Università di Catania

DIEEI
Dipartimento di Ingegneria Elettrica Elettronica e Informatica

Corso di Distribuited Systems and Big Data

Lorenzo Basile 1000055691, Antonio Santo Buzzone 1000055698

Relazione Progetto

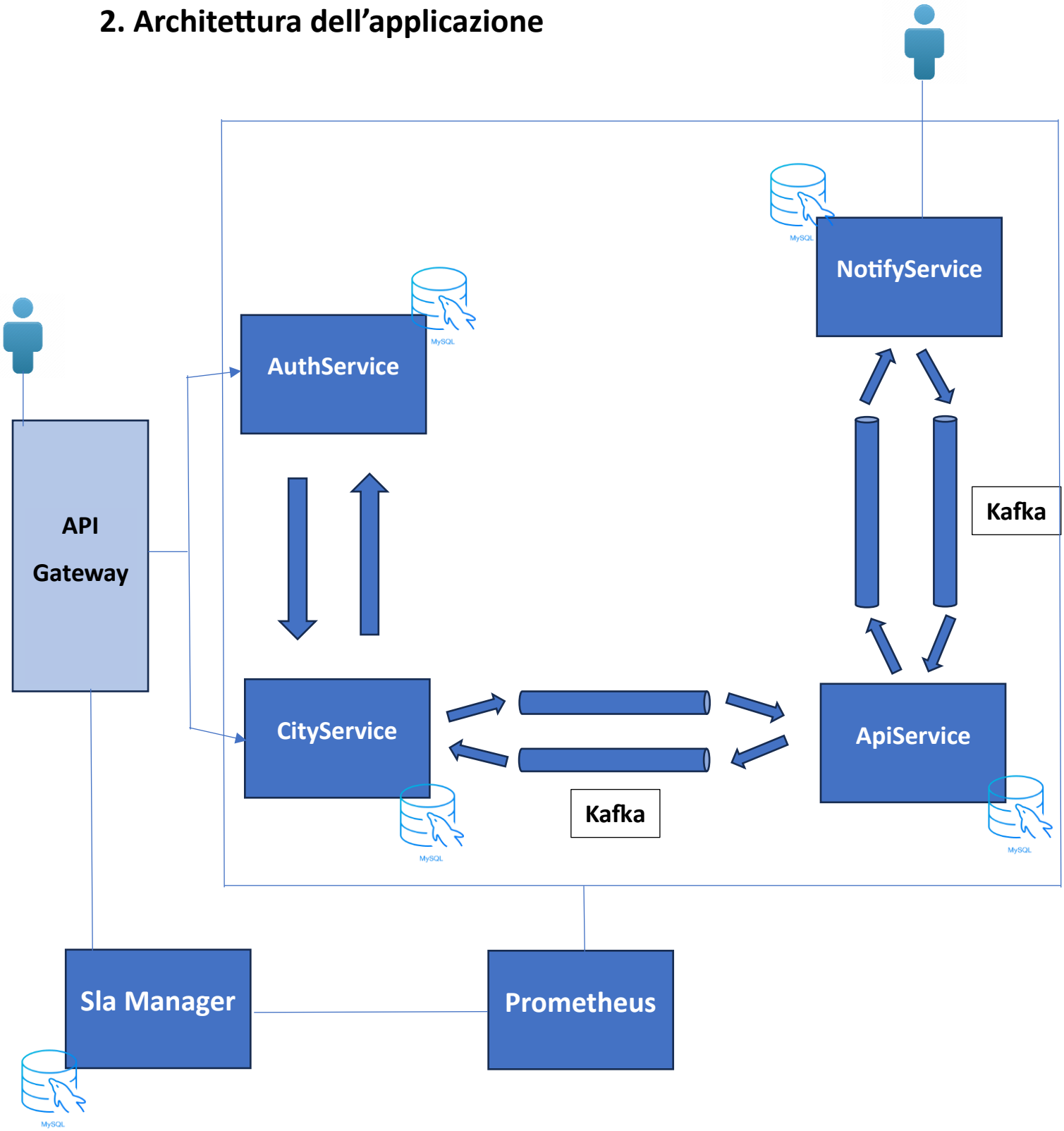
1. Introduzione

Lo scopo dell'elaborato è lo sviluppo di una piattaforma che permette all'utente di essere avvisato tramite notifica telegram, ad intervalli regolari, su particolari informazioni meteorologiche. L'utente dopo essersi autenticato ha a disposizione una funzione d'iscrizione a degli eventi meteorologici di una città di sua scelta, fornendo alcuni vincoli, tra cui temperatura minima, massima e presenza pioggia o neve.

Una volta trovata la città verranno confrontati i vincoli inseriti dall'utente con le informazioni in tempo reale della città desiderata e successivamente verrà inviata una notifica all'utente se i vincoli vengono violati. Abbiamo scelto di utilizzare come linguaggio python con l'uso del framework flask. Per la persistenza dei dati abbiamo preferito l'uso del database MySQL, è stata considerata la scelta di associare un database ad ogni servizio dove risulta necessario avere persistenza dei dati. Abbiamo implementato una comunicazione asincrona tra alcuni servizi tramite i servizi Apache Kafka.

Inoltre, è stato usato il servizio Prometheus per la raccolta delle metriche, quest'ultime tramite il servizio SLA possono essere valutate per identificare violazioni attuali e passate. Infine, sono stati creati dei client che permetto d'interfacciarsi coi servizi AuthService, CityService ed SLA_Manager.

2. Architettura dell’applicazione



L’architettura è composta dai seguenti servizi:

- AuthService
- CityService
- ApiService
- NotifyService

- Prometheus
- Kafka
- Zookeeper
- Sla_manager
- Gateway

AuthService è il microservizio che si occupa dell'autenticazione dell'utente. Le sue funzioni sono: esporre API per l'inserimento di dati d'autenticazione, interfacciarsi col database MySQL e codificare un token in fase d'autenticazione contenente i dati dell'utente. Se l'utente non risulta essere registrato allora è possibile farlo inserendo alcuni dati come (username, telegram chat id, password), le informazioni dell'utente verranno inserite all'interno della tabella *user* e all'utente verrà assegnato un id univoco. Altrimenti se risulta essere registrato è possibile effettuare il login tramite username e password. Nella fase di registrazione il telegram chat id risulta essenziale in quanto all'utente verranno inviate delle notifiche tramite app telegram. La comunicazione tra authService e cityService avviene tramite lo scambio del token necessario per l'associazione dei dati della città con l'utente stesso.

Il secondo microservizio è **CityService**, dopo avere ricevuto i dati dall'utente tra cui la città che si desidera cercare e le varie constraint, il suo compito è quello d'inserire questi dati e il corrispettivo id dell'utente all'interno di una tabella chiamata info_meteo. CityService fa uso di Kafka per inviare in modo asincrono questi dati ad ApiService. La comunicazione tra questi due microservizi avviene appunto tramite l'associazione del messaggio da parte di cityService al topic kafka: "WeatherInformations".

ApiService è il microservizio che, dopo esser stato avviato, esegue in parallelo tre thread. Il primo di ascolto dei messaggi ricevuti da cityService tramite kafka iscrivendosi al topic "WeatherInformations" e l'inserimento di tali dati nel db, il secondo si occupa d'interrogare costantemente il database per recuperare i dati da info_meteo, in particolare la città di cui si vogliono ricevere informazioni. Tramite un servizio web che mette a disposizione delle API che consentono di ottenere molteplici informazioni metereologiche, effettua un confronto tra i vincoli inseriti dall'utente le info recuperate dal servizio web e si occupa di occupa di pubblicare nel topic Kafka "WeatherNotification" queste informazioni. Questi eventi saranno consumati dal microservizio NotifyService.

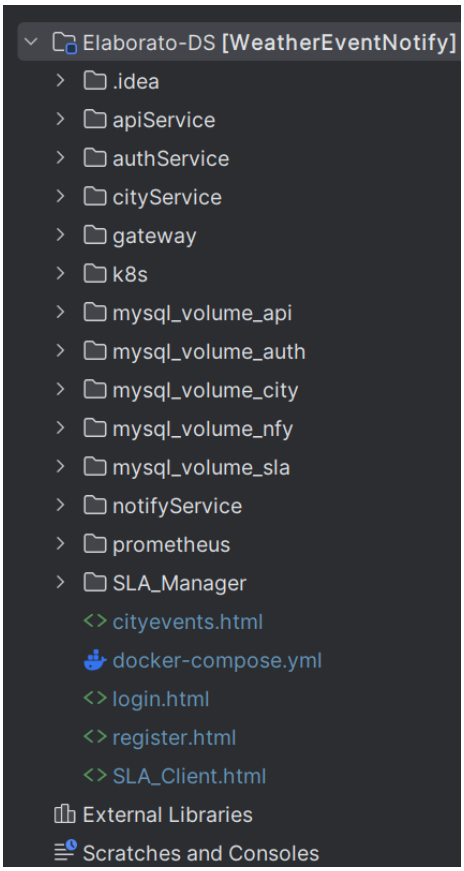
NotifyService è l'ultimo microservizio progettato per questa piattaforma, che ha come obiettivo primario quello d'inviare notifiche all'utente. Attraverso la ricezione di un evento tramite kafka estrae le informazioni, tra cui id_utente, telegram chat id, città e vari vincoli e si occupa di inviare tramite il telegram chat id una notifica all'utente.

L'elaborato contiene anche il servizio **Prometheus**: si tratta di un tool di monitoraggio e alerting che archivia metriche in un database proprietario di timeseries. In particolare, la scelta è stata orientata verso l'implementazione di cAdvisor, nodeExporter e customExporter contenente metriche customizzate come per esempio: numero di utenti registrati, tempo di risposta di un api, numero di accessi al database e infine percentuale di cpu, memoria e spazio su disco utilizzati da ogni servizio.

Un altro servizio implementato è **SLA_Manager**, che permette di monitorare un set di metriche a scelta dell'utente collezionate da Prometheus. Questo servizio permette i seguenti utilizzi: aggiunta e rimozione di una metrica al db, valutazione del valore attuale della metrica inserita tramite query al server Prometheus, conteggio delle violazioni delle metriche da monitorare avvenute nelle ultime 1, 3 e 6 ore.

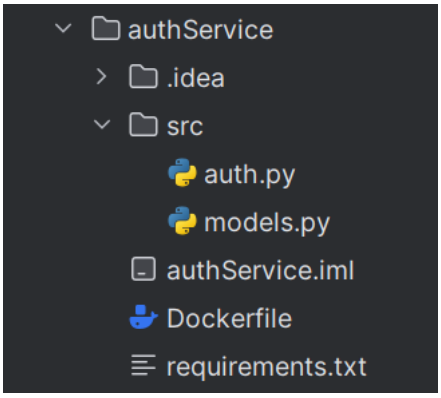
È presente inoltre un **API Gateway**, che prende tutte le chiamate API dai clienti e le instrada al giusto microservice usando il routing della richiesta. Così facendo, lato client si utilizza solamente un unico punto di ingresso piuttosto che effettuare richieste API a diversi url in base al microservizio che si intende utilizzare.

3. Descrizione dei servizi del sistema distribuito



Come IDE per lo sviluppo del codice e per l'organizzazione dei package è stato utilizzato intellij, in particolare Python con l'uso di flask. All'interno dell'elaborato sono presenti i servizi, i volumi dichiarati per i vari database utilizzati, i file docker-compose e kind-compose rispettivamente per l'esecuzione su docker e k8s.
Infine, abbiamo implementato dei client per semplificare l'accesso alle api dell'applicazione, in particolare sono stati creati: login.html e register.html per interfacciarsi con il servizo Auth, cityevents.html per comunicare con cityService e infine SLA_Client per inviare richieste al servizio SLA_Manager.

AuthService




```python
db = SQLAlchemy()

7 usages  abuzz
class User(db.Model):
    __tablename__ = 'user'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.Text, nullable=False)
    password = db.Column(db.Text, nullable=False)
    telegram_chat_id = db.Column(db.Text, nullable=False)
```

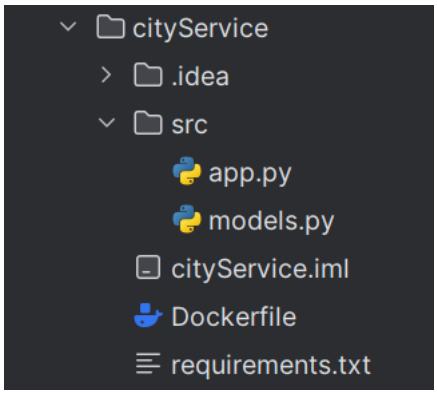
In questo servizio inseriamo dei modelli tramite il file model.py associati a delle tabelle nel database utilizzando oggetti Python invece di scrivere query SQL dirette. Nel file auth.py inizialmente si hanno delle istruzioni di configurazione: recupero variabili d'ambiente da docker compose per la configurazione del db e della chiave segreta per la codifica del token, configurazione app flask e inizializzazione db. L'applicazione Flask è configurata per utilizzare SQLAlchemy per la gestione del database, con le informazioni di connessione specificate tramite le variabili d'ambiente. Le metriche di Prometheus vengono inizializzate e associate all'applicazione Flask. I metodi principali di questo servizio sono :

- **user_register():** gestisce una richiesta POST inviata all'endpoint "/register" che permette la registrazione di nuovi utenti. Verifica che i campi obbligatori (username, password, telegramChatId) siano compilati e controlla se l'utente esiste già nel database. Se l'utente è nuovo, viene creato, aggiunto al database e generato un token. Vengono anche registrate metriche, come il numero totale di connessioni al database e il numero di utenti registrati. Il risultato della registrazione viene restituito sotto forma di un JSON che include uno stato di successo o un indicatore di fallimento.

- **user_login():** gestisce il processo di login degli utenti attraverso una richiesta POST all'endpoint "/login". La funzione verifica se la richiesta è di tipo POST e recupera le credenziali utente (username e password) dalla richiesta. Successivamente, esegue una query nel database per verificare la presenza di un utente con le credenziali fornite. Se l'utente esiste, viene generato un nuovo token JWT valido. In caso di successo, viene restituito un JSON con uno stato di successo (state=0) e il token generato. Se le credenziali sono invalide, viene restituito uno stato di errore (state=1) con un messaggio appropriato. La funzione tiene anche traccia di metriche e gestisce un meccanismo per rimuovere gli utenti dalla lista di token non validi.

- **measure_metrics():** Questo metodo implementa una routine di misurazione periodica di metriche di sistema, come percentuale d'utilizzo della memoria, la percentuale di utilizzo della CPU e lo spazio disco utilizzato. Utilizza le librerie psutil e shutil per ottenere queste informazioni. La libreria schedule viene impiegata per pianificare l'esecuzione della funzione di misurazione a intervalli regolari, ad esempio ogni minuto. Inoltre, il processo di misurazione viene eseguito in un thread separato (scheduler_thread), assicurando così una raccolta continua di dati di sistema in background.

CityService




```python
db = SQLAlchemy()
4 usages  abuzz
class User(db.Model):
    __tablename__ = 'user'

    id = db.Column(db.Integer, primary_key=True)
    telegram_chat_id = db.Column(db.Text, nullable=False)

2 usages  abuzz *
class Info_meteo(db.Model):
    __tablename__ = 'info_meteo'

    info_id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    city = db.Column(db.Text, nullable=False)
    t_max = db.Column(db.Integer, nullable=True)
    t_min = db.Column(db.Integer, nullable=True)
    rain = db.Column(db.Boolean, nullable=True)
    snow = db.Column(db.Boolean, nullable=True)

    user = relationship('User')
```

In cityService vengono inserite delle classi tramite “models.py”. Queste classi definiscono la struttura delle tabelle nel database. La classe Info_meteo è collegata alla classe User tramite una relazione di chiave esterna, consentendo l'associazione di informazioni meteo specifiche a un utente. In “app.py” viene configurata un'applicazione Flask e inizializzate diverse variabili d'ambiente, inclusa la chiave segreta (SECRET_KEY) e le credenziali di accesso al database MySQL. L'applicazione Flask è configurata per utilizzare SQLAlchemy per la gestione del database, con le informazioni di connessione specificate tramite le variabili d'ambiente. Le metriche di Prometheus vengono inizializzate e associate all'applicazione Flask per monitorare le prestazioni del servizio.

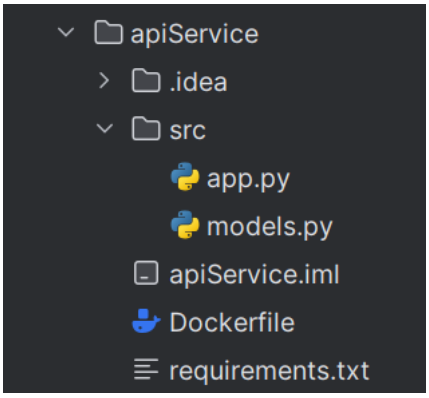
I metodi principali di questo servizio sono :

- **Home():** Questo metodo gestisce le richieste POST all'endpoint '/cityevents/<token>' in un'applicazione Flask. Verifica se la richiesta è di tipo POST e analizza i dati ricevuti, tratta i dati ricevuti (come il nome della città, la presenza di pioggia/neve, temperatura massima e minima) e li adatta per l'inserimento nel database, gestendo i casi di dati mancanti. Decodifica il token ricevuto nel parametro token utilizzando la chiave segreta (SECRET_KEY), questo passaggio avviene per poter utilizzare l’id dell’utente che attualmente ha fatto il login. Aggiunge un nuovo record di informazioni meteo al database utilizzando le informazioni analizzate dalla richiesta, crea un messaggio contenente informazioni sull'utente e le informazioni meteo, che successivamente verrà inviato tramite send_kafka(). Infine, restituisce una risposta JSON indicando uno stato di successo (state: 0) per confermare l'avvenuto inserimento delle informazioni meteo nel sistema. In questo metodo viene calcolata la metrica (response-time), tramite il tempo di inizio della richiesta (request_start_time) utilizzando il modulo time, il tempo di fine della richiesta (request_end_time), la differenza viene quindi registrata nella metrica (api_response_time).

- **Send_kafka** si occupa di inviare un messaggio JSON al topic 'weatherInformations' su un server Kafka, utilizzando un produttore Kafka configurato.

- **Measure_metrics():** funzionamento analogo al servizio precedente.

ApiService



In questo servizio vengono inserite delle classi tramite “models.py”. Queste classi definiscono la struttura delle tabelle nel database. La classe Info_meteo è collegata alla classe User tramite una relazione di chiave esterna, consentendo l'associazione di informazioni meteo specifiche a un utente. Struttura analoga al servizio “cityService”. In “app.py” viene configurata un'applicazione Flask e inizializzate diverse variabili d'ambiente, inclusa la chiave segreta (SECRET_KEY) e le credenziali di accesso al database MySQL.

I metodi principali di questo servizio sono:

- **Consuma_da_kafka():** progettata per consumare messaggi da un topic Kafka denominato “weatherInformations”. Il consumatore viene configurato per leggere continuamente i messaggi dal topic specificato. Per ogni messaggio, verifica se l'utente associato esiste nel database; se non esiste, crea un nuovo utente. Successivamente, aggiunge le informazioni meteo al database utilizzando i dati estratti dal messaggio.
- **Check_weather():** esegue ciclicamente la verifica delle condizioni meteorologiche per gli utenti registrati nel sistema. Per ciascun utente, recupera le città associate agli eventi meteorologici richiesti e effettua richieste API al servizio OpenWeatherMap per ottenere i dati meteorologici correnti. La funzione confronta quindi i dati ottenuti con le preferenze dell'utente per pioggia, neve, temperatura massima e minima. Se le condizioni specificate dagli utenti corrispondono alle condizioni attuali, viene creato un messaggio contenente tali informazioni, e il messaggio viene inviato a Kafka tramite la funzione send_kafka. Questo processo consente agli utenti di ricevere notifiche in tempo reale sulle condizioni meteorologiche che corrispondono alle loro preferenze specifiche.

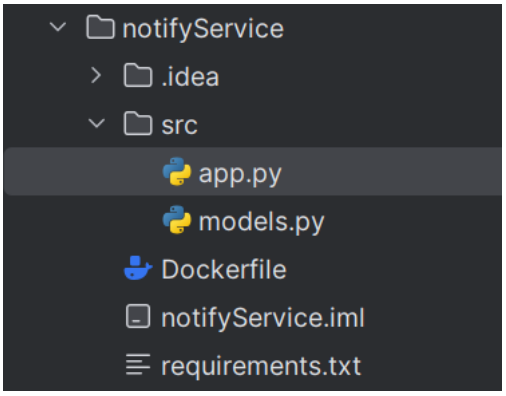
- **Send_kafka():** si occupa di pubblicare messaggi contenenti notifiche meteorologiche sul topic “weatherNotification” di un server Kafka, consentendo a potenziali consumatori di ricevere informazioni in tempo reale sugli eventi meteorologici corrispondenti alle preferenze degli utenti.

- **Measure_metrics**(): metodo utilizzato per la misurazione di metriche di sistema, come percentuale d’utilizzo della memoria, la percentuale di utilizzo della CPU e lo spazio disco utilizzato. Utilizza le librerie psutil e shutil per ottenere queste informazioni.

- Infine, la funzione **loop_execution()** avvia tre thread paralleli per eseguire le seguenti attività in modo asincrono: avvia un thread (consuma_da_kafka_thread) per eseguire la funzione consuma_da_kafka, che si occupa di consumare messaggi da un topic Kafka specifico ("weatherInformations"). Avvia un thread (measure_metrics_thread) per eseguire la funzione measure_metrics, che misura e aggiorna metriche relative alle risorse di sistema e infine il thread (check_weather_thread) per eseguire la funzione check_weather,

che ciclicamente controlla le condizioni meteorologiche per gli utenti registrati e invia notifiche tramite Kafka in caso di corrispondenza.

NotifyService



Il microservizio notify inizialmente recupera le variabili d'ambiente per impostare la chiave segreta, l'utente del database MySQL, la password, il nome del database e il nome del servizio. Utilizza Flask per configurare l'applicazione web, definendo la chiave segreta e l'URL del database MySQL. Viene fornito un token di accesso per l'API di Telegram. Infine, sono definiti metriche Prometheus per monitorare l'utilizzo della memoria, della CPU e lo spazio su disco utilizzato dal servizio.

I metodi principali di questo servizio sono:

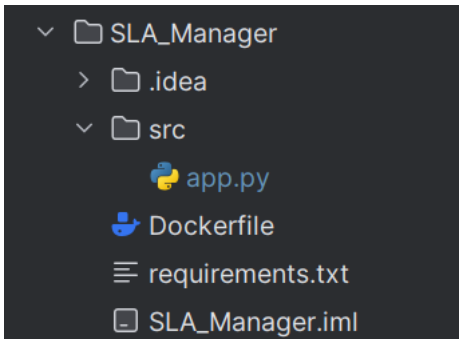
- **Consuma_da_kafka():** La funzione consuma messaggi da un topic kafka denominato 'weatherNotification' su un server Kafka. I messaggi sono decodificati da JSON e inseriti in un dizionario. Successivamente, vengono eseguite operazioni di inserimento nel database SQLAlchemy, verificando la presenza di utenti e informazioni meteorologiche già esistenti. La funzione è progettata per essere eseguita in modo continuo all'interno di un ciclo while True.

- **Send_t_message():** la funzione recupera tutte le tuple di utenti e informazioni meteorologiche correlate attraverso una query. Successivamente, per ogni coppia di utente e informazione meteorologica e crea un messaggio da inviare tramite Telegram. I messaggi contengono informazioni come la città, la presenza di pioggia o neve e le temperature minime e massime. Infine, la funzione utilizza la libreria telepot per inviare i messaggi tramite l'API di Telegram ai rispettivi utenti identificati tramite telegram_chat_id.

- **Measure_metrics():** metodo utilizzato per la misurazione di metriche di sistema, come percentuale d'utilizzo della memoria, la percentuale di utilizzo della CPU e lo spazio disco utilizzato. Utilizza le librerie psutil e shutil per ottenere queste informazioni.

- Infine, la restante parte contiene tre funzioni principali eseguite in loop: **consuma_loop():** esegue continuamente la funzione consuma_da_kafka ogni 30 secondi. **Measure_loop():** esegue continuamente la funzione measure_metrics ogni 15 secondi. **Send_loop():** esegue continuamente la funzione send_t_message ogni 900 secondi (15 minuti). Questa funzione è responsabile per l'invio di messaggi meteorologici tramite Telegram agli utenti interessati.

SLA_Manager




```python
# Definizione del modello del SLA
14 usages  Lorenzo +1
class SLA_table(db.Model):
    __tablename__ = 'sla'

    id = db.Column(db.Integer, primary_key=True)
    metric_name = db.Column(db.String(50), nullable=False)
    desired_value = db.Column(db.Float, nullable=False)
    job_name = db.Column(db.String(50), nullable=False)

# Definizione del modello delle violazioni
11 usages  Lorenzo +1 *
class Violation(db.Model):
    __tablename__ = 'violations'
    id = db.Column(db.Integer, primary_key=True)
    sla_id = db.Column(db.Integer, db.ForeignKey('sla.id'), nullable=False)
    value = db.Column(db.Float, nullable=False)
    timestamp = db.Column(db.DateTime, nullable=False)

    sla = relationship('SLA_table')
```

Questo servizio configura un'applicazione Flask che interagisce con un database MySQL, gestendo informazioni riguardanti Service Level Agreements (SLA) e le relative violazioni. Le variabili d'ambiente vengono utilizzate per recuperare credenziali come la chiave segreta e i dettagli del database. Due modelli di dati sono definiti: SLA_table rappresenta le specifiche degli SLA con attributi come nome della metrica, valore desiderato e nome del job, mentre Violation rappresenta le violazioni degli SLA, includendo dettagli come il valore della violazione e un timestamp. . La classe Violations è collegata alla classe SLA_table tramite una relazione di chiave esterna, consentendo l'associazione di informazioni di violazioni specifiche a una metrica.

I metodi principali sono:

- **Add_metric():** la route '/add_metric' gestisce richieste POST per aggiungere nuove metriche al database. Prima di procedere, verifica se esiste già una metrica con la stessa combinazione di nome della metrica e nome del job. Se sì, restituisce un messaggio indicando che la metrica è già presente. Altrimenti, crea un nuovo record SLA_table con i dati forniti nella richiesta POST, includendo nome della metrica, nome del job e valore desiderato. Questo nuovo record viene aggiunto al database mediante SQLAlchemy. La risposta JSON restituisce un messaggio '0' se l'operazione ha successo o '1' se la metrica era già presente.

- **Remove_metric():** la route Flask '/remove_metric' gestisce richieste POST per rimuovere metriche dal database. Inizialmente, recupera il record SLA_table corrispondente alla combinazione specificata di nome della metrica e nome del job dalla richiesta POST. Successivamente, ottiene tutte le violazioni associate a questa metrica attraverso una query sulla tabella delle violazioni (Violation). Se la metrica è presente, procede alla rimozione di tutte le violazioni associate e successivamente elimina la metrica stessa dal database. La risposta JSON restituisce un messaggio '0' se l'operazione ha successo (metrica e violazioni rimosse), o '1' se la metrica specificata non esiste nel database.

- **Prometheus_request():** gestisce richieste al server Prometheus per ottenere dati relativi a una specifica metrica. Accetta il nome della metrica come parametro, configura l'URL del server Prometheus (http://prometheus:9090/api/v1/query) e i parametri per la richiesta,

includendo la metrica specificata; quindi, esegue una richiesta POST al server Prometheus. Se la risposta ha uno status code 200, interpreta i dati JSON ricevuti e li restituisce. La funzione fornisce un mezzo per interrogare dinamicamente il server Prometheus e ottenere informazioni in tempo reale sulle metriche di monitoraggio.

- **Monitor_system_metrics:** la funzione implementa un sistema di monitoraggio continuo basato su Service Level Agreements (SLA) attraverso l'interazione con un server Prometheus. Inizialmente, recupera gli SLA dal database. Successivamente, per ogni SLA, richiede dati in tempo reale al server Prometheus. Analizza i risultati della richiesta per identificare eventuali violazioni degli SLA. Verifica se i risultati corrispondono al job specificato nell'SLA e confronta il valore attuale con il valore desiderato. Quando viene rilevata una violazione, registra i dettagli, inclusi l'id dello SLA, il valore rilevato e il timestamp, nella tabella Violation del database. L'esecuzione periodica della funzione è pianificata con la libreria schedule, attualmente ogni minuto, e avviene in background attraverso un thread separato gestito dalla funzione run_scheduler.

- **Get_sla_status():** La route Flask '/sla_current_state' gestisce richieste POST per ottenere lo stato corrente di un SLA. La funzione inizialmente, recupera l'SLA corrispondente dalla richiesta. Se l'SLA non è presente, restituisce uno stato indicante l'assenza. Successivamente, esegue una richiesta al server Prometheus per ottenere dati in tempo reale sulla metrica dell'SLA. Analizza i risultati per determinare il valore attuale della metrica. Verifica se il valore attuale supera il valore desiderato dell'SLA, identificando eventuali violazioni. Infine, restituisce una risposta JSON contenente lo stato dell'operazione: il valore attuale, il valore desiderato e l'indicazione di violazione.

- **Get_sla_past_violations():** La route Flask '/sla_past_violations/' gestisce richieste POST per recuperare le violazioni passate di un SLA. La funzione inizialmente, recupera l'SLA corrispondente dalla richiesta e verifica la sua esistenza. Se l'SLA non è presente, restituisce uno stato indicante l'assenza. Successivamente, calcola il numero di violazioni dell'SLA nelle ultime 1, 3 e 6 ore. Utilizza la tabella delle violazioni e filtra i dati in base ai timestamp delle violazioni rispetto all'orario attuale. Restituisce una risposta JSON contenente lo stato dell'operazione, il valore desiderato dell'SLA e il numero di violazioni registrate per ciascun intervallo temporale specificato. La funzione fornisce quindi informazioni dettagliate sullo storico delle violazioni.

- **get_sla_probability_violations():** il codice estrae i parametri dalla richiesta POST, effettua una richiesta a un server Prometheus per ottenere i dati delle metriche, esegue una previsione futura utilizzando un modello ARIMA sulle metriche temporali ottenute, calcola gli intervalli di confidenza e infine determina la probabilità di violazione rispetto al valore desiderato. La probabilità calcolata viene restituita come risposta JSON.

Prometheus

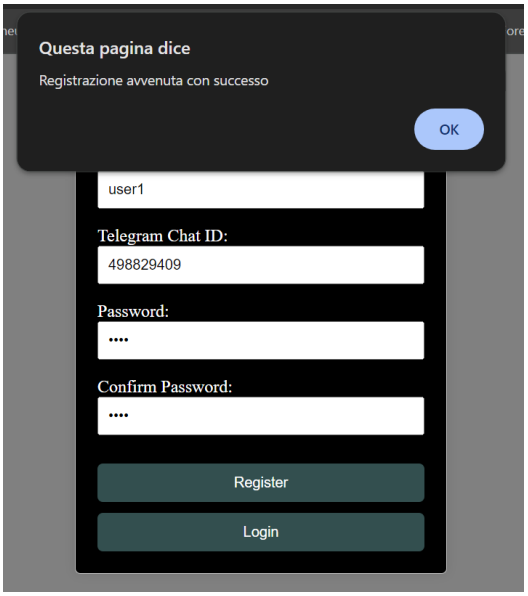
Il file .YML descrive la configurazione di un server Prometheus per il monitoraggio di diversi job tramite la raccolta di metriche. L'intervallo di raccolta delle metriche è di 15 secondi per tutti i job. Ogni job raccoglie metriche da servizi specifici attraverso le rispettive porte specificate. Le metriche raccolte per i microservizi authServ, cityServ, apiServ, nfyServ sono: le metriche per utilizzo della memoria, cpu, e spazio su disco. Inoltre, sono state raccolte altre metriche come numero di connessioni al database, numero di registrazioni al servizio e tempo di risposta si un api. Inoltre è stato implementato nodeExporter, è uno strumento di monitoraggio specifico per il sistema host, che esporta statistiche del kernel, della CPU, della memoria e di altri sottosistemi del sistema operativo

Gateway

Contenuto del file “roting.conf” fornisce la configurazione Nginx. Definisce tre upstream per i servizi denominati 'auth_serv', 'city_serv', e 'sla_manager', ciascuno in esecuzione su una porta specifica. Inoltre, configura un server Nginx in ascolto sulla porta 80 che agisce come proxy inverso per indirizzare le richieste in arrivo ai rispettivi servizi di backend tramite gli upstream definiti.

4. Testing e risultati

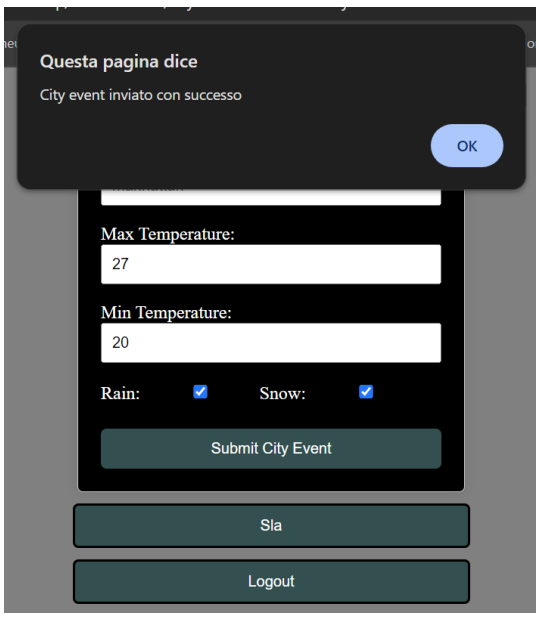
Il testing verrà effettuato attraverso build e deployment su docker.
Per il testing, verranno effettuate delle richieste tramite i vari client creati e sarà verificata la corretta comunicazione con i vari servizi e per ogni servizio la corretta comunicazione con il proprio database. La prima parte comprende la creazione di un nuovo utente tramite l'inserimento di alcuni dati, il microservizio associato a questa operazione è **auth_serv.**




```
mysql> select * from user;
+----+----------+----------+------------------+
| id | username | password | telegram_chat_id |
+----+----------+----------+------------------+
|  1 | user1    | 1234     | 498829409        |
+----+----------+----------+------------------+
1 row in set (0.00 sec)
```

Dopo aver inserito i dati, essi vengono memorizzati all'interno della tabella user del database associato a questo microservizio (authDb). In questo servizio viene generato un token che sarà utilizzato dal servizio successivo per poter associare le informazioni meteo all'utente detentore di quel token.

In seguito, sono state inserite alcune città di prova con varie constraint per verificare il corretto funzionamento, il microservizio utilizzato è **cityServ.**

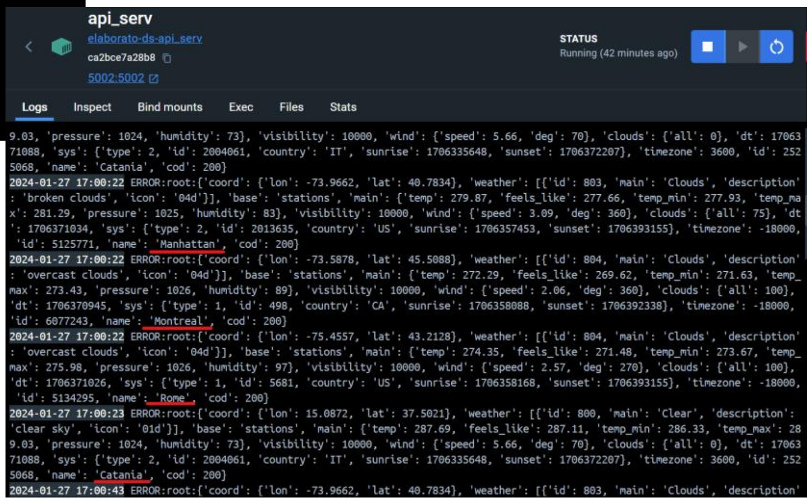



```
mysql> select* from user;
+----+------------------+
| id | telegram_chat_id |
+----+------------------+
|  1 | 498829409        |
+----+------------------+
1 row in set (0.01 sec)
```

```
+---------+---------+-----------+-------+-------+------+------+
| info_id | user_id | city      | t_max | t_min | rain | snow |
+---------+---------+-----------+-------+-------+------+------+
|       1 |       1 | manhattan |    27 |    20 |    1 |    1 |
|       2 |       1 | montreal  |    22 |    18 |    1 |    1 |
|       3 |       1 | roma      |    28 |    13 |    1 |    0 |
|       4 |       1 | catania   |    35 |     9 |    1 |    0 |
+---------+---------+-----------+-------+-------+------+------+
4 rows in set (0.00 sec)
```

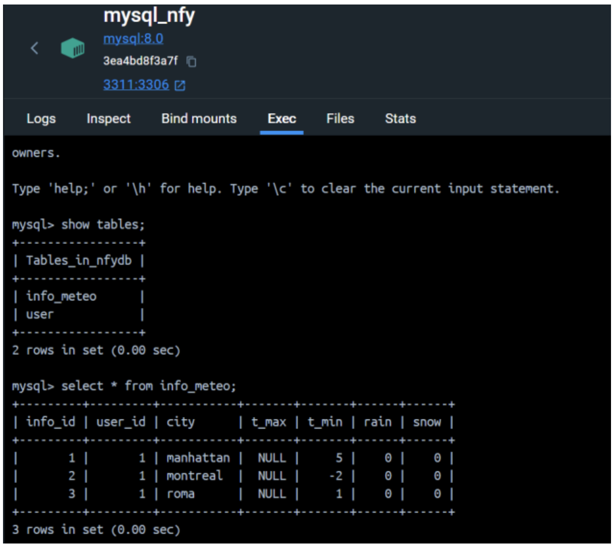
Come si può notare questi dati vengono salvati facendo riferimento allo stesso utente e verranno inviati tramite kafka, associando tali messaggi al topic "weatherInformation" al microservizio successivo

Una volta ricevute i dati tramite kafka, vengono memorizzati nel db associato al servizio **api_serv,** in seguito queste informazioni vengono utilizzate per effettuare una richiesta API al servizio OpenWeatherMap, la figura sotto mostra il risultato restituito dal servizio in formato json in cui sono state evidenziate le città di cui è stata fatta richiesta



In seguito, queste informazioni vengono elaborate, ovvero vengono confrontati i dati in tempo reale con le varie constraint inserite dall'utente e vengono inviate tramite kafka, associando tali messaggi al topic "Weather information", al servizio nfy_serv che si occuperà di notificare l'utente.

Infine, il microservizio **nfy_serv** riceve questi dati kafka e li memorizza all'interno del suo database che successivamente verranno inviati all'utente ad intervalli regolari di 15 minuti, come si può notare dalla figura.



Questa parte di testing riguarda il servizio **SLA_Manager** in cui sono richieste le seguenti operazioni: create/update metrica in sla con range di valori, query dello stato della sla, query del numero di violazioni delle ultime 1,3 e 6 ore. I dati sono resi persistenti attraverso la memorizzazione in un database.