ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

ATTIVITA' PROGETTUALE

in

Sistemi Distribuiti M

Realizzazione e analisi di un'applicazione
a Microservizi basata su Spring Cloud

STUDENTE:
Andrea Sturiale

PROFESSORE:
Paolo Bellavista

Sommario

Introduzione

I **Microservizi** corrispondono a una tecnica di sviluppo software tramite la quale un'applicazione viene strutturata come una collezione di componenti debolmente accoppiati tra loro. Questi componenti, chiamati *servizi*, sono dei moduli software, autonomamente modificabili, che implementano certe funzionalità specifiche del contesto di dominio in cui si sta lavorando. Ciascun di esso mette a disposizione delle proprie API rappresentanti le uniche operazioni utilizzabili dai relativi client esterni. Inoltre ogni componente di un sistema a microservizi può essere caratterizzato da un'architettura interna e un'implementazione che talvolta differisce totalmente dalle altre unità software. Ad esempio capita, per motivi legati al dominio, che siano realizzati in linguaggi differenti tra loro.
Questo pattern architetturale si propone in contrapposizione ad un'architettura di tipo Monolitico, ovvero ad un'applicazione in cui tutte le funzionalità sono inglobate in un singolo componente eseguibile su di un server. In una situazione del genere, se vogliamo apportare una modifica al nostro sistema dovremo necessariamente ogni volta riscrivere ed effettuare nuovamente il deployment di una nuova versione di tutta l'applicazione. Naturalmente con le sempre crescenti esigenze del mercato fare in modo che la propria applicazione rimanga continuamente al passo con i tempi risulta sempre più difficile e controproducente. Tali motivi hanno quindi spinto molti verso la ricerca di processi di sviluppo sempre più agili, veloci e in grado di far fronte in modo tempestivo ad eventuali cambiamenti impattando il minimo sull'intero sistema, portando così allo sviluppo di diversi approcci architetturale per la realizzazione di applicazioni.

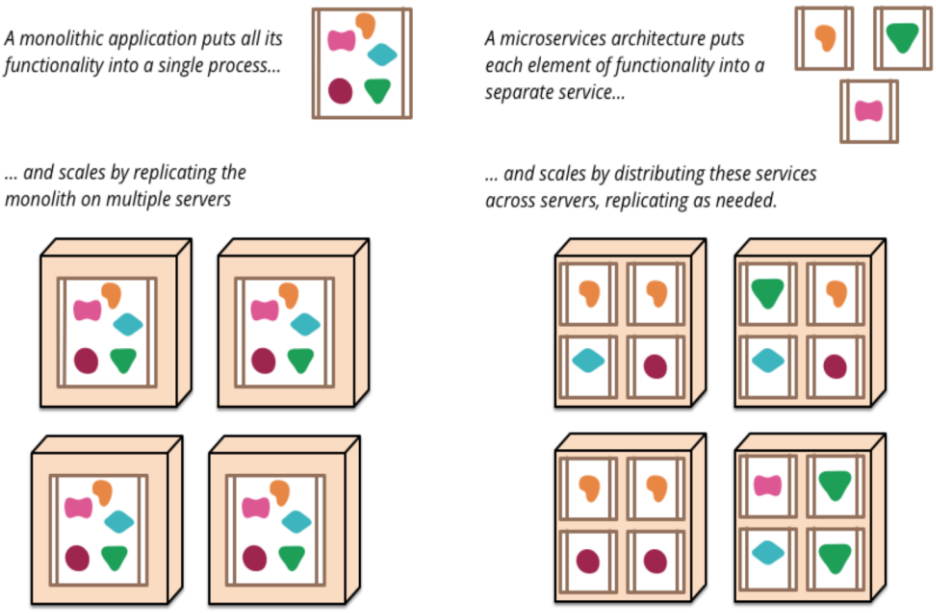

Figura 1: Applicazione monolitica vs Microservizi

Vantaggi e Svantaggi

Servizi singoli e semplici sono più facili da sviluppare rispetto al classico monolite e favoriscono maggiore scalabilità al sistema. Il loro disaccoppiamento è facilitato dal fatto che ciascun modulo controlla le proprie risorse rendendole accessibili dall'esterno passando necessariamente attraverso le opportune API. Ciò rende possibile che ogni servizio non solo possa essere scalato in modo indipendente dagli altri ma anche distribuito su piattaforme aventi risorse adatte e specifiche per quel componente. Questo è differente rispetto al caso del monolite, dove i moduli con diversi requisiti di risorse, ad esempio potremmo avere componenti CPU-intensive o memory-intensive, devono essere disposti insieme sullo stesso nodo. La modularità del sistema facilita inoltre la realizzazione e l'esecuzione di test automatizzati per ciascuno di esso.
Sviluppando un'applicazione a microservizi si ottengono altri vantaggi in termini di organizzazione del lavoro di sviluppo: ogni servizio può essere affidato ad un piccolo team di sviluppatori che può lavorare in modo completamente indipendente da tutto il resto, utilizzando anche linguaggi di programmazione, framework e tecnologie di archiviazione dati differenti, a patto di rimanere coerenti con l'interfaccia accordata precedentemente per le comunicazioni con gli altri. Lo sviluppo indipendente delle funzionalità vuol dire anche isolare eventuali malfunzionamenti; se si verifica un errore infatti, questo rimane circoscritto al servizio che l'ha generato evitando di compromettere l'esecuzione dell'intero sistema. Tutto ciò facilita l'adesione al modello *DevOps,* insieme di pratiche che conducono ad una rapida, frequente e affidabile distribuzione di applicazioni software con cui un'azienda è in grado di aumentare la velocità di soddisfacimento delle richieste di mercato.
Sfortunatamente, per quanto tutto possa sembrare bello e funzionale, ci sono anche degli aspetti negativi che non si possono assolutamente trascurare. Se da un lato tale architettura migliora la modularità e il disaccoppiamento, dall'altro introduce la complessità e i problemi legati ai sistemi distribuiti. Ad esempio ci si deve occupare della comunicazione fra i vari moduli software distribuiti, che è ben più complessa di una semplice chiamata locale di un sistema monolite.
Inoltre un servizio deve essere realizzato con lo scopo di riuscire a gestire fallimenti parziali durante la sua esecuzione, o talvolta deve interfacciarsi con componenti remoti non disponibili, cercando di mantenere la più bassa latenza possibile. In queste condizioni, effettuare dei test sull'intero sistema, potrebbe non risultare così semplice come invece accade per software monolitici. A ciò si aggiunge il fatto che solitamente ciascun servizio possiede il proprio database: se da un lato facilita il disaccoppiamento, dall'altro crea complessità nel gestire la consistenza dei dati, transazioni e query tra i vari microservizi.
In questo contesto bisogna ricorrere a pattern specifici per ogni singolo caso.

Tecnologie impiegate

Adesso passiamo all'analisi degli strumenti a disposizione nell'ambiente java per realizzare applicazioni tramite questo pattern architetturale specificando i benefici e i limiti per ciascuna tecnologia.

Spring Boot

Per quanto semplice il concetto di architetture a microservizi possa sembrare, iniziare a sviluppare ex novo un'applicazione di questo genere potrebbe non risultare cosa facile e immediata. Uno dei difetti di tale approccio è legato infatti alla complessità delle operazioni di setup per ogni singolo servizio, che potrebbero richiedere un considerevole ammontare di tempo per via delle svariate configurazioni da effettuare. Spring Boot è la soluzione ideale per chi vuole saltare tali step e passare direttamente allo sviluppo vero e proprio dei microservizi. Esso rappresenta infatti il punto di partenza per costruire applicazioni stand-alone basate su Spring (framework open source per piattaforme Java sviluppato dalla **Pivotal Software**) che possono essere avviate e implementate con semplicità. È realizzato quindi secondo il paradigma *convention over configuration*, che garantisce appunto configurazione minima per lo sviluppatore che utilizza un determinato framework. Ad esempio con Spring Boot troveremo già integrati all'interno dell'applicazione stessa container necessari per la gestione di richieste http quali Tomcat (default), Jetty e Undertow auto-configuranti e pronti all'uso.

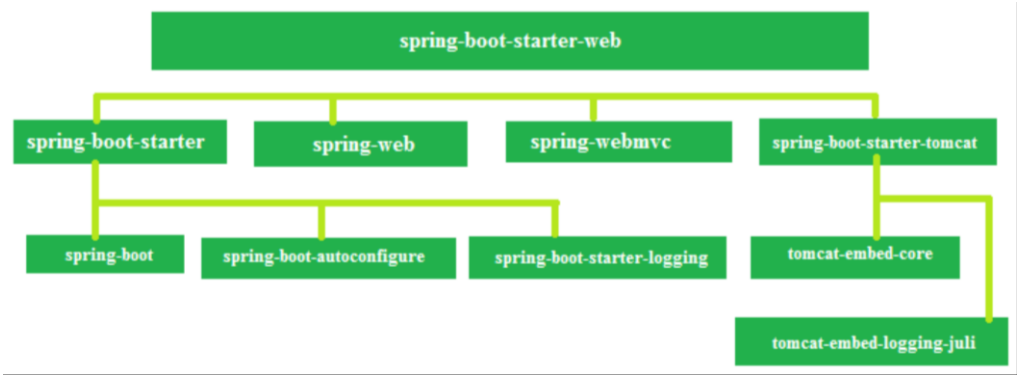

Figura 2: Spring Boot

Spring Initializr

Spring Initializr è un tool realizzato da Pivotal per il bootstrap e l'inizializzazione dei singoli servizi in maniera molto semplice. Basta infatti recarsi all'indirizzo https://start.spring.io/ per ritrovarsi davanti un'interfaccia grafica intuitiva di inizializzazione per il nostro progetto, come mostrato di seguito in figura. A questo punto occorrerà selezionare il tipo di progetto tra Maven e Gradle, il linguaggio di programmazione da utilizzare tra Java, Kotlin e Groovy e la versione di Spring Boot tra quelle disponibili. Dopodiché, una volta selezionato le dipendenze necessarie, verrà generato un archivio contenente il progetto, pronto per essere importato e avviato nell'IDE preferito. Successivamente, sulla base delle dipendenze inserite, viene avviato

il meccanismo di creazione automatica del workspace che provvederà ad importare tutte le librerie di cui si ha bisogno.

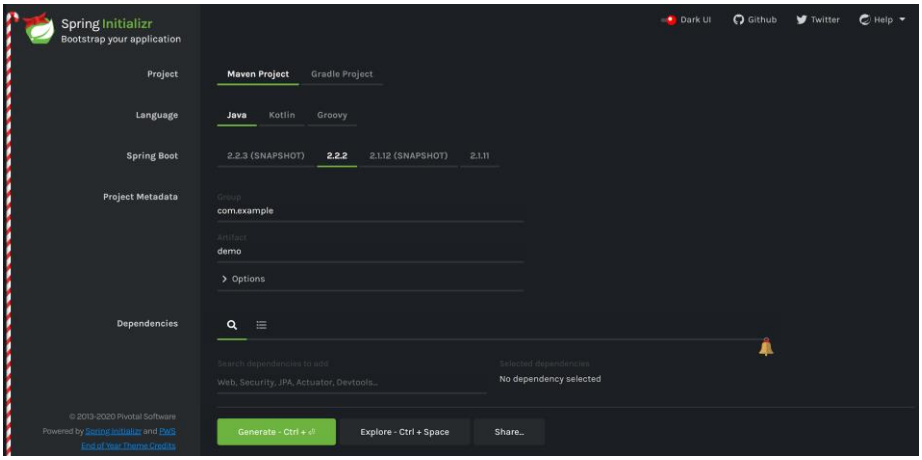

Figura 3: Spring Initializr

Spring Cloud

Spring Cloud è un progetto che nasce con l'intento di offrire agli sviluppatori Spring strumenti per la risoluzione, sempre in maniera rapida e veloce, di alcuni dei più comuni problemi legati alle architetture a microservizi e ai sistemi distribuiti in generale. In particolare per questa attività progettuale si farà riferimento a:

- **Spring Cloud Netflix:** progetto che fornisce un insieme di framework e librerie open source di Netflix con lo scopo di risolvere problemi relativi a sistemi distribuiti su larga scala. Nello specifico, si sfrutterà **Eureka** per il servizio di discovery all'interno del sistema.
- **Spring Cloud Stream:** framework adatto a realizzare microservizi altamente scalabili e connessi con sistemi a scambio di messaggi condivisi.
- **Spring Cloud Gateway:** componente di che si occupa di ricevere le richieste dei vari client e di inoltrarle agli appositi microservizi aggiungendo funzionalità cross cutting come ad esempio sicurezza, monitoring ecc.
- **Resilience4j:** libreria alternativa a Netflix Hystrix con lo scopo di garantire fault tolerance basandosi su Java 8 e sulla programmazione funzionale.

Grazie a tali strumenti sarà possibile, utilizzando semplici annotazioni e apposite configurazioni, costruire robusti sistemi sulla base dei migliori componenti ad oggi riconosciuti in tale campo.

Discovery Service

Ogni volta che si realizza un modulo software che invoca un servizio tramite delle REST Api, per mandare la propria richiesta, c'è la necessità di conoscere l'indirizzo IP e la porta dell'istanza del servizio. In un'applicazione tradizionale che esegue su un hardware fisico, la

posizione nella rete di una sua istanza è stabilita staticamente tramite un apposito file di configurazione. Ma in un sistema a microservizi basato su infrastrutture cloud la situazione è dinamica.
L'allocazione sulla rete di un servizio è assegnata dinamicamente, ma non solo, anche il numero di istanze presenti a run time cambia a causa di autoscaling, fallimenti ecc. Queste motivazioni portano all'utilizzo di un sistema di service discovery. La sua funzione principale è quella di mantenere la posizione corrente delle istanze delle varie applicazioni registrate. Il meccanismo di discovery viene aggiornato ogni qual volta che un'istanza inizia l'esecuzione o si ferma. Al momento dell'invocazione di un servizio da parte di un client, il framework lato cliente ottiene dal server una lista delle istanze disponibili e instrada le richieste a una di quelle. Tra i vari strumenti che si trovano in commercio il più diffuso è il framework *Eureka*, l'unico componente di Netflix OSS che ancora non è stato messo in modalità di mantenimento. Nello specifico Eureka si avvale di due pattern:

- **Client-side discovery:** in questo caso è il client che, una volta ottenuta la posizione dell'istanza del servizio da richiedere, comunica direttamente con il server. Una possibile alternativa, non adottata in questo caso dal framework, è quella di aggiungere un router tra client e server in grado di instradare le richieste comunicando direttamente con il meccanismo di discovery. Così facendo si toglie parte del carico dai vari clienti, aumentando però il numero di componenti da configurare e gestire.

- **Self registration pattern:** è il servizio a registrare la propria posizione sulla rete presso l'Eureka Server. Inoltre al momento della registrazione fornisce un "*health check URL"* utilizzato dal registry per verificare che l'istanza sia effettivamente disponibile per gestire le richieste. Lo stesso servizio deve invocare periodicamente un segnale "heartbeat" per prevenire la cancellazione della propria registrazione che viene mantenuta in memoria non persistente.

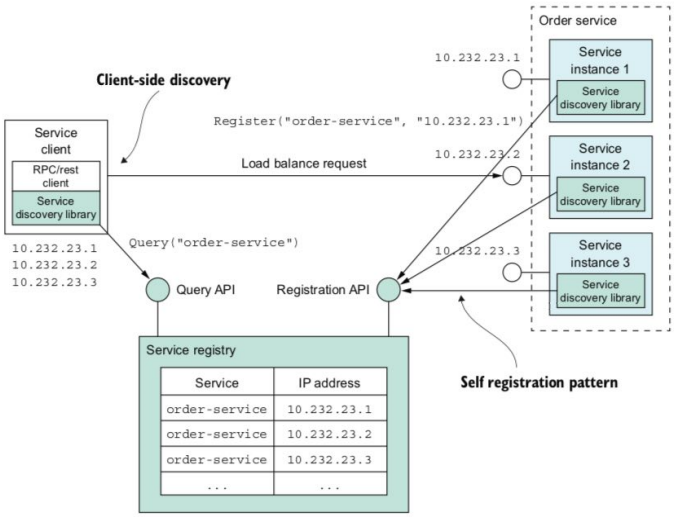

Figura 4: Eureka framework

In aggiunta, Eureka prevede all'interno della propria libreria presente lato client, un load balancer built-in in grado di bilanciare le richieste tra la lista delle istanze attualmente disponibili tramite un algoritmo round-robin. Infine c'è la possibilità di salvare su cache i risultati ottenuti evitando così di sovraccaricare la comunicazione fra Eureka Server e Eureka client.
Un beneficio di avere un sistema di discovery a livello di applicazione piuttosto che a livello di infrastruttura di deployment è che garantisce l'indipendenza da quest'ultima. Ad esempio si potrebbe avere il caso in cui una parte del sistema esegue su Kubernetes e la restante parte su un ambiente legacy, in questo caso l'Eureka server farebbe da tramite attraverso diverse piattaforme.
Uno svantaggio invece sarebbe la necessità di utilizzare una libreria di servizio di discovery per ogni linguaggio o framework differente che si utilizza. Spring Cloud facilita l'integrazione di Eureka all'interno delle applicazioni, ma ciò non è detto che accada utilizzando altri linguaggi quali Go o Javascript. Infine, da non sottovalutare, bisogna andare a configurare e gestire un ulteriore componente applicativo, il Discovery Server.

Spring Cloud Stream

Spring Cloud Stream è un framework per la creazione di applicazioni di microservizi incentrato su modello a scambio di messaggi. Esso si basa su Spring Boot per creare applicazioni autonome fornendo connettività con i broker di messaggi di diversi vendor, aggiungendo i concetti di "gruppi di consumatori" e partizioni.
L'applicazione comunica con l'esterno tramite dei *bindings* tra le destinazioni esposte dai broker e le funzioni realizzate nel codice sorgente. A far da collante vi sono i *Binders*, componenti specifici per il vendor che si sta usando che si occupano di gestire la connessione con essi. L'astrazione fornita da questi binder consente a un'applicazione di essere flessibile nel modo in cui ci si connette al middleware. Ad esempio, gli sviluppatori possono scegliere dinamicamente, in fase di esecuzione, il mapping tra le destinazioni esterne (come topic di Kafka o gli exchange di RabbitMQ) e la funzione che gestisce la ricezione dei messaggi a livello applicativo. Tale configurazione può essere fornita tramite proprietà esterne o in qualsiasi forma supportata da Spring Boot (inclusi argomenti dell'applicazione, variabili di ambiente e file application.yml o application.properties).
Spring Cloud Stream rileva automaticamente e utilizza il binder trovato nelle dipendenze del progetto, in aggiunta è possibile utilizzare diversi tipi di middleware con lo stesso codice. Per fare ciò, bisogna includere un binder diverso per ciascuno di esso.
La comunicazione tra le applicazioni che sfruttano Spring Cloud Stream segue un modello di pub-sub, in cui i dati vengono trasmessi attraverso canali condivisi. In questo modo si riduce la complessità sia del produttore che del consumatore e consente di aggiungere nuove applicazioni alla topologia senza interrompere il flusso esistente.
Molto spesso c'è la necessità di scalare il sistema creando più istanze dei vari componenti, ma nel fare ciò quest'ultime devono essere inserite in una relazione di concorrenza dove solo una di esse dovrebbe gestire un determinato messaggio. Spring Cloud Stream modella questo comportamento attraverso il concetto di un gruppo di consumatori al di là del fatto se il broker lo supporta o meno. Ogni associazione di consumatori può utilizzare la proprietà

spring.cloud.stream.bindings.<bindingName>.group per specificare un nome di gruppo. Per i consumatori mostrati nella figura seguente, questa proprietà verrà impostata come *spring.cloud.stream.bindings.<bindingName>.group=hdfsWrite* o *spring.cloud.stream.bindings.<bindingName>.group=average*.

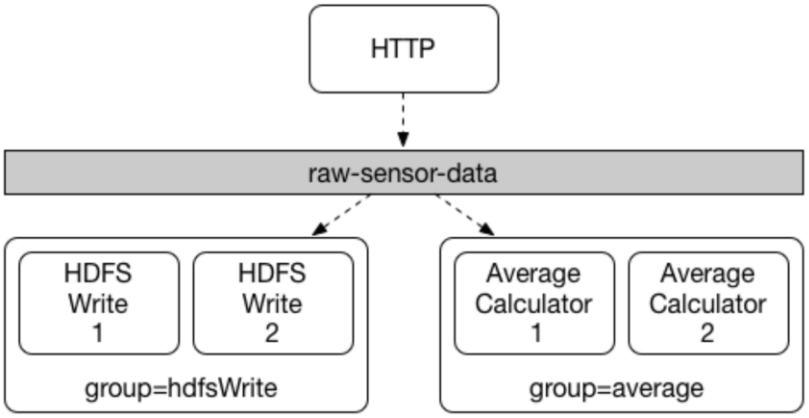

Figura 5: Spring Cloud Stream groups

Tutti i gruppi che si iscrivono a una determinata destination ricevono una copia dei dati pubblicati, ma solo un membro di ciascun gruppo riceve un determinato messaggio da quella destinazione. In questo modo si impedisce alle istanze dell'applicazione di ricevere messaggi duplicati. Per impostazione predefinita, quando non viene specificato un gruppo, Spring Cloud Stream assegna l'applicazione a un gruppo di consumatori anonimo e indipendente a membro singolo che è in una relazione di pub / sub con tutti gli altri gruppi di consumatori.
Inoltre gli abbonamenti ai gruppi di consumatori possono essere "*durable"*. In altre parole, il binder assicura che le iscrizioni ai gruppi siano persistenti e che, una volta creata almeno una sottoscrizione per un gruppo, il gruppo riceva i messaggi, anche se vengono inviati mentre tutte le applicazioni associate a quel gruppo non sono in esecuzione.
Spring Cloud Stream fornisce questa astrazione comune per implementare indipendentemente dal fatto che il broker stesso lo supporti naturalmente (ad esempio Kafka) o meno (ad esempio RabbitMQ).
Considerando il modello di programmazione invece, per conoscere meglio il funzionamento di questo framework è necessario vedere i seguenti concetti chiave:

- **Destination Binders:** componenti di Spring Cloud Stream che forniscono la configurazione e l'implementazione necessaria per facilitare l'integrazione con i sistemi di messaggistica esterni. Questa integrazione è responsabile della connettività, instradamento dei messaggi da e verso produttori/consumatori, conversione del tipo di dati, invocazione del codice utente e altro. Essi gestiscono molte funzionalità boiler-plate che altrimenti ricadrebbero sulle spalle dello sviluppatore. Tuttavia, per ottenere ciò, il destination binders necessita di un minimo set di istruzioni da parte dell'utente, che in genere si presenta sotto forma di configurazione dei *bindings*.

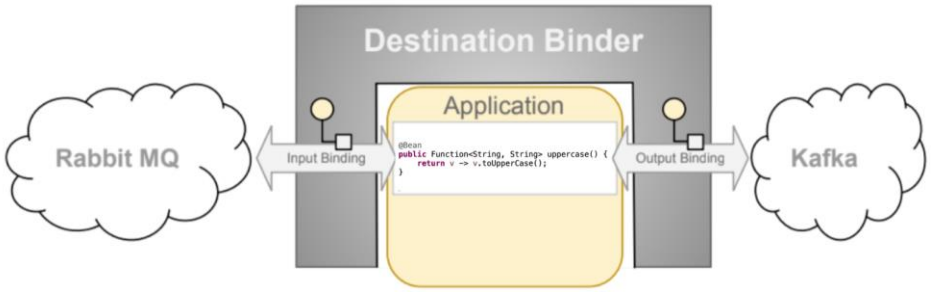

Figura 6: Spring Cloud Stream Binder

- **Binding:** astrazione che rappresenta il ponte tra le sorgenti/destinazioni esposte dal binder e il codice utente. Questa associazione codice-binder necessita di un nome configurabile tramite apposita proprietà *spring.cloud.stream.bindings.input.destination=myQueue*. In questo caso il binding si chiamerà "input" e sarà legato a una destination identificata con "myQue".
  A livello di codice, il framework mette a disposizione due alternative per specificare la funzione da chiamare al momento della ricezione o invio dei messaggi: approccio funzionale o basato su annotazioni. Optando per quest'ultimo, bisogna specificare innanzitutto il componente su cui si vuole eseguire il binding tramite l'annotazione *@EnableBinding*, specificando al suo interno l'interfaccia in cui è definita la sottoscrizione di riferimento, in questo caso *InputChannel*. Infine tramite l'annotazione *@StreamListener* si associa al binding il metodo invocato al momento della ricezione dei messaggi nella coda associata. Di seguito viene riportato un esempio di componente che riceve messaggi tramite il binding "input".

```
@EnableBinding(InputChannel.class)
public class VoteHandler {

  @Autowired
  VotingService votingService;

  @StreamListener(InputChannel.SINK)
  public void handle(Vote vote) {
    votingService.record(vote);
  }
```

```
public interface InputChannel {

  String SINK = "input";

  @Input(SINK)
  SubscribableChannel input();
}
```

- **Messaggio:** la struttura dei dati canonica utilizzata da produttori e consumatori per comunicare con Destination Binders (e quindi altre applicazioni tramite sistemi di messaggistica esterni) è caratterizzata da: un *Header*, insieme di coppie nome-valore tra cui distinguiamo il Content-Type del messaggio, il canale a cui inviare un'eventuale risposta, l'ID del messaggio ecc. e il *Payload* effettivo. Inoltre il framework fornisce già una lista di *MessageConverters* per gestire i casi d'uso più comuni di conversione del payload in oggetto Java.

Spring Cloud Gateway

Un Gateway è un servizio che fa da punto di ingresso dell'applicazione per il mondo esterno. Esso provvede una serie di API per i propri clienti, nascondendo l'architettura interna del sistema e semplificando il codice per gli sviluppatori, similmente a quanto viene fatto con il

Pattern Facade nella programmazione a oggetti. Le responsabilità chiavi di un API Gateway sono tre:

- **Routing delle richieste:** il componente si occupa di instradare le richieste in arrivo al servizio corrispondente. Quando esso riceve una chiamata, l'API Gateway consulta una mappa di routing che specifica a quale servizio mandare la richiesta che sta gestendo. Una possibile tabella potrebbe, ad esempio, consistere nel mappare un metodo HTTP e l'URL HTTP di un determinato servizio.
- **Composizione di API:** molto spesso accade che alcune operazioni del Gateway siano realizzate sfruttando chiamate multipli a vari componenti. Nell'esempio sottostante il client per ottenere un ordine, non è costretto a eseguire una richiesta per ciascun servizio necessario, ma sarà il Gateway a processare la richiesta recuperando i risultati dei diversi servizi chiamati in causa in modo da ridurre il carico di lavoro dei clienti.

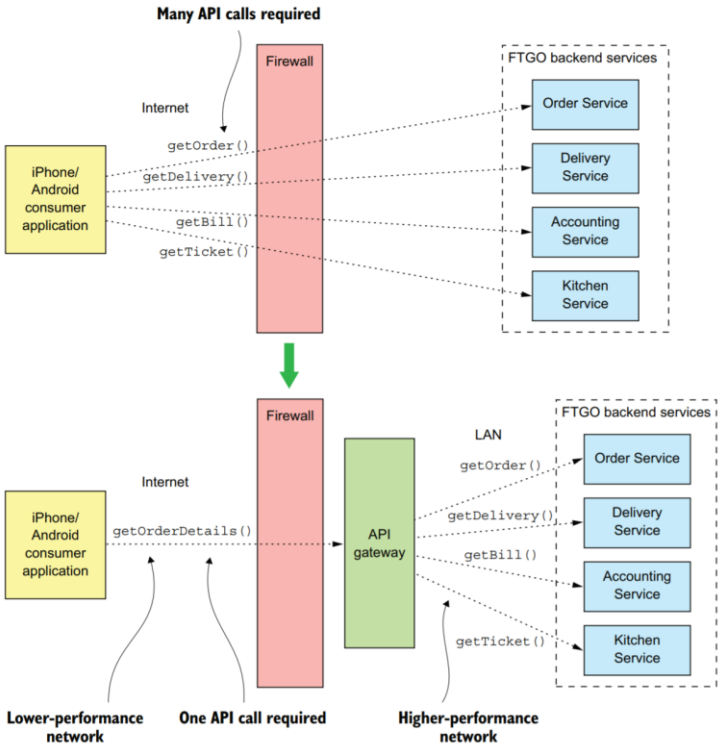

Figura 7: ApiGateway

- **Trasformazione di protocolli:** talvolta può accadere che il protocollo utilizzato per la comunicazione con l'esterno sia differente con con quello usato per chiamare i singoli servizi interni del sistema. In questo caso è compito del Gateway di fare da tramite tra i due ambienti. Ad esempio potrebbe fornire un'interfaccia REST per client esterni, anche se i servizi dell'applicazione utilizzano diverse combinazioni di protocolli internamente, quali REST o gRPC, protocollo per scambio di messaggi in formato binario con semantica sincrona.

Oltre a queste funzionalità primarie, all'interno del Gateway possono essere implementate le cosiddette "*funzionalità di confine*", ovvero funzioni che vengono realizzate ai confini di

un'applicazione o di un sistema. Gli esempi più tipici sono: *Autenticazioni* e *Autorizzazioni* delle richieste, *logging* delle operazioni, *monitoraggio del sistema* ai morsetti ecc. Ovviamente ciò comporta l'aumento di complessità del componente con il rischio di renderlo un collo di bottiglia sia durante la fase di sviluppo, sia durante l'esecuzione del sistema dato che un eventuale crash del Gateway provocherebbe la fine della comunicazione con i clienti.
Passando ora a Spring Cloud Gateway, esso rappresenta la soluzione che Spring offre per la realizzazione di un proprio componente personalizzabile. Spring Cloud Gateway si fonda su Spring Boot 2.x, Spring WebFlux, and Project Reactor. Proprio per quest'ultimo viene adottato un approccio *asincrono:* invece di avere un thread dedicato per ogni connessione con il client creando un limite di richieste concorrenti che l'API Gateway può gestire dovuti alla piattaforma in cui sta eseguendo, in un modello non bloccante un singolo processo in loop va ad eseguire la distribuzione delle richieste al gestore dell'evento corrispondente.
Spring Cloud Gateway si basa su tre concetti principali:

- **Route**: elemento base del gateway. È definito da un ID, un URI di destinazione, una raccolta di predicati e una raccolta di filtri. Una route viene abbinata a una richiesta se il predicato associato è verificato.
- **Predicato**: è un oggetto che verifica se la richiesta fornita soddisfa una determinata condizione. Per ogni route, si possono definire uno o più predicati che, se soddisfatti, accetteranno richieste per il servizio configurato dopo aver applicato i filtri.
- **Filter**: si tratta di istanze di Spring Framework GatewayFilter che sono state costruite con una fabbrica specifica. Qui, è possibile modificare richieste e risposte prima o dopo l'inoltro della richiesta al componente previsto.

Una volta che una richiesta arriva, la prima cosa che il gateway fa, nello specifico l'handler mapping, è verificare se la richiesta corrisponde a una delle route disponibili in base al predicato corrispondente. Una volta che una route coincide con la chiamata quest'ultima si sposta al Web Handler. Questo gestore manda la richiesta al servizio adatto passando attraverso una catena di filtri che possono eseguire la proprio logica sia prima che dopo il service la processa. Si parla in questo caso rispettivamente di "pre-filter" e "post-filter". Esistono molti filtri predefiniti forniti dal framework stesso per modificare l'header e il body della richiesta. Tra questi vi sono i "global filter" che possono essere applicati a tutte le chiamate in ingresso per eseguire ad esempio l'autenticazione e l'autorizzazione di ogni chiamata in un unico posto.

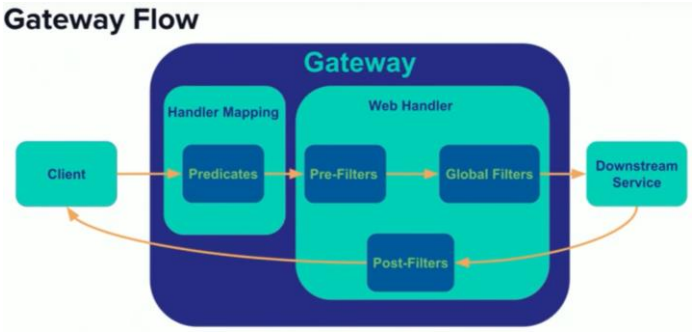

Figura 8: Spring Cloud Gateway architettura

Resilience4j

In un sistema distribuito, ogni volta che un servizio effettua una richiesta sincrona a un altro bisogna tenere a mente la possibilità di un fallimento parziale del nodo remoto. Poiché il client e il servizio sono processi separati, quest'ultimo potrebbe non essere in grado di rispondere in modo tempestivo ad esempio a causa di un errore o per manutenzione. O ancora il server potrebbe essere sovraccarico e rispondere molto lentamente alle richieste in arrivo. Poiché il client è bloccato in attesa di una risposta, c'è il rischio che l'errore o il ritardo possa propagarsi versi "i clienti" del cliente e così via, causando un'interruzione generale.
Per queste motivazioni ogni qualvolta che un servizio chiama sincronicamente un altro, dovrebbe salvaguardarsi combinando i seguenti meccanismi:

- **Timeout**: il cliente non si deve mai bloccare per un tempo indeterminato e al contrario bisogna utilizzare sempre i timeout durante l'attesa di una risposta garantendo che le risorse in uso non siano mai bloccate indefinitamente.
- **Limitare il numero di richieste da un client a un server:** talvolta bisogna imporre un limite al numero di richieste in sospeso che un cliente può effettuare a un determinato servizio. Se il limite è stato raggiunto, è probabilmente inutile continuare a inviare richieste aggiuntive e tali tentativi dovrebbero fallire immediatamente
- **Circuit Breaker Pattern:** esso si basa sul tenere traccia del numero di richieste riuscite con successo o fallite. Se il tasso di errore supera una certa soglia, viene aperto un "circuito virtuale" che blocca ulteriori tentativi impedendo probabili fallimenti. Infatti, un elevato numero di richieste non riuscite suggerisce che il servizio non risulti disponibile momentaneamente e che l'invio di successive richieste sia inutile. Inoltre dopo un periodo di timeout, viene lasciata la possibilità al client di riprovare e, in caso di successo, l'interruttore del circuito è chiuso riprendendo il flusso normale di esecuzione.

Netflix ha messo a disposizione una propria libreria open-source, Hystrix, che implementa i precedenti meccanismi e non solo. Ma da un anno a questa parte la stessa Netflix ha annunciato che non avrebbe supportato Hystrix in maniera attiva ma solo in modalità di mantenimento. Come conseguenza sono stati implementati vari progetti alternativi, tra cui quello di maggior successo è *Resilience4j*, realizzato appositamente per Java 8 e pienamente integrato all'interno di Spring Boot. Esso è framework usato per garantire tolleranza ai guasti e *resilienza,* ovvero rendere il sistema in grado di riprendere il flusso normale di esecuzione in seguito ad alterazioni.
Le funzioni centrali che questo progetto offre sono: *Circuit Breaker, Automatic Retry, Rate Limiter* e *Bulkhead pattern.* Per l'applicazione sono state utilizzate i primi due componenti.

Circuit Breaker Pattern

In Resilience4j il circuit breaker viene gestito come se fosse una macchina caratterizzati da tre stati finiti: CLOSED, OPEN, HALF_OPEN. Il passaggio da uno stato all'altro viene eseguito

in base al rateo di fallimento delle richieste salvato e aggiornato utilizzando una “sliding window” che aggrega le chiamate man mano che arrivano. Viene data la possibilità di scegliere tra due tipologie di finestre: *count-based* che aggrega gli ultimi n-esiti registrati o *time-based* con cui viene calcolato il risultato delle richieste negli ultimi n-secondi.
Lo stato del CircuitBreaker parte da CLOSED durante il quale vengono accettate le prime richieste e iniziato conteggio. Una volta che il rateo di fallimento è uguale o maggiore di una soglia configurabile esso passa ad OPEN, ad esempio quando più del 40% delle chiamate registrate ha avuto esito negativo. Per impostazione predefinita, tutte le eccezioni vengono considerate come un errore, ma è possibile definire un elenco di eccezioni che devono essere contate come errore e anche quelle che possono essere ignorate in modo da non considerarle né un fallimento né un successo.
Inoltre il CircuitBreaker può cambiare stato anche quando la percentuale di chiamate definite “slow” è uguale o maggiore di un’altra soglia configurabile. Ad esempio, quando oltre il 50% delle chiamate registrate ha impiegato più di 5 secondi per essere risolte. Questo aiuta a ridurre il carico su un sistema esterno prima che in realtà non riesca più a rispondere del tutto.
Quando il componente si trova nello stato OPEN rifiuta le chiamate in arrivo lanciando *CallNotPermittedException*. Successivamente, dopo che è trascorso un tempo di attesa, esso passa da OPEN a HALF_OPEN e consente un numero configurabile di chiamate per vedere se il servizio è tornato funzionante o meno. Se la percentuale di failure e la frequenza di chiamate lente sono inferiori alla rispettiva soglia, lo stato torna a CLOSED, altrimenti OPEN.

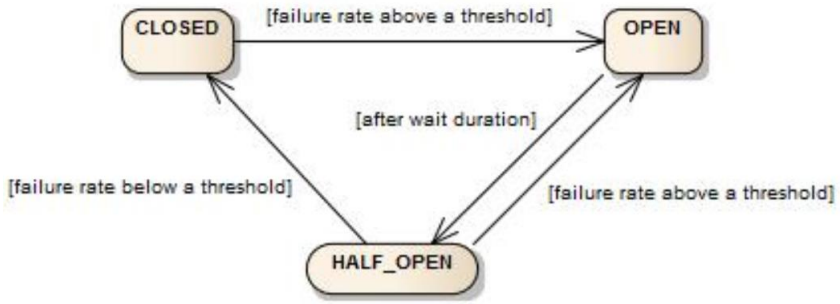

Figura 9: Resilience4j Circuit Breaker

In aggiunta il CircuitBreaker è thread-safe per i seguenti motivi:

- Il suo stato viene memorizzato tramite l’AtomicReference, classe che fornisce un riferimento ad una istanza letta e modificata atomicamente.
- CircuitBreaker utilizza operazioni atomiche per aggiornare lo stato con funzioni prive di effetti collaterali.
- La registrazione delle chiamate e il calcolo del rateo d’errore tramite la finestra scorrevole è gestita tramite apposito lock.

Ciò significa che solo un thread è in grado di aggiornare lo stato o la finestra scorrevole in un determinato momento. Ma l’esecuzione effettiva del metodo corrispondente alla richiesta in esame non fa parte della sezione critica. Altrimenti il CircuitBreaker introdurrebbe un calo di prestazione enorme e un collo di bottiglia.

Automatic Retry

Resilience4j-Retry è un componente che permette di riprovare l'invio di una richiesta a un servizio in modo del tutto automatico in seguito alla generazione di un'eccezione o al soddisfacimento di una specifica condizione. Esso consente di configurare diverse opzioni quali ad esempio il numero massimo di chiamate da ripetere, intervallo di tempo tra una richiesta e l'altre, le eccezioni da ignorare ecc.
Resilience4j-Retry viene di solito usato per quelle operazioni che vanno ad aggiornare o eliminare degli oggetti remoti o, alternativamente, per richiedere un servizio che tipicamente è stabile. Inoltre è spesso sfruttato in combinazione con altri pattern, come ad esempio il sopracitato CircuitBreaker, per garantire maggior robustezza al componente che si sta sviluppando. Ovviamente in quest'ultimo caso i tentativi aggiuntivi vengono conteggiati nel calcolo del rateo d'errore.

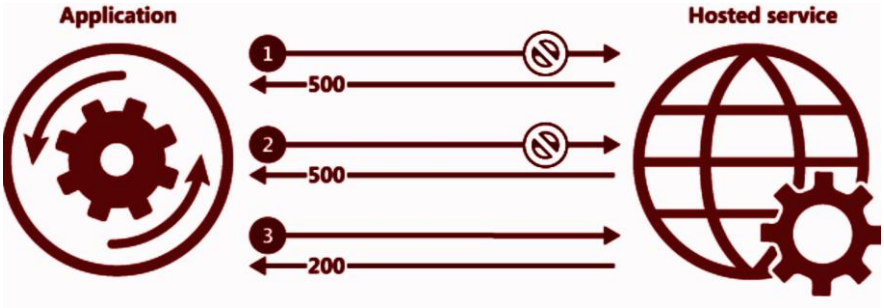

Figura 9: Resilience4j Automatic Retry

Architettura del sistema e implementazione

Il dominio applicativo prevede la realizzazione di un sistema che sia in grado di ricevere ed elaborare uno streaming di informazioni generate da un sensore di temperatura. Questi dati vengono mandati a un Message Broker tramite un'apposita configurazione del sensore o della piattaforma in cui viene installato. Le informazioni sono elaborate, man mano che il sensore le genera, da un apposito microservizio realizzato in Java che li salva in una serie di file d log e contemporaneamente notifica gli utenti tramite mail quando il valore ricevuto dalla sorgente supera una certa soglia. Inoltre per dare la possibilità di integrazione con una futura applicazione mobile o web è stato inserito un API Gateway all'interno dell'architettura con lo scopo di offrire le funzionalità del sistema tramite interfaccia standard e indipendente dai singoli microservizi.
Vista la natura dell'attività progettuale, ho preferito verticalizzare le mie conoscenze sulle tecnologie di Spring Cloud utilizzate per la realizzazione del sistema piuttosto che sulla configurazione della comunicazione fra il sensore e il broker dei messaggi. Per questo il caso d'uso sulla quale mi baserò sarà caratterizzato da un singolo sensore installato direttamente nella stanza o nel componente che si vuole monitorare remotamente. Quest'ultimo deve essere impostato in modo da inviare i messaggi direttamente al message broker remoto che può essere distribuito su qualsiasi piattaforma. Il sistema inoltre, che eseguirà remotamente all'interno di una infrastruttura cloud in fase di produzione deve dare la possibilità agli utenti di specificare quale sia la soglia oltre cui devono essere notificati e la mail corrispondente.
Di seguito saranno analizzati i singoli componenti del sistema dettagliando le scelte implementative impiegate sulla base di quanto detto prima.

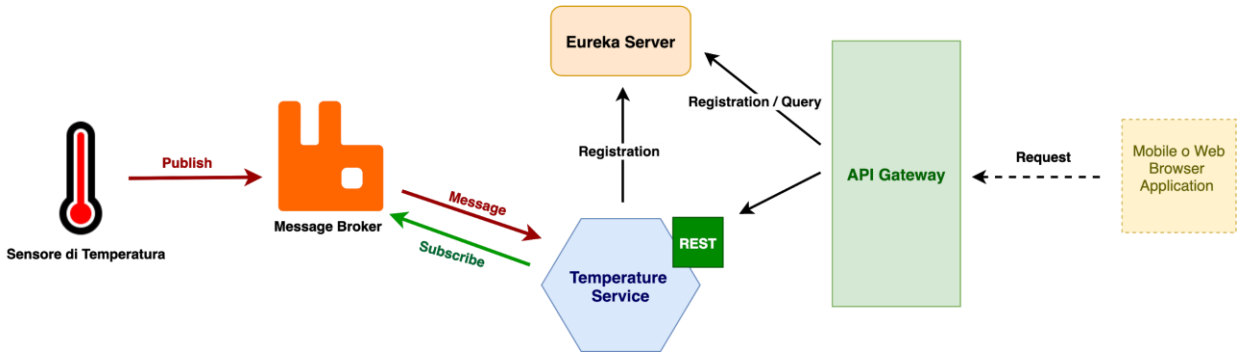

Figura 10: Architettura logica del sistema

Sensore di Temperatura

Il sensore di temperatura corrisponde alla sorgente dello streaming di dati che, raccogliendo informazioni sulla temperatura dell'ambiente circostante, li invia tramite Mqtt al message broker. Ma non essendo in possesso di un sensore vero e proprio, esso viene simulato tramite uno script scritto in python reperito dal seguente link di dominio pubblico "MQTT Generator Script". Questo script, sfruttando la libreria *Eclipse paho*, permette di inviare messaggi Mqtt dandoti la possibilità di modificarne il body e la frequenza di invio in un apposito file di

configurazione. Inoltre bisogna specificare l'indirizzo IP e la porta del nodo in cui esegue il broker, nel mio caso rispettivamente *localhost* e *1883*, dato che l'applicativo è implementato in un unico nodo nella fase di sviluppo, e infine il topic dove vengono pubblicati i pacchetti, ovvero *temperature*.
Per quanto riguarda il body dei messaggi inviati dal sensore ho stabilito che sia caratterizzati dal seguente schema riportato di seguito insieme a un esempio con valori numerici:

- **Unit:** stringa che specifica l'unità di misura corrispondente al valore che il sensore sta trasmettendo.
- **Range:** valore registrato dal sensore. Ho supposto che sia compreso all'interno dell'intervallo [0, 50], estremi compresi.
- **Description:** breve descrizione riguardante un aspetto del sensore, ad esempio la posizione del sensore.

```
{
   "unit" : "C",
   "range" : "37",
   "description" : "Main Entrance"
}
```

Message Broker

Un broker è un modello architetturale per la convalida, la trasformazione e il routing dei messaggi. Media la comunicazione tra le applicazioni, minimizzando la consapevolezza reciproca che le applicazioni dovrebbero avere l'una con l'altra per poter scambiare messaggi, implementandone efficacemente il disaccoppiamento. Essi sono componenti che supportano la comunicazione fra gli utilizzatori tramite un modello a scambio di messaggi definiti in modo formale. Tra i più utilizzati nell'ambiente dei sistemi distribuito troviamo *RabbitMQ*, un message-oriented middleware basato sul protocollo Advanced Message Queuing Protocol (**AMQP**).
Rispetto il tipico scenario dello scambio di messaggi, RabbitMQ introduce, oltre la presenza del **Publisher**, del **Consumer** e della **Queue/Topic**, un nuovo elemento: l'**Exchange**. In questo caso il Publisher non invia più il dato direttamente alla coda, ma passa per l'Exchange che si occupa di inoltrare il messaggio alla coda corretta in base ad una *chiave di routing* associata al pacchetto che si sta inviando.

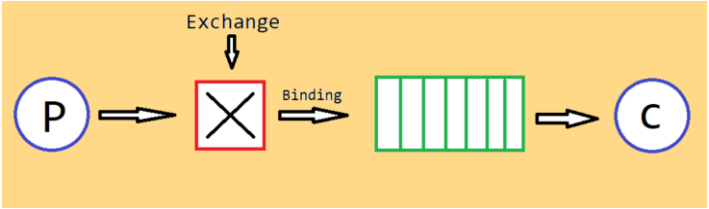

Figura 11: RabbitMQ exchange

Nonostante sia differente rispetto a AMQP, la grande diffusione di **MQTT** nell'ambiente mobile e IoT ha portato all'integrazione di questo protocollo all'interno di RabbitMQ tramite un'apposita estensione.
Invece di avere un protocollo fondato sugli exchange di AMQP, MQTT è limitato al classico modello pub/sub. Naturalmente, questa limitazione ha un impatto minore se si utilizza RabbitMQ, poiché i messaggi, i publisher, i subscriber MQTT sono trattati come se fossero i corrispettivi AMQP. Ci sono differenze nei messaggi pubblicati tramite MQTT e AMQP che sottolineano lo scopo di ciascuno protocollo. Essendo un protocollo leggero, MQTT è migliore tutte quelle volte in cui si ha a disposizione dell''hardware limitato senza connessioni di rete affidabili, mentre AMQP è progettato per essere più flessibile ma richiede ambienti di rete più robusti e affidabili.
I messaggi inviati tramite MQTT sono lunghi massimo 256 KB e in esso viene specificato il nome del topic a cui inviarlo. Quest'ultimo viene usato come *routing key* al momento del routing del messaggio MQTT sfruttando solo exchange di tipo topic, in particolare l'exchange *amq.topic è* il canale di comunicazione di default.
Dal punto di vista del subscriber invece quando ci si connette a RabbitMQ tramite MQTT, verrà creata una nuova coda. Essa verrà denominata usando il formato mqtt-subscriber-[NAME] qos [N], dove [NAME] è il nome client univoco e [N] è il livello QoS impostato sulla connessione. Ad esempio, una coda denominata mqtt-subscriber-facebookqos0 verrebbe creato per un abbonato di nome Facebook con un'impostazione QoS pari a 0. Una volta che la coda viene creata per una richiesta di sottoscrizione, sarà vincolata all'exchange topic precedentemente configurato utilizzando la semantica delle routing key prevista da AMQP.

Eureka Server

Come spiegato in precedenza l'Eureka Server è un componente che mantiene le informazioni di tutti i servizi che stanno eseguendo all'interno del sistema, nello specifico indirizzo IP e porta di ciascuno.
L'integrazione di Spring Cloud con le librerie messe a disposizione da Netflix OSS facilita l'implementazione dell'Eureka Server limandolo a tre semplici passi:

1. Bisogna aggiungere *spring-cloud-starter-netflix-eureka-server* alle dipendenze del proprio progetto. Questo può essere eseguito direttamente alla creazione del progetto se si utilizzasse la piattaforma Spring Initialzr, come nel mio caso. Oppure si aggiorna successivamente il pom.xml in caso di un'applicazione basato su Maven o alternativamente il file build.gradle nel caso di Gradle. Dato che ho sfruttato Maven per familiarizzare con un altro strumento rispetto a quelli presentati durante il corso, riporto di seguito solo la dipendenza aggiunta nel pom.xml.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

2. Dopo bisogna abilitare il server Eureka tramite l'apposita annotazione *@EnableEurekaServer* direttamente nella classe che esegue il main dell'Applicazione.

```
@EnableEurekaServer
@SpringBootApplication
public class DiscoveryServiceApplication {

	public static void main(String[] args) {
	   SpringApplication.run(DiscoveryServiceApplication.class, args);
	}
}
```

3. Infine si devono specificare le proprietà dell'applicazione in un apposito file, denominato *application.yml* in base al dominio del sistema. In questo caso ho configurato la porta del componente, *8761,* e aggiunto altri due attributi per avviarlo in modalità che si definisce *standalone.* Questo perchè a default ogni server Eureka è anche un client Eureka e richiede (almeno uno) un URL per individuare altre istanze di Eureka server presso cui registrarsi. Se non lo si fornisce, il servizio viene eseguito e funziona lo stesso, ma riempie la console di numerosi errori. Per evitare questo si va eliminare il comportamento da client tramite due apposite proprietà settate come false e riportate di seguito. Infatti considerando la semplicità del sistema sviluppato composto da Api Gateway e Temperature Service, la combinazione di cache lato client di e di heartbeat periodico rende l'Eureka server sufficientemente resistente ai fallimenti anche in modalità a singola istanza.

```
server:
   port: 8761
 eureka:
   client:
    registerWithEureka: false
    fetchRegistry: false
```

Infine automaticamente il framework offre una dashboard in cui poter monitorare lo stato del server al seguente indirizzo *http://eurekaAddress:port.* Non solo ti permette di verificare quali servizi si sono registrati ma fornisce altre informazioni ad esempio la memoria disponibile, quella occupata, da quanto tempo il server è funzionante ecc. Al momento dell'acquisizione della figura sottostante erano registrati solo due servizi: Api-Gateway e Temperature-Service. Come si può notare per quest'ultimo vi sono due indirizzi registrati poiché Temperature-Service è stato replicato in due istanze differenti in ascolto sulle porte 8082 e 8083. In questo modo quando un client chiederà l'indirizzo in corrispondenza del nome logico *temperature-service* riceverà sia http://localhost:8082 che http://localhost:8083, scegliendone uno per l'invio della richiesta tramite ad esempio il load balancer di default fornito da Eureka o uno custom.

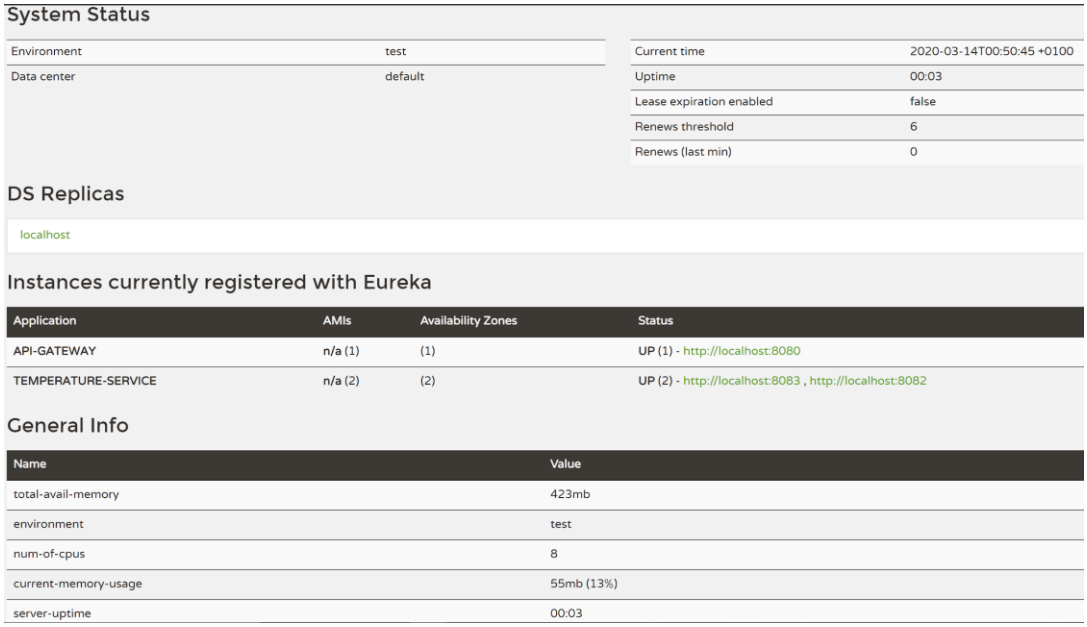

Figura 12: Eureka Server dashboard

Spring Cloud Gateway

Così come è stato fatto per l'Eureka server, per implementare Spring Cloud Gateway bisogna aggiungere la dipendenza corrispondente al progetto su cui si sta lavorando direttamente con Spring Initializr o successivamente alla creazione.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Inoltre, essendo che viene sfruttato Eureka per trovare la posizione della rete del nodo a cui instradare le richieste in arrivo, bisogna aggiungere la dipendenza *spring-cloud-netflix-eureka-client* e abilitare il comportamento da client con l'apposita annotazione *@EnableEurekaClient.* Così facendo l'applicazione andrà a registrarsi automaticamente presso il discovery server e sarà possibile eseguire delle query per la risoluzione dei nomi logici di cui si ha bisogno.

Per quanto riguarda la definizione delle *routes* necessarie per instradare le richieste provenienti dall'esterno, vengono offerte due possibilità di approcci: o tramite file di configurazione esterno, ad esempio *application.yml*, o alternativamente in un bean specifico nel codice sorgente. Avrei optato per la prima possibilità poichè permette di cambiare le route a tempo di esecuzione senza nessuna modifica del codice sorgente.

Essendo che in questo caso l'architettura interna è composta solamente dal TemperatureService, nell' application.yml è stata inserita solo una route. Quest'ultima permetterà di inoltrare qualsiasi richiesta in arrivo che farà match con "*http://apiGatewayAddress:apiGatewayPort/temperature/users/***" a un'istanza del

microservizio corrispondente. Per evitare di fissare l'indirizzo di quest'ultimo nella configurazione, si fa riferimento ad un nome logico che poi verrà usato al momento dell'arrivo di una richiesta per la sua risoluzione tramite query all'Eureka server in modo del tutto trasparente. Ciò è specificato con l'apposita proprietà *spring.cloud.gateway.routes.uri: lb://TEMPERATURE-SERVICE*. Come si può notare prima del nome logico è stata inserita l'espressione "lb" per abilitare il client load balancer basato sulla libreria Ribbon di NetflixOSS che usa un algoritmo round robin a default. Alternativamente si può pensare di realizzare un proprio load balancer custom sfruttando politiche di bilanciamento del carico più complesse come il traffico corrente ecc.
Infine è stato aggiunto un filtro apposito corrispondente a un CircuitBreaker con cui gestire le chiamate a TemperatureService sfruttando la libreria Resilience4j. Questo per garantire maggior robustezza possibile all'ApiGateway ad eventuali crash del microservizio e del servizio di discovery.
Dopo aver aggiunto le dipendenze necessarie al pom.xml tra cui *spring-cloud-starter-circuitbreaker-reactor-resilience4j* e *resilience4j-spring-boot2*, bisogna configurare il circuito. Per questo componente, ho optato di tenere in considerazione il risultato delle ultime 100 chiamate effettuate per il calcolo del rateo di errore e del rateo di chiamate considerate lente se eseguono con un ritardo di 2.5s, impostando la chiusura del circuito quando viene superata la soglia del 25% per uno dei due valori. Infine Spring Cloud Gateway dà la possibilità di configurare un uri a cui inoltrare la chiamata ogni volta che viene rifiutata una richiesta o viene generato un errore sia perchè il TemperatureService non risponde, sia perchè il discovery server è down. In questo caso ho realizzato un controller interno al Gateway che restituisce un messaggio di errore di default per le operazioni che prevedono la modifica di un'entry del database. Invece per una richiesta GET si va a cercare se l'entità in questione è presente nella cache condivisa con tutte le istanze di TemperatureService restituendo così un risultato anche in caso di fallimento di quest'ultimo. Alternativamente si potrebbe pensare di inoltrare la richiesta a un servizio esterno apposito per la situazione di recovery.
Per andare ad applicare il Circuit Breaker al filtro corrispondente infine c'è bisogno di realizzare un bean di nome *ReactiveResilience4JCircuitBreakerFactory* all'interno del progetto. Questo componente permette al supporto di istanziare il circuit breaker corretto, con relative proprietà, in base al nome specificato nella configurazione, in questo caso "*temperatureService*", in modo trasparente allo sviluppatore.
Di seguito riporto solo una parte dell'application.yml del progetto.

```
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lowerCaseServiceId: true
      routes:
```

```
- id: temperatureModule
uri: lb://TEMPERATURE-SERVICE
predicates:
- Path=/temperature/users/**
filters:
- name: CircuitBreaker
  args:
    name: temperatureService
    fallbackUri: forward:/fallback/temperature
```

Temperature Service

Adesso si passa a descrivere il componente centrale del sistema, realizzato secondo il pattern Database per Service. Esso è costituito da due moduli principali: il primo svolge il compito di gestire i messaggi Mqtt inviati dal sensore tramite il broker RabbitMQ e di notificare per mail gli utenti sulla base del messaggio corrente; il secondo invece permette agli utenti di aggiungere la mail o modificare la soglia di temperatura oltre cui vogliono essere avvertiti salvando queste informazioni in un apposito database esterno. Entrambi i moduli sono strutturati su tre livelli in modo da separare il livello che risponde alle chiamate esterne (in verde), a quello che implementa la business logic del microservizio (in viola) interagendo con un servizio esterno tramite il supporto di Spring Boot che fornisce rispettivamente le *Repository* per i database e *MailSender* per inviare una mail.

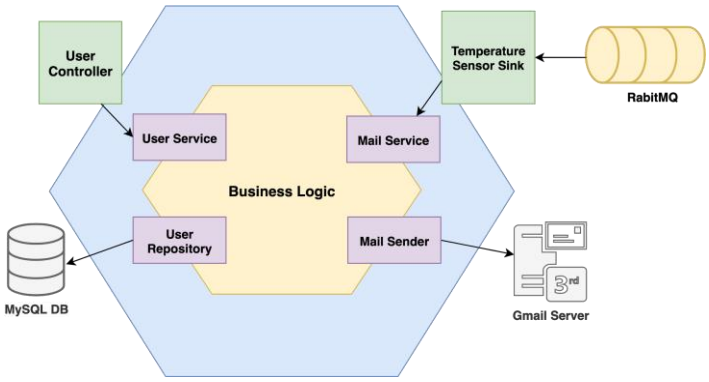

Figura 13: Temperature Service

Temperature Module

Per la ricezione dei messaggi dal message broker RabbitMQ questo modulo sfrutta Spring Cloud Stream e sulla base del loro contenuto vengono notificati gli utenti che hanno precedentemente registrato la propria mail. In aggiunta il body viene salvato nel proprio file system locale attraverso Spring AOP e il framework di log *Logback* che viene automaticamente chiamato al momento della ricezione del pacchetto.
Inizialmente bisogna aggiungere la dipendenza per Spring Cloud Stream e per il binder, rispettivamente *spring-cloud-stream e spring-cloud-stream-binder-rabbit.* Dopo si passa alla configurazione del binding specificandone il nome, *message-sink* e la destination del broker a cui collegarsi, in questo caso *amq.topic.* Per il binder, invece, devono essere definite le proprietà con cui viene eseguita la sottoscrizione a rabbitMQ. In questo caso

essa dovrà essere durevole, il tipo di exchange invece sarà topic, mentre la routing key associata alla coda da cui prendere i messaggi è *temperature* (che corrisponde al topic del Mqtt Generator) e infine l'exchange è durevole, ovvero i messaggi devono essere salvati su memoria persistente dal broker.

```
spring:
 cloud:
  stream:
    bindings:
      message-sink :
        destination: amq.topic
        binder: rabbit
        group: temperature-consumer-group
        consumer :
         concurrency: 1
    rabbit:
     bindings:
      message-sink:
        consumer:
           bindQueue: true
           bindingRoutingKey: temperature
           durableSubscription: true
           declareExchange: true
           exchangeDurable: true
           exchangeType: topic
           queueNameGroupOnly: true
```

Da notare che è stato specificato inoltre la proprietà *spring.cloud.stream.bindings.group* in modo da sfruttare l'omonima astrazione fornita da Spring Cloud Stream. In questo modo replicando più istanze di questo microservizio non si avrà il problema di processare messaggi duplicati poichè solo un'istanza fra le appartenenti allo stesso gruppo di nome *temperature-consumer-group*, riceverà il messaggio appena accodato. Di conseguenza l'utente non si vedrà email duplicate nella propria casella postale.
All'avvio dell'applicazione Spring Cloud Stream, tramite il binder specifico di RabbitMQ, va a sottoscrivere il sistema considerando le proprietà precedentemente definite e genera una coda con lo stesso nome del gruppo, associandola all'exchange amq.topic con annessa routing key temperature.
Ovviamente dovranno essere fornite a parte le credenziali e l'indirizzo con cui accedere al nodo dove esegue il message broker.

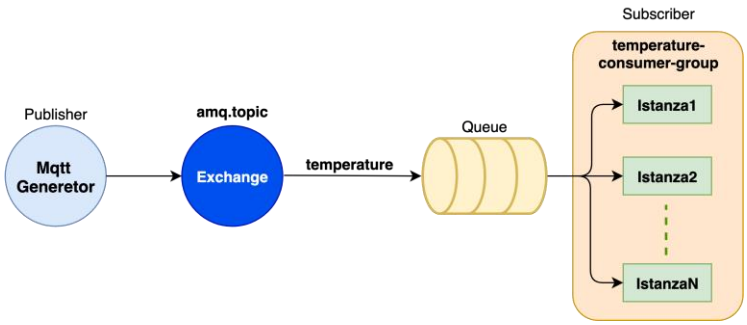

Figura 14: Consumers group

Una volta finito con la configurazione del canale di comunicazione fra l'applicazione e il message broker si passa all'implementazione del controller che gestisce i messaggi di nome

TemperatureSensorSink. Questa classe si avvale dell'annotazione @StreamListener per segnalare il metodo "handle" da invocare al momento della ricezione del pacchetto. Al momento dell'intercettazione del massaggio dal canale di input denominato message-sink, il body viene automaticamente convertito in un oggetto Java, in questo caso *TemperatureSensorMessage*, sulla base della tipologia del contenuto, a default *application/json*.

```
@StreamListener(InputChannel.SINK)
public void handle(TemperatureSensorMessage message) {
   try {
        mailService.sendEmailToUsers(message);
   } catch (Exception e) {
       log.error("Error in sending mail: " + e.getMessage());
   }
 }
```

Come si può notare dal codice la classe TemperatureSensoreSink delega l'invio delle mail a un altro componente detto *MailService*. Quest'ultimo legge il valore della temperatura corrente dal body e prende dal database una lista di utenti che devono essere avvisati poichè la temperatura è salita rispetto al limite previsto.
Per inviare una la mail è stata usata la libreria *JavaMail* che necessita della dipendenza *spring-boot-starter-mail*. Questo framework fornisce l'interfaccia *JavaMailSender* che permette sia l'invio di messaggi semplici, come nel mio caso, poichè contenenti solo del testo oppure di tipo mime. Il passaggio successivo consiste nello specificare le proprietà del server di posta che si vuole utilizzare, in questo caso Gmail, nel file application.yml utilizzando lo spazio dei nomi spring.mail. *.

```
spring:
  mail:
    host: smtp.gmail.com
    port: 587
    username: mail
    password: password
```

Gmail permette di inviare mail tramite un account dando l'accesso al server SMTP che utilizza la porta 587. Bisogna tenere presente che la password per l'account non è quella ordinaria, ma rappresenta un token generato appositamente per fare gestire l'email da applicazioni terze. Ovviamente Gmail impone un limite sia nel numero di messaggi giornalieri che possono essere inviati sia nel numero di mail che possono essere aggiunti in copia nell'invio di un messaggio, rispettivamente 2000 e 100. Per lo sviluppo e il testing dell'applicativo non ci sono problemi per queste limitazioni, alternativamente si deve cercare un provider alternativo per un contesto di produzione.
Essendo che si sta facendo affidamento a un servizio esterno, in questo caso Gmail, si è pensato di inserire un circuit breaker in modo da filtrare le chiamate al server. Ad esempio in caso in cui il server sia in sovraccarico o momentaneamente non disponibile, si va a chiudere il circuito in modo da impedirne chiamate successive per un lasso di tempo di 5s

con lo scopo di non rallentare la lettura dei messaggi Mqtt dal broker. Dato che inoltre il server Gmail non è direttamente gestito da me nè tanto meno è collegato tramite una rete affidabile, ho impostato il ritardo per considerare una chiamata "slow" a 3s rispetto ai valori settati per gli altri moduli. Per lo stesso motivo è stato configurato l'error rate a valori più alti oltre cui attivare il circuito, nello specifico 35% delle chiamate effettuate.

Users Module

Lo user module offre la possibilità di aggiungere o modificare l'email o il valore della temperatura oltre cui si vuole ricevere un avvertimento. Questo componente del microservizio è stato realizzato tramite il pattern "Controller, Service e Repository" in modo da isolare le chiamate esterne dalla fonte dati usata. Nello specifico essendo che le informazioni di un utente sono relativamente semplici si è optato per un database Mysql per la gestione e la persistenza dei dati nella fase di sviluppo. Adesso andiamo nel dettaglio nei due sottocomponenti principali.

User Controller

Nell'ambiente Spring quando si devono fornire dei servizi a dei client esterni esponendo le proprie API viene tipicamente usato RESTful, stile architetturale basato sullo scambio di messaggi sincrono che sfrutta direttamente il protocollo HTTP. Alternativamente si potrebbe pensare di utilizzare un altro protocollo come gRPC che, non essendo limitato a HTTP, permette la realizzazione di servizi con un ricco insieme di operazioni ed è più efficiente nello scambio di messaggi di grandi dimensioni. Ma non avendo il contesto queste necessità si è optato per il primo approccio.
Il concetto chiave in REST è la *risorsa* che rappresenta un singolo oggetto della logica di business. In questo caso la risorsa centrale è l'Entity User che, sfruttando i metodi messi a disposizione da HTTP, viene referenziata tramite vari URL. Ad esempio una richiesta GET restituisce una rappresentazione della risorsa in questione, tipicamente nella forma di JSON, mentre una POST permette di creare un nuovo user all'interno del sistema e così via.
Di conseguenza il controller in Spring è stato realizzato annotando la classe UserController con *@RestController* che indica al container di serializzare i valori di ritorno dei metodi in formato JSON e associa il componente alla risorsa User esponendo gli URL necessari per la modifica, inserimento, eliminazione e reperimento degli oggetti.
Di seguito viene riportato per semplicità come sono stati realizzati gli endpoint per le richieste GET e POST, le restanti PUT e DELETE sono gestite in modo analogo.

```
@RequestMapping(value = "/temperature/users/{userId}", method = RequestMethod.GET)
public User getUserById(@PathVariable int userId) {
    return userService.getUserById(userId);
}
```

```
@RequestMapping(value = "/temperature/users", method = RequestMethod.POST)
public User addUser(@Valid @RequestBody User user) {
    return userService.saveUser(user);
}
```

Per associare un metodo del controller a una determinata richiesta HTTP si utilizza l'annotazione *@RequestMapping.* Ad esempio quando verrà effettuata una richiesta GET all'url "*http://ip:porta/temperature/users/1*" verrà restituito lo user con id 1. Inoltre è possibile automatizzare il processo di validazione dell'input tramite *@Valid*, nello specifico viene controllato se l'utente inserisca un'email conforme o se la soglia rispetta i vincoli imposti dal sensore.
Infine è stato aggiunto un controller apposito, *ExceptionHandlerController* che gestisce le eccezioni lanciate da UserController in modo da mandare messaggi di errore personalizzati e standard. Ad esempio se un client richiede uno user il cui id non è presente riceverà il seguente messaggio:

```
{
   "timestamp": "2020-09-04 11:09:05",
   "status": 400,
   "error": "NOT_FOUND",
   "message": "Not found any user with id: 1"
}
```

User Service

Corrisponde al layer che si interfaccia direttamente con la sorgente dei dati rendendo lo UserController indipendente dalla tipologia di persistenza adottata.
Per quanto riguarda la logica di business del sistema, questo componente si limita a mappare le richieste che arrivano dal controller nei rispettivi metodi messi a disposizione da UserRepository per reperire o modificare i dati degli utenti. Ad esempio all'arrivo di una richiesta GET il controller invocherà il metodo *getUserById* di UserService che a suo volta prenderà il bean corrispondente tramite il Repository dal database Mysql. Analogamente è stato fatto per le altre richieste.
In aggiunta, con lo scopo di migliorare le performance del sistema in termini di latenza di risposta che il microservizio offre, è stato applicato un livello di cache in-memory al componente UserService. Inoltre la cache permette di diminuire una parte del carico di lavoro del database che viene concorrentemente acceduto sia dal modulo che gestisce gli utenti sia dal modulo della temperatura che si interfaccia con il sensore.
Tra tutte le possibilità che sono in commercio è stato scelto di utilizzare *Hazelcast,* sistema open-source di accesso dati direttamente in memoria Ram basato su Java. Hazelcast offre la possibilità di realizzare un livello di cache distribuito su vari nodi di un cluster, di conseguenza i dati sono partizionati e divisi tra di essi. Inoltre vengono messi a disposizione vari pattern da poter applicare all'interno dell'ambiente dei microservizi, ad esempio si potrebbe pensare di gestire la cache in un cluster fisicamente distante dell'ambiente di deployment del proprio sistema. Ma visto che sia l'ApiGateway che TemperatureService

sono stati realizzati tramite Spring e quindi eseguono su JVM ho sfruttato il pattern definito *embedded distributed cache.* Nello specifico ciascuna istanza avrà una propria cache locale che a livello applicativo viene vista come se fosse privata, ma che in realtà appartiene a un cluster di nodi che condividono i dati tra loro.

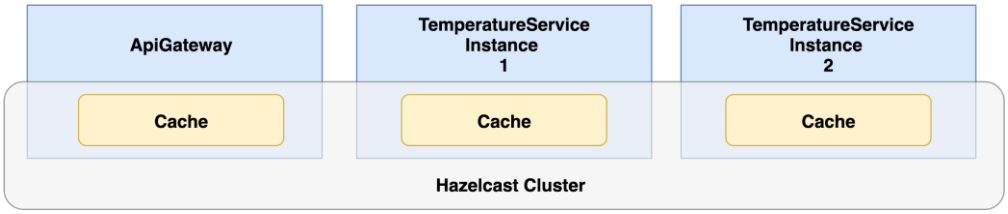

Figura 15: Hazelcast cluster

Questo tipo di architettura garantisce una minor latenza rispetto al caso precedente poichè gli accessi alla cache sono eseguiti localmente o al massimo in un nodo locale al cluster e non a un servizio di deployment esterno. Ma impone il vincolo di realizzare microservizi basati su Java poichè Hazelcast in questo caso sfrutta la comunicazione tra più JVM per la sincronizzazione e la ripartizione dei dati tra i vari host.
La cache viene gestita da Hazelcast come una Mappa, ovvero un insieme di coppie di oggetti "*<chiave,valore>",* divisa su più nodi in modo tale che ogni entry della struttura dati si trovi solo su di una singola istanza del cluster. Quest'ultima viene definita copia *primaria* poiché è la sola responsabile dell'esecuzione delle operazioni sulla partizione che gestisce. In aggiunta vengono garantite diverse repliche per ciascuna entry nei restanti nodi detti di backup in modo da rendere il sistema più robusto in seguito a un crash del nodo primario di quella determinata frazione.

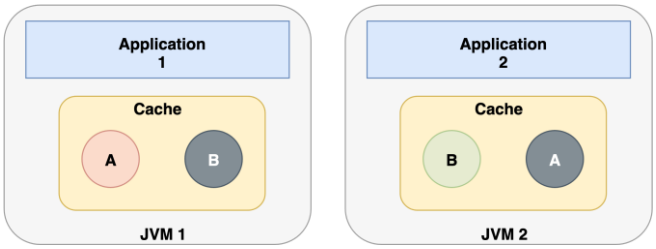

Figura 16: Hazelcast distributed cache

Quando si legge o si scrive una voce della mappa, si comunica in modo trasparente con il membro Hazelcast a cui è assegnata la replica primaria della partizione corrispondente. In questo modo, ogni richiesta raggiunge la versione più aggiornata di una particolare voce di dati in un cluster stabile. Le repliche di backup rimangono in modalità standby fino a quando la copia primaria rimane funzionante. In caso di errore della copia primaria, una tra quelle di backup viene promossa al ruolo principale.
Ogni nodo nel cluster è configurato per avere le stesse funzionalità condividendo la *tabella delle partizioni* che comprende diverse informazioni come i dettagli dei membri, lo stato del cluster, le informazioni riguardanti i backup e il partizionamento, ecc. Ma tra di essi è il primo nodo creato a gestire gli altri membri eseguendo automaticamente l'assegnazione

dei dati. Se quello più vecchio smette di funzionare, il secondo più vecchio riprende a coordinare i restanti e così via.
Tutti i nodi rimanenti sono definiti *smart client*, ovvero hanno i metadati relativi al cluster ma con informazioni restrittive rispetto al coordinatore corrente. Pertanto, un client può connettersi direttamente alla replica primaria del dato per cui è arrivata la richiesta riducendo eventuali ritardi causati della rete poichè non si deve passare attraverso il coordinatore.
Per aumentare le performance inoltre gli update dei backup vengono eseguite in modo *lazy:* quando la copia primaria riceve un'operazione di aggiornamento per una chiave, la si esegue localmente e la si propaga alle repliche di backup senza però aspettare che l'operazione di update abbia successo, non garantendo così piena consistenza fra copia primaria e backup.
Per quanto riguarda, invece, l'integrazione della cache all'interno del microservizio e dell'ApiGateway è stato sfruttato Spring Boot Cache. Questo framework fornisce pieno supporto per l'aggiunta trasparente della memorizzazione in cache a un'applicazione con lo scopo di ridurre il numero di esecuzioni di metodi in base alle informazioni disponibili. La logica di memorizzazione viene applicata in modo trasparente, senza alcuna interferenza con il chiamante. Spring Boot automaticamente configura l'infrastruttura della cache in base alla dipendenza che trova all'interno del classpath del progetto, in questo caso *com.hazelcast.hazelcast-spring*, purché il supporto sia abilitato tramite *@EnableCaching*.
Questo livello di astrazione dallo specifico provider che si sta usando è ottenuto sfruttando le seguenti annotazioni:

- *@Cacheable:* indica che il risultato dell'invocazione di un metodo (o di tutti i metodi in una classe) può essere memorizzato nella cache. Ogni volta che viene invocato il metodo associato, verrà controllato se nella cache è già presente il valore di ritorno per la corrente chiave passata come parametro di ingresso. Se non è trovato alcun valore, il metodo target viene richiamato e il risultato verrà memorizzato nella cache associata, alternativamente l'oggetto reperito dalla cache è restituito senza l'esecuzione della funzione.
  Questa annotazione è stata usata solo per "getUserById" poichè è l'unico metodo che prevede il reperimento delle informazioni dal database e quindi l'unico di cui ha senso eseguire il caching.

```
@CircuitBreaker(name = "userService", fallbackMethod = "fallbackGetUser")
@Cacheable(value = "userCache", key = "#id")
public User getUserById(int id) {
    User user=userRepository.findById(id).orElseThrow(() -> new UserNotFoundException("Not
    found any user with id: "+id));
    return user;  }
```

- *@CachePut:* contrariamente all'annotazione @Cacheable, il metodo in questo caso viene eseguito per ogni chiamata e inserisce i risultati nella cache eseguendo un refresh del dato se presente in precedenza. Per questo l'annotazione è stata associata

a tutti quei metodi che vanno a inserire o modificare le entità nel database mantenendo così la cache aggiornata.

```
@Retry(name = "retryService", fallbackMethod = "fallbackAddUser")
@CachePut(value = "userCache", key = "#user.id")
public User saveUser(User user) {
    return userRepository.save(user);
}
```

- *@CacheEvict:* annotazione che permette di eliminare una entry nella cache in corrispondenza di una determinata chiave. Questa è stata associata al metodo "deleteUserById" in modo da togliere e risparmiare spazio ogni qual volta venga eliminato uno User dal database.

```
@Retry(name = "retryService", fallbackMethod = "fallbackDeleteUser")
@CacheEvict(value="userCache",key="#id", beforeInvocation = false)
public void deleteUserById(int id) {
    try {
        userRepository.deleteById(id);
    } catch (EmptyResultDataAccessException e) {
       throw new UserNotFoundException("Not found any user with id: "+id);
    }
}
```

Infine essendo che il database su cui vengono salvati i dati può essere causa di rallentamenti sia dovuti a eventuali crash del nodo in cui esegue o poichè processa troppe chiamate rispetto al carico che riesce a reggere, è stato aggiunto un circuit breaker per regolare le richieste. Con lo scopo di ridurre il più possibile la latenza media delle risposte che vengono inoltrate al Gateway sono marcate come "slow" le richieste con un ritardo al di sopra di 2.5s.
Visto che il database è parte integrante dell'architettura e non un servizio esterno sono stati imposti vincoli più restrittivi rispetto ai parametri usati per il MailService. Quindi è stato stabilito un massimo per il rateo di errore e il rateo di chiamate lente del 25% oltre cui viene aperto il circuito bloccando le richieste in entrata.

```
resilience4j.circuitbreaker:
  backends:
    userService:
        register-health-indicator: true
        slidingWindowSize: 100
        minimumNumberOfCalls: 10
        slow-call-duration-threshold: 2000
        slow-call-rate-threshold: 25
        permittedNumberOfCallsInHalfOpenState: 25
        automaticTransitionFromOpenToHalfOpenEnabled: true
        waitDurationInOpenState: 4000
        failureRateThreshold: 25
        ignoreExceptions:
            - org.springframework.dao.DataIntegrityViolationException
            - it.distributedsystems.projectactivity.temperatureservice
            .exception.UserNotFoundException
```

Come si può notare sono state precisate due eccezioni tramite la proprietà *ignoreExceptions* indicando quelle che non devono essere considerate nel calcolo del rateo di errore poichè sono gestite all'interno della business logic del componente. Tra queste troviamo DataIntegrityViolation che viene lanciata ogni qualvolta è violato un vincolo SQL nella table corrispondente e UserNotFound per quando si vuole accedere a uno User il cui id non è presente nel sistema.
Inoltre Resilience4j permette di definire un metodo di fallback che viene automaticamente eseguito nel momento in cui un metodo fallisce lanciando un eccezione. Per la gestione dell'errore in getUserById viene semplicemente mandato un messaggio personalizzato poichè comporta che non è stato trovato l'utente in cache e che non si è riusciti nemmeno a reperire le informazioni dal database. Per i restanti metodi è stato aggiunto il componente *Retry* della libreria Resilience4j con la seguente configurazione.

```
resilience4j.retry:
   instances:
     retryService:
       maxRetryAttempts: 3
       waitDuration: 500
       ignoreExceptions:
             - org.springframework.dao.DataIntegrityViolationException
             - it.distributedsystems.projectactivity.temperatureservice
             .exception.UserNotFoundException
```

Visto che per questi metodi non ha senso salvare su cache i risultati precedenti poichè provocano la modifica delle tabelle del database, l'utilizzo di Resilience4j-Retry serve per garantire maggior robustezza.
Se per esempio la chiamata del metodo "saveUser" dovesse fallire la prima volta per qualsiasi motivo che non sia a causa di eccezioni gestite appositamente, viene riprovato altre due volte intervallando 500ms di attesa tra un tentativo e un altro. Da specificare che tutti le ripetizioni eseguite sono conteggiate per il rateo di errore del circuito agendo in combinazione con esso.
Di seguito vengono proposti due grafici rappresentanti una sequenza di funzionamento del circuit breaker con il relativo rateo di errore.
Fin dall'avvio del componente il circuito è nello stato CLOSED (in grigio) permettendo la ricezione delle chiamate in arrivo mentre il rateo di errore è pari a -1. Solo all'arrivo della 10 richiesta, come specificato da "minimumNumberOfCalls", inizia il calcolo del rateo di fallimento e nel momento in cui viene superato o raggiunto il 25% il circuito passa a OPEN (in viola) automaticamente. Dopo 4s in cui le chiamate allo UserService sono rifiutate, il circuito diventa HALF_OPEN (in verde) permettendo fino a un massimo di 25 richieste. Da notare che durante quest'ultima transazione il rateo si azzera e solo dopo l'arrivo della decima chiamata viene ricalcolato. Infine successivamente il circuito ritorna CLOSED poichè il rateo è rimasto sotto il 25%.

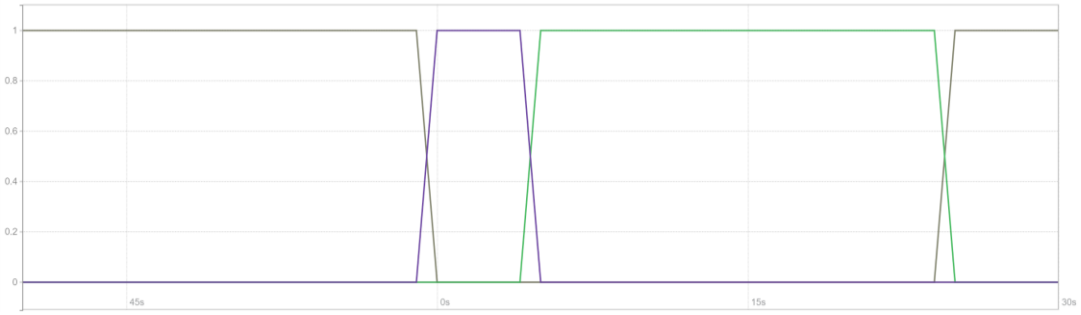

Figura 17: Stato del CircuitBreaker al trascorrere del tempo

Figura 18: Rateo di errore in % del CircuitBreaker al trascorrere del tempo

Test sulle performance del sistema

Una volta conclusa la fase di implementazione e di testing dei singoli componenti si è passato a eseguire i test sulle performance dell'intero sistema. Quest'ultimi sono stati realizzati tramite il software Apache Jmeter sfruttando la possibilità di eseguire richieste HTTP concorrenti al Gateway. I seguenti test vertono su quanto incide l'utilizzo della cache nel ridurre la latenza media delle risposte e saranno applicati in due scenari differenti: il primo con una sola istanza di TemperatureService e l'altro con due istanze replicate. In entrambi i casi i test sono stati eseguiti simulando 300 richieste concorrenti in 5s di cui il 50% di esse richiedono operazioni in scrittura sul database e il restante 50% operazioni di lettura. I componenti del sistema eseguiranno sullo stesso nodo fisico, ovvero il mio portatile in ambedue gli scenari.

Scenario 1

In questo caso sono stati condotti test multipli considerando sempre solo una singola istanza di TemperatureService ma alternando l'utilizzo della cache o meno con il completo funzionamento del microservizio o meno.

Label	**#Samples**	**Avarage**	**Min**	**Max**	**Std. Dev**	**Error %**	**Throughput**
HTTP Post	150	45	32	103	13.78	0	15.1/sec
HTTP Get	150	15	12	24	1.99	0	30.4/sec
Total	300	30	12	103	17.98	0	59.5/sec

Tabella 1: Test senza l'utilizzo della cache e senza ricezione di messaggi Mqtt

Label	**#Samples**	**Avarage**	**Min**	**Max**	**Std. Dev**	**Error %**	**Throughput**
HTTP Post	150	60	34	138	23.63	0	15.1/sec
HTTP Get	150	15	12	79	6.88	0	30.4/sec
Total	300	38	12	138	27.18	0	59.5/sec

Tabella 2: Test senza l'utilizzo della cache e con ricezione di messaggi Mqtt

Label	**#Samples**	**Avarage**	**Min**	**Max**	**Std. Dev**	**Error %**	**Throughput**
HTTP Post	150	46	35	83	10.02	0	15.1/sec
HTTP Get	150	5	4	11	1.04	0	30.5/sec
Total	300	25	4	83	17.98	0	59.3/sec

Tabella 3: Test con l'utilizzo della cache e senza ricezione di messaggi Mqtt

Label	#Samples	Avarage	Min	Max	Std. Dev	Error %	Throughput
HTTP Post	150	64	39	153	28.78	0	15.1/sec
HTTP Get	150	7	4	17	1.99	0	30.4/sec
Total	300	35	4	153	31.24	0	59.5/sec

Tabella 4: Test con l'utilizzo della cache e con ricezione di messaggi Mqtt

Considerando tutti i casi nel complesso si nota che le operazioni più costose sono gli accessi al database che provocano un aumento dei tempi di risposta, in particolare quelle che richiedono scritture nel database. Inoltre come si può vedere l'utilizzo del livello di caching comporta un considerevole miglioramento delle performance provocando un abbassamento dei tempi medi di risposta portando rispettivamente la media da 30ms a 25ms in un caso e da 38ms a 35ms nel secondo caso.
Comunque sia con o senza cache le performance del sistema subiscono un lieve peggioramento quando il modulo della temperatura riceve i messaggi dal message broker e invia email parallelamente alla gestione degli utenti.

Scenario 2

Similmente a quanto fatto per lo scenario 1 caso sono stati eseguiti test multipli alternando l'ultilizzo della cache o meno con il completo funzionamento del microservizio o meno, ma stavolta erano in esecuzione due istanze di TemperatureService.

Label	#Samples	Avarage	Min	Max	Std. Dev	Error %	Throughput
HTTP Post	150	44	33	99	11.21	0	15.1/sec
HTTP Get	150	17	12	25	1.94	0	30.4/sec
Total	300	31	12	99	13.57	0	59.6/sec

Tabella 1: Test senza l'utilizzo della cache e senza ricezione di messaggi Mqtt

Label	#Samples	Avarage	Min	Max	Std. Dev	Error %	Throughput
HTTP Post	150	60	33	167	27.44	0	15.1/sec
HTTP Get	150	15	12	39	4.76	0	30.4/sec
Total	300	37	12	167	27.18	0	59.3/sec

Tabella 2: Test senza l'utilizzo della cache e con ricezione di messaggi Mqtt

Label	#Samples	Avarage	Min	Max	Std. Dev	Error %	Throughput
HTTP Post	150	47	35	101	15.02	0	15.1/sec

HTTP Get	150	6	5	25	2.04	0	30.3/sec
Total	300	27	5	101	19.19	0	59.5/sec

Tabella 3: Test con l'utilizzo della cache e senza il modulo della temperatura che esegue

Label	**#Samples**	**Avarage**	**Min**	**Max**	**Std. Dev**	**Error %**	**Throughput**
HTTP Post	150	64	38	204	35.73	0	15.1/sec
HTTP Get	150	8	4	35	3.69	0	30.4/sec
Total	300	36	4	204	41.24	0	59.1/sec

Tabella 4: Test con l'utilizzo della cache e con il modulo della temperatura che esegue

Analizzando i dati riportati in precedenza sono stati ottenuti risultati del tutto simili a quelli dello scenario 1. Effettivamente la replicazione del microservizio non comporta la riduzione della latenza delle risposte poichè quello dipende dagli accessi al database che corrispondono alle operazioni più costose. Infatti per questo punto di vista è l'uso della cache ciò che fa abbassare la latenza media delle richieste processate.
Ma è stato condotto un ultimo test in cui per verificare il massimo carico che il sistema supporta nei due scenari. Ed è risultato che mentre nello scenario 1 il carico massimo arriva a circa 500 richieste in 5s, nello scenario 2 invece si riesce a gestire circa 1000 richieste tra scrittura dal database e lettura della cache. Quindi replicando il sistema si ottiene la possibilità di risolvere più richieste parallele in maniera approssimativamente proporzionale al numero di istanze. In quest'ultimo caso però bisogna tener presente che le performance continuano essere legate all'utilizzo di un semplice database Mysql.

Conclusioni e sviluppi futuri

In sintesi è stato realizzato un sistema che è in grado di interfacciarsi ad un sensore di temperatura da un lato e in grado di gestire informazioni degli utenti dall'altro. Sono state sfruttate a pieno le tecnologie che Spring Cloud offre sfruttando Spring Cloud Stream per la gestione dello streaming dei messaggi inviati tramite Mqtt mentre i restanti framework sono stati usati per gestire il modulo degli utenti con la relativa comunicazione REST.
Seppur il sistema sia relativamente semplice allo stato attuale sono presenti tutti i componenti necessari per ingrandirlo, quali ad esempio l'Eureka Server e l'ApiGateway. Infatti se in futuro volessimo aggiungere un altro sensore da monitorare, basterebbe realizzare un microservizio che gestisce appositamente il flusso dei messaggi e aggiungere le proprie Api all'ApiGateway fornendo l'accesso alle relative informazioni. Allo stesso modo si può realizare un'applicazione client che sia in grado di fornire una vista di tutti i sensori monitorati dal sistema agli utenti agganciandosi agli endpoint del Gateway.
Il passo successivo sarebbe quello di occuparsi del deployment del sistema ad esempio su Kubernetes, una piattaforma portatile e open source per la gestione di carichi di lavoro e servizi containerizzati facilitando sia la configurazione dichiarativa che l'automazione. In questo modo si potrebbe sostituire l'Eureka Server con il servizio di discovery proprio della piattaforma stessa non dovendolo gestire più a livello applicativo. Inoltre Hazelcast mette a disposizione un pattern differente rispetto a quello usato in precedenza definito *sidecar cache,* in cui viene associato una cache locale a ciascun container indipendentemente dal linguaggio di programmazione, non avendo più la limitazione di dover usare Java con relativa JVM.

Fonti

Spring Framework: Pivotal, https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/ .
Spring Boot: Pivotal, https://docs.spring.io/spring-boot/docs/2.1.6.RELEASE/reference/html/ .
Martin Fowler: https://martinfowler.com/microservices/ .
Microservices Pattern with examples in Java: Chris Richardson, 2019.
Spring Cloud Stream: Pivotal, https://cloud.spring.io/spring-cloud-static/spring- cloud-stream/current/reference/html/index.html .
Spring Cloud Gateway: Pivotal, https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.2.2.RELEASE/reference/html .
Resilience4j: https://resilience4j.readme.io .
Spring Cloud Netflix: Pivotal, https://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/2.2.2.RELEASE/reference/html/ .
RabbitMQ in depth: Gavin M. Roy, 2018.
MQTT Generator Script: Mariano Guerra https://gist.github.com/marianoguerra/be216a581ef7bc23673f501fdea0e15a .
Hazelcast: https://docs.hazelcast.org/docs/latest-dev/manual/html-single/ .
Jmeter: Apache, http://jmeter.apache.org/ .