Distributed System and Big Data

A.A. 2023-2024

News Notifier App

Nunzio Saitta 1000025485

Francesco Denaro 1000050594

Indice

1. Abstract

Il progetto proposto si focalizza sulla progettazione e implementazione di un sistema di distribuzione di notizie informative, mirando a ottimizzare l'esperienza degli utenti attraverso l'utilizzo di microservizi. Questi microservizi, organizzati in container dedicati mediante la tecnologia **Docker**, costituiscono un modello architetturale robusto e flessibile.

Docker, attraverso la sua capacità di creare container isolati con tutte le dipendenze necessarie, semplifica la distribuzione e l'esecuzione dell'applicazione. Ciò non solo favorisce la portabilità su diverse infrastrutture, ma migliora anche la scalabilità e la gestione efficiente delle risorse, ottimizzando il deployment.

Inoltre, è stato integrato **Kubernetes** come orchestratore di container per gestire la distribuzione, la scalabilità e il monitoraggio del sistema. **Kubernetes** offre un ambiente orchestrato che semplifica la gestione dei container in un'infrastruttura distribuita. Questo arricchisce l'architettura basata, consentendo una gestione avanzata e dinamica dell'infrastruttura distribuita, garantendo una distribuzione efficiente, scalabile e affidabile del sistema di distribuzione delle notizie.

L'applicativo simula un ecosistema di distribuzione di notizie in cui gli utenti possono iscriversi e specificare le loro preferenze di categoria. Contemporaneamente, il sistema è in grado di recuperare notizie da un servizio esterno come **NewsAPI**, filtrando le richieste in base a delle categorie specifiche che avranno una corrispondenza con le categorie selezionate dagli utenti. Questo processo consente al sistema di notificare gli utenti in modo personalizzato, garantendo un flusso di informazioni rilevante e di loro interesse. Per fare ciò si è utilizzato un sistema architetturale del tipo **Publisher-Subscriber**, facendo uso della tecnologia offerta da **Kakfa**.

L'architettura basata su microservizi non solo offre una maggiore modularità e scalabilità ma facilita anche la manutenzione e l'espansione dell'applicazione con l'implementazione di nuove funzionalità. L'approccio adottato mira a fornire un sistema efficiente e flessibile, adattabile alle esigenze dinamiche degli utenti e alle evoluzioni del contesto informativo.

2. Schema Architetturale

2.1. Introduzione

Il Sistema è composto quindi da un **Producer Service**, il quale raccoglie periodicamente articoli da un servizio esterno e li distribuisce su **Kafka** in diversi topic, mentre sarà presente un **Subscriber Service,** che quando riceverà degli aggiornamenti su uno specifico topic, invierà delle notifiche agli utenti interessati tramite l'invio di e-mail.

La gestione degli utenti è prevista mediante l'implementazione di altri tre diversi microservizi: **Customer Service**, **Category Service** e **Authentication Service.** Gli utenti, quindi, possono registrarsi all'applicativo e scegliere le categorie di loro interesse. Tutte queste informazioni sono conservate in appositi database.
Per quanto riguarda il Publisher, quest'ultimo è schedulato in modo tale da recuperare le news periodicamente, filtrate per categoria, andando così a creare/aggiornare un topic per ogni categoria.
Il Subscriber, non appena è avviato, effettua una sottoscrizione per ogni categoria e quando verrà

aggiornato un topic, contatterà il Category Service per recuperare le e-mail degli utenti che hanno selezionato quella categoria come preferenza in modo da poterli notificare con le nuove notizie.

Inoltre, è stato inserito un sistema di **QoS management**. L'implementazione prevede l'utilizzo di **cAdvisor** e **Prometheus** per il monitoraggio delle prestazioni dei container presenti nell'applicativo e la raccolta di metriche. In aggiunta, è stato integrato uno **SLA Manager** come microservizio, il quale supervisiona gli accordi **SLA**, contribuendo alla gestione ottimale delle risorse e all'assicurazione della qualità complessiva del servizio.

2.2. Diagrammi

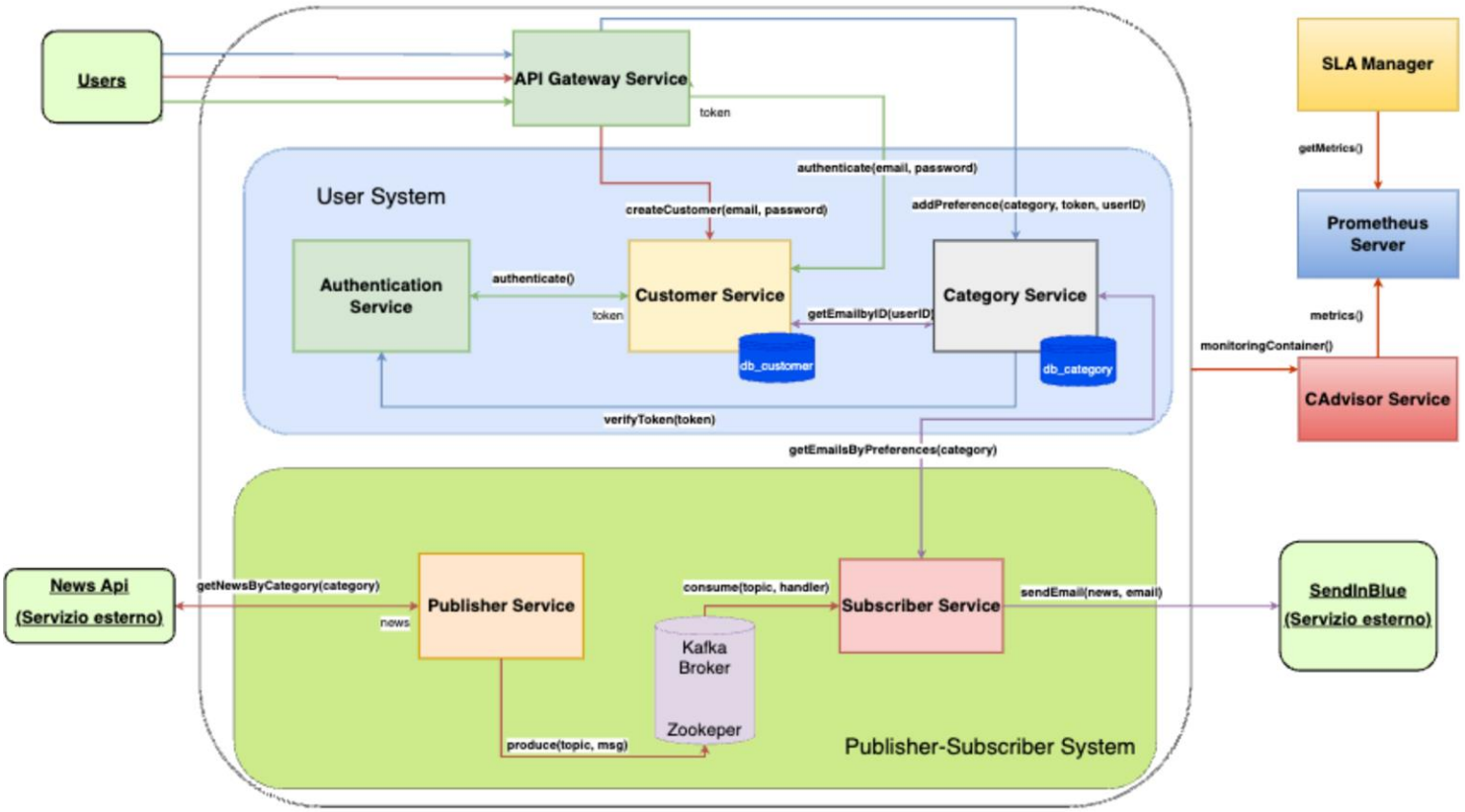


Figura 1 Diagramma con relazioni applicativo

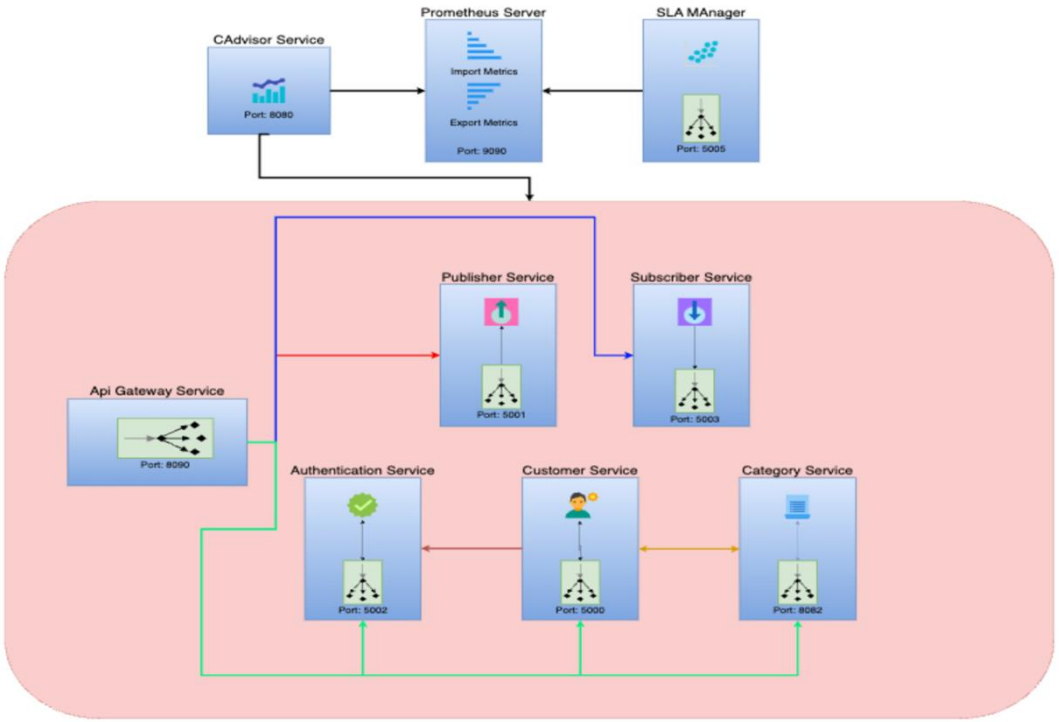


Figura 2 Diagramma Microservizi realizzati

3. Customer Service

Il microservizio **Customer** è dedicato alla gestione del customer e offre la possibilità di utilizzare le operazioni **CRUD** per gestire in modo completo i dati relativi ai customer. Infatti, consente l'inserimento di nuovi customer nel database, il recupero di informazioni specifiche sui customer, l'aggiornamento dei dettagli esistenti e l'eliminazione di record customer, offrendo così una gestione completa e flessibile delle interazioni con i dati del cliente nel contesto del database associato. Per garantire una connessione sicura ed efficiente con il database customer, il microservizio utilizza SQLAlchemy come Object-Relational Mapping (ORM), facilitando la gestione delle interazioni complesse con il database e contribuendo alla coerenza dei dati e alla sicurezza delle transazioni.

Ovviamente questo servizio si occupa di fornire l'accesso a tutte le operazioni utili per i customer, ma ai fini dell'esecuzione del progetto verrà utilizzato principalmente l'endpoint **'/customers'** per la registrazione di un nuovo utente, in modo da inserire nell'apposito DB le informazioni dell'utente e l'endpoint **'/customers/auth'** per generare un token di autenticazione. Per fare questo viene utilizzata la tecnologia JWT Token e in questo caso il Customer Service dovrà contattare l'Authentication Service (*di cui parleremo successivamente)* per generare il token, il quale verrà utilizzato dall'utente per interagire con il **Category Service**.

API	**Method**	**Path**
Retrieve all customers	GET	/customers
Create a customer	POST	/customers
Retrieve a customer by ID	GET	/customers/{customer_id}
Delete a customer	POST	/customers/delete
Authenticate a customer	POST	/customers/auth

4. Category Service

Il microservizio **Category** assume il ruolo chiave nella gestione delle categorie di notizie nell'applicativo, sovrintendendo alle interazioni con i clienti. Una volta registrati, i clienti hanno la possibilità di definire le loro preferenze, ossia le categorie di notizie di loro interesse, determinando così la ricezione di notifiche relative alla pubblicazione di nuovi articoli appartenenti alle categorie specificate.

Analogamente al Customer Service, anche il Category Service adotta l'utilizzo di **SQLAlchemy** per interagire con il database delle categorie (db_category). Tra le operazioni principali che vengono utilizzato nel funzionamento principale dell'applicativo troviamo l'aggiunta delle preferenze da parte di un cliente tramite l'endpoint **'/categories/add-preference'** (richiedendo l'utilizzo del token) e la richiesta di recuperare tutte le e-mail degli utenti che hanno selezionato una determinata categoria (endpoint **'/categories/preferences'**) che, come vedremo, è una richiesta effettuata dal **Subscriber Service** per inviare le notifiche.

In merito all'aggiunta di una preferenza, per eseguire l'operazione il Category Service invia il **token**, che deve essere necessariamente aggiunto alla chiamata all'endpoint, all'Authentication Service, il quale effettua un'operazione di verifica del token e quindi permette lo svolgimento delle operazioni, fino al salvataggio in DB della preferenza.

API	**Method**	**Path**
Retrieve all categories	GET	categories
Create a category	POST	/categories
Retrieve a categories by ID	GET	/categories/{category_id}
Retrieve a categories by name	GET	/categories?category-name=
Add customer preference to category	POST	/categories/add-preference
Retrieve customer emails for preference	GET	/categories/preferences

5. Authentication Service

Il microservizio **Authentication** assume la responsabilità della generazione e verifica dei token mediante l'impiego della tecnologia JWT Token. Esso espone esclusivamente due endpoint esterni, finalizzati all'esecuzione delle già menzionate operazioni.
Tale servizio è sollecitato dal **Customer Service** per la creazione del token, il quale verrà utilizzato dall'utente, nonché dal **Category Service** nel contesto dell'interazione dell'utente con quest'ultimo per l'inserimento delle preferenze di categoria.

API	**Method**	**Path**
Authenticate customer	POST	/authenticate
Verify token	POST	/verify_token

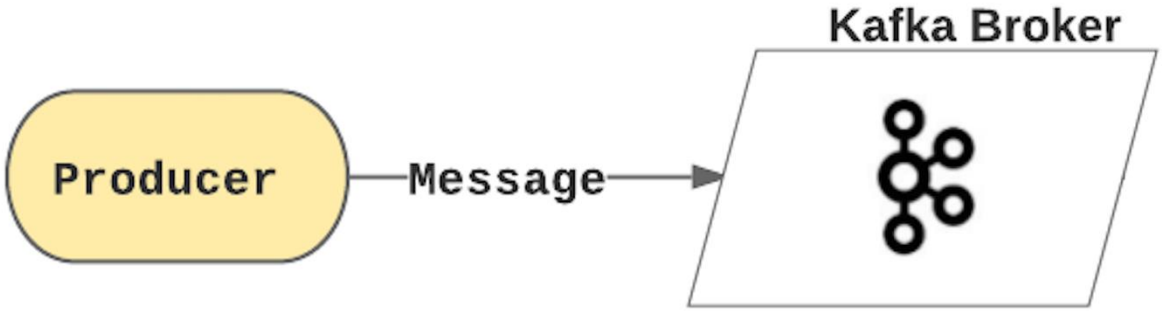
6. Publisher Service

Il microservizio **Publisher**, riveste la responsabilità di agire come publisher nel contesto della comunicazione fornita da Kafka, gestendo la pubblicazione di messaggi all'interno dei topics. Vengono creati dinamicamente dei topics specifici per ogni categoria direttamente all'avvio di Kafka, adottando una strategia in cui il Publisher Service recupera gli articoli filtrati per categoria da un servizio esterno, **News API**, e li pubblica nel topic associato alla categoria corrispondente. Questa configurazione agevola anche il processo di sottoscrizione lato Subscriber, consentendo di associare le news a categorie specifiche.

Per il recupero delle notizie, il Publisher Service interagisce con News API, aggiungendo la categoria come parametro dell'API, garantendo così il ritorno di news riferite esclusivamente a quella categoria. Successivamente, il publisher si connette al topic specifico associato a tale categoria e procede con la pubblicazione delle notizie.

Per attivare il publisher, sono state implementate due soluzioni intercambiabili, basate su una variabile di ambiente definita nei file di configurazione docker-compose.yml e deployment.yml per ambienti Docker e Kubernetes, rispettivamente. La prima implementazione consiste in uno scheduler che esegue le operazioni ciclicamente ogni X ore, attivato al momento del lancio del container. La seconda implementazione, invece, offre un endpoint esposto, particolarmente utile per scopi di testing, permettendo il triggering asincrono del publisher.

Per l'invocazione del servizio esterno, è impiegata la tecnologia del **Circuit Breaker**, integrata con un meccanismo di gestione del retry per la richiesta, limitato a un numero massimo di tentativi. Questo offre un meccanismo controllato di gestione degli errori. Garantisce la resilienza del sistema, limitando il numero di tentativi di chiamata e prevenendo il verificarsi di situazioni di degrado prolungato.



7. Subscriber Service

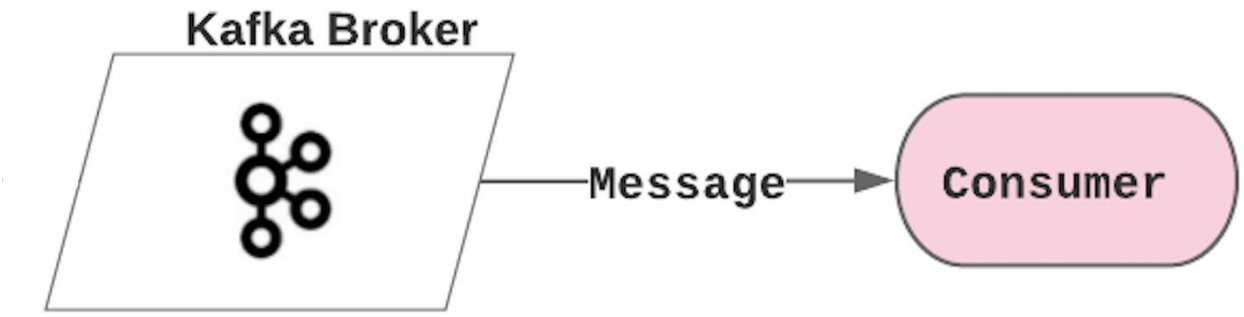
Il **Subscriber Service** è progettato per sottoscriversi ai diversi topic Kafka creati (uno per ogni categoria) e inviare notifiche agli utenti via e-mail in base alle preferenze di categoria di notizie che hanno specificato. Viene verificata la presenza dei topic Kafka necessari, come "*general_topic*", "*sports_topic*", "*science_topic*", "*technology_topic*", che, come detto in precedenza, vengono creati all'avvio di Kafka. Se non sono presenti, attende fino a quando non vengono creati, ovvero fino a quando il servizio Kafka non è completamente avviato.

Una volta che i topic sono presenti, l'applicativo si sottoscrive a ciascun topic corrispondente a una categoria di notizie. Ad esempio, se ci sono categorie come sport, scienza, tecnologia, si sottoscriverà ai topic "*sports_topic*", "*science_topic*", "*technology_topic*" rispettivamente.

Il meccanismo della sottoscrizione è fondamentale per il funzionamento effettivo dell'applicativo. Infatti, questo permette di poter comprendere quando verranno pubblicate nuove notizie in merito ad una data categoria. Così facendo si saprà sempre qual è la categoria aggiornata e in questo modo il Subscriber Service potrà contattare il **Category Service** per ottenere le e-mail di tutti gli utenti interessati a quella categoria (ossia che hanno scelto quella categoria come preferenza).

Una volta ricevute le e-mail, il Subscriber Service, facendo uso di un servizio esterno quale **SendInBlue**, potrà generare il corpo di una mail, in cui saranno presenti i nuovi articoli con le informazioni riguardo al titolo e al link della notizia specifica e provvederà a inviare le mail di notifiche a tutti gli utenti ritornati dal **Category Service**.

Anche per l'interazione tra il Subscriber Service e il Category Service, così come per l'interazione con il servizio esterno SendInBlue, viene utilizzato il **Circuit Breaker** integrato con il meccanismo di gestione del retry.



8. API Gateway Service

L'API Gateway, implementato attraverso il framework **Flask** in Python, assume un ruolo centrale nella gestione delle richieste all'interno dell'architettura dell'applicativo. Esso agisce come intermediario tra i clienti e i vari microservizi sottostanti, facilitando la comunicazione e l'inoltro delle richieste in modo efficiente. Il ruolo chiave del gateway è evidente nella sua capacità di instradare le richieste in arrivo ai servizi appropriati. Il modello di **routing** flessibile consente di indirizzare le richieste in base al tipo di servizio richiesto, garantendo una distribuzione efficace del traffico. Le URL dei servizi, come il Customer Service e il Category Service, anche se è presente la configurazione anche per l'endpoint di test del Publisher Service, sono configurate dinamicamente per adattarsi alle esigenze specifiche dell'applicativo.

La gestione delle richieste viene effettuata attraverso due principali route: una per i servizi con path specifico *(/api/<string:service>/<string:subpath>)* e l'altra per i servizi senza subpath (*/api/<string:service>).* Questa struttura consente una gestione flessibile delle richieste, adattandosi alle diverse modalità di interazione con i vari servizi. La modularità del codice facilita l'aggiunta di nuovi servizi senza richiedere modifiche estensive al gateway stesso. La logica di inoltro delle richieste, basata sul metodo HTTP e sulla presenza di un percorso specifico, contribuisce alla chiarezza e alla coerenza delle operazioni di routing.

Quindi l'API Gateway Service svolge un ruolo cruciale nell'orchestrazione delle comunicazioni tra i clienti e i microservizi sottostanti. La sua implementazione, insieme a pratiche di routing efficienti, favorisce una distribuzione bilanciata del traffico e contribuisce al mantenimento di un'architettura modulare e scalabile per l'applicativo.

9. SLAManager Service

Lo **SLAManager Service** offre una serie di funzionalità per il monitoraggio e il mantenimento degli SLA (Service Level Agreement) attraverso l'uso di metriche recuperate tramite l'utilizzo di **Prometheus**. Grazie all'utilizzo di **Flask** e altre librerie Python, l'applicazione consente di interagire con i dati provenienti da **Prometheus** e gestire le metriche e gli **SLA** in modo efficiente.

Una delle funzionalità principali dell'applicativo è la capacità di recuperare metriche da **Prometheus** utilizzando la libreria **prometheus_api_client**. Questo permette di ottenere dati accurati e aggiornati sulle prestazioni dei servizi monitorati. Inoltre, l'applicativo fornisce API per la gestione delle metriche e degli **SLA** associati ad esse. Utilizzando le diverse funzioni implementate, è possibile eseguire varie operazioni come ottenere tutte le metriche disponibili, recuperare una metrica specifica per ID o nome, aggiungere e rimuovere SLA associati alle metriche, e ottenere informazioni sulle violazioni degli **SLA**.

L'applicativo offre anche la possibilità di eseguire previsioni sulle violazioni degli **SLA** utilizzando il modello **ARIMA** implementato nella libreria '***pmdarima***'. Il modello **ARIMA** è una tecnica di analisi delle serie temporali che combina l'effetto autoregressivo, l'integrazione e la media mobile per modellare e prevedere andamenti nei dati nel tempo. L'utilizzo del modello **ARIMA** nell'applicativo consente agli utenti di anticipare potenziali violazioni degli **SLA** in base alle tendenze storiche dei dati di monitoraggio.

Analizzando le serie temporali, è possibile identificare modelli che indicano possibili deviazioni dai SLA, consentendo una gestione proattiva delle prestazioni del sistema. In questo modo, il modello **ARIMA** si presenta come uno strumento prezioso per migliorare la capacità predittiva e la tempestività delle risposte alle variazioni delle prestazioni.

API	**Method**	**Path**
Retrieve all metrics	GET	/metrics/all
Retrieve metric by ID	GET	/metrics/int:metric_id
Retrieve metric by name	GET	/metrics?metric-name
Create a metric	POST	/metrics
Add/Update SLA	POST	/add_sla
Delete sla	POST	/delete_sla
Retrieve all SLA	GET	/sla_metrics/all
Retrieve violations	GET	/violations?time-range
Check status SLA	GET	/check_sla
Forecast violations	GET	/forecast_violations?time-range

10. Distribuzione

La distribuzione dell'applicazione su **Docker** e **Kubernetes** rappresenta una scelta strategica che offre numerosi vantaggi in termini di gestione, scalabilità e affidabilità. La nostra applicazione è stata suddivisa in microservizi, ciascuno dei quali è contenuto in un container **Docker** autonomo. Questi container possono essere facilmente creati, distribuiti e gestiti in qualsiasi ambiente che supporti **Docker**, garantendo un ambiente consistente e isolato per ciascun servizio.

L'orchestrazione di questi container è stata affidata a **Kubernetes**, che fornisce una gestione centralizzata dei servizi, consentendo la scalabilità dinamica in risposta al carico di lavoro. In particolare, è stato definito il servizio dell'api gateway come load balancer in modo da esporre il servizio su un indirizzo IP esterno e di gestire il bilanciamento del carico delle richieste.

Inoltre, le funzionalità di monitoraggio e gestione delle risorse fornite da **Kubernetes**, insieme a strumenti come **cAdvisor** e **Prometheus**, contribuiscono a garantire prestazioni ottimali e una diagnosi tempestiva di eventuali problemi.

I deployment riflettono la struttura a microservizi dell'applicazione, consentendo una gestione e un'evoluzione flessibili. Ogni componente è isolato, facilitando lo sviluppo, la distribuzione e la manutenzione dell'intero sistema.

11. Prometheus e cAdvisor

L'integrazione di **cAdvisor** e **Prometheus** all'interno dell'architettura dell'applicativo svolge un ruolo chiave nel monitoraggio delle risorse e delle prestazioni dei container. Questa sezione esplorerà le funzioni di **cAdvisor** e il modo in cui **Prometheus** cattura e analizza metriche significative per garantire un ambiente operativo stabile. Questa, infatti, offre un livello avanzato di visibilità e controllo sulle prestazioni dei container. Monitorare metriche specifiche permette il mantenimento di un ambiente operativo stabile e di prendere decisioni informate per ottimizzare le prestazioni complessive del sistema.

11.1. cAdvisor: Monitoraggio dei container

cAdvisor è un tool di monitoraggio che fornisce informazioni dettagliate sulle risorse utilizzate dai container in esecuzione. Esso si concentra su diverse metriche chiave, ma nel caso specifico dell'applicativo **News Notifier** si è scelto di monitorare le seguenti:

- **Utilizzo della Memoria del Container** (*container_memory_usage_bytes*): Questa metrica misura la quantità di memoria effettivamente utilizzata da un container. Il monitoraggio di questa metrica è cruciale per evitare problemi legati alla gestione della memoria, prevenire eventuali limiti di risorse e ottimizzare l'allocazione di memoria nei container.

- **Tempo di Avvio del Container** (*container_start_time_seconds*): Indica il tempo di avvio di un container. Questa informazione è utile per tracciare il ciclo di vita dei container, valutare le prestazioni in relazione al tempo e identificare eventuali ritardi nell'avvio dei container.

- **Errori di Trasmissione di Rete del Container** (*container_network_transmit_errors_total*): Questa metrica rappresenta il numero totale di errori di trasmissione di rete del container. Monitorare questa metrica è essenziale per identificare potenziali problemi nella comunicazione di rete dei container, contribuendo a garantire una connettività affidabile.

11.2. Prometheus: Raccolta e Analisi delle Metriche

Prometheus, un sistema di monitoraggio gioca un ruolo centrale nella cattura e nell'analisi delle metriche provenienti da **cAdvisor**. Tra le caratteristiche chiave di Prometheus vi sono:

- **Configurazione Flessibile**: **Prometheus** consente una configurazione flessibile delle fonti di dati, consentendo agli operatori di definire job specifici per il recupero di metriche da diverse fonti, come nel seguente caso con **cAdvisor**.
- **Linguaggio di Query PromQL**: **Prometheus** utilizza il linguaggio di query **PromQL** per eseguire analisi avanzate sulle metriche raccolte. Questo consente allo SLA Manager di definire query custom per ottenere le informazioni specifiche.

Per l'applicativo in questione si è scelto di configurare **Prometheus** con un intervallo di scraping di 15 secondi. **Prometheus** recupera regolarmente le informazioni esposte da **cAdvisor** attraverso il job definito nella sua configurazione.

Le metriche raccolte, ossia '*container_memory_usage_bytes'*, '*container_start_time_seconds'* e '*container_network_transmit_errors_total'*, forniscono una panoramica dettagliata delle risorse e delle attività dei container.