** raccoglie dati finanziari utilizzando la libreria **yfinance**. Per ogni ticker (titolo azionario), esegue una richiesta per ottenere l'ultimo valore disponibile.

  Una volta raccolto il dato, il **DataCollector** invia un messaggio notificando l'aggiornamento a un **topic Kafka** specifico. Nel caso specifico, i dati vengono inviati al topic **to-alert-system**.

– AlterSystem ↔ Database:

- L'**AlterSystem** interroga il **Database** per ottenere ulteriori informazioni sui ticker come recuperare i valori minimi, massimi.
- Il **Database** risponde con i dati necessari per completare l'elaborazione dell'**AlterSystem**, che poi procederà a generare un messaggio di allerta se vengono soddisfatte le condizioni predefinite.

– AlterSystem ↔ Kafka:

- Dopo aver elaborato le informazioni e verificato le condizioni, l'**AlterSystem** produce messaggi di allerta e li invia a un altro **topic Kafka** (to-notifier). **Kafka** riceve i messaggi dal **DataCollector** e li rende disponibili nei suoi topic. L'**AlterSystem** consuma questi messaggi dal topic **to-alert-system** e li elabora.Durante l'elaborazione, l'**AlterSystem** esegue ulteriori controlli tramite il Database per determinare se i dati ricevuti soddisfano determinati criteri (ad esempio, se un ticker ha raggiunto un valore che richiede una notifica).
- Una volta verificato che le condizioni siano soddisfatte, l'**AlterSystem** invia il messaggio di allerta al topic **to-notifier**, utilizzando un altro produttore Kafka.

– Kafka ↔ AlertNotifierSystem:

- **AlertNotifierSystem** consuma i messaggi di allerta generati dall'**AlterSystem** e disponibili su Kafka.

- Questi messaggi vengono processati dall'**AlertNotifierSystem** per creare notifiche indirizzate all'utente. L'**AlertNotifierSystem** può inviare notifiche via **e-mail** o **Telegram**.

– Telegram bot↔ database:

- Il bot ha un'interazione chiave con il database per memorizzare e validare le informazioni degli utenti. Le azioni specifiche sono:
  - **Validazione Dati**: Verifica se il chat_id è già associato a un'email nel database. Se sì, informa l'utente che l'associazione esiste già.
  - **Persistenza Dati**: Salva l'associazione (chat_id-email) nel database se non esiste già. Questo consente di inviare notifiche future.
- Il database esegue lettura (per verificare associazione) e scrittura (per memorizzare nuova associazione).

– Telegram bot↔ Client:

Il bot interagisce direttamente con l'utente tramite l'API di Telegram, guidando il processo di raccolta e gestione delle informazioni. Le azioni specifiche verso l'utente sono:

- Gestione delle Richieste: Riceve chat_id e e-mail dagli utenti per configurare l'account.
- Comunicazione: Invia messaggi di conferma, errore o guida:
  - Conferma se l'associazione è riuscita.
  - Notifica se l'e-mail è già associata al chat_id.
  - Guida l'utente se l'e-mail è errata o se è necessario completare l'associazione.

– Prometheus ↔ /Metrics **(varie componenti)**

Prometheus raccoglie metriche esposte da diverse componenti tramite endpoint /metrics.

- Ogni componente del sistema (ad esempio Data Collector, Kafka, Alert System, ecc.) espone metriche sul proprio stato o sui dati elaborati tramite l'endpoint /metrics.
- Prometheus interroga periodicamente questi endpoint per raccogliere informazioni, come il numero di richieste gestite, tempi di risposta, errori e altre metriche operative.Questi dati vengono poi salvati e resi disponibili per ulteriori analisi o visualizzazioni.

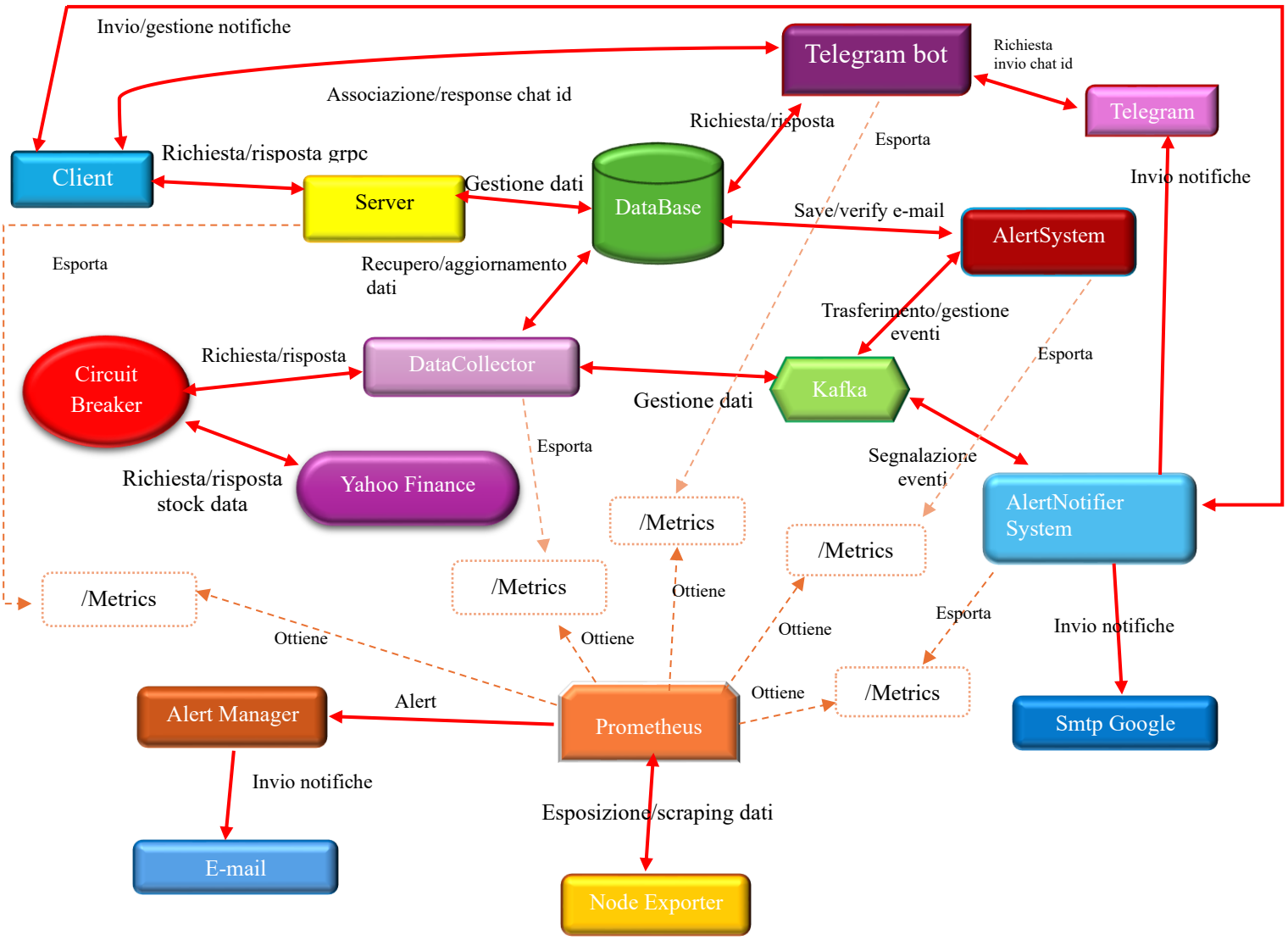
– Prometheus → Alert Manager

- Prometheus invia alert all'Alert Manager, che analizza le metriche raccolte e le confronta con regole definite (alerting rules). Quando una condizione specifica (ad esempio, un valore fuori soglia) viene soddisfatta, Prometheus genera un alert e lo invia all'Alert Manager.

– Alert Manager→E-mail

- Una volta ricevuto un avviso, l'Alert Manager invia notifiche utilizzando vari canali di comunicazione. In questo caso, le notifiche vengono inviate via **e-mail** al destinatario configuratoNode Exporter

- Il Node Exporter raccoglie informazioni dal sistema operativo e dall'hardware di ogni nodo (come CPU, memoria, spazio su disco, e traffico di rete). Queste informazioni vengono esposte su un endpoint HTTP accessibile da Prometheus.
- Prometheus raccoglie(**scraping)** regolarmente i dati esposti dal Node Exporter ed è effettuato ogni pochi secondi o minuti



V. API Implementate

- **RegisterUser**: Permette di registrare un nuovo utente, fornendo la propria e-mail, un ticker che deve essere valido (presente nel file CSV) e i valori massimo e minimo per il ticker. Il sistema restituisce un messaggio di successo o errore, con un valore booleano che indica se la registrazione è avvenuta con successo
- **LoginUser**: Permette all'utente già registrato di effettuare il login al sistema, inviando l'e-mail. Dopo che il sistema ha verificato l'effettiva presenza dell'e-mail nel database, viene

mandato un messaggio e valore boolean che mostra se la richiesta è avvenuta con successo o si è verificato un errore.

- **AddTickerUtente**: Permette all'utente di aggiungere un nuovo ticker. Il ticker deve essere valido (presente nel file CSV) ed è introdotta la possibilità di fornire i valori massimo e minimo per il ticker. Il sistema aggiunge il ticker al database se non è presente, altrimenti aggiunge semplicemente l'associazione e restituisce un messaggio di conferma o errore.
- **ShowTickersUser**: Permette di visualizzare tutti i ticker associati all'utente introducendo la visualizzazione dei relativi valori di max e min, senza dover fornire manualmente l'e-mail poiché l'utente è già autenticato. Il sistema restituisce i ticker associati all'utente e un messaggio che indica se l'operazione è stata completata con successo
- **UpdateUser**: Permette all'utente di sostituire un proprio ticker con la possibilità di inserire dei valori di max e min per il ticker che si modifica. L'e-mail non deve essere fornita, poiché l'utente è già autenticato. Il sistema aggiorna il ticker nel database e restituisce un messaggio di successo o errore.
- **DeleteTickerUser:** Permette di eliminare un ticker associato all'utente. Viene inserito un ticker e viene mandato al server che eliminerà il ticker associato a quell'utente e se non associati ad altri utenti verrà eliminato anche dal sistema.
- **GetLatestValue**: Restituisce il valore più recente del ticker inserito. Non è necessario mandare l'e-mail perché il sistema è già a conoscenza grazie al login effettuato in precedenza. Il sistema risponde alla richiesta con l'ultimo valore del ticker insieme al suo timestamp e restituendo un messaggio e un valore boolean che mostra se l'operazione è avvenuta con successo o si è verificato un errore.
- **GetAverageValue**: Calcola e restituisce il valore medio del ticker inserito. Non è necessario mandare l'e-mail perché il sistema è già a conoscenza grazie al login effettuato in precedenza, ma viene chiesto di quanti valori effettuare la media. Il sistema risponde alla richiesta con la media dei valori, il suo timestamp e restituendo un messaggio e un valore boolean che mostra se l'operazione è avvenuta con successo o si è verificato un errore.
- **DeleteUser**: Permette di eliminare l'utente ed eliminare i ticker che sono associati esclusivamente ad esso e non ad altri user. Non è necessario inviare alcun dato, poiché il sistema successivamente al login è già a conoscenza della e-mail. Il sistema risponde alla richiesta con un messaggio e un valore boolean che mostra se l'eliminazione è avvenuta con successo o si è verificato un errore.
- **UpdateMinMaxValue**: Permette di aggiornare i valori massimo e minimo di un ticker. Il sistema aggiorna i valori nel database e restituisce un messaggio di conferma o errore e il ticker aggiornato.

API	Descrizione	Input	Output
RegisterUser	Registra un nuovo utente e associa un ticker con valori massimo e minimo.	E-mail, ticker, max_value, min_value	Messaggio, valore boolean in caso di conferma o errore
LoginUser	Login utente	E-mail	Messaggio, valore boolean in caso di conferma o errore

AddTickerUtente	Aggiunge un ticker all'utente con valori massimo e minimo.	Ticker, max_value, min_value	Messaggio, valore boolean in caso di conferma o errore
ShowTickersUser	Mostra i ticker associato all'utente	Non viene mandato nulla perché è implementata la funzione login.	valore boolean in caso di conferma o errore. Ritorna tutti i ticker associati all'utente.
UpdateUser	Aggiorna un ticker esistente con un nuovo ticker e nuovi valori massimo e minimo.	Vecchio ticker, Nuovo ticker, max_value, min_value	Messaggio, valore boolean in caso di conferma o errore
DeleteTickerUser	Elimina un ticker associato all'utente	Ticker	Messaggio, valore boolean in caso di conferma o errore
GetLatestValue	Restituisce l'ultimo valore per un ticker	Ticker	Messaggio, valore boolean in caso di conferma o errore. Il ticker, ultimo valore e timestamp
GetAverageValue	Restituisce la media dei valori per un ticker	Ticker e numero che indica la media di quanti valori si vuole fare	Messaggio, valore boolean in caso di conferma o errore, Il ticker, la media e il timestamp
DeleteUser	Elimina un utente e i dati associati	Non viene mandato nulla perché è implementato la funzione login e-mail.	Messaggio, valore boolean in caso di conferma o errore
UpdateMinMaxValue:	Modifica valori di max e min che vengono monitorati da un dato ticker	Ticker,new_max_value,new_min_value	Messaggio, valore boolean in caso di conferma o errore, ticker aggiornato

Tabella 1 Mostra le API implementate

VI. Struttura Database

Si divide in quattro tabelle: Users, che gestisce gli utenti tramite e-mail, Tickers, gestisce i codici univoci per i ticker finanziari, UserTickers, che associa gli utenti ai ticker d'interesse e TickerData permette di memorizzare i dati storici relative a ticker.

– Users

Contiene gli utenti registrati nel sistema e vengono identificati in maniere univoca tramite il campo e-mail che costituisce la chiave primaria.

Nome Campo	Tipo di Dato	Chiave	Relazione	Descrizione
e-mail	VARCHAR (255)	Primary key		Indirizzo e-mail univoco che identifica l'utente
Chat id	VARCHAR (255)			Id che identifica la chat con il telegram bot (default:null)

Tabella 2 USERS

– Tickers

Contiene i ticker associati a titoli finanziari. Il ticker costituisce la chiave primaria, permettendo l'identificazione in maniera univoca.

Nome	Tipo di Dato	Chiave	Relazione	Descrizione
ticker	VARCHAR (10)	Primary key		Identificativo univoco

Tabella 3 Tickers

– UserTickers

Permette di stabilire la relazione tra utenti e ticker, gestendo le soglie massima e minima dei tickers che l'utente desidera monitorare. La combinazione di user e ticker costituisce la chiave primaria. Inoltre, entrambi si collegano come chiave esterna alla tabella User e Ticker, garantendo l'autoeliminazione nel caso viene eliminato un utente o un ticker associato a quell'utente grazie al comando ON DELETE CASCADE.

Nome	Tipo di Dato	Chiave	Relazione	Descrizione
user	VARCHAR (255)	Primary key, Foreign Key	Collegato a **Users.email** (ON DELETE CASCADE)	E-mail dell'utente che possiede il ticker.
ticker	VARCHAR (10)	Primary key, Foreign Key	Collegato a **Tickers.ticker** (ON DELETE CASCADE)	Ticker posseduto dall'utente.
Max_value	FLOAT			Soglia massima del ticker da controllare
Min_value	FLOAT			Soglia minima del ticker da controllare

Tabella 4 UserTickers

– TickerData

Contiene i dati storici relative ai ticker. Il campo id viene utilizzato come chiave primaria ed è univoco, generato automaticamente. La colonna ticker chiave esterna collega alla tabella tickers. La colonna value memorizza il valore numerico del ticker mentre la colonna timestamp tiene conto della data e l'ora in cui viene registrato. Tramite il comando ON DELETE CASCADE ticker permette l'autoeliminazione nel caso in cui non siano collegato a nessun ticker.

Nome	Tipo di Dato	Chiave	Relazione	Descrizione
id	INT AUTO_INCREMENT	Primary Key		Identificativo univoco
ticker	VARCHAR (10)	Foreign Key	Collegato a **Tickers.ticker** (ON DELETE CASCADE)	Identificativo del ticker
value	FLOAT			Valore associato al ticker.
timestamp	TIMESTAMP			Data e ora in cui è stato registrato il valore.

Tabella 5 TickerData

Documento redatto da:

Massimiliano Finocchiaro, Dario Rovito