



**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**

Dipartimento di
Ingegneria Gestionale, dell'Informazione e della Produzione

Corso di laurea in
Ingegneria Informatica

Classe n. L-8

Pattern architetturali per la resilienza di applicazioni a micro-servizi nel framework Spring

Candidato:
Andrea Ingoglia

Matricola n.1068751

Relatore:
Chiar.ma Prof.ssa Patrizia Scandurra

Anno Accademico
2021/2022

Indice

Introduzione.....	5
1. Introduzione al problema affrontato	5
2. Obiettivo della tesi	6
3. Presentazione dei capitoli.....	7
1. La resilienza nelle applicazioni a servizi.....	8
1.1. Architettura a micro-servizi	8
1.2. Caratteristiche e problematiche di un sistema a micro-servizi	9
1.2.1. Scalabilità e distribuzione.....	9
1.2.2. Guasti parziali	9
1.2.3. Granularità di un micro-servizio e overhead di comunicazione	10
1.2.4. Sincronismo delle comunicazioni	10
1.3. Definizione di resilienza	11
1.4. Stato di un sistema.....	11
1.4.1. Classificazione degli stati di un sistema	11
1.5. Classificazione della resilienza	12
2. Tecnologie di sviluppo utilizzate.....	14
2.1. Introduzione	14
2.2. Tecnologie utilizzate	14
2.2.1. Java.....	14
2.2.2. IntelliJ IDEA.....	15
2.2.3. GIT	15
2.2.4. GitHub	16
2.2.5. Database H2.....	16

2.2.6.	Postman	16
2.2.7.	Spring	17
2.2.8.	Maven	18
2.2.9.	Spring Data JPA.....	18
2.2.10.	Spring MVC.....	19
2.3.	Il modello esagonale.....	20
3.	Pattern e meccanismi in Spring per la resilienza.....	22
3.1.	Introduzione	22
3.2.	Panoramica delle tecniche per la resilienza e per l'osservabilità.....	22
3.3.	Resilience4j: la libreria per la resilienza.....	23
3.3.1.	Circuit Breaker e API di fallback.....	23
3.3.2.	Time Limiter	24
3.3.3.	Rate Limiter	24
3.4.	Componenti essenziali per un'architettura a micro-servizi	25
3.4.1.	API Gateway.....	26
3.4.1.1.	Utilizzo dei filtri in Spring Cloud Gateway.....	26
3.4.1.2.	Integrazione tra Spring Cloud Gateway e il Load Balancer	27
3.4.2.	Registro dei servizi.....	28
3.4.2.1.	Pannello di visualizzazione.....	29
3.4.2.2.	Tabella dei servizi del prototipo	29
3.5.	Definizione di Osservabilità	30
3.6.	Logging.....	30
3.6.1.	Sleuth.....	31
3.6.2.	Elasticsearch, Logstash, Kibana.....	31

3.7.	I log su Cloud: integrazione tra Logback ed ELK	31
3.7.1.	Logger.....	32
3.7.2.	Appender.....	32
3.7.3.	Configurazione di Kibana.....	34
3.8.	Health Check.....	35
3.9.	Prometheus.....	36
3.9.1.	Configurazione di Prometheus.....	36
3.9.2.	Monitoraggio dello stato di salute di un servizio	38
3.9.3.	Generazione di grafici per il monitoraggio delle prestazioni	39
4.	RNMA: Resilient Newsletter Microservice Application	40
4.1.	Introduzione	40
4.2.	Idea di progetto	40
4.3.	Architettura del sistema	41
4.4.	Processo di registrazione	44
4.4.1.	Prima fase: acquisizione dei dati di registrazione	44
4.4.2.	Seconda fase: invio dell'email di verifica	46
4.4.3.	Terza fase: conferma dell'identità.....	47
4.4.4.	One way interaction.....	49
4.5.	Validazione del processo di registrazione	50
4.6.	API di supporto	52
4.6.1.	Flusso di aggiornamento.....	52
4.6.1.1.	Validazione del processo di aggiornamento	54
4.6.2.	Flusso di disiscrizione	54
4.6.2.1.	Validazione del flusso di disiscrizione	56

4.6.3. Stampa degli utenti.....	57
4.6.3.1. Validazione del flusso di stampa	58
4.7. Riepilogo della struttura delle API	59
5. RNMA all'opera: validazione dei pattern per la resilienza e per l'osservabilità	61
5.1. Introduzione	61
5.2. Circuit Breaker in azione	61
5.3. Time Limiter in azione	63
5.4. Rate Limiter in azione	64
5.5. Aggregazione dei log.....	65
Sviluppi futuri per l'applicazione	67
Docker	67
Kubernetes.....	68
Indice delle figure	69
Bibliografia.....	71

Introduzione

1. Introduzione al problema affrontato

Internet fu uno dei principali fattori che portò architetti e sviluppatori a ridisegnare le strategie con cui un'applicazione può essere progettata e distribuita. Con lo sviluppo della rete, infatti, i sistemi software iniziarono a mutare la propria conformazione, passando da massicci mainframe centralizzati ai più attuali container, ovvero, componenti applicativi distribuiti su diversi nodi della rete Internet.

A cominciare dagli anni '70 la diffusione dello standard TCP/IP permise di collegare sempre più reti locali tra loro e a partire dagli anni '80, con la nascita del protocollo HTTP, si arrivò allo sviluppo del Network Computing.

Il Network Computing consiste nell'utilizzo della rete per fare comunicare tra loro client e server, tramite chiamate a procedura remota, che consentono di interrogare e di richiedere risorse ad un server.

Dagli anni duemila, come conseguenza al Cloud Computing, nacque la tendenza di frammentare i sistemi software in più componenti applicativi dotati di vita propria, ma in costante contatto e interconnessione tra loro. Questa tendenza permise agli sviluppatori di semplificare il rilascio e la distribuzione dei sistemi, sfruttando anche tecnologie che ne permettono la containerizzazione.

Oggi un utente è in grado di usufruire dei servizi in rete, interrogando le applicazioni tramite le API che esse espongono, sul modello REST HTTP.

In questo contesto nascono i micro-servizi, un pattern architetturale, che pone nuove linee guida per lo sviluppo di applicazioni distribuite.

Insieme ad essi, nascono anche i pattern per la resilienza, che permettono la protezione di tali sistemi dai malfunzionamenti e dagli errori introdotti dalla comunicazione in rete.

2. Obiettivo della tesi

Il presente lavoro di tesi ha l'intento di realizzare un prototipo di sistema software basato su micro-servizi, che sia in grado di funzionare in modo affidabile e continuativo in presenza di guasti e di malfunzionamenti, grazie ai pattern per la resilienza; e di cui si possa monitorare lo stato @Runtime, grazie ai pattern per l'osservabilità.

L'implementazione di tecniche per la resilienza richiede che gli architetti del software definiscano quali pattern debbano essere utilizzati per reagire agli eventi che possono modificare la stabilità di un sistema, in modo tale da innescare l'azione di controllo più adeguata per ripristinare le condizioni che permettono al sistema di erogare correttamente i propri servizi.

All'interno dell'elaborato verrà presentato il prototipo RNMA¹ : un sistema per la gestione degli utenti iscritti ad un servizio di newsletter. All'interno dell'applicazione, verranno sfruttate le potenzialità del framework Spring Cloud per renderla resiliente.

Tale sistema, potrà essere utilizzato per inviare regolarmente notizie, articoli e promozioni al gruppo di persone che abbia scelto di riceverle. Dato che, tali comunicazioni avvengono principalmente via email, è importante avere un sistema che possa verificare l'attualità dell'indirizzo fornito dall'utente in fase di registrazione. A tale scopo, il "sistema di verifica email" dovrà inviare all'indirizzo specificato un link di conferma, grazie al quale, l'utente cliccando, potrà confermare la propria identità.

L'obiettivo dell'elaborato è sviluppare un componente con queste funzionalità, che riesca a garantire qualità come robustezza, fast recovery e osservabilità.

¹ Alias di "Resilient Newsletter Microservice Application". Codice sorgente e artefatti sono disponibili al seguente indirizzo: <https://github.com/AndreaIngolia/RNMA-Resilient-Newsletter-Microservice-Application>

3. Presentazione dei capitoli

Il primo capitolo dopo questa introduzione descrive dal punto di vista infrastrutturale i limiti di un'architettura a micro-servizi e ne mette in evidenza le fragilità. Introdotta le problematiche più comuni, verranno enunciati alcuni concetti, facenti parte della teoria dell'affidabilità dei sistemi, elaborata dallo studioso Jean-Claude Laprie [1].

Il secondo capitolo presenta tutte le tecnologie utilizzate per lo sviluppo del prototipo e il cosiddetto “modello esagonale”, un pattern progettuale che definisce le linee guida da adottare quando si vuole sviluppare un micro-servizio.

Il terzo capitolo esamina come possono essere implementati i pattern e i meccanismi per la resilienza, utilizzando il framework Spring Cloud. In questo capitolo vengono anche descritti i sistemi che possono essere utilizzati per monitorare lo stato e il comportamento del sistema in tempo reale.

Nei capitoli quattro e cinque verrà presentato il prototipo, il cui acronimo è RNMA, un sistema di gestione e registrazione degli utenti, che desiderano iscriversi ad una newsletter. La descrizione dei flussi è supportata da diagrammi di sequenza UML e per ciascuna API verrà mostrato come è possibile valorizzare le richieste su Postman.

La tesi si conclude discutendo gli sviluppi futuri che potrebbero essere adottati per il prototipo, discutendo di tecnologie come Docker e Kubernetes.

1. La resilienza nelle applicazioni a servizi

1.1. Architettura a micro-servizi

Il termine “micro-servizi” [2] fu coniato dagli informatici Martin Fowler e James Lewis nel 2014 quando pubblicarono il saggio intitolato *Microservices*. I micro-servizi si configurano come un approccio architetturale per lo sviluppo di applicazioni, che grazie a questo pattern, vengono composte da più servizi dislocati in rete, che comunicano tra loro per raggiungere obiettivi computazionali comuni, tramite API. Nonostante il sistema finale sia fortemente decentralizzato, esso appare agli utenti come unico, coeso e coerente.

Ciascun componente dell’architettura modella logiche di business ben definite. Inoltre, ciascun micro-servizio deve essere dotato di tutti gli elementi, hardware e software, che permettono di renderlo il più auto-sufficiente possibile, come ad esempio, una propria banca dati.

L’indipendenza dei micro-servizi comporta il vantaggio che ciascuno di essi può essere implementato con uno stack tecnologico indipendente dagli altri, purché sia esso in grado di comunicare con l’esterno usando dei protocolli di comunicazione comuni a tutti i servizi che compongono il sistema.

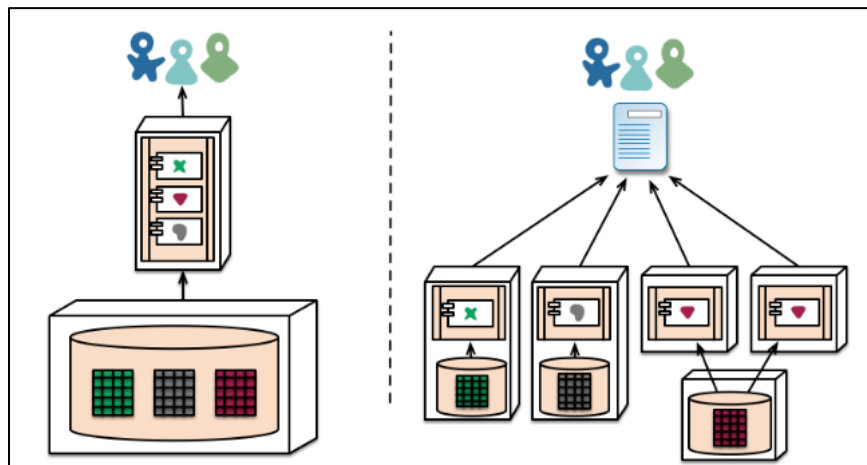


Figura 1.1 Confronto tra architettura monolitica e architettura a micro-servizi

1.2. Caratteristiche e problematiche di un sistema a micro-servizi

1.2.1. Scalabilità e distribuzione

A differenza delle applicazioni monolitiche, che vengono rilasciate come un unico artefatto eseguito su una singola macchina, ciascun micro-servizio può essere distribuito in modo indipendente dagli altri, spesso ricorrendo alla tecnica della containerizzazione, appoggiandosi a tecnologie come Docker.

La scalabilità orizzontale [2] è una tecnica, che permette di aumentare la capacità di elaborazione e la tolleranza ai guasti di un sistema. L'obiettivo viene raggiunto mediante la replicazione delle istanze dei micro-servizi, evitando così di fare affidamento su un singolo nodo di rete.

La replicazione e la distribuzione di più istanze dello stesso micro-servizio permettono di aumentare la disponibilità generale del sistema, poiché si accresce la probabilità che esista almeno un'istanza attiva per ogni micro-servizio in grado di soddisfare le richieste in arrivo. Inoltre, poiché questa configurazione permette di bilanciare il carico di richieste in arrivo tra le istanze attive di un micro-servizio, si riduce la possibilità che si verifichi la degradazione centralizzata delle prestazioni.

1.2.2. Guasti parziali

La configurazione distribuita di un'architettura a micro-servizi introduce la necessità di dovere gestire il verificarsi di guasti parziali. Per guasto parziale [2], si intende un malfunzionamento solo di un sotto-insieme dei componenti del sistema. Questa condizione può gravemente compromettere le prestazioni, causando - per esempio - un aumento dei tempi di risposta, o nel peggiore dei casi, portare all'indisponibilità totale del sistema, a causa di malfunzionamenti propagati a cascata.

1.2.3. Granularità di un micro-servizio e overhead di comunicazione

Lo stato dell'arte [2] prevede che ciascun servizio del sistema debba implementare una quantità limitata e coerente di funzionalità. Tuttavia, un servizio con un livello di granularità troppo fine, potrebbe richiedere una quantità eccessiva di cooperazione con altri componenti del sistema per terminare con successo un flusso di lavoro completo. Uno dei fattori principali che influisce negativamente sulle prestazioni di un'architettura a micro-servizi è proprio la latenza dovuta alla comunicazione in rete.

Per evitare un overhead di comunicazione in rete eccessivo, i micro-servizi coinvolti in questi flussi devono essere raggruppati, per creare un componente a dimensione maggiore, che sia auto-sufficiente, così da diminuire il traffico di rete.

1.2.4. Sincronismo delle comunicazioni

Un altro fattore che incide sulla degradazione delle prestazioni è il tipo di protocollo utilizzato per la comunicazione tra i componenti. Uno dei protocolli di rete più diffusi è HTTP, la cui caratteristica principale è quella di essere di tipo sincrono.

Pertanto, il suo svantaggio [2] principale risiede nel tempo d'attesa o *blocking time* che un client deve affrontare quando aspetta una risposta dal backend del sistema a cui ha inviato la richiesta. Il mittente, quindi si blocca e rimane in attesa di una risposta da parte del server.

Uno dei meccanismi più utilizzati per la risoluzione del problema è l'adozione di comunicazioni asincrone qualora i sistemi non siano in condizione di offrire una risposta in tempi adeguati.

Il protocollo HTTP prevede l'utilizzo di verbi per identificare il tipo di operazione CRUD, che può essere applicata alle risorse del sistema, come ad esempio le tuple di un database relazionale, secondo il seguente schema:

1. GET recupera una risorsa o un insieme di risorse.
2. POST crea una nuova risorsa.
3. PUT modifica una risorsa già esistente.
4. DELETE rimuove una risorsa.

1.3. Definizione di resilienza

La resilienza [1] è un concetto chiave nella progettazione e nella gestione dei sistemi informatici. Secondo la definizione dell'informatico Jean-Claude Laprie, la resilienza è la capacità di un sistema di mantenere o ripristinare un livello accettabile di prestazioni in seguito a una perturbazione. Laprie, inoltre, classifica e descrive le perturbazioni in base alla loro natura e in base all'effetto che hanno sullo stato del sistema.

Pertanto, la resilienza si riferisce alla capacità di un sistema di continuare a funzionare anche in caso di malfunzionamenti. La gestione dei guasti consiste nell'avere processi in atto alla loro individuazione e al ripristino di uno stato corretto di funzionamento.

1.4. Stato di un sistema

Secondo la definizione di Jean-Claude Laprie [1], lo stato di un sistema è l'insieme delle proprietà fisiche, logiche e funzionali che descrivono la configurazione del sistema in un dato momento. Lo stato di un sistema rappresenta il suo "stato attuale" in termini di dati, configurazioni, parametri e altre proprietà che determinano come il sistema sta funzionando. L'insieme delle combinazioni dei valori che gli attributi assumono definisce *lo spazio degli stati*.

1.4.1. Classificazione degli stati di un sistema

Nella teoria della resilienza di Jean-Claude Laprie [1], lo spazio degli stati è un concetto chiave e rappresenta l'insieme di tutti gli stati possibili in cui un sistema può trovarsi.

Lo spazio degli stati si divide in tre macro-insiemi disgiunti, classificati come:

1. L'insieme degli stati accettabili. Esso, nella teoria della resilienza di Laprie rappresenta l'insieme di tutti gli stati che rispettano criteri di accettabilità ottimali. Gli stati accettabili garantiscono che il sistema sia in grado di svolgere la propria funzione in modo sicuro e affidabile, anche in presenza di eventi di disturbo.

L'insieme degli stati accettabili include anche gli stati di "degrado controllato", in cui il sistema è in grado di continuare a svolgere la propria funzione con un grado di qualità minore.

2. Spazio di sopravvivenza. Il *survival space* è il sottoinsieme dello spazio degli stati da cui il sistema è in grado di ripristinare le proprie funzionalità dopo un evento di disturbo. Il *survival space* è un concetto chiave nella teoria della resilienza di Laprie, poiché consente di analizzare la capacità di un sistema di riprendersi da un evento di disturbo e di identificare le azioni di controllo necessarie al recupero della stabilità.
3. Lo spazio degli stati morti. È l'insieme degli stati da cui non si è in grado di ripristinare uno stato accettabile. Gli stati morti possono essere causati da eventi di disturbo che danneggiano irreparabilmente il sistema o da una mancata attuazione di azioni di controllo. All'interno di questo spazio nessuna azione di controllo è in grado di riportare il sistema ad una condizione di stabilità.

1.5. Classificazione della resilienza

Il concetto di resilienza può essere classificato ulteriormente in sottocategorie. Esse descrivono in che modo un sistema reagisce ad un cambiamento "evolutivo".

Per cambiamento evolutivo, o *evolutionary change* [1], ci si riferisce ad un cambiamento strutturale del sistema. Questi cambiamenti possono riguardare, per esempio, la modifica della struttura fisica del sistema, come l'aggiunta di nuovi componenti e la rimozione di componenti esistenti, o l'aggiornamento delle tecnologie utilizzate. Essi possono anche essere involontari ed essere scatenati da perturbazioni non considerate in fase di progettazione.

La classificazione della resilienza [1] rispetto a cambiamenti evolutivi è la seguente:

1. Robustezza. È la capacità del sistema di resistere alle perturbazioni, mantenendo sempre attive le proprie funzionalità, al massimo della qualità.
2. Degradabilità elegante. Questo termine descrive un sistema che in caso di malfunzionamenti è in grado di funzionare con qualità ridotta, ma accettabile rispetto ai criteri definiti in fase di progettazione.

3. Recuperabilità. È la capacità di un sistema di tornare alla propria condizione di equilibrio dopo essere stato perturbato o danneggiato; ovvero passare dalla regione di sopravvivenza, alla regione degli stati stabili.

In conclusione, la resilienza può essere ottenuta attraverso l'implementazione di tecniche specifiche, scelte in funzione del livello di protezione e del grado di continuità del servizio, che si vuole vengano garantiti.

2. Tecnologie di sviluppo utilizzate

2.1. Introduzione

In questo capitolo verranno esplorate le tecnologie che sono state utilizzate per lo sviluppo del prototipo, ponendo un accento particolare sul framework Spring e sulle sue estensioni, come Spring Data JPA e Spring MVC.

Nel capitolo successivo, verranno discussi i pattern per la resilienza, offerti dal framework Spring Cloud, che ne fornisce le implementazioni.

2.2. Tecnologie utilizzate

2.2.1. Java

Java [3] è un linguaggio di programmazione sviluppato da James Gosling presso Sun Microsystems, nel 1995. Uno dei tratti caratteristici del linguaggio è la sua portabilità, poiché ha la capacità di essere eseguito su qualsiasi macchina, nella quale cui è installato il JDK, *Java Development Kit*, indipendentemente dal sistema operativo. Java è un linguaggio di programmazione fortemente tipizzato ad oggetti, il che significa che fa utilizzo delle classi per organizzare e strutturare il codice.



Figura 2.1 Logo di Java

2.2.2.IntelliJ IDEA

IntelliJ IDEA [4] è un IDE, ambiente di sviluppo integrato, sviluppato da JetBrains e progettato per essere compatibile con un elevato numero di versioni del linguaggio di programmazione Java.

IntelliJ offre una serie di caratteristiche per aiutare gli sviluppatori a scrivere, eseguire e debuggare codice in modo efficiente; alcune delle sue funzionalità sono:

1. Supporto per la sintassi di Java, inclusi i “code hints” e la segnalazione in tempo reale della presenza di errori di compilazione.
2. Integrazione con sistemi di controllo di versione, come Git, consentendo di utilizzarne i comandi direttamente all’interno dell’ambiente di sviluppo.
3. Strumenti per la navigazione e per il refactoring del codice.
4. Supporto per la creazione di progetti con i principali dei framework sul mercato, come Spring.



Figura 2.2 Logo di IntelliJ

2.2.3.GIT

Git [5] è un sistema di controllo di versione open-source, che consente agli sviluppatori di tenere traccia delle modifiche al codice sorgente.



Figura 2.3 Logo di Git

2.2.4.GitHub

GitHub [6] è una piattaforma, all'interno della quale gli sviluppatori possono pubblicare i propri progetti in uno spazio di lavoro condiviso. Esso permette agli sviluppatori di collaborare tra loro in modo efficiente, poiché espone funzionalità come: la possibilità di creare branch e pull request e la possibilità di tenere traccia dei bug, dei lavori in corso e degli sviluppi futuri, con il pannello di issue tracking. GitHub è diventato uno dei luoghi più popolari per la collaborazione e lo sviluppo di software open-source e, più recentemente, nel 2018 è stato acquisito da Microsoft.



Figura 2.4 Logo di GitHub

2.2.5.Database H2

H2 [7] è un database relazionale open-source scritto in Java. È un database in-memory, il che significa che i dati vengono salvati nella memoria RAM del dispositivo anziché su disco; pertanto, l'accesso ai dati, in lettura o in scrittura è più rapido, rispetto a quanto offerto da database che si appoggiano a strumenti di memoria persistente. Per questo motivo è particolarmente utilizzato durante le fasi di sviluppo e testing.

2.2.6.Postman

Postman [8] è una popolare applicazione che consente di testare le API, *Application Programming Interface*, per mezzo di chiamate HTTP, che possono essere create e inviate direttamente dall'applicazione.



Figura 2.5 Logo di Postman

2.2.7.Spring

Spring [10] è un framework open-source, che permette di semplificare aspetti legati alla configurazione e allo sviluppo di applicazioni Java. Il fattore innovativo di Spring comprende diversi nuclei tematici, tra cui, le modalità con le quali vengono gestite le dipendenze di un'applicazione, il modo in cui viene gestita la comunicazione con un DBMS e la possibilità di creare servizi web, con uno sforzo di configurazione minimo.

Il cuore del framework è rappresentato da *Spring Core*: un modulo mandatorio, importato di default. Esso fornisce le funzionalità di base del framework, come la gestione dei Bean e l'iniezione delle dipendenze; tecnica attraverso la quale Spring implementa il design pattern *Inversion of Control*.

Il Core può essere esteso da vari moduli, che consentono di introdurre funzionalità aggiuntive ad un progetto; alcuni di esse, sono:

1. Spring Web: fornisce le funzionalità per la creazione di applicazioni web con protocollo HTTP.
2. Spring Data JPA: permette di fare comunicare un'applicazione con un'ampia gamma di DBMS, relazionali e non relazionali.
3. Spring Cloud: fornisce le funzionalità per l'implementazione dei componenti fondamentali di un'architettura a micro-servizi, come ad esempio l'API gateway, il Load Balancer e i pattern per la resilienza.
4. Java Mail. Questo modulo permette di inviare messaggi di posta elettronica da applicazione, servendosi di server di posta elettronica di fornitori esterni.



Figura 2.6 Logo di Spring

2.2.8.Maven

Maven [9] è uno strumento per gestione delle dipendenze, cioè le librerie esterne che devono essere importate per essere utilizzate da un progetto. Maven utilizza un file in formato XML, detto POM, *Project Object Model*, all'interno del quale è possibile specificare il nome dell'artefatto e la versione da importare. In fase di compilazione, Maven scarica dai repository centrali tutte le dipendenze definite nel POM e le memorizza sulla macchina su cui l'applicazione è in esecuzione.



Figura 2.7 Logo di Maven

2.2.9.Spring Data JPA

Spring Data JPA [11] è un framework, che permette ad un'applicazione di accedere a dati relazionali tramite il framework JPA, *Java Persistence API*.

JPA è un *Object-Relational Mapping*, ovvero uno strumento che consente di lavorare con dati relazionali come se fossero oggetti Java.

Spring Data JPA rende immediata la creazione di tabelle, per mezzo dell'interfaccia `@Repository`. Le entità gestite da ciascuna repository sono annotate con `@Entity`.

A titolo esemplificativo si riporta un esempio ricavata dal prototipo RNMA:

```
@Entity
public class MemberEntity {
    @Id
    @Column (name = "EMAIL", unique = true, nullable = false)
    private String email;
    @Column (name = "NAME", nullable = false)
    private String name;
    @Column (name = "SURNAME", nullable = false)
```

```

    private String surname;
    @Column (name = "CONFIRMED_IDENTITY")
    private String isIdentityConfirmed = "F";
}

```

La seguente classe rappresenta un'entità di tipo “MemberEntity”, caratterizzata dalla chiave primaria “email” di tipo stringa e da altri campi, come “name” e “surname”.

Il campo “isIdentityConfirmed” è un campo ufficiosamente booleano. Esso può contenere i soli valori stringa F e T. In fase di progettazione, è stato deciso di rappresentare il campo, come una stringa, poiché non tutti i DBMS supportano nativamente il tipo booleano, come ad esempio il DBMS di Oracle.

Per la memorizzazione delle entità precedenti è sufficiente definire la seguente interfaccia:

```

public interface MemberRepository
    extends JpaRepository <MemberEntity, String> {}

```

La tabella prende il nome di “MemberRepository” e permette di memorizzare oggetti di tipo “MemberEntity”, che hanno chiave primaria di tipo stringa.

Spring Data JPA implementa i metodi CRUD: *create*, *read*, *update*, *delete* per ciascun tipo di entità.

2.2.10. Spring MVC

Spring MVC [12] è un framework ottimizzato per lo sviluppo di applicazioni web, con il pattern Model View Controller. Esso fornisce i meccanismi, che rendono un sistema in grado di elaborare richieste HTTP in arrivo dalla rete.

Quando una chiamata arriva alle porte di un micro-servizio, il sistema utilizza un oggetto di tipo DispatcherController, per raccogliere tali richieste e per indirizzarle al Controller interno più appropriato, da cui partirà l'elaborazione vera e propria.

Un Controller è caratterizzato da una collezione di API, che possono essere annotate con: `@GetMapping`, `@PostMapping`, `@PutMapping` o `@DeleteMapping`, in funzione del verbo HTTP associato alla chiamata.

Il `DispatcherController` agisce come il punto d'ingresso di un servizio, prima che le richieste vengano computate. L'operazione di instradamento verso il Controller interno viene portata a termine sulla base della URI e del verbo HTTP della richiesta. Al termine dell'elaborazione, il controller restituisce all'utente una risposta, tipicamente in formato JSON.

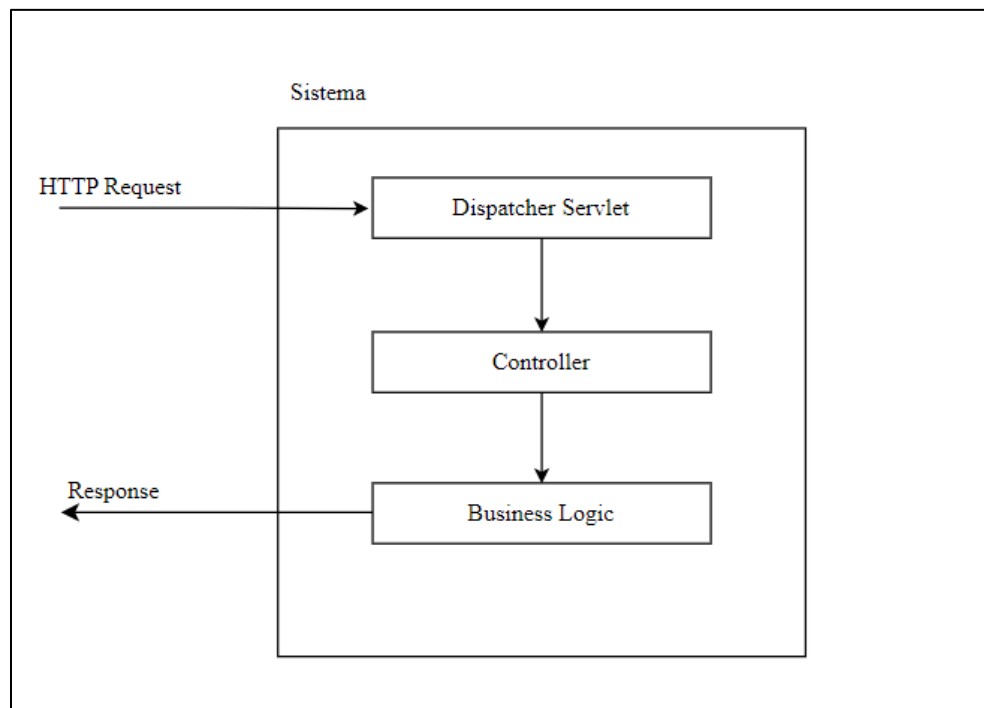


Figura 2.8 Funzionamento del front controller

2.3. Il modello esagonale

Il modello esagonale [14] è un pattern teorizzato da Alistair Cockburn, un noto esperto di ingegneria del software, scrittore e padre della metodologia Agile. Esso definisce i principi e le linee guida che devono essere applicate per l'implementazione di un micro-servizio.

Il modello esagonale risolve il problema dell'infiltrazione di codice della logica di business verso gli strati esterni dell'applicazione.

L'esagono ha lo scopo di evidenziare visivamente:

1. L'allontanamento dall'immagine unidimensionale a strati con cui tipicamente viene rappresentato un sistema software.
2. La presenza di un numero finito di porte, che vengono utilizzate per fare comunicare un servizio con l'esterno.

La logica di business rappresenta il nucleo dell'applicazione, attorno a cui vengono integrati tutti i componenti che dovranno interagirvi.

L'interazione tra le parti deve avvenire unicamente attraverso interfacce, dette porte. Per ciascuna porta è possibile prevedere la presenza di adattatori, che consentono di specificare con ulteriore dettaglio i protocolli a cui i componenti esterni devono adeguarsi per potersi connettere al micro-servizio. Per via di questa struttura, il modello esagonale è anche detto *architettura a porte e adattatori*.

Il modello esagonale permette di migliorare la modularità di un micro-servizio, garantendo una maggiore flessibilità di sviluppo e di manutenzione, poiché ciascun componente può essere sviluppato e testato indipendentemente dagli altri. L'alterazione di un componente non deve impattare il funzionamento del resto del sistema.

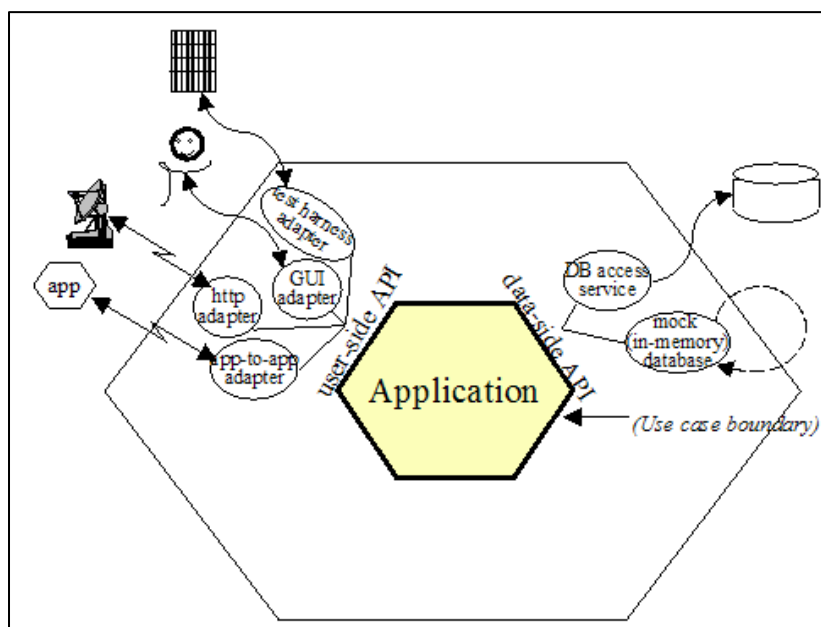


Figura 2.9 Rappresentazione concettuale del modello esagonale di Alistair Cockburn

3. Pattern e meccanismi in Spring per la resilienza

3.1. Introduzione

In questo capitolo verrà discusso come utilizzare i pattern per la resilienza per costruire applicazioni robuste e tolleranti agli errori. Le tecniche sono implementate per mezzo della libreria Resilience4j, del framework Spring Cloud.

Come discusso nei precedenti capitoli, le strategie per la resilienza permettono di gestire i malfunzionamenti, che si manifestano durante le chiamate alle API di un servizio, garantendo che un'applicazione preservi il più possibile la propria integrità.

3.2. Panoramica delle tecniche per la resilienza e per l'osservabilità

La resilienza è un aspetto fondamentale per le applicazioni, che operano in ambienti distribuiti e altamente dinamici. In questo paragrafo, verranno esplorate alcune delle tecniche più comuni per migliorare la resilienza delle applicazioni; alcune di esse sono:

1. Progettazione ridondante e Load Balancer [2]. La progettazione ridondante è una strategia utilizzata per garantire un'ampia disponibilità di uno o più servizi di un'applicazione. Essa consiste nel replicare il numero di istanze di un servizio in modo che, in caso di problemi, le richieste possano essere indirizzate alla prima istanza attiva disponibile. Il Load Balancer è il componente, che concretamente provvede allo smistamento delle richieste.
2. Gestione dei guasti parziali [2]. Nel caso in cui la ridondanza non sia sufficiente e si verifichi l'indisponibilità di uno o più servizi, si può ricorrere all'utilizzo di pattern che permettono di non sovraccaricare il sistema con richieste, che a priori è noto non potranno essere eseguite.
3. Timeout. Questo pattern viene utilizzato quando è necessario limitare la durata di una chiamata rivolta ad un'API di un servizio, soprattutto quando noto è che potrebbe impiegare una quantità indefinita di tempo per essere completata. Nel caso di

applicazioni distribuite risulta essere utile quando si cerca di comunicare con componenti di terze parti o con servizi che operano su reti instabili.

4. Monitoraggio del sistema [2]. I sistemi di monitoraggio servono a raccogliere e ad analizzare i dati relativi alla salute del sistema e al suo stato di funzionamento, affinché sia possibile identificarne le anomalie.

3.3. Resilience4j: la libreria per la resilienza

Resilience4j [15] è la libreria di Spring Cloud [13], da cui si ottengono le implementazioni dei pattern per la resilienza per un'applicazione Java.

Le tecniche di resilienza possono essere divise in due categorie:

1. Reattive. Le tecniche di resilienza reattive gestiscono gli errori solo dopo che essi si sono verificati e hanno come obiettivo di minimizzare i loro effetti sull'applicazione e sugli utenti che ne fanno utilizzo. Un pattern d'esempio è il Circuit Breaker.
2. Attive. Le tecniche di resilienza attive, invece, si concentrano sulla prevenzione dei malfunzionamenti attraverso la progettazione di un'architettura robusta e tollerante. Alcune delle tecniche che soddisfano questa proprietà sono: la replicazione dei servizi, il Time Limiter e il Rate Limiter.



Figura 3.1 Logo di Resilience4j

3.3.1. Circuit Breaker e API di fallback

Il Circuit Breaker [2] è un interruttore che viene attivato quando un servizio non può garantire il proprio funzionamento. Il Circuit Breaker interrompe temporaneamente la comunicazione al fine di evitare la propagazione a cascata dell'errore, cioè che l'invio di ulteriori richieste possa esaurire tutte le risorse computazionali del sistema, peggiorando così la situazione.

In un'ottica di basso livello, il Circuit Breaker viene implementato come un contatore che tiene traccia del numero di richieste fallite. Quando il numero di richieste fallite supera una determinata soglia di tolleranza, l'interruttore viene attivato e tutte le future comunicazioni in arrivo verranno abortite di default. Quando il Circuit Breaker è attivato, è possibile utilizzare un'API di fallback per gestire le richieste in arrivo e restituire un messaggio di errore preimpostato. L'API di fallback è un'alternativa di backup, che viene utilizzata per fornire un comportamento di default quando l'API principale non è accessibile o non risponde come previsto.

Dopo un periodo di tempo, detto *timeout*, il Circuit Breaker verifica se il servizio coinvolto è tornato a funzionare correttamente. Se ciò è accaduto, l'interruttore viene aperto e le comunicazioni ripristinate; altrimenti il Circuit Breaker rimane attivato per un altro periodo di tempo, fino al controllo successivo.

3.3.2.Time Limiter

Il Time Limiter [2] è un pattern che gestisce le richieste che impiegano troppo tempo per essere elaborate. Quando il loro tempo di elaborazione supera un intervallo di tempo, detto *timeout*, la richiesta viene abortita e viene restituito un messaggio di errore. L'obiettivo è evitare che il sistema venga bloccato da richieste lente o incapaci di ottenere risposta.

Tale pattern può essere utilizzato per proteggere il sistema da richieste volontariamente lente, come gli attacchi di tipo Denial of Service, impedendo che il sistema venga sottoposto a un carico computazionale eccessivo.

È necessario che il timeout venga impostato in modo appropriato, per evitare di interrompere le richieste legittime, che tuttavia potrebbero impiegare più tempo del previsto per essere elaborate.

3.3.3.Rate Limiter

Il Rate Limiter [2] è un pattern utilizzato per gestire e limitare il carico di richieste che una API può processare in un intervallo di tempo, per evitare che il sistema venga sottoposto a

un carico di lavoro eccessivo. L'obiettivo del Rate Limiter è garantire che un servizio non collassi a fronte di picchi di traffico inaspettatamente elevati.

Ogni volta che una richiesta viene respinta dal Rate Limiter, viene restituito il codice d'errore HTTP 429, *too many requests*.

Con Resilience4j, per implementare tale funzionalità per un'API è sufficiente aggiungere la seguente annotazione sopra la segnatura del metodo da proteggere:

```
@RateLimiter (name = "rateLimiterUserManager")
```

Dove “rateLimiterUserManager” è il nome dell'oggetto istanziato come Time Limiter, che nel caso del prototipo, deve essere configurato per processare un massimo di dieci richieste al minuto:

```
resilience4j.ratelimiter:  
  instances:  
    rateLimiterUserManager:  
      limitForPeriod: 10  
      limitRefreshPeriod: 1m  
      timeoutDuration: 0
```

3.4. Componenti essenziali per un'architettura a micro-servizi

In aggiunta alle tecniche per la resilienza, un'architettura a micro-servizi deve essere munita di alcuni componenti essenziali [2], tra cui l'API Gateway, il registro dei servizi e il Load Balancer; le cui funzionalità sono:

1. API Gateway: è un componente che funge da punto di ingresso per le richieste inviate al sistema. Esso gestisce le richieste in arrivo, le inoltra ai micro-servizi appropriati e restituisce le risposte agli utenti.

2. Load Balancer: è un componente che distribuisce le richieste in arrivo, tra le istanze attive del micro-servizio a cui esse sono rivolte; esso garantisce che nessuna delle istanze venga sovraccaricata eccessivamente.
3. Registro dei servizi. Il registro dei servizi è un componente che tiene traccia degli indirizzi delle istanze disponibili, per consentire ad esse di ritrovarsi e di comunicare tra loro.

3.4.1.API Gateway

L'API gateway in un'architettura a microservizi funge da punto di accesso unico per le richieste in arrivo nel sistema. La principale funzione di un Gateway è il routing delle richieste, ovvero definire a quale micro-servizio indirizzare una richiesta. Questa operazione viene generalmente compiuta utilizzando una strategia di tipo *path-based*, che agisce in funzione dell'URI della richiesta.

Spring Cloud permette di implementare un Gateway con Spring Cloud Gateway, un componente molto flessibile, che oltre a svolgere l'operazione di routing, può essere integrato agilmente con altre unità fondamentali di un'architettura a micro-servizi, come il registro dei servizi e il Load Balancer.

Spring Cloud Gateway [13] si basa su un'architettura interna a *routing e filtro*. Di seguito viene illustrato un caso concreto di utilizzo dei filtri.

3.4.1.1. Utilizzo dei filtri in Spring Cloud Gateway

I filtri [13] in Spring Cloud Gateway sono oggetti, che consentono di eseguire operazioni su una richiesta in ingresso o su una risposta in uscita. Nell'ultimo caso, possono essere utilizzati per gestire i messaggi di errore, che devono essere generati in seguito a malfunzionamenti specifici.

Ad esempio, la seguente configurazione del file di proprietà del Gateway utilizza il filtro denominato *CircuitBreaker*, per incrementare il numero di richieste fallite del Circuit Breaker “userManagerCircuitBreaker”, solo quando il risultato di un'elaborazione sottostante restituisce i codici di errore: 500, 503 o 504.

```

spring:
  cloud:
    gateway:
      routes:
        - id: USER-MANAGER
          uri: lb://USER-MANAGER
          predicates:
            - Path=/members/**
          filters:
            - name: CircuitBreaker
              args:
                name: userManagerCircuitBreaker
                fallbackUri: forward:/fallback
                statusCodes:
                  - 500
                  - 503
                  - 504

```

Grazie ai filtri è quindi possibile gestire i casi di errore che un Circuit Breaker deve tenere in considerazione per modificare il proprio stato, ovvero aprirsi o chiudersi. Inoltre, grazie ad essi è possibile restituire dei messaggi di fallback, adeguati in funzione del tipo di errore intercettato dai micro-servizi sottostanti. In questo caso, il messaggio di errore è generato dall'API di fallback, sita all'interno del Gateway.

3.4.1.2. Integrazione tra Spring Cloud Gateway e il Load Balancer

Spring Cloud Gateway [13] permette di integrarsi con il Load Balancer, per distribuire il traffico delle richieste tra diverse copie di un micro-servizio.

Nel file di configurazione precedente, tale funzionalità viene gestita nel campo “uri”, grazie alla sintassi “lb://nomeDelServizio”, dove la scrittura “lb” indica, che l'operazione di

instradamento deve essere competenza del Load Balancer, che come accennato in precedenza, si occupa di distribuire il traffico in ingresso, per ottenere una distribuzione equa del carico di lavoro. Il Load Balancer di Spring Cloud Gateway distribuisce le richieste utilizzando l'algoritmo Round-Robin.

Le informazioni statiche di indirizzamento dell'istanza selezionata dal Load Balancer vengono recuperate dal registro dei servizi.

3.4.2.Registro dei servizi

Il registro dei servizi del prototipo è stato implementato con Spring Cloud Netflix [13], una libreria di Spring Cloud, che mette a disposizione il componente Eureka Server per l'attività di service discovery.

Il service discovery è un meccanismo che consente ai componenti di un'applicazione distribuita di comunicare tra di loro in un ambiente dinamico, senza la necessità che essi posseggano il numero di porta e il nome dell'host dei servizi a cui vogliono connettersi; quindi, senza specificare informazioni relative alla loro localizzazione statica all'interno della rete. È un pattern molto utilizzato per applicazioni distribuite su cloud, dove i servizi vengono spesso scalati dinamicamente.

Ogni volta che, un servizio diventa indisponibile viene rimosso dal registro, rendendolo irraggiungibile per i micro-servizi che vorranno comunicare con esso. Di default, quando si tenta di comunicare con un micro-servizio che non ha alcuna istanza disponibile viene restituito il codice di errore 503, *service unavailable*.

3.4.2.1. Pannello di visualizzazione

Eureka mette a disposizione un pannello sulla porta 8761, da cui è possibile monitorare le istanze che possono essere messe in comunicazione tra loro.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
EMAIL-SENDER	n/a (2)	(2)	UP (2) - host.docker.internal:email-sender:8083 , host.docker.internal:email-sender:8082
GATEWAY	n/a (1)	(1)	UP (1) - host.docker.internal:gateway:8080
USER-MANAGER	n/a (1)	(1)	UP (1) - host.docker.internal:user-manager:8081

Figura 3.2 Elenco dei servizi del progetto registrati a Eureka

Dal pannello precedente è possibile visualizzare le informazioni statiche di indirizzamento di ciascuna istanza connessa.

3.4.2.2. Tabella dei servizi del prototipo

Nome servizio	Gateway	Eureka server	User manager	Email sender
Host	localhost	localhost	localhost	localhost
Numero di porta	8080	8761	8081	8082, 8083

Figura 3.3 Elenco dei servizi, host e numero di porta del prototipo

È possibile notare, che il micro-servizio “Email Sender” è caratterizzato da due istanze attive sulle porte 8082 e 8083.

Le specifiche di ciascun servizio verranno esplorate nel capitolo successivo.

3.5. Definizione di Osservabilità

Per osservabilità [2] ci si riferisce alla capacità di monitorare tutti gli oggetti che compongono un sistema, al fine di studiarne le prestazioni e determinarne le criticità in tempo reale.

L'osservabilità può concretizzarsi attraverso la raccolta di metriche, come ad esempio tempi di risposta o percentuali di utilizzo della CPU e della memoria; o attraverso la tracciabilità delle richieste per mezzo dei log. Le informazioni raccolte possono essere utilizzate per risolvere tempestivamente i problemi, soprattutto negli ambienti di produzione.

La combinazione di pattern per la resilienza e per l'osservabilità permette di ottenere un sistema a micro-servizi trasparente, che sia in grado di funzionare correttamente in tempo reale, nonostante le situazioni impreviste.

Nei seguenti paragrafi verranno discussi i pattern più comuni per monitorare il comportamento di un sistema.

3.6. Logging

Il logging [2] è utilizzato per memorizzare informazioni in formato testuale relative ai flussi di esecuzione, che un sistema gestisce durante il suo ciclo di vita. I log possono essere registrati su file, in un database, o possono essere inviati a uno o più sistemi remoti per la raccolta centralizzata, in modo da semplificarne l'analisi.

Il pattern più diffuso per l'analisi dei log è quella dell'aggregazione tramite *correlation id*. Un *correlation id* è un identificativo univoco, che viene assegnato ad ogni transazione all'interno di un sistema distribuito. Esso, permette di ricostruire completamente il flusso di esecuzione di una richiesta, correlando tra loro i log generati dai diversi componenti del sistema. Con questo pattern è possibile identificare eventuali punti di blocco e bug nel codice. *Java.util.logging* fu la prima libreria che garantì la possibilità di generare log all'interno di un'applicazione Java. Nel tempo, a partire da essa sono stati sviluppati framework, in grado di garantire una gestione dei log più completa ed efficiente, tra cui Logback, che consente di generare log e di inviarli a una o più destinazioni per il loro monitoraggio, come ad esempio Elasticsearch, Logstash e Kibana.

3.6.1.Sleuth

Sleuth [13] è una libreria open-source di Spring Cloud per la tracciabilità distribuita. Grazie ad essa, è possibile generare gli identificatori (*correlation id*) per la tracciabilità delle richieste che attraversano il sistema, in modo da poter comprendere come i componenti interagiscono tra loro.

3.6.2.Elasticsearch, Logstash, Kibana

Lo stack ELK, acronimo di Elasticsearch [17], Logstash [16], Kibana [18], è un insieme di strumenti open-source per la raccolta, l'elaborazione e la visualizzazione dei log.

Nello specifico:

1. Elasticsearch è un motore di ricerca open source utilizzato per gestire grandi quantità di dati testuali, come i log generati da applicazioni. A differenza dei database tradizionali, esso si concentra sull'indicizzazione e sulla ricerca dei dati per mezzo di ricerche di tipo full-text.
2. Logstash è una pipeline utilizzata per raccogliere i log generati da un'applicazione; esso si occupa di trasformarli in un formato standard e di inviarli a una varietà di destinazioni diverse; come ad esempio Elasticsearch e Kibana.
3. Kibana è un'interfaccia web integrata con Elasticsearch, attraverso la quale è possibile creare dashboard per l'esplorazione dei dati.

3.7. I log su Cloud: integrazione tra Logback ed ELK

In questo paragrafo verrà mostrato come è possibile, che i log generati in un micro-servizio vengano trasferiti e resi disponibili su Cloud. Tale requisito viene raggiunto attraverso l'integrazione di due sistemi: la libreria Logback, la cui dipendenza è iniettata all'interno di ciascun micro-servizio, che deve essere in grado di generare log; e lo stack Elasticsearch, Logstash e Kibana.

La libreria Logback utilizza due classi principali, Logger e Appender.

3.7.1.Logger

Un oggetto di tipo Logger, quando viene utilizzato all'interno della logica di business di un'applicazione, consente di generare dei log. A livello implementativo si presenta come segue:

```
private static final Logger LOG =  
    LoggerFactory.getLogger(MemberController.class);
```

Dove “MemberController” è una classe contenente della logica di business. Nella fattispecie, un log viene generato come segue:

```
LOG.info("The system is receiving too many requests, the rate  
limiter tripped.")
```

3.7.2.Appender

Gli Appender rappresentano il meccanismo attraverso cui i log vengono trasmessi a punti di raccolta esterni.

Essi sono configurati in file XML, uno per micro-servizio; la loro configurazione basa il proprio funzionamento su due tag principali: “appender” e “destination”:

```
<appender name="logstash" class=  
"net.logstash.logback.appender.LogstashTcpSocketAppender">  
  
<destination>localhost:5000</destination>
```

Essi permettono di creare un appender dal nome “logstash”, che invierà i log dell’applicazione alla destinazione definita nell’attributo “class”, tramite protocollo TCP, attraverso la porta 5000.

Definito il primo fronte di comunicazione, è necessario che anche il servizio Logstash venga configurato per poter ricevere tali informazioni. Il suo file di configurazione è il seguente:

```
input {  
  tcp {  
    port => "5000"  
    type => syslog  
    codec => json_lines  
  }  
}  
  
output {  
  elasticsearch {  
    hosts => ["http://elasticsearch:9200"]  
    index => "newsletter"  
  }  
}
```



Figura 3.4 Logo di Logstash

Il file è suddiviso in due blocchi, denominati input e output. L’input configura Logstash per ricevere i dati inviati da un’applicazione, sulla porta 5000 tramite protocollo TCP.

A sua volta, tramite il plugin di output, Logstash potrà inoltrare le informazioni ricevute verso ulteriori destinatari. In questo caso il destinatario è unico ed è Elasticsearch, connesso alla

porta 9200. Prima di essere inviate, le informazioni vengono raggruppate in un indice, il cui nome è "newsletter". Un indice in Logstash raggruppa logicamente un insieme di variabili testuali.



Figura 3.5 Logo di Elasticsearch

3.7.3. Configurazione di Kibana

Kibana è un servizio containerizzato, così come Logstash ed Elasticsearch. La sua configurazione in Docker è la seguente:

```
kibana:
  image: kibana:7.14.1
  container_name: kibana
  restart: always
  ports:
    - 5601:5601
  environment:
    - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
```

Il portale di Kibana è raggiungibile dalla porta 5601 e fa uno scraping dei dati contenuti in Elasticsearch, attraverso la porta 9200.

Dopo avere configurato il collegamento tra Kibana ed Elasticsearch è possibile utilizzare il sistema di ricerca testuale, per ricostruire il flusso di una transazione, fornendo il *correlation id* come chiave per l'aggregazione dei dati.



Figura 3.6 Logo di Kibana

3.8. Health Check

Il pattern Health Check [2] è comunemente utilizzato per monitorare la salute di un sistema o di un servizio. Esso consiste nell'esecuzione di controlli periodici, per verificare che tutto il sistema sia operativo e funzioni correttamente. In base ai risultati dei controlli, il sistema viene considerato "healthy" o "unhealthy".

Spring Boot Actuator [13] è la dipendenza di Spring, grazie alla quale è possibile monitorare lo stato di salute di un'applicazione. Tale dipendenza permette ad un servizio di esporre l'endpoint `actuator/prometheus`, attraverso il quale il servizio esterno Prometheus può ricavare le informazioni sul suo stato di salute. I file di proprietà dei servizi da monitorare devono avere la seguente configurazione:

```
management:
  endpoints:
    web:
      exposure:
        include: prometheus
```

3.9. Prometheus

Prometheus [20] è un servizio che permette di generare grafici e reportistica a partire dalle metriche ottenute da un servizio durante la sua esecuzione; alcune di esse sono:

1. Utilizzo delle risorse del sistema: come CPU, memoria e spazio su disco.
2. Prestazioni delle applicazioni: come la latenza delle richieste.
3. Stato di salute dei servizi: “healthy” o “unhealthy”



Figura 3.7 Logo di Prometheus

3.9.1. Configurazione di Prometheus

Affinché un micro-servizio sia monitorabile, è necessario che il server Prometheus abbia nel proprio file di proprietà le “configurazioni di scraping”, che gli consentono di recuperare le metriche di un servizio attraverso l’endpoint esposto precedentemente:

```
global:
  scrape_interval: 5s

- job_name: 'API-GATEWAY-MONITOR'
  metrics_path: '/actuator/prometheus'
  scrape_interval: 5s
  static_configs:
    - targets: ['localhost:8080']
```

- job_name: 'USER-MANAGER-MONITOR'
metrics_path: '/actuator/prometheus'
scrape_interval: 5s
static_configs:
 - targets: ['localhost:8081']

- job_name: 'EMAIL-SENDER-MONITOR'
metrics_path: '/actuator/prometheus'
scrape_interval: 5s
static_configs:
 - targets: ['localhost:8082', 'localhost:8083']

- job_name: 'EUREKA-REGISTRY-MONITOR'
metrics_path: '/actuator/prometheus'
scrape_interval: 5s
static_configs:
 - targets: ['localhost:8761']

3.9.2. Monitoraggio dello stato di salute di un servizio

Nel caso in cui tutti i servizi dell'architettura del prototipo siano sani e disponibili, verrà presentata la seguente videata:

Targets		
<div>All Unhealthy Collapse All</div> <div>Filter by endpoint or labels</div>		
API-GATEWAY-MONITOR (1/1 up) show less		
Endpoint	State	Labels
http://localhost:8080/actuator/prometheus	UP	instance="localhost:8080" job="API-GATEWAY-MONITOR"
EMAIL-SENDER-MONITOR (2/2 up) show less		
Endpoint	State	Labels
http://localhost:8082/actuator/prometheus	UP	instance="localhost:8082" job="EMAIL-SENDER-MONITOR"
http://localhost:8083/actuator/prometheus	UP	instance="localhost:8083" job="EMAIL-SENDER-MONITOR"
EUREKA-REGISTRY-MONITOR (1/1 up) show less		
Endpoint	State	Labels
http://localhost:8761/actuator/prometheus	UP	instance="localhost:8761" job="EUREKA-REGISTRY-MONITOR"
USER-MANAGER-MONITOR (1/1 up) show less		
Endpoint	State	Labels
http://localhost:8081/actuator/prometheus	UP	instance="localhost:8081" job="USER-MANAGER-MONITOR"

Figura 3.8 Stato di salute dei servizi del prototipo

Altrimenti, in caso di indisponibilità di qualche servizio, lo stato di salute verrà modificato e la perturbazione sarà visibile anche dal pannello di controllo di Prometheus:

API-GATEWAY-MONITOR (0/1 up) show less		
Endpoint	State	Labels
http://localhost:8080/actuator/prometheus	DOWN	instance="localhost:8080" job="API-GATEWAY-MONITOR"

Figura 3.9 Servizio con stato di salute "unhealthy"

3.9.3. Generazione di grafici per il monitoraggio delle prestazioni

1. Percentuale di utilizzo della CPU del processo “User Manager”

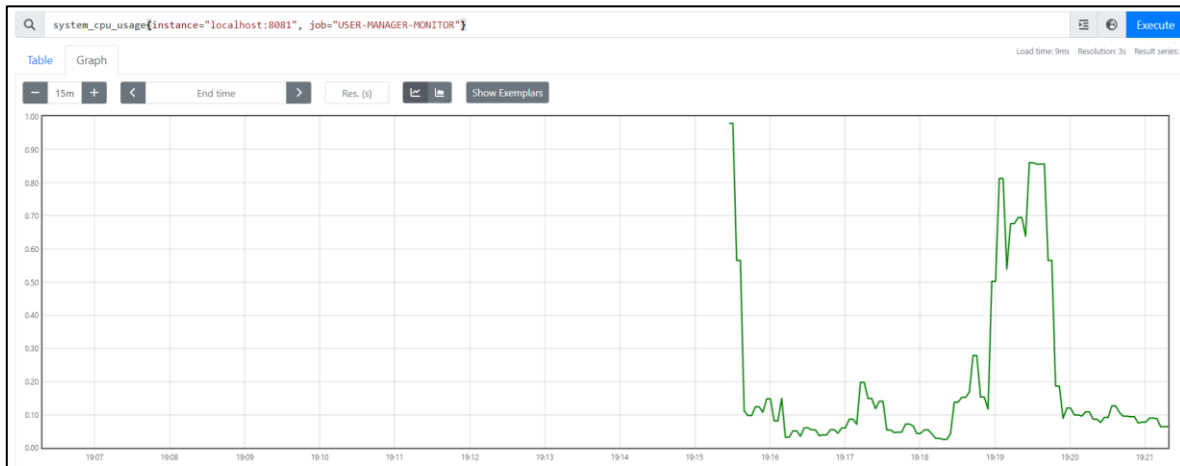


Figura 3.10 Utilizzo della CPU di “User Manager”

2. Percentuale di utilizzo della CPU del processo “Email Sender”

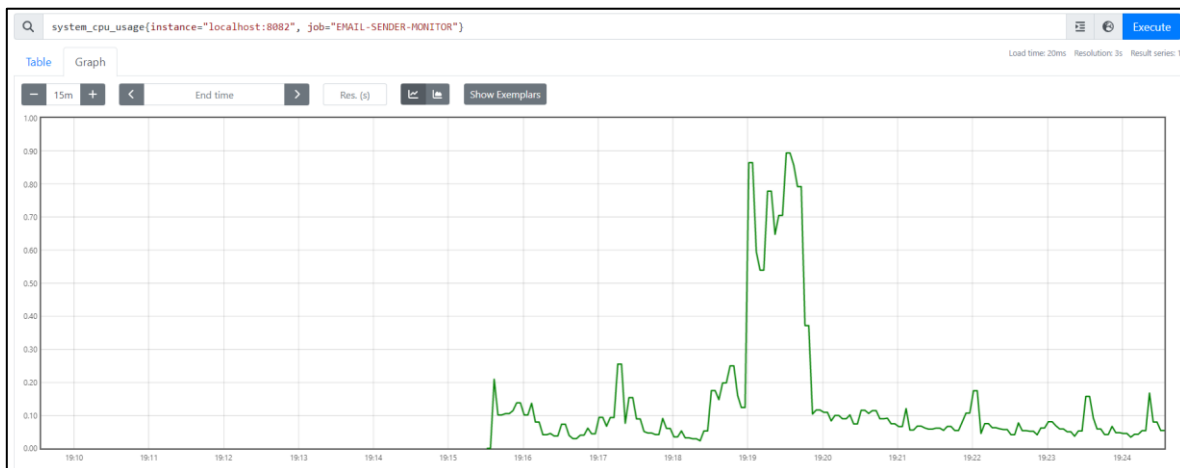


Figura 3.10 Utilizzo della CPU di “Email Sender”

4. RNMA: Resilient Newsletter Microservice Application

4.1. Introduzione

In questo capitolo verranno presentati i requisiti e i flussi del progetto: un sistema distribuito, che si basa su architettura a micro-servizi, all'interno del quale sarà possibile integrare i pattern per la resilienza e per l'osservabilità, con l'obiettivo di creare un'applicazione che sia in grado di gestire eventuali malfunzionamenti e di garantire la possibilità di monitorare il suo stato interno. In questo capitolo verrà descritta in dettaglio l'organizzazione architetturale del sistema.

4.2. Idea di progetto

Una newsletter è un mezzo di comunicazione utilizzato per inviare informazioni periodiche, come notizie, articoli o promozioni, a un gruppo di persone che si sono iscritte per riceverle. Le newsletter si basano tipicamente su comunicazioni via email, pertanto si rende necessaria la presenza di un sistema, che sia in grado di verificare che un indirizzo di posta sia attivo e che sia utilizzato dall'utente che ha fatto richiesta di registrazione. Tale componente è chiamato *sistema di verifica mail*.

Esso in fase di registrazione invia automaticamente una email all'indirizzo fornito dall'utente, con un link di verifica. L'utente accedendo alla propria casella di posta elettronica, potrà fare clic sul link e confermare di essere il proprietario dell'indirizzo fornito al sistema.

L'obiettivo del progetto è la creazione di un componente con queste funzionalità, che verrà reso resiliente sfruttando le potenzialità di Spring Cloud.

4.3. Architettura del sistema

Il cuore sistema è costituito da due micro-servizi: il micro-servizio degli utenti, anche detto “User Manager” e il micro-servizio per l'invio delle email, chiamato “Email Sender”.

“User Manager” fornisce le API per gestire i flussi di registrazione, di aggiornamento e di cancellazione degli utenti, mentre “Email Sender” si occupa dell'invio della mail di verifica appoggiandosi al servizio di posta elettronica Gmail.

Affinché “User Manager” possa innescare l'invio di una mail di conferma per mezzo del micro-servizio “Email Sender”, del quale sono state realizzate più istanze, è necessario implementare il registro dei servizi Eureka Server, per il discovery dinamico.

All'interno dell'architettura sono stati inclusi un sistema per la gestione dei log e un sistema per il monitoraggio delle metriche.

Il sistema di gestione dei log è costituito dai servizi Elasticsearch, Logstash e Kibana. Esso consente di raccogliere e visualizzare i log prodotti dai vari servizi. Il sistema di monitoraggio, Prometheus, offre la possibilità di monitorare le metriche e lo stato di salute dei servizi.

L'integrazione dei due sistemi precedenti con l'applicazione è necessaria per garantire una maggiore osservabilità dell'applicazione e per la rilevazione tempestiva di eventuali problemi.

Per richiedere un servizio all'applicazione, l'utente deve farne richiesta rivolgendosi al gateway dell'architettura.

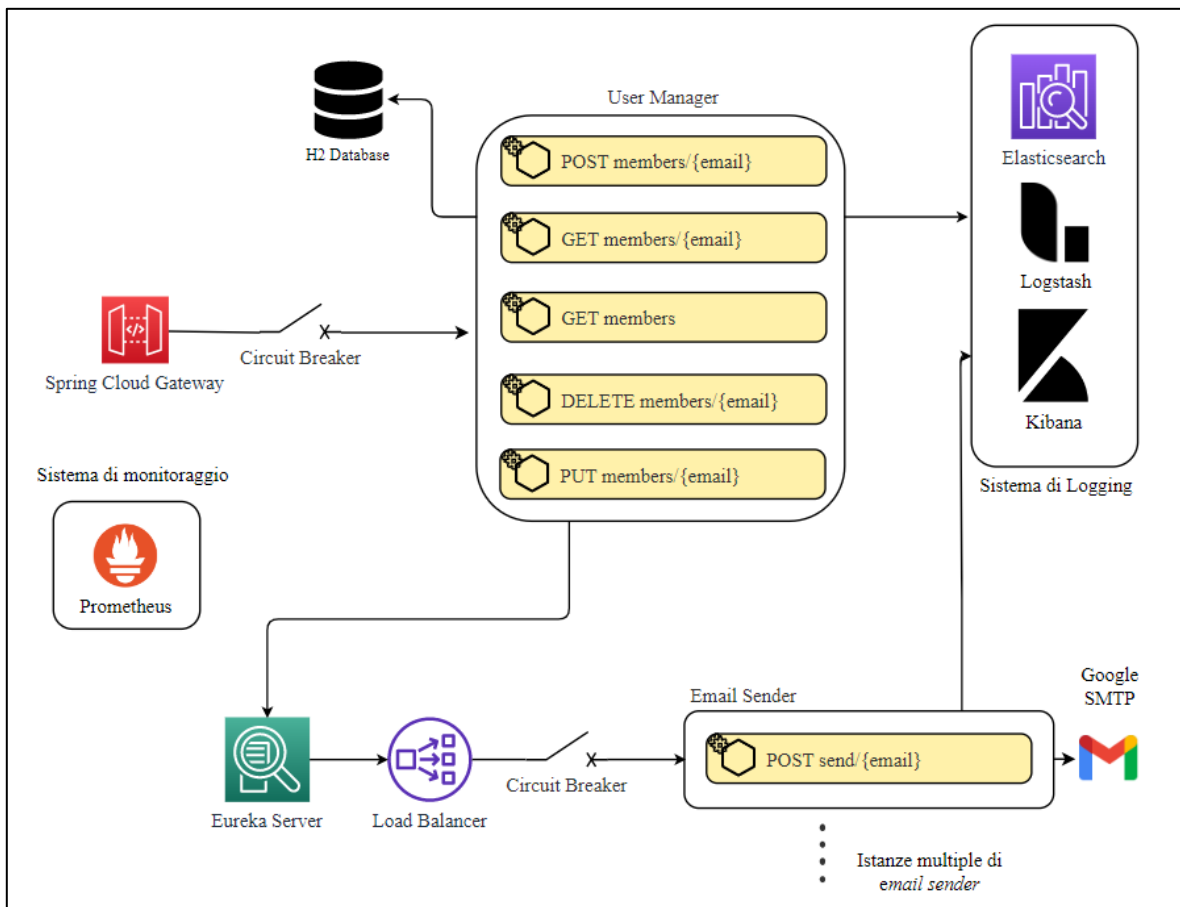


Figura 4.1 Diagramma a componenti dell'architettura del sistema

Di seguito il diagramma dei casi d'uso, in cui vengono mostrate schematicamente le operazioni che possono essere effettuate dagli utilizzatori del sistema:

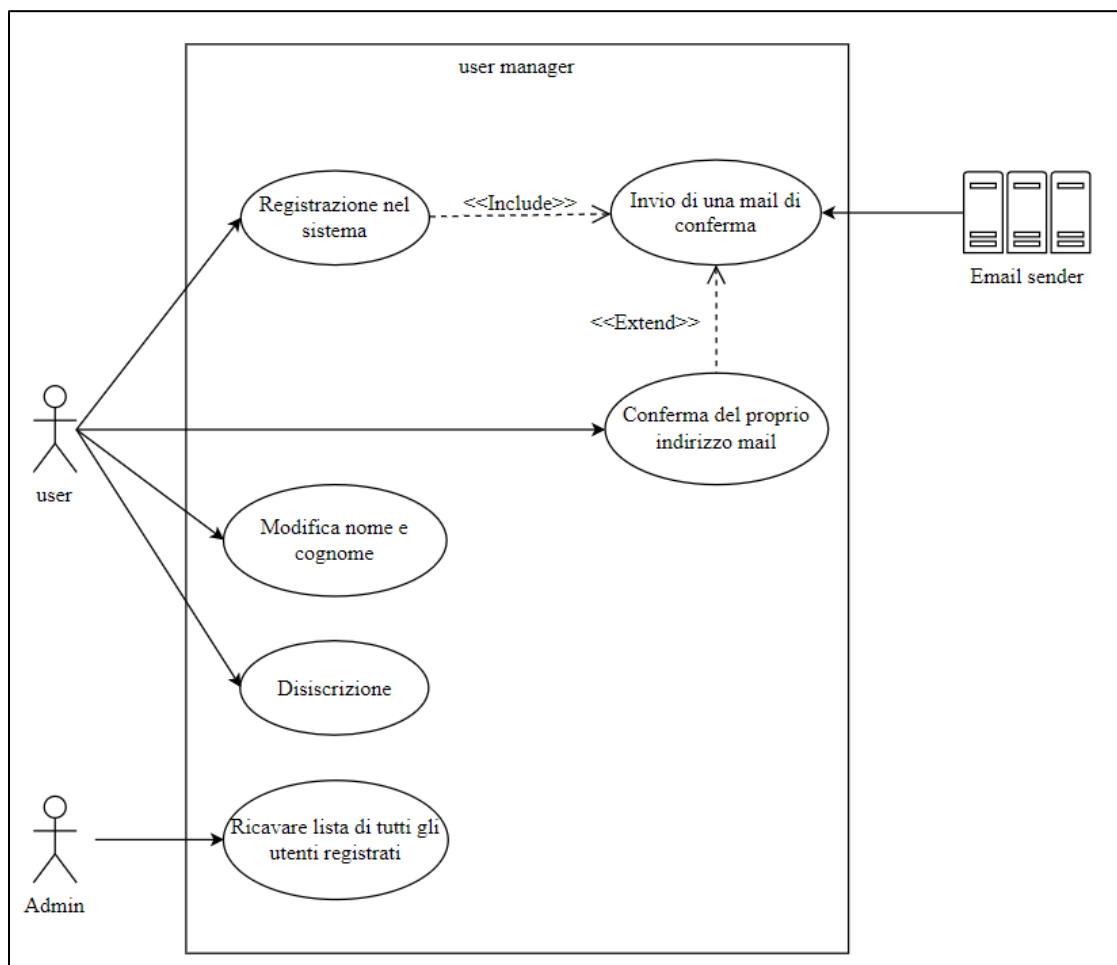


Figura 4.2 Diagramma dei casi d'uso

4.4. Processo di registrazione

Il processo di registrazione serve per memorizzare le informazioni del nuovo utente che ne fa richiesta e per verificare che l'indirizzo email fornito sia valido e che identifichi in maniera univoca l'utente che decide di iscriversi al servizio di newsletter.

Il processo di registrazione comprende diverse fasi, che nel prototipo sono:

1. Raccolta dell'indirizzo email e di altre informazioni personali, come il nome e il cognome dell'utente.
2. Invio di una email di verifica all'indirizzo fornito: Il sistema genera un URI, contenente il riferimento ad un token di conferma associato all'indirizzo a cui è stata inviata l'email.
3. L'utente deve cliccare sull'URI generato nella fase precedente. Al clic, viene verificato che il token sia valido rispetto all'indirizzo email da verificare. Se il processo ha esito positivo, lo stato dell'utente stato viene impostato a "verificato".

4.4.1. Prima fase: acquisizione dei dati di registrazione

Il processo inizia quando un utente invia una richiesta di registrazione al sistema fornendo i propri dati di registrazione, come indirizzo mail, nome e cognome. Se l'utente esiste già, il sistema rifiuta la richiesta e restituisce un errore. Altrimenti, genera e memorizza nel database un codice di conferma univoco, il token, associato all'utente.

Dopo avere creato il link di conferma cliccabile, "User Manager" richiede ad "Email Sender", che esso venga recapitato via email all'utente. Nel caso in cui la comunicazione tra i due micro-servizi fallisca viene restituito un messaggio di errore generato dall'API di fallback del Circuit Breaker.

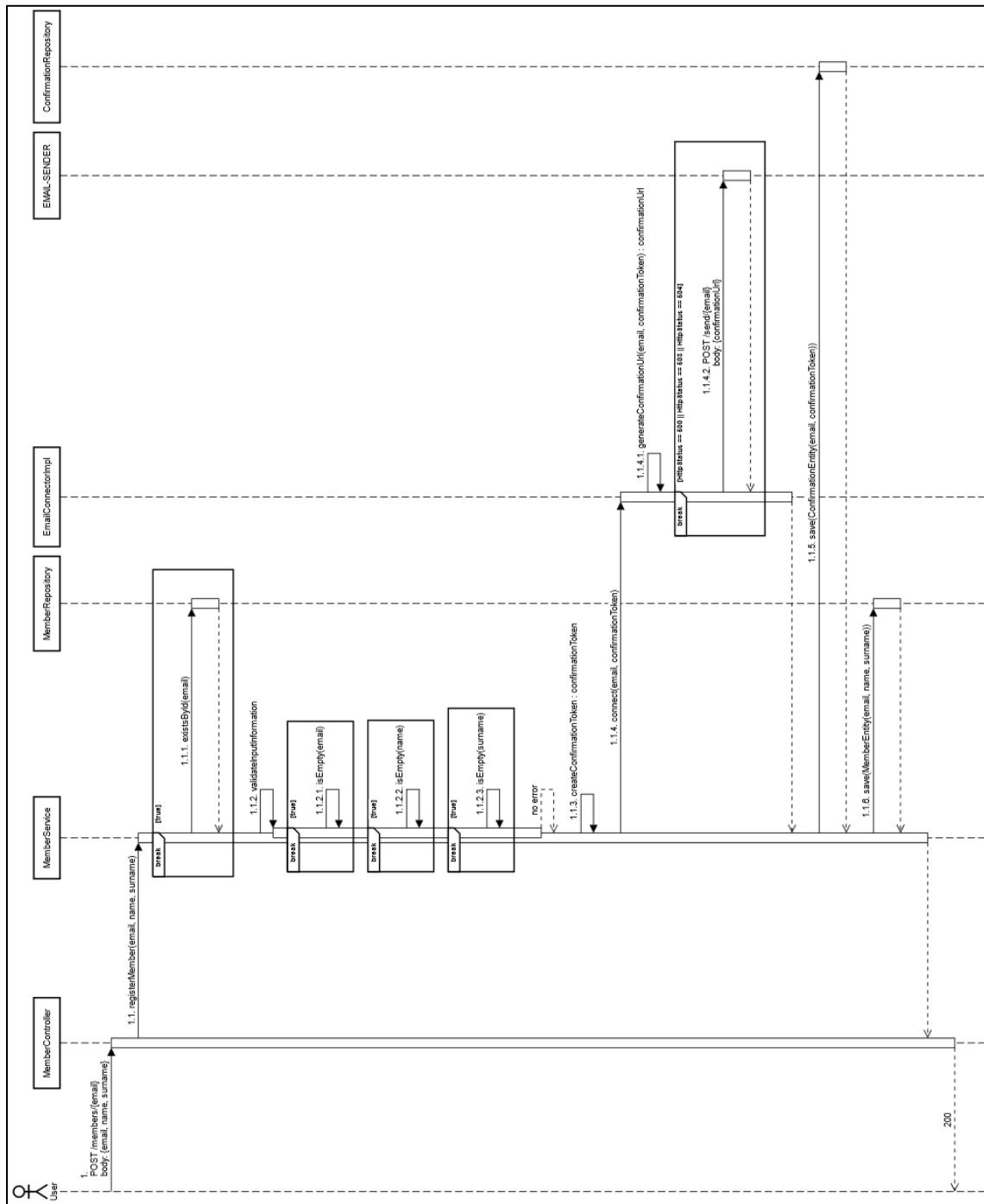


Figura 4.3 Primo step flusso di registrazione

4.4.2. Seconda fase: invio dell'email di verifica

Quando “User Manager” richiede ad “Email Sender” di occuparsi dell'invio del link di verifica, il comando del flusso viene passato nelle mani di quest'ultimo. Se l'invio dell'email ha esito positivo, “User Manager” riceve codice HTTP 200, *ok*. Altrimenti vengono restituiti i codici di errore 500, *internal server error* o 503, *service unavailable*, che spingerebbero il Circuit Breaker a restituire un messaggio di errore e ad incrementare il proprio contatore interno delle richieste fallite.

“Email Sender” si appoggia a JavaMail, una libreria per l'invio di email tramite protocolli di posta elettronica come SMTP, POP3 o IMAP. All'interno del progetto sono utilizzati i servizi di posta di Google, che fanno utilizzo del protocollo SMTP, *Simple Mail Transfer Protocol*.

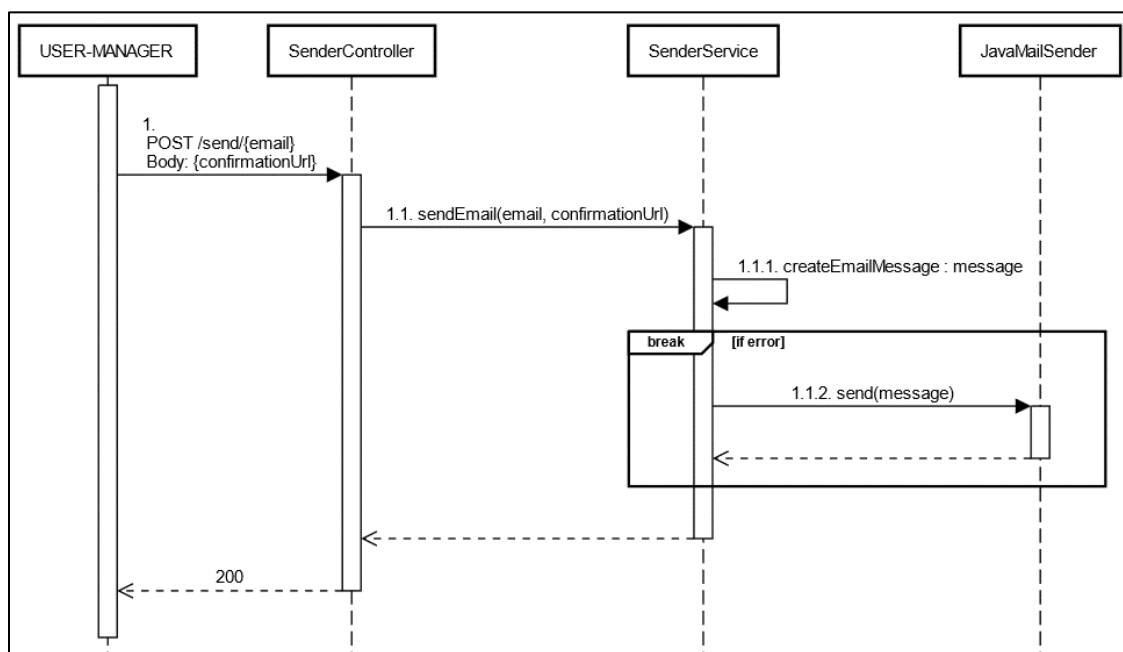


Figura 4.4 Secondo step flusso di registrazione

Affinché “Email Sender” sia in grado di inviare un messaggio di posta elettronica, per conto di un indirizzo email preconfigurato, è necessario impostare le seguenti proprietà:

```
spring:
  mail:
    host: smtp.gmail.com
    port: 587
    username: newsletter.rnma@gmail.com
    password: ltsjfnzomiwekljt
    properties:
      mail:
        smtp:
          auth: true
          starttls:
            enable: true
```

4.4.3.Terza fase: conferma dell'identità

Quando un utente fa clic sull'URI ricevuto, inizia la procedura per il controllo della correttezza del token di validazione ricevuto. L'URI cliccabile contiene i riferimenti al token generato nella prima fase e all'indirizzo email da cui parte la richiesta.

Se l'utente ha già confermato la propria identità o all'indirizzo email è associato un token differente da quello ricevuto, la richiesta fallisce e viene restituito un messaggio di errore. Altrimenti, l'utente viene validato e le modifiche vengono propagate all'interno del database.

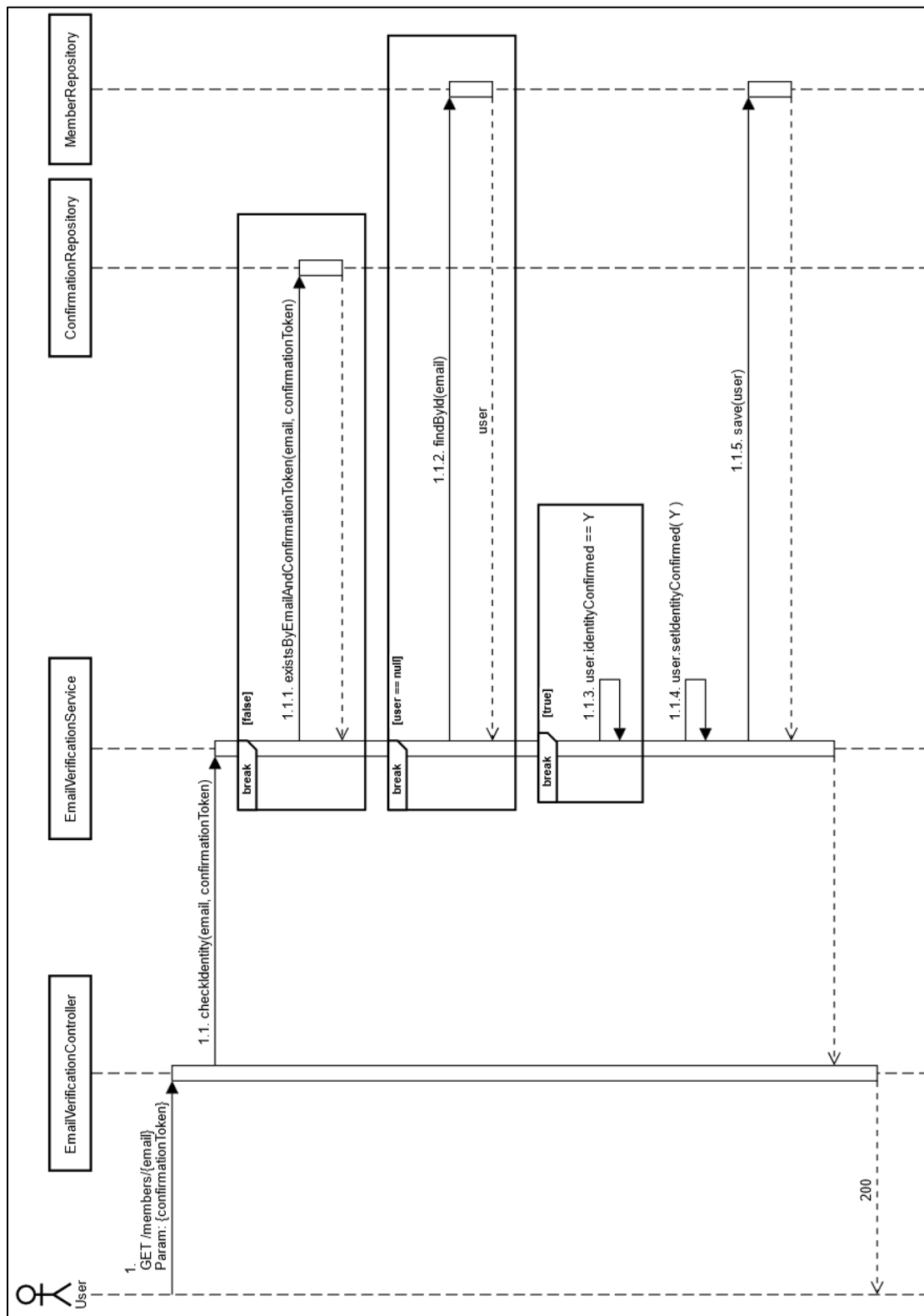


Figura 4.5 Terzo step flusso di registrazione

4.4.4. One way interaction

Il clic dell'URI innesca una chiamata HTTP di tipo GET. L'utilizzo del verbo GET semplifica la progettazione di questa fase, poiché consente ad un utente di confermare la propria identità semplicemente cliccando sul link ricevuto.

Altrimenti, più correttamente, l'utente avrebbe dovuto inserire il token ricevuto in una form, la cui submission avrebbe dato origine ad una richiesta di tipo PUT.

L'utilizzo del verbo PUT rappresenta il cambiamento di stato dell'utente da “non verificato” a “verificato”.

A differenza dell'utilizzo classico di una GET, che prevede la ricezione di una risposta contenente un'entità o una collezione di entità; una richiesta GET di tipo “one way” provoca un effetto unidirezionale, come il cambiamento dello stato di un oggetto del sistema (o del database), senza prevedere la ricezione di alcuna entità in risposta.

4.5. Validazione del processo di registrazione

1. Invio della richiesta di registrazione POST `members/{email}`

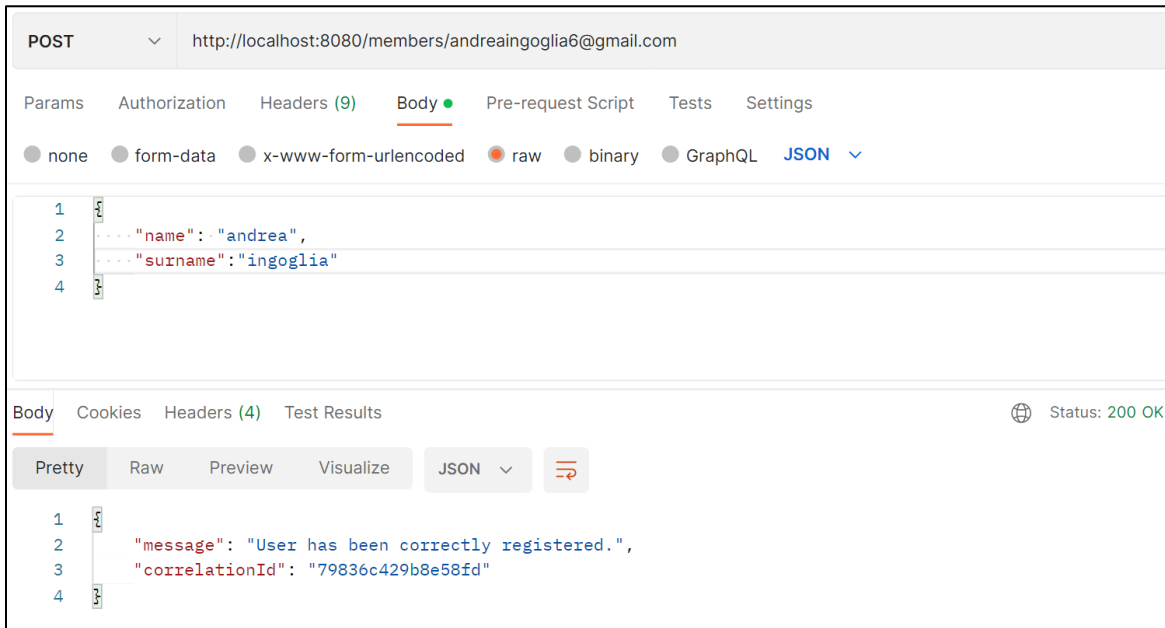


Figura 4.6 Validazione primo step di registrazione

Tale chiamata deve essere indirizzata al gateway del sistema, sulla porta 8080. Essa prevede, che nel corpo della richiesta siano presenti il nome e il cognome dell'utente da registrare in formato JSON.

2. Invio dell'email di conferma

Tale fase è gestita interamente dall'applicazione e risulta nascosta all'utilizzatore. Il micro-servizio "User Manager" genera l'URI su cui cliccare per la verifica e chiede al servizio "Email Sender" di occuparsi dell'inoltro dell'email, tramite il seguente metodo:

```

public void connect (String email, String confirmationToken)
{
    ConfirmationUrl url = new
    ConfirmationUrl(generateConfirmationUrl(email,
    confirmationToken));
    HttpEntity<ConfirmationUrl>httpEntity=newHttpEntity<>(url);
    restTemplate.postForEntity("http://EMAIL-SENDER/send/" +
    email, httpEntity, String.class);
}

```

3. Clic sull'indirizzo di verifica

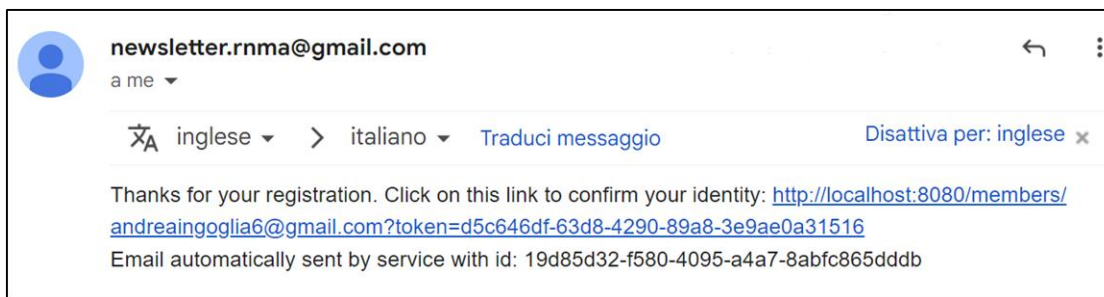


Figura 4.7 Validazione terzo step di registrazione

Nell'email viene riportato anche l'identificativo dell'istanza di "Email Sender" da cui è stata inviata l'email. In questo modo è possibile verificare che il Load Balancer commuti il traffico, da "User Manager" a "Email Sender", secondo l'algoritmo Round Robin.

4.6. API di supporto

In aggiunta alle API per la registrazione, il sistema ne espone altre per l'aggiornamento del nome e del cognome di un utente, per la disiscrizione di un utente della newsletter o per ottenere la lista completa di tutti gli utenti registrati.

4.6.1. Flusso di aggiornamento

L'API di aggiornamento consente agli utenti di modificare i propri nome e cognome. Il processo prevede che un utente sia già registrato all'interno del sistema e che le nuove informazioni non siano valori nulli. Qualora i dati siano validi il servizio imposta i nuovi campi e salva le modifiche nel database. Al termine del processo, viene restituito un codice di stato 200 per indicare che l'aggiornamento è avvenuto con successo.

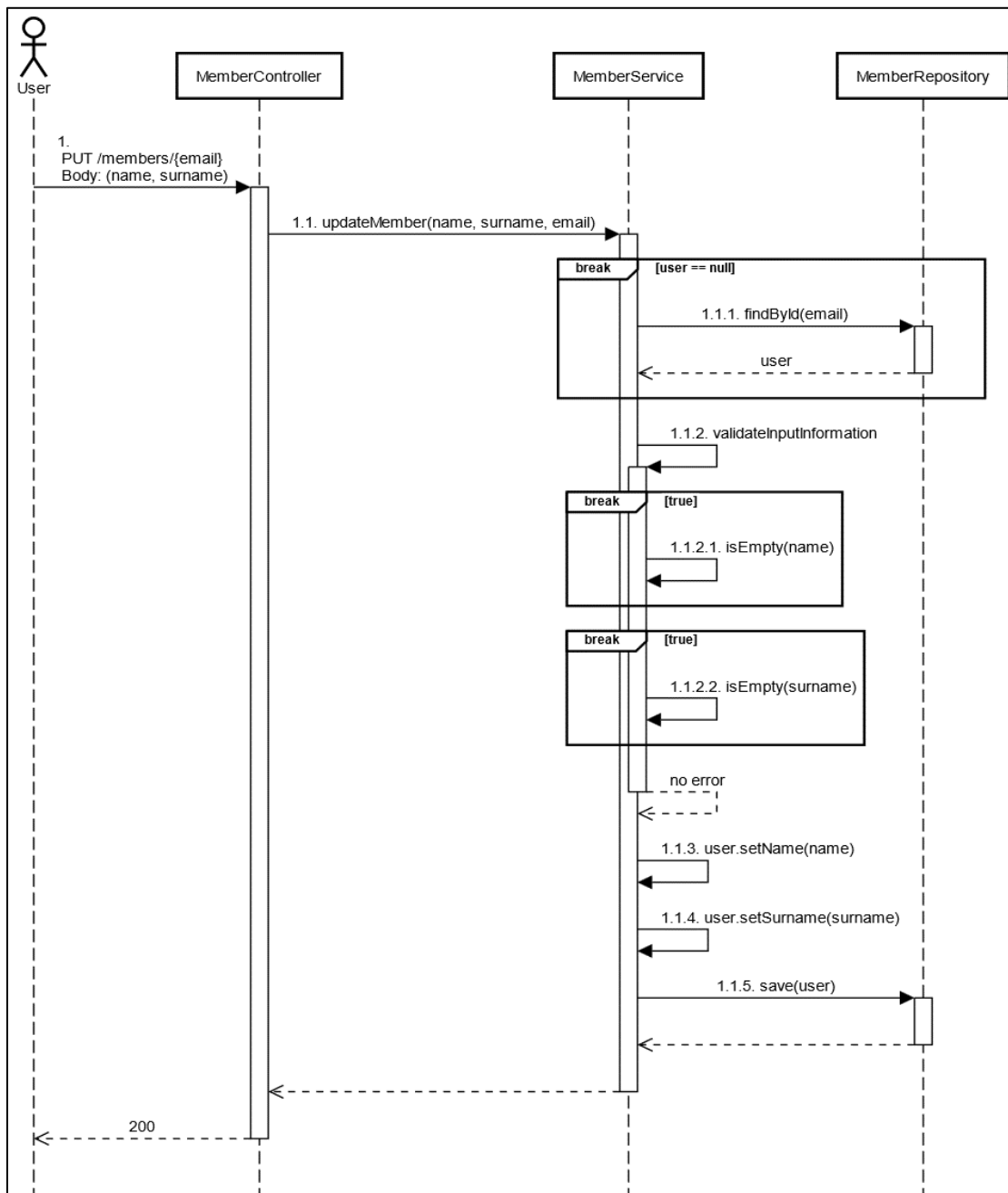


Figura 4.8 Flusso di aggiornamento

4.6.1.1. Validazione del processo di aggiornamento

La richiesta PUT `members/{email}`, prevede che vengano passati in format JSON i nuovi valori di nome e cognome nel corpo della chiamata.

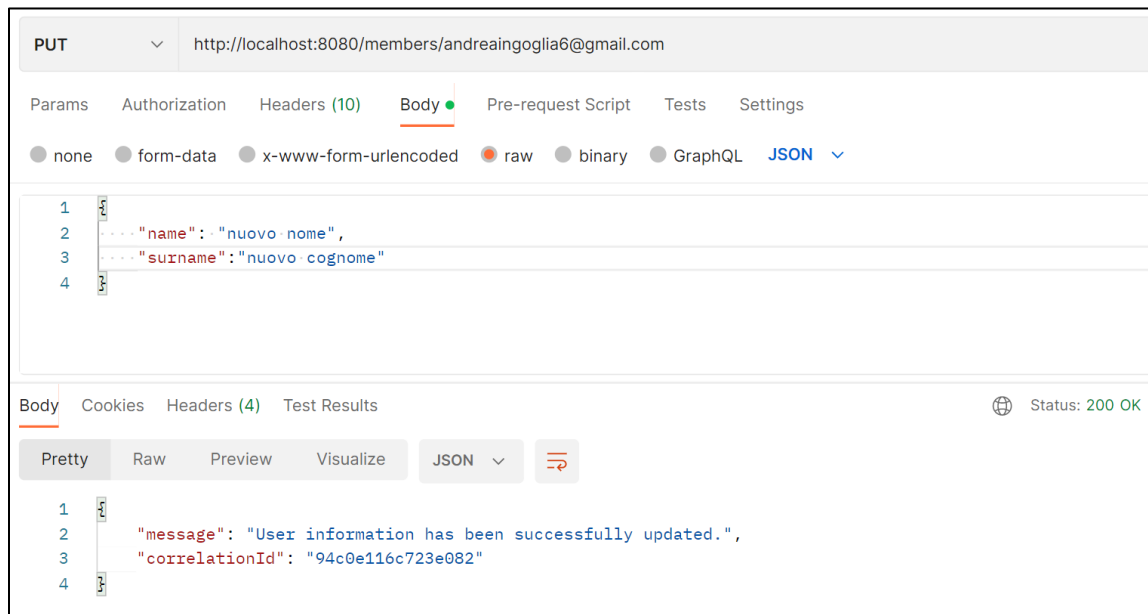


Figura 4.9 Validazione flusso di aggiornamento

Anche in questo caso, la chiamata è rivolta verso la porta 8080, sulla quale è connesso Spring Cloud Gateway.

4.6.2. Flusso di disiscrizione

Il flusso di cancellazione serve affinché un utente possa eliminare le proprie informazioni dal sistema. È necessario che l'utente fornisca la propria email e una chiave valorizzata a YES, per confermare la propria volontà di procedere con la cancellazione.

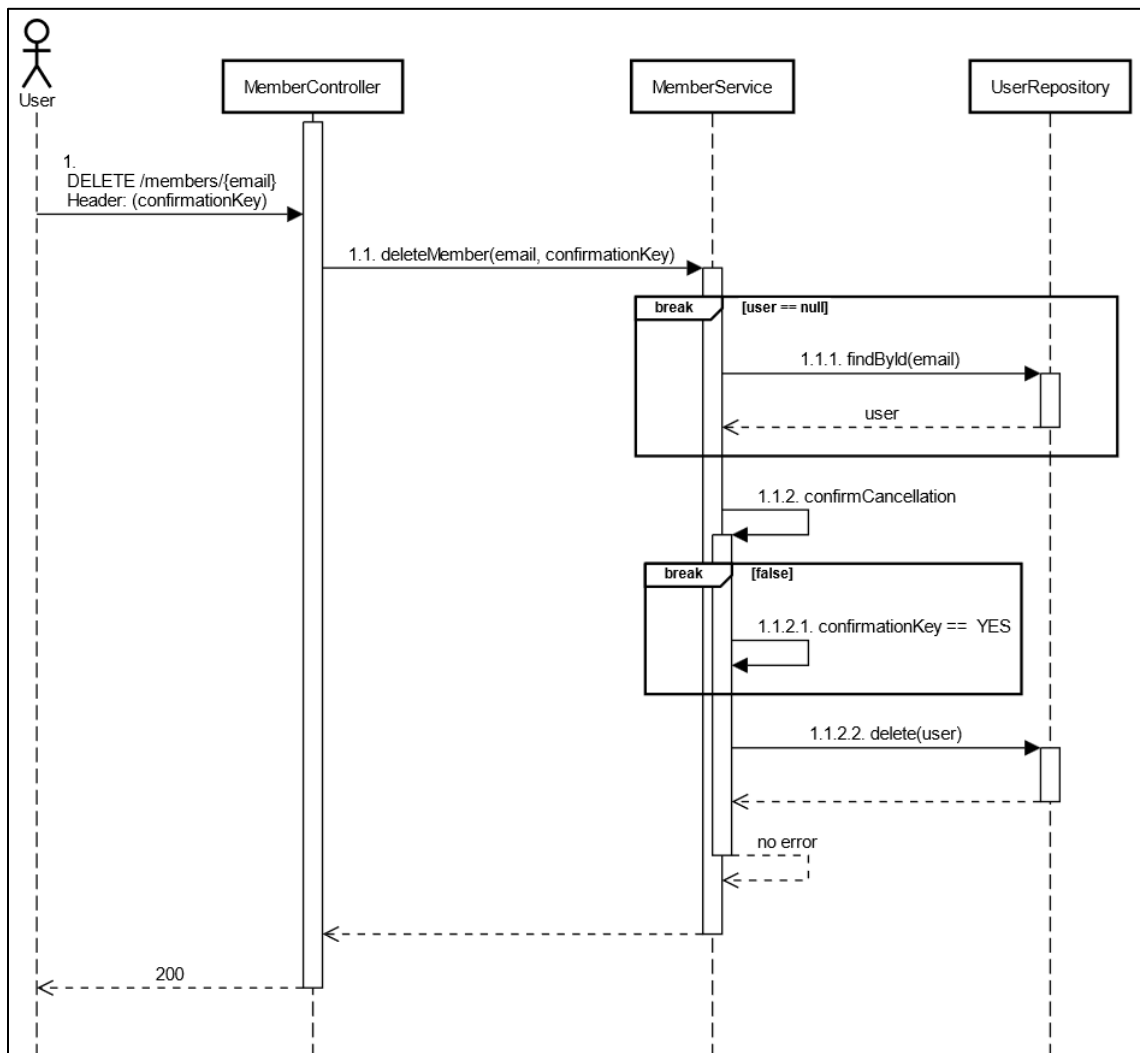


Figura 4.10 Flusso di disiscrizione

4.6.2.1. Validazione del flusso di disiscrizione

La richiesta DELETE `members/{email}`, prevede che tra i campi di testata, venga valorizzata la chiave `confirmationKey` a YES, affinché l'utente possa confermare la propria scelta.

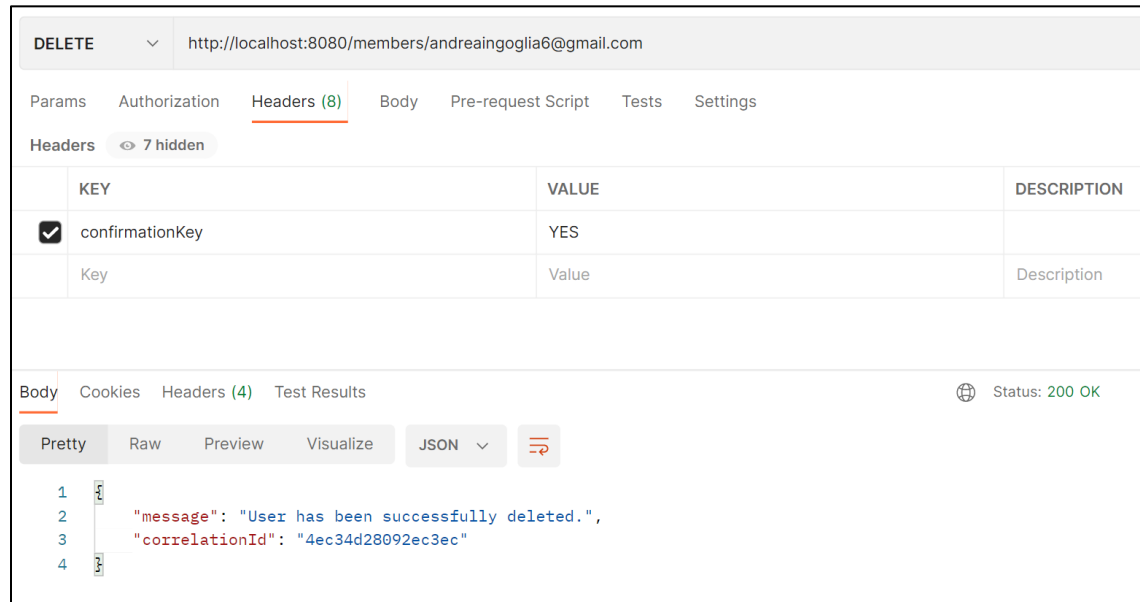


Figura 4.11 Validazione flusso di disiscrizione

4.6.3. Stampa degli utenti

Quest'ultimo processo permette di ricavare l'elenco completo degli utenti registrati nel sistema.

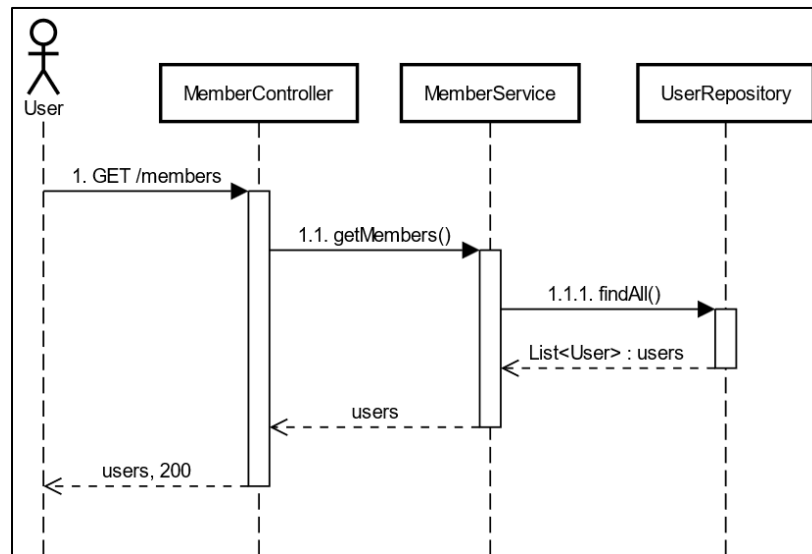


Figura 4.12 Flusso di stampa

4.6.3.1. Validazione del flusso di stampa

Affinché un utente amministratore possa ricavare l'elenco completo degli utenti registrati è stato esposto l'endpoint GET `members`.

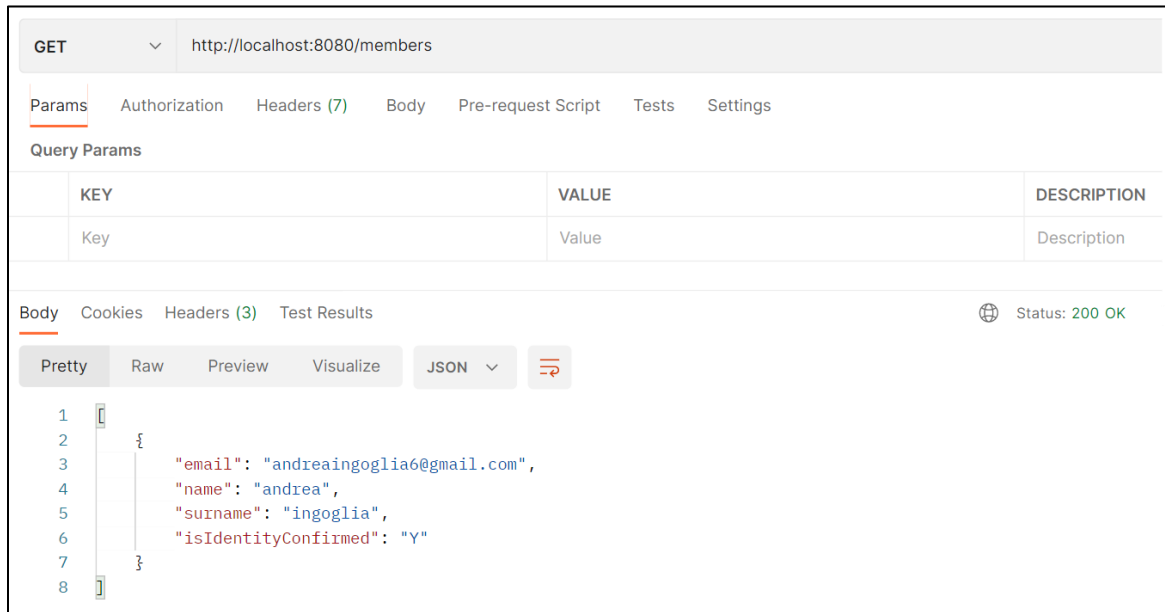


Figura 4.13 Validazione flusso di stampa

4.7. Riepilogo della struttura delle API

Servizio coinvolto	Nome API	Header	String Parameter	Body	Funzionalità
user manager	POST members/{email}	X	X	Name, Surname	Registrazione delle informazioni dell'utente e generazione del link di conferma
email sender	POST send/{email}	X	X	ConfirmationURI	Invio automatico dell'email di conferma all'utente
user manager	GET members/{email}	X	Token	X	Al click di questo URL, viene verificato che il token ricevuto sia valido per la rispettiva email
user manager	GET members	X	X	X	Permette di ricavare tutti gli utenti registrati in formato JSON
user manager	DELETE members/{email}	SafeKey	X	X	Permette di cancellare un utente dalla banca dati
user manager	PUT members/{email}	X	X	Name, Surname	Permette di modificare il nome e il cognome associati all'email, con i nuovi valori passati nel body

Figura 4.14 Elenco delle API

I risultati ottenuti precedentemente sono replicabili per mezzo di una collection Postman, che contiene le richieste verso le API esposte dal prototipo ².

² Collection Postman disponibile al seguente indirizzo: <https://github.com/AndreaIngolia/RNMA-Resilient-Newsletter-Microservice-Application/tree/main/Collection%20Postman>

5. RNMA all'opera: validazione dei pattern per la resilienza e per l'osservabilità

5.1. Introduzione

In questo capitolo, dopo avere mostrato le funzionalità dell'applicazione, verranno mostrate alcune tipologie di errore, che porteranno all'attivazione di differenti strategie per la resilienza. Nello specifico, ci si concentrerà sul Circuit Breaker, sul Time Limiter e sul Rate Limiter.

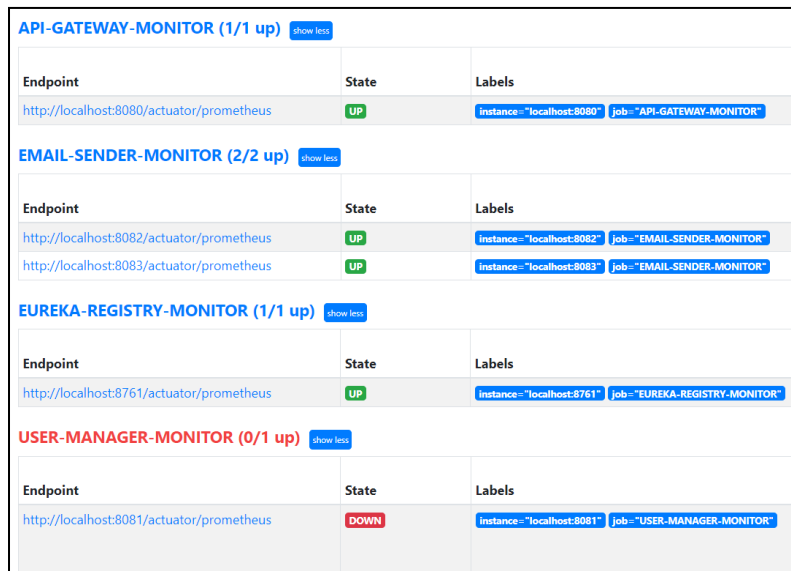
Verrà illustrato come è possibile sfruttare Elasticsearch, Logstash e Kibana per aggregare i log generati da una transazione.

5.2. Circuit Breaker in azione

Il Circuit Breaker entra in azione in presenza di un guasto parziale, ovvero nella situazione in cui tutte le istanze di un particolare micro-servizio siano indisponibili.

A tale scopo sarà necessario ricondursi ad una delle due configurazioni:

1. Indisponibilità totale del servizio "User Manager"



API-GATEWAY-MONITOR (1/1 up) show less		
Endpoint	State	Labels
http://localhost:8080/actuator/prometheus	UP	instance="localhost:8080" job="API-GATEWAY-MONITOR"
EMAIL-SENDER-MONITOR (2/2 up) show less		
Endpoint	State	Labels
http://localhost:8082/actuator/prometheus	UP	instance="localhost:8082" job="EMAIL-SENDER-MONITOR"
http://localhost:8083/actuator/prometheus	UP	instance="localhost:8083" job="EMAIL-SENDER-MONITOR"
EUREKA-REGISTRY-MONITOR (1/1 up) show less		
Endpoint	State	Labels
http://localhost:8761/actuator/prometheus	UP	instance="localhost:8761" job="EUREKA-REGISTRY-MONITOR"
USER-MANAGER-MONITOR (0/1 up) show less		
Endpoint	State	Labels
http://localhost:8081/actuator/prometheus	DOWN	instance="localhost:8081" job="USER-MANAGER-MONITOR"

Figura 5.1 Indisponibilità del servizio "User Manager"

In questo caso, per ogni tentativo di raggiungere una delle qualsiasi API appartenente a “User Manager”, verrà mostrato il messaggio di errore generato dall’API di fallback del Circuit Breaker, poiché il sistema in sé, non è in grado di fornire una risposta appropriata:

```
{
  "error": "Circuit Breaker: The service is actually
    unavailable. Please, contact our development team for
    further information."
}
```

2. Indisponibilità totale del servizio “Email Sender”

API-GATEWAY-MONITOR (1/1 up) show less		
Endpoint	State	Labels
http://localhost:8080/actuator/prometheus	UP	instance="localhost:8080" job="API-GATEWAY-MONITOR"
EMAIL-SENDER-MONITOR (0/2 up) show less		
Endpoint	State	Labels
http://localhost:8082/actuator/prometheus	DOWN	instance="localhost:8082" job="EMAIL-SENDER-MONITOR"
http://localhost:8083/actuator/prometheus	DOWN	instance="localhost:8083" job="EMAIL-SENDER-MONITOR"
EUREKA-REGISTRY-MONITOR (1/1 up) show less		
Endpoint	State	Labels
http://localhost:8761/actuator/prometheus	UP	instance="localhost:8761" job="EUREKA-REGISTRY-MONITOR"
USER-MANAGER-MONITOR (1/1 up) show less		
Endpoint	State	Labels
http://localhost:8081/actuator/prometheus	UP	instance="localhost:8081" job="USER-MANAGER-MONITOR"

Figura 5.2 Indisponibilità del servizio “Email Sender”

In questo caso, verrà mostrato il precedente messaggio di errore, se e solo se, verrà richiamata la prima API di registrazione `POST members/{email}`, ovvero l’unica API che prevede il coinvolgimento di “Email Sender” per l’invio automatico delle email di conferma.

5.3. Time Limiter in azione

Il sistema deve essere in grado di proteggersi da richieste che impiegano volontariamente un intervallo di tempo troppo elevato per l'elaborazione della richiesta. Il Time Limiter implementato per il prototipo abortisce di default tutte le richieste che impiegano un tempo superiore a sessanta per essere completate.

Affinché fosse possibile simulare un comportamento di tale genere, è stata predisposta un'API di supporto così composta:

```
@GetMapping(path = {"/timeout"})
public void slowEndpoint() throws InterruptedException
{
    Thread.sleep(70000);
}
```

Il metodo `Thread.sleep(long millis)` consente di simulare una richiesta volontariamente lenta.

L'API è raggiungibile interrogando Spring Cloud Gateway sulla porta 8080, con la seguente richiesta, senza specificare ulteriori parametri:

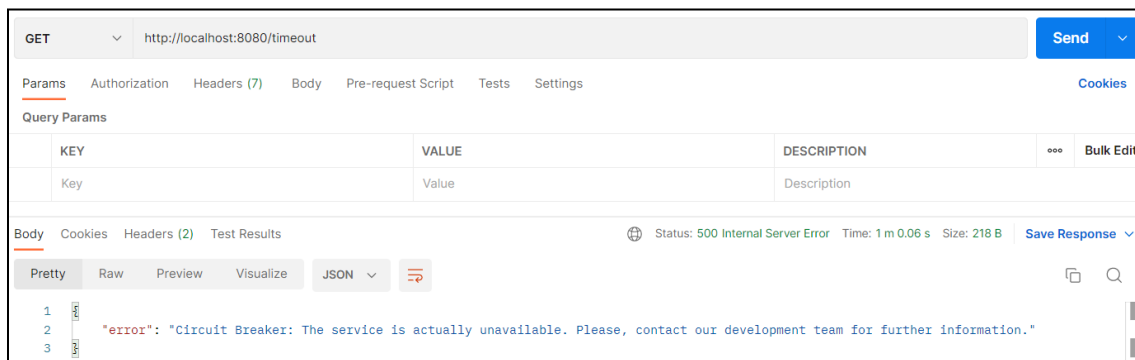


Figura 5.3 Api di fallback per il Time Limiter

5.4. Rate Limiter in azione

Il prototipo permette agli utilizzatori di interrogare l'API di registrazione `POST members/{email}` ad una velocità massima di dieci richieste al minuto, poiché si tratta di una funzionalità che necessita di un elevato overhead di rete per essere completata.

Qualora il numero di richieste al minuto superi la soglia prefissata, verrà restituito il seguente messaggio di errore:

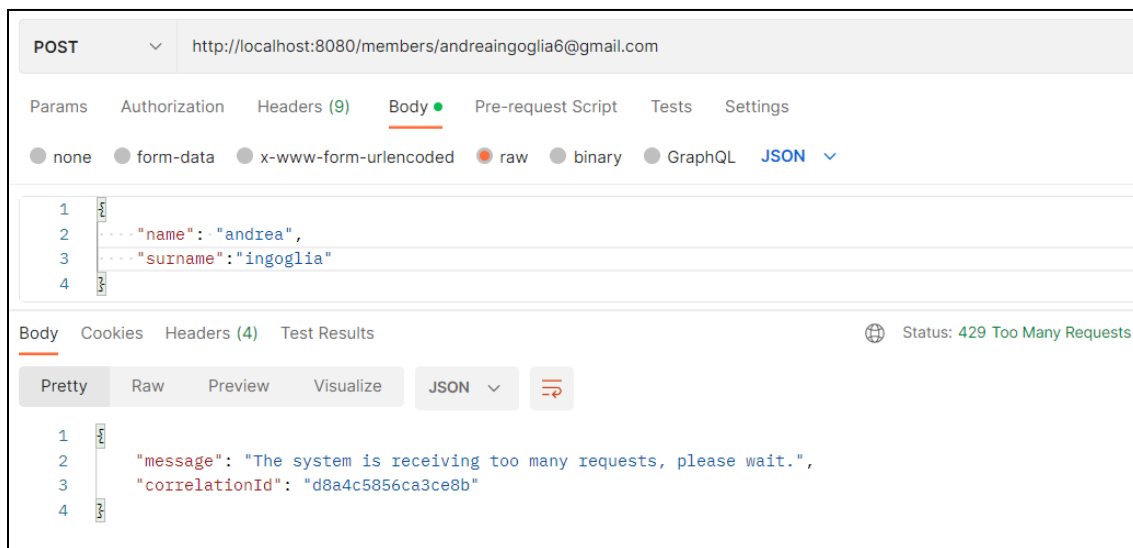


Figura 5.4 Attivazione del Rate Limiter

5.5. Aggregazione dei log

Data una richiesta, potrà essere utilizzato il correlation id fornito in risposta dal sistema, per aggregare i log della richiesta. Si supponga l'esempio della cancellazione di un utente dal sistema:

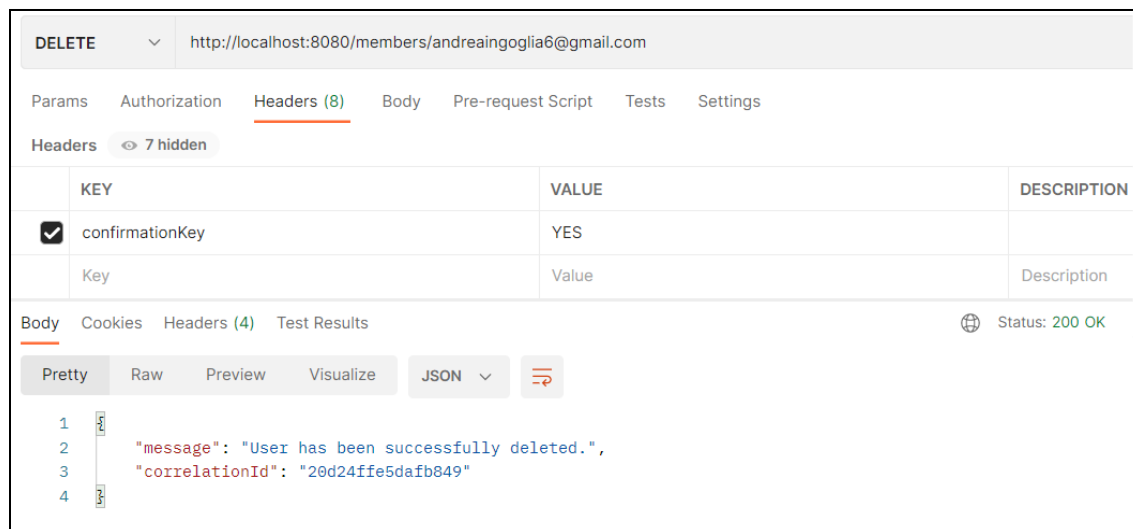


Figura 5.5 Esempio cancellazione di un utente

Da Kibana, utilizzando il codice fornito, verrà mostrata la seguente situazione:



Figura 5.6 Dashboard di Kibana

Per la chiave fornita sono stati generati i seguenti log:

1. *“Member with email andreaingoglia6@gmail.com is trying to unsubscribe from the newsletter.”*
2. *“The provided key is correct, user with email andreaingoglia6@gmail.com has been unsubscribed from the newsletter.”*

Sviluppi futuri per l'applicazione

Da quanto trattato nei capitoli precedenti, è possibile notare come l'adozione dell'architettura a micro-servizi abbia mutato il paradigma con cui un'applicazione può essere progettata. È necessario, tuttavia, tenere in considerazione anche gli aspetti sistemistici legati alla sua distribuzione.

Ad oggi, per il rilascio di un prodotto software, sempre più spesso ci si affida all'utilizzo di strumenti avanzati, come Docker e Kubernetes, due soluzioni popolari per la gestione e per la scalabilità dei servizi containerizzati.

In questo capitolo, verranno descritti Docker e Kubernetes ad un livello introduttivo e ne verranno analizzati i vantaggi. In particolare, si vedrà Docker come tecnologia per la creazione di container e Kubernetes come sistema per la scalabilità automatica.

Docker

Docker [21] è una piattaforma di containerizzazione che permette di impacchettare un servizio e tutte le sue dipendenze in un unico elemento, detto container, che può essere eseguito su qualsiasi macchina in cui è presente Docker, il che garantisce che l'applicazione funzioni allo stesso modo su qualsiasi sistema operativo e hardware.

Docker fonda il proprio funzionamento sul concetto di “immagine”. Le immagini sono dei modelli preconfigurati che contengono tutte le informazioni e i riferimenti necessari, come il codice sorgente, le dipendenze e le configurazioni di un servizio, da utilizzare come impronta per la creazione di container. I container possono essere catalogati come istanze eseguibili di un'immagine.

Ciascun container rappresenta un'istanza isolata di un servizio, in questo modo, è possibile eseguire più container che utilizzano la stessa immagine, garantendo un'ampia disponibilità del servizio.

Con Docker è possibile creare, testare e distribuire applicazioni in modo rapido ed efficiente su qualsiasi ambiente senza tenere conto del suo sistema operativo.

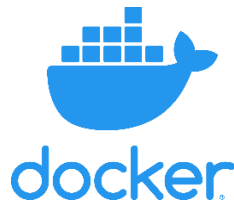


Figura 6.1 Logo di Docker

Kubernetes

Kubernetes [22] è una piattaforma open-source, che permette di scalare i servizi containerizzati di un sistema in modo automatizzato [2].

Docker e Kubernetes sono tecnologie che possono essere utilizzate in modo complementare. Docker consente di creare e distribuire container, mentre Kubernetes ne consente la scalabilità. Combinando le tecnologie, è possibile implementare il pattern per la resilienza Auto Scaling.

Sfruttando questo pattern sarebbe possibile replicare automaticamente i servizi di un sistema in funzione del loro carico lavoro e del loro stato di salute, aumentandone o diminuendone le istanze attive, per garantire maggiore potenza di calcolo o per liberare le risorse inutilizzate.



Figura 6.2 Logo di Kubernetes

Indice delle figure

Figura 1.1 Confronto tra architettura monolitica e architettura a micro-servizi

Figura 2.1 Logo di Java

Figura 2.2 Logo di IntelliJ

Figura 2.3 Logo di Git

Figura 2.4 Logo di GitHub

Figura 2.5 Logo di Postman

Figura 2.6 Logo di Spring

Figura 2.7 Logo di Maven

Figura 2.8 Funzionamento del front controller

Figura 2.9 Rappresentazione concettuale del modello esagonale di Alistair Cockburn

Figura 3.1 Logo di Resilience4j

Figura 3.2 Elenco dei servizi del progetto registrati a Eureka

Figura 3.3 Elenco dei servizi, host e numero di porta del prototipo

Figura 3.4 Logo di Logstash

Figura 3.5 Logo di Elasticsearch

Figura 3.6 Logo di Kibana

Figura 3.7 Logo di Prometheus

Figura 3.8 Stato di salute dei servizi del prototipo

Figura 3.9 Servizio con stato di salute "unhealthy"

Figura 3.10 Utilizzo della CPU di "User Manager"

Figura 3.11 Utilizzo della CPU di "Email Sender"

Figura 4.1 Diagramma dei casi d'uso

Figura 4.2 Diagramma a componenti dell'architettura del sistema

Figura 4.3 Primo step flusso di registrazione

Figura 4.4 Secondo step flusso di registrazione

Figura 4.5 Terzo step flusso di registrazione

Figura 4.6 Validazione primo step di registrazione

Figura 4.7 Validazione terzo step di registrazione

Figura 4.8 Flusso di aggiornamento

Figura 4.9 Validazione flusso di aggiornamento

Figura 4.10 Flusso di disiscrizione

Figura 4.11 Validazione flusso di disiscrizione

Figura 4.12 Flusso di stampa

Figura 4.13 Validazione flusso di stampa

Figura 4.14 Elenco delle API

Figura 5.1 Indisponibilità del servizio “User Manager”

Figura 5.3 Api di fallback per il Time Limiter

Figura 5.4 Attivazione del Rate Limiter

Figura 5.5 Esempio cancellazione di un utente

Figura 5.6 Dashboard di Kibana

Figura 6.1 Logo di Docker

Figura 6.2 Logo di Kubernetes

Bibliografia

- [1] A conceptual framework for resilience: fundamental, definitions, strategies and metrics. Jesper Andersson, Vincenzo Grassi, Raffaella Mirandola, Diego Perez-Palacin. 15 dicembre 2020
- [2] Microservices Patterns, Chris Richardson.
- [3] What is Java?: https://www.java.com/it/download/help/whatis_java.html
- [4] IntelliJ: <https://www.jetbrains.com/idea/>
- [5] Git: <https://git-scm.com/>
- [6] GitHub: <https://github.com/>
- [7] H2 database: <https://www.h2database.com/html/main.html>
- [8] Postman: <https://www.getpostman.com/>
- [9] Maven: <https://maven.apache.org/>
- [10] Spring Framework: <https://docs.spring.io/spring-framework/docs/5.0.7.RELEASE/spring-framework-reference/index.html>
- [11] Spring Data JPA: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>
- [12] Spring MVC: <https://docs.spring.io/spring-framework/docs/5.0.7.RELEASE/spring-framework-reference/web.html#spring-web>
- [13] Spring Cloud: <https://spring.io/projects/spring-cloud>
- [14] Modello esagonale: <https://alistair.cockburn.us/hexagonal-architecture/>
- [15] Resilience4j: <https://resilience4j.readme.io/docs>
- [16] Logstash: <https://www.elastic.co/logstash>
- [17] Elasticsearch: <https://www.elastic.co/what-is/elasticsearch>
- [18] Kibana: <https://www.elastic.co/what-is/kibana>
- [19] Logback: <https://logback.qos.ch/manual/>
- [20] Prometheus: <https://prometheus.io>
- [21] Docker: <https://www.docker.com/>
- [22] Cos'è Kubernetes?: <https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/>

[23] JavaMail: <https://docs.spring.io/spring-framework/docs/5.0.7.RELEASE/spring-framework-reference/integration.html#mail>