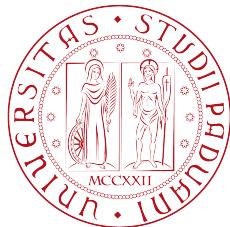


**Università degli Studi di Padova**

---

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”  
Corso di Laurea in Informatica



**Migrazione di una Web application  
con architettura monolitica  
verso un'architettura a microservizi**

Tesi di laurea triennale

Relatore  
**Prof. Davide Bresolin**

Laureando  
**Matteo Marchiori**

.....

.....

---

**Anno Accademico 2018–2019**

Matteo Marchiori: *Migrazione di una Web application con architettura monolitica verso un'architettura a microservizi*, Tesi di laurea triennale, © 26 settembre 2019.

Per le immagini riportate in questa tesi :

Ballet: si veda la figura 18 a pagina 17

Fonte: <https://pixabay.com/it/photos/balletto-teatro-danza-illuminazione-2682291>

Foto originale di Bruno Riomar, 2017

© <https://pixabay.com/it/photos/balletto-teatro-danza-illuminazione-2682291>

Licenza: <https://pixabay.com/it/service/license>

Orchestra: si veda la figura 18 a pagina 17

Fonte: [https://upload.wikimedia.org/wikipedia/commons/3/39/MIT0\\_Orchestra\\_Sinfonica\\_RAI.jpg](https://upload.wikimedia.org/wikipedia/commons/3/39/MIT0_Orchestra_Sinfonica_RAI.jpg)

Foto originale di MITO SettembreMusica, 2008

© [https://upload.wikimedia.org/wikipedia/commons/3/39/MIT0\\_Orchestra\\_Sinfonica\\_RAI.jpg](https://upload.wikimedia.org/wikipedia/commons/3/39/MIT0_Orchestra_Sinfonica_RAI.jpg)

Licenza: Creative Commons

Circuit breaker: si veda la figura 21 a pagina 20

Fonte: [https://upload.wikimedia.org/wikipedia/commons/6/6a/Circuit\\_breaker\\_2\\_pole\\_on\\_DIN\\_rail.JPG](https://upload.wikimedia.org/wikipedia/commons/6/6a/Circuit_breaker_2_pole_on_DIN_rail.JPG)

Foto originale di Alexey Khrulev, 2008

© [https://upload.wikimedia.org/wikipedia/commons/6/6a/Circuit\\_breaker\\_2\\_pole\\_on\\_DIN\\_rail.JPG](https://upload.wikimedia.org/wikipedia/commons/6/6a/Circuit_breaker_2_pole_on_DIN_rail.JPG)

Licenza: Creative Commons

Tutti i marchi riportati appartengono ai legittimi proprietari.

# SOMMARIO

Questo documento descrive il lavoro da me svolto durante il periodo di stage presso l'azienda Sync Lab s.r.l. di Padova. Lo stage ha avuto una durata complessiva di 304 ore e si è svolto sotto la supervisione di Fabio Pallaro.

Gli obiettivi da raggiungere erano i seguenti:

- studio di architetture a microservizi;
- studio e ripasso di linguaggi e tecnologie per il backend;
- studio di tecnologie per il frontend;
- implementazione dei microservizi per la migrazione di un'applicazione con architettura monolitica in uso dall'azienda.

Il documento è stato suddiviso in:

**CAPITOLO 1** Fornisce alcuni dettagli sulla struttura del testo e su Sync Lab.

**CAPITOLO 2** Descrive il progetto, gli obiettivi del progetto in dettaglio e le tecnologie adottate.

**CAPITOLO 3** Qui descrivo la parte relativa allo studio teorico delle architetture software e delle tecnologie usate nel progetto.

**CAPITOLO 4** Fornisce alcuni dettagli relativi alla progettazione dei microservizi.

**CAPITOLO 5** Qui descrivo l'implementazione dei microservizi.

**CAPITOLO 6** Considerazioni finali e possibili miglioramenti.



## RINGRAZIAMENTI

*Ringrazio il professor Davide Bresolin, relatore della mia tesi, per la disponibilità e per l'aiuto, oltre che per la passione che mi ha trasmesso per l'informatica.*

*Ringrazio Fabio Pallaro, il mio tutor, e tutti i colleghi e amici si Sync Lab. Senza il loro sostegno e senza i loro consigli non sarebbe stata la stessa esperienza.*

*Ringrazio Alberto che dispensa sempre ottimi consigli, perché senza la sua ispirazione oggi farei tutt'altro.*

*Ringrazio mamma Carla, papà Andrea e tutta la mia famiglia per l'appoggio e perché mi ricorda sempre che ho un senso e una certa importanza.*

*Ringrazio David, Giovanni, Victor, Nicola, Mirko e tutti i colleghi e amici che in questi tre anni mi hanno permesso di realizzare alcuni sogni nel cassetto e, almeno in parte, di raggiungere una certa autonomia.*

*Ringrazio tutti gli Alpha6, gli amici di The Crew, i miei professori del Severi e tutti quelli che hanno portato gioia nel mio percorso.*

*Ultimi ma non ultimi, ringrazio i professori Tommaso Gordini, Tullio Vardanega, Massimo Marchiori e Yuval Noah Harari, per aver cercato di farmi vedere la foresta oltre ai singoli alberi.*

*Padova, 26 settembre 2019*

**Matteo Marchiori**



# INDICE

Frontespizio	i
Sommario	iii
Ringraziamenti	v
Indice	vii
Elenco delle figure	ix
Elenco delle tabelle	xi
<b>1 INTRODUZIONE</b>	<b>1</b>
1.1 L’azienda	1
1.1.1 Profilo aziendale	1
1.2 Contesto applicativo	1
1.2.1 Servizi e prodotti offerti	2
1.2.2 Ricerca e sviluppo	3
1.3 Organizzazione del testo	3
<b>2 DESCRIZIONE</b>	<b>5</b>
2.1 Introduzione al progetto	5
2.2 Obiettivi	6
2.2.1 Obiettivi obbligatori	6
2.2.2 Obiettivi desiderabili	6
2.2.3 Obiettivi facoltativi	6
2.3 Requisiti	7
2.3.1 Requisiti obbligatori	7
2.3.2 Requisiti desiderabili	7
2.3.3 Requisiti facoltativi	8
2.4 Tecnologie	8
2.4.1 Tecnologie per i microservizi	8
2.4.2 Tecnologie per il frontend	11
2.4.3 Altre tecnologie	12
<b>3 STUDIO TEORICO</b>	<b>13</b>
3.1 Linee generali	13
3.1.1 Architettura software	13
3.1.2 Architettura a microservizi	14
3.2 Comparazione con l’approccio monolitico	15
3.3 Comparazione con le architetture orientate ai servizi	16
3.4 Pattern utilizzati	18
3.4.1 API Gateway	18
3.4.2 Service Registry/Discovery	19
3.4.3 Circuit Breaker	20
3.4.4 Database per service	21
3.4.5 Saga	22
<b>4 PROGETTAZIONE</b>	<b>25</b>
4.1 Descrizione della progettazione	25

<b>5 IMPLEMENTAZIONE</b>	<b>27</b>
<b>5.1 Descrizione dell'implementazione</b>	<b>27</b>
<b>5.1.1 Microservizio document</b>	<b>27</b>
<b>5.1.2 Microservizio rabbitmq</b>	<b>27</b>
<b>5.1.3 Microservizio logging</b>	<b>28</b>
<b>5.1.4 Microservizio eureka</b>	<b>28</b>
<b>5.1.5 Microservizio apigateway</b>	<b>28</b>
<b>5.2 Problemi e soluzioni</b>	<b>28</b>
<b>5.2.1 Più servizi, meno DRY</b>	<b>28</b>
<b>5.2.2 Controlli sui punti critici</b>	<b>29</b>
<b>5.3 Verifica</b>	<b>29</b>
<b>5.4 Collaudo</b>	<b>29</b>
<b>6 CONCLUSIONI</b>	<b>31</b>
<b>6.1 Consuntivo finale</b>	<b>31</b>
<b>6.2 Soddisfacimento degli obiettivi</b>	<b>31</b>
<b>6.3 Possibili miglioramenti</b>	<b>32</b>
<b>6.3.1 Prestazioni e scalabilità</b>	<b>32</b>
<b>6.3.2 Misure</b>	<b>32</b>
<b>6.3.3 Logging</b>	<b>32</b>
<b>6.4 Conoscenze acquisite</b>	<b>33</b>
<b>6.5 Valutazioni personali</b>	<b>33</b>
<b>Bibliografia</b>	<b>35</b>

## ELENCO DELLE FIGURE

Figura 1	Logo Sync Lab	1
Figura 2	Obiettivo: dal monolite ai microservizi	5
Figura 3	Logo Java	8
Figura 4	Logo Spring	8
Figura 5	Logo Spring Boot	9
Figura 6	Logo Spring Cloud	9
Figura 7	Logo RabbitMQ	10
Figura 8	Logo Resilience4J	10
Figura 9	Logo Log4J	11
Figura 10	Logo Angular	11
Figura 11	Logo MongoDB	12
Figura 12	Logo Gradle	12
Figura 13	Logo Git	12
Figura 14	Esempi di architettura a confronto	13
Figura 15	Architettura a microservizi	14
Figura 16	Architettura monolitica	15
Figura 17	Confronto SOA e microservizi	17
Figura 18	Confronto tra architettura SOA e architettura a microservizi per metodo di coordinamento	17
Figura 19	Esempio di API gateway	18
Figura 20	Come funziona il service discovery	19
Figura 21	Circuit breaker	20
Figura 22	Esempio di database per service	21
Figura 23	Esempio di saga pattern con coreografia	22



## ELENCO DELLE TABELLE

Tabella 1	Confronto tra monolite e microservizi	16
Tabella 2	Confronto tra servizi e microservizi	17
Tabella 3	Risultati dei test di collaudo	30
Tabella 4	Esito degli obiettivi	31



# 1

## INTRODUZIONE

In questo capitolo fornisco alcune informazioni riguardanti Sync Lab s.r.l., l'azienda presso cui ho svolto lo stage, e descrivo alcune norme tipografiche adottate nel documento.

### 1.1 L'AZIENDA



Figura 1: Logo Sync Lab

#### 1.1.1 Profilo aziendale

Sync Lab s.r.l. è una società di consulenza informatica fondata nel 2002 con sedi a Napoli, Roma, Milano e Padova. Fin dai primi anni Sync Lab è rapidamente cresciuta nel mercato dell'Information and Communications Technology (ICT), consolidando rapporti con clienti e collaboratori, raggiungendo un organico aziendale di oltre 200 risorse, una solida base finanziaria e una diffusione sul territorio con quattro sedi.

La grande attenzione alla gestione delle risorse umane ha fatto di Sync Lab un riferimento per coloro che volessero avviare o far evolvere in chiave professionale la propria carriera. Il basso ricambio del personale testimonia la voglia dei collaboratori di condividere il progetto comune, assumendo all'interno di esso ruoli e responsabilità che solo un processo evolutivo così intenso può offrire. Grazie all'approccio di adattamento al nuovo e a una buona diversificazione sul mercato, i ricavi dell'azienda hanno avuto un incremento proporzionale alla sua crescita.

### 1.2 CONTESTO APPLICATIVO

L'azienda supporta le esigenze di innovazione di tutte le organizzazioni per ogni settore di mercato nell'ambito Information Technology (IT), fornendo servizi negli ambiti di consulenza al business, finanziamento ai progetti e consulenza ai settori dell'IT.

Sync Lab ha come punti di forza la qualità certificata dei servizi offerti (ISO 9001, ISO 14001, ISO 27001, OHSAS 18001) e un'accurata gestione delle risorse umane.

L'approfondita conoscenza di processi e tecnologie, maturata in esperienze altamente significative e qualificanti, fornisce l'esperienza e le competenze necessarie per gestire progetti di elevata complessità, dominando l'intero ciclo di vita del software: studio di fattibilità, progettazione, implementazione, gestione e manutenzione.

L'offerta di consulenza specialistica trova l'eccellenza nella progettazione di architetture software avanzate, siano esse per applicativi di dominio, per sistemi Business Support System (BSS), per sistemi di integrazione (ad esempio Enterprise Application Integration (EAI), e Service-Oriented Architecture (SOA, si veda la sezione [3.3 a pagina 16](#)) o per sistemi di controllo applicativo/territoriale. Il laboratorio Research and Development (R&D) dell'azienda è sempre al passo con i nuovi paradigmi tecnologici e di comunicazione, ad esempio *Big Data* (raccolta di dati talmente estesa in termini di velocità, volume e varietà da richiedere apposite tecnologie e metodi per l'estrazione), *Cloud Computing* (erogazione di servizi offerti a un cliente da un fornitore attraverso il Web), Internet of Things (IoT), Mobile e Sicurezza IT, per supportare i propri clienti nella creazione e nell'integrazione di applicazioni, processi e dispositivi.

#### 1.2.1 Servizi e prodotti offerti

Sync Lab si sta sempre più specializzando in vari settori d'impiego: dal mondo delle banche alle assicurazioni con una nicchia importante nell'ambito sanità in cui vanta un prodotto d'eccellenza per la gestione delle cliniche private.

Alcuni prodotti offerti da Sync Lab sono elencati in seguito.

**SYNCLINIC** Sistema informativo sanitario per la gestione integrata di tutti i processi clinici e amministrativi di ospedali, cliniche e case di cura. Gestisce, organizza e controlla tutte le fasi del percorso di cura del paziente.

**STREAMCRUSHER** È in grado di raccogliere, indicizzare ed interpretare l'enorme mole di dati che un centro di elaborazione dati di un'organizzazione è in grado di generare ogni giorno: log di applicazioni, log di sistema, avvisi, *clickstream data* (analisi del flusso di navigazione degli utenti), data feeds, dati di configurazione, modifiche ai file system, code di messaggi, eccetera. Da questi dati StreamCrusher estrapola informazioni utili all'IT management per guadagnare nuovi livelli di visibilità e scoprire nuove opportunità di business.

**DPS 4.0** È l'applicazione Web che supporta titolari, responsabili del trattamento e Data Protection Officier (DPO) nel governo della compliance della privacy General Data Protection Regulation (GDPR) nel rispetto del principio di *accountability* (principio di responsabilità, da parte degli amministratori che impiegano risorse finanziarie pubbliche, di rendicontarne l'uso sia sul piano della regolarità dei conti sia su quello dell'efficacia della gestione.).

L'azienda inoltre ha recentemente fondato una collegata Sync Security che si occupa espressamente del mondo della sicurezza informatica in generale.

### 1.2.2 Ricerca e sviluppo

Uno degli obiettivi aziendali è quello di dar vita a un nucleo di professionisti di alto profilo occupati in tematiche di ricerca avanzata. L'azienda collabora con diverse università, tra cui l'Università degli Studi di Padova, il Politecnico di Milano, la Sapienza di Roma e l'Università degli Studi di Napoli, portando avanti diversi progetti di ricerca.

**BIG-ASC** Big Data and Advanced Analytics for Secure Mobile Commerce ha come obiettivo la creazione di una piattaforma Big Data che sappia rispondere a requisiti stringenti delle piattaforme di *Mobile Commerce* (capacità di gestire il commercio elettronico attraverso l'uso di un dispositivo mobile), come scalabilità, autonomia e performance.

**EHEALTHNET** Un progetto per servizi di *eHealth* (complesso delle risorse, soluzioni e tecnologie informatiche di rete applicate alla salute ed alla sanità) con un approccio orizzontale, dove è possibile connettere tutti gli attori della sanità, abilitando lo scambio di informazioni.

**SECURE CLOUD** Un ecosistema di servizi cloud caratterizzato da garanzie di sicurezza superiori, che forniscono protezione dagli attacchi di utenti privilegiati (ad esempio il provider cloud o l'amministratore di sistema) e software (ad esempio l'*hypervisor* (componente centrale di un sistema basato sulle macchine virtuali)).

## 1.3 ORGANIZZAZIONE DEL TESTO

Per facilitare la lettura del documento ho adottato le seguenti norme tipografiche:

- i termini ambigui o di uso non comune vengono definiti la prima volta che vengono usati nel documento tra parentesi tonde e sono evidenziati in corsivo;
- gli acronimi vengono anch'essi definiti la prima volta e riportati tra parentesi tonde, successivamente vengono usati normalmente;
- il nero viene usato solamente per i titoli, per le intestazioni e per i termini da definire in una lista di definizioni, così come il maiuscoletto;
- il blu viene usato per evidenziare collegamenti all'interno del documento, mentre i collegamenti ipertestuali sono di colore rosso in stile dattilografico;
- le cifre dei numeri indicanti una quantità aritmetica esatta vengono scritte in maiuscolo, così come le cifre di norme e requisiti.



# 2 | DESCRIZIONE

In questo capitolo descrivo il progetto di stage, gli obiettivi relativi a esso e le tecnologie che mi sono state utili nel portare a termine il lavoro.

## 2.1 INTRODUZIONE AL PROGETTO

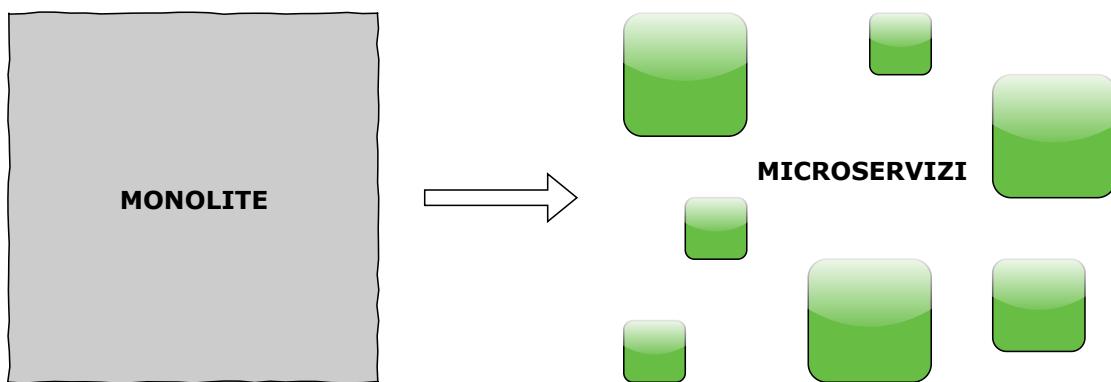


Figura 2: Obiettivo: dal monolite ai microservizi

Il progetto di stage SyncRec è stato proposto dall'azienda Sync Lab per migliorare l'attuale portale già esistente usato nella registrazione dei profili di nuovi candidati interessati a svolgere un percorso formativo all'interno dell'azienda.

Il vecchio portale Skill Management System (SMS) consente di amministrare i profili delle persone interessate a un colloquio ricavando alcune informazioni partendo da un modulo compilato online manualmente dal personale di Sync Lab. Altre informazioni vengono aggiunte successivamente partendo da un foglio di calcolo inviato tramite email e compilato dal candidato stesso.

Dato il gran numero di persone ospitate dall'azienda in percorsi di formazione, sono richiesti molti controlli da parte del personale dell'azienda, dato che per ogni candidato è necessario l'intervento da parte del personale per riportare i dati dal foglio di calcolo al portale online.

Un'altra considerazione opportuna è che tutte le operazioni (registrazione del candidato, invio delle email, inserimento dei dati del profilo, logging) avvengono in un ambiente centralizzato, portando a un ambiente difficilmente gestibile e poco scalabile.

Per rimediare a questi inconvenienti l'azienda ha iniziato lo sviluppo di un'applicazione Web con architettura a microservizi (si veda la sezione [3.1.2 a pagina 14](#)), in modo da alleggerire il carico di lavoro sulla piattaforma. I candidati potranno inserire i dati direttamente dal portale, senza dover richiedere l'intervento del personale di Sync Lab.

La progettazione e lo sviluppo dell'applicazione è stata suddivisa tra più tirocinanti. La mia parte consiste nella progettazione e nello sviluppo di alcuni dei microservizi e all'applicazione dei pattern più comuni adottati in tale tipo di architettura.

Per poter familiarizzare con le tecnologie lo stage è stato suddiviso in due periodi. Il primo periodo incentrato sullo studio teorico dell'architettura a microservizi e dei *design pattern* (descrizione o modello logico da applicare per la risoluzione di un problema che può presentarsi in diverse situazioni durante le fasi di progettazione e sviluppo del software) usati, mentre il secondo incentrato sullo studio delle tecnologie usate per la realizzazione dei microservizi e sull'implementazione.

## 2.2 OBIETTIVI

Gli obiettivi del progetto, definiti all'inizio dello stage con Fabio Pallaro, sono stati suddivisi in obbligatori, desiderabili e facoltativi. Gli obiettivi obbligatori sono da portare a termine perché il progetto possa essere considerato sufficiente, mentre gli obiettivi desiderabili e facoltativi non sono considerati fondamentali per la conclusione dello stage, ma concorrono a ottenere una preparazione migliore nell'ambito dei microservizi e un progetto immediatamente usabile da Sync Lab.

### 2.2.1 Obiettivi obbligatori

- Acquisizione di competenze sull'architettura a microservizi e sulle tecnologie usate per lo sviluppo del progetto.
- Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il crono programma.
- Portare a termine le funzionalità richieste dal cliente con una percentuale di superamento degli item di collaudo pari al 50%, in particolare relativamente a:
  1. microservizio document;
  2. microservizio rabbitmq.

### 2.2.2 Obiettivi desiderabili

- Acquisire competenze su Spring Cloud.
- Portare a termine le funzionalità richieste dal cliente con una percentuale di superamento degli item di collaudo pari all' 80%, in particolare relativamente a:
  1. microservizio logging.

### 2.2.3 Obiettivi facoltativi

- Implementazione delle funzionalità di Service Registry e Service Discovery mediante Spring Cloud, quindi:
  1. microservizio eureka;
  2. microservizio apigateway.

## 2.3 REQUISITI

Partendo dagli obiettivi esposti inizialmente nel piano di lavoro, Fabio Pallaro e io abbiamo stabilito alcuni requisiti per i diversi microservizi da implementare. Così come gli obiettivi, i requisiti sono stati suddivisi in obbligatori, desiderabili e facoltativi. Riporto in seguito alcuni dei requisiti individuati per tipologia.

### 2.3.1 Requisiti obbligatori

- RO01: i documenti memorizzati nel servizio document devono avere un nome e il contenuto del file
- RO02: è possibile aggiungere tipi di documenti diversi
- RO03: il servizio può restituire una lista con tutti i documenti
- RO04: il servizio permette la ricerca di un documento per id
- RO05: è possibile eliminare un documento per id
- RO06: è possibile modificare il nome e il contenuto di un documento, identificato da un codice univoco
- RO07: deve essere restituito un errore in caso il documento non possa essere memorizzato
- RO08: deve essere restituito un errore in caso il documento non possa essere eliminato
- RO09: deve essere restituito un errore in caso il documento non possa essere modificato
- RO10: deve essere restituito un errore in caso venga cercato un documento inesistente
- RO11: il servizio rabbitmq deve gestire una o più code di messaggi
- RO12: il servizio rabbitmq deve gestire uno o più inviatori di messaggi
- RO13: il servizio rabbitmq deve gestire uno o più scodatori di messaggi, ognuno iscritto a una coda
- RO14: deve essere restituito un errore nel caso il messaggio non possa essere inoltrato dagli scodatori

### 2.3.2 Requisiti desiderabili

- RD01: deve essere possibile capire quale microservizio ha effettuato la richiesta di log
- RD02: il log deve essere tracciabile per tipo di messaggio
- RD03: il log deve essere tracciabile per tempo di memorizzazione del messaggio di log
- RD04: ogni microservizio deve poter richiedere la memorizzazione di un messaggio di log

### 2.3.3 Requisiti facoltativi

- RF01: i microservizi devono registrarsi sul service registry Eureka
- RF02: i microservizi devono poter essere raggiungibili attraverso il microservizio apigateway

## 2.4 TECNOLOGIE

Per la progettazione e lo sviluppo della piattaforma ho preso in studio le tecnologie attualmente più utilizzate nello sviluppo di *backend* (parte che permette l'effettivo funzionamento delle iterazioni che un utente compie attraverso la parte frontend di un'applicazione) a microservizi e *frontend* (parte dell'applicazione con cui l'utente può interagire) per applicazioni Web.

### 2.4.1 Tecnologie per i microservizi

#### *Java SE*

Si tratta della versione standard del linguaggio di programmazione Java orientato agli oggetti. Questa è stata presa in ripasso partendo dagli appunti presi dai corsi di Object Oriented Programming (OOP) e Programmazione Concorrente e Distribuita (PCD) e dal libro Effective Java di Joshua Bloch.

Ho usato Java perché il framework Spring è basato su questo linguaggio di programmazione.



Figura 3:  
Logo Java

#### *Spring*

Spring è un *framework* (implementazione logica di uno o più design pattern su cui un software può essere progettato e realizzato) Java che facilita lo sviluppo di applicazioni enterprise. È composto da molti moduli che possono essere inclusi in un progetto, moduli indipendenti tra loro che consentono di aggiungere funzionalità diverse senza includere tutto il framework.

Spring adotta delle configurazioni che possono essere scritte in molti modi diversi, a seconda delle preferenze dello sviluppatore. La stessa configurazione può essere scritta in un file xml, oppure in un file di *properties* (proprietà descritte in una gerarchia che possono avere sottoproprietà), in un file *yaml* (formato simile alle properties, ma più compatto) oppure all'interno del codice sorgente attraverso annotazioni Java. Queste configurazioni consentono allo sviluppatore di descrivere alcuni comportamenti necessari per permettere al framework di gestire l'applicazione.



Figura 4:  
Logo Spring

### **Spring Core**

Questa tecnologia è la parte centrale del framework Spring. Consente di sviluppare applicazioni sfruttando il principio della *dependency injection* (design pattern della OOP dove si trovano dichiarate le dipendenze di cui un componente necessita per facilitarne sviluppo e testabilità), secondo la quale le dipendenze delle classi vanno esposte verso l'esterno, in modo da rendere facile l'individuazione delle dipendenze tra classi ed evitare le dipendenze nascoste.

Spring porta all'estremo questo principio attraverso l'inversione di controllo. L'inversione di controllo permette al framework stesso di gestire la vita degli oggetti che fanno parte del contesto dell'applicazione, quindi è Spring che istanzia gli oggetti quando è necessario e li inietta nelle classi che hanno una dipendenza verso di essi.

### **Spring MVC**

Il modulo Model View Controller (MVC) di Spring consente di realizzare il pattern MVC attraverso poche annotazioni, in modo da separare le funzionalità esposte all'esterno (View) dal core business dell'applicativo (Controller) e dai dati su cui si opera (Model).

### **Spring Boot**

Questo modulo segue il principio di convention over configuration.

Attraverso poche annotazioni è possibile avere un'applicazione funzionante che usa molte configurazioni di default, in modo da dover scrivere meno codice e usare le migliori caratteristiche di Spring, evitando di reinventare la ruota.



Figura 5:  
Logo Spring  
Boot

### **Spring Data MongoDB**

Tale modulo consente al framework Spring di usare il database Mongo tramite una *repository* (ambiente di un sistema informativo in cui vengono gestiti metadati). Tale repository fornisce molte funzionalità e serve per far corrispondere una classe Java che rappresenta un modello con la struttura logica del database. È possibile aggiungere funzionalità alla repository, ad esempio query personalizzate, semplicemente scrivendo metodi nell'interfaccia con nomi che rispettino determinate convenzioni, oppure è possibile definire query complesse tramite annotazioni Java.

### **Spring Data REST**

Attraverso questo modulo Spring fornisce automaticamente la logica per le chiamate Representational State Transfer (REST) verso una risorsa rappresentata attraverso un modello. È sempre possibile modificare il comportamento predefinito o aggiungere ulteriori operazioni attraverso Spring MVC.

### **Spring Cloud Gateway**

Si tratta di un modulo Spring che consente la creazione di un API gateway (si veda la sezione [3.4.1 a pagina 18](#)), ovvero di un'interfaccia usabile da tutte le applicazioni interne o esterne per trovare servizi di cui necessitano per il funzionamento. Permette pertanto di realizzare il pattern omonimo in un'architettura a microservizi in modo semplice e immediato.



Figura 6:  
Logo Spring  
Cloud

### *Spring Cloud Netflix Eureka*

Questo modulo Spring consente di creare un registro con tutti i servizi di un'applicazione composta da più servizi. Nell'architettura a microservizi rende concreto il pattern service registry/discovery (si veda la sezione [3.4.2 a pagina 19](#)).

### *Spring Cloud Circuit Breaker*

È un modulo di recente concezione che va a sostituire Spring Cloud Netflix Hystrix. Serve per realizzare un circuit breaker (si veda la sezione [3.4.3 a pagina 20](#)), che è una componente che interviene in caso un microservizio richiesto per il funzionamento dell'applicazione non sia attivo o non venga trovato. Quando si verifica una situazione eccezionale, il circuit breaker si attiva evitando ripercussioni sugli altri microservizi e segnalando che qualcosa all'interno del sistema non funziona correttamente.

### *Spring AMQP*

Con questo modulo Spring rende gestibile il *broker* (software che mette in comunicazione un client con un oggetto remoto) RabbitMQ. È possibile definire le code per i messaggi, incodatori e scodatori.

Nel progetto ho adottato il pattern publish/subscribe, in cui vi è una coda (subject) che viene riempita da un incodatore e che può essere presa in ascolto da più scodatori (observer). Nel caso siano presenti più observer dello stesso tipo, viene adottata una politica di scodamento Round Robin, quindi a turno viene scelto uno scodatore per il messaggio in elaborazione.

### *RabbitMQ*

Si tratta di un message broker che supporta il protocollo Advanced Message Queueing Protocol (AMQP). Un message broker è un software in grado di gestire messaggi tra mittenti e destinatari, consentendo di trasformare i messaggi in un linguaggio comprensibile dai destinatari partendo dal linguaggio dei mittenti. Può inoltre gestire code di messaggi a cui i destinatari possono iscriversi, offrendo quindi il supporto necessario per applicare il pattern publish/subscribe.



Figura 7:  
Logo  
RabbitMQ

### *Resilience4J*

Resilience4J è una libreria ispirata a Netflix Hystrix che consente di realizzare un circuit breaker.

Si tratta di un componente con meno dipendenze rispetto a Hystrix, inoltre è fatto apposta per essere integrato con le *espressioni lambda* (funzioni definite senza essere legate a un identificatore) introdotte in Java 8, rendendo di fatto il codice più leggibile.



Figura 8:  
Logo  
Resilience4J

## *Log4J*

Log4J è una tecnologia sviluppata da Apache per la gestione dei log in Java ed è uno standard de facto.

Nel progetto l'ho usata per gestire i log in un microservizio atto a mantenere i log di tutti gli altri e nei singoli microservizi per gestire un log locale nel caso il microservizio remoto non sia disponibile.



Figura 9:  
Logo Log4J

### 2.4.2 Tecnologie per il frontend

#### *JavaScript*

JavaScript è un linguaggio orientato agli oggetti per la programmazione Web lato client. Essendo stato ampiamente trattato nel corso di tecnologie Web, ho ripassato quanto necessario dagli appunti presi a lezione.

#### *TypeScript*

Poiché JavaScript è un linguaggio orientato agli oggetti tramite prototipazione, non mette a disposizione molte funzionalità che solitamente sono presenti in linguaggi di OOP più evoluti. Per sopperire a tali mancanze Microsoft ha sviluppato TypeScript, un linguaggio che estende JavaScript con firme dei metodi ben definite, classi, interfacce e alcuni tipi elementari.

#### *Angular*

Angular è una piattaforma per lo sviluppo frontend di applicazioni Web. È basata su TypeScript e l'elaborazione avviene lato client. Un'applicazione sviluppata in Angular è formata dalle seguenti componenti.



Figura 10:  
Logo Angular

**COMPONENTS** Definiscono la logica delle componenti visive. Ogni componente è formato da una classe, un template HTML e un foglio di stile CSS.

**SERVICES** Sono servizi trasversali alle viste, che possono essere usati per diversi scopi, ad esempio per la validazione di dati o per fare richieste al backend.

**MODULES** I moduli di Angular possono includere più componenti e servizi, e vengono usati per delineare il dominio del codice.

Inoltre Angular fornisce il supporto per template HTML intelligenti, dove è possibile definire strutture di controllo direttamente nel codice, ad esempio istruzioni condizionali e ciclomatiche, ed è possibile basarsi su proprietà degli oggetti, ad esempio per verificare se un input è stato modificato o semplicemente attivato dall'utente.

### 2.4.3 Altre tecnologie

#### *MongoDB*

Il Database Management System (DBMS) Mongo è una piattaforma che permette di gestire database Not Only SQL (NoSQL) basati sui documenti. Non esiste il concetto di relazione tra entità, che devono essere pertanto realizzate manualmente, ad esempio memorizzando in un campo l'identificativo del documento correlato. Le basi di dati a documenti sono molto efficienti in caso di memorizzazione di dati distribuita tra più microservizi. Nel progetto pertanto non vi è un unico database, ma ogni microservizio ha il proprio database.



Figura 11:  
Logo MongoDB

#### *JavaServer Pages (JSP) e Servlet*

Tali tecnologie sono state prese in considerazione puramente dal punto di vista teorico, in modo da poter comprendere il funzionamento del vecchio sistema SMS, realizzato interamente in JSP.

#### *Gradle*

Gradle è uno strumento di *build automation* (l'atto di automatizzare un'ampia varietà di compiti, come la compilazione, l'esecuzione dei test, il rilascio in sistemi di produzione) basato su Maven. Viene quindi usato per gestire le dipendenze di un progetto e per stabilire alcune azioni che vengono automaticamente svolte in alcune fasi del ciclo di vita dell'applicazione. Nel progetto l'ho usato in tutti i microservizi in modo da risolvere automaticamente le dipendenze per Java e Spring.



Figura 12:  
Logo Gradle

#### *Git*

Git è un sistema per il controllo del versionamento del codice sorgente. Serve per tenere traccia di tutte le modifiche fatte al codice, in modo da sapere a ogni commit cosa è stato modificato rispetto al precedente. Essendo un sistema distribuito è possibile apportare molte modifiche in locale, sincronizzando il lavoro con una repository remota solo all'occorrenza.



Figura 13:  
Logo Git

# 3 | STUDIO TEORICO

scrivo in seguito alcune nozioni teoriche e i design pattern che ho usato per capire in modo più approfondito lo scopo del progetto e come raggiungerlo al meglio.

## 3.1 LINEE GENERALI

### 3.1.1 Architettura software

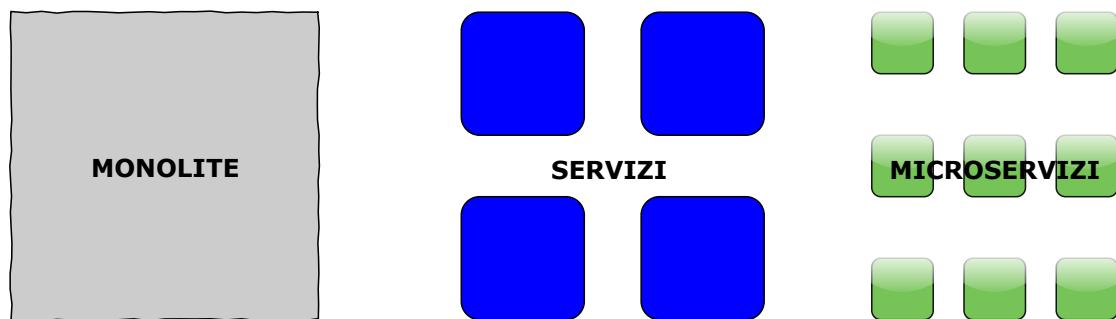


Figura 14: Esempi di architettura a confronto

Un'architettura software è un modello di organizzazione delle componenti di un sistema software. L'architettura di un software determina come le componenti sono organizzate, come interagiscono tra loro e con l'esterno. La decisione di quale architettura software scegliere influisce sulla progettazione, lo sviluppo e la verifica delle componenti.

### 3.1.2 Architettura a microservizi

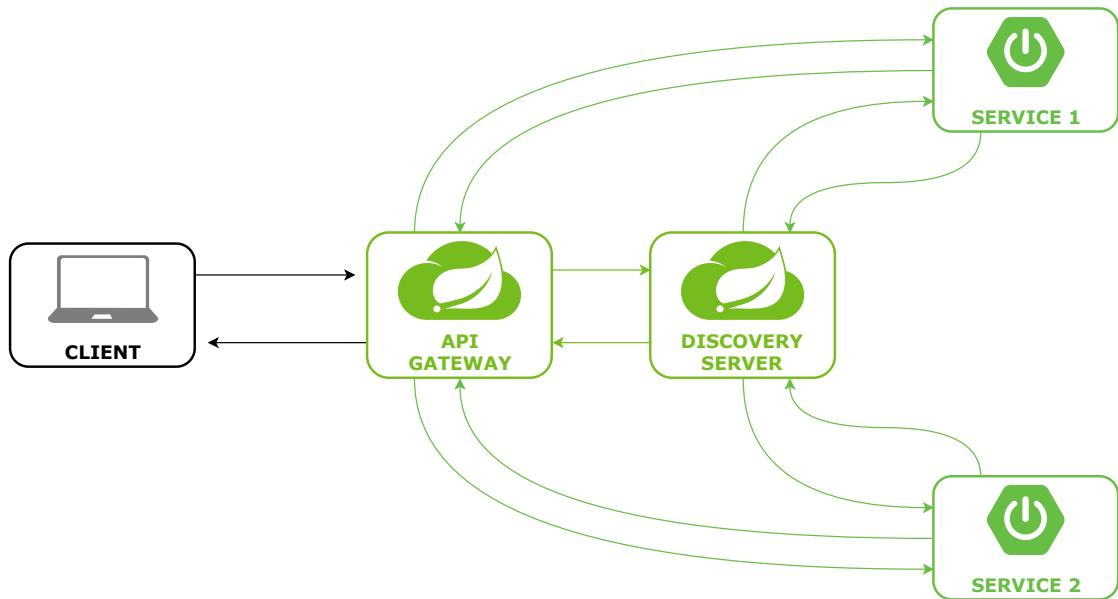


Figura 15: Architettura a microservizi

Una definizione di architettura a microservizi è la seguente:

[...]insieme di piccoli servizi, ognuno dei quali esegue in un processo separato, che comunicano attraverso messaggi (ad esempio chiamate HTTP attraverso un'Application Programming Interface (API)). Tali servizi vengono progettati attorno alle capacità di business e sono rilasciabili in modo indipendente [...] [Fowler 2014]

Questa architettura segue il single responsibility principle, secondo cui ogni componente di un'architettura software deve essere il più indipendente possibile dalle altre componenti e deve occuparsi solamente delle sue funzionalità.

Le caratteristiche di un'architettura a microservizi sono descritte in seguito [Fowler e Lewis 2014].

**COMPOSIZIONE DI SERVIZI** Ogni componente è un'unità del software, indipendente dalle altre, rimpiazzabile e aggiornabile. Tali servizi devono esporre un'interfaccia per essere utilizzati (API).

**ORGANIZZAZIONE INCENTRATA SUL BUSINESS** Incentrando ogni componente su un aspetto del business indipendente dagli altri è possibile organizzare il lavoro in piccoli team di sviluppo indipendenti.

**ENDPOINT INTELLIGENTI, CANALI FUTILI** Non esiste alcuna organizzazione centralizzata delle comunicazioni tra i microservizi, i quali si appoggiano a un sistema di comunicazione che deve solamente inoltrare i messaggi.

**GESTIONE DECENTRALIZZATA** Poiché non esiste una soluzione perfetta per qualsiasi tipo di problema, nell’architettura a microservizi ogni componente può adottare tecnologie diverse e linguaggi diversi. Tale caratteristica è riscontrabile nelle architetture in cui i contratti tra microservizi sono ben strutturati e si adottano pattern come il *tolerant reader* (pattern che prevede un lettore tollerante, ovvero che legge solamente i campi di cui ha necessità, senza imporre obblighi non necessari sulla struttura dei messaggi [Fowler 2011]) e il *consumer-driven contracts* (pattern in cui i consumatori delle funzionalità determinano tutte quelle che devono essere fornite dai produttori, e come esse devono essere strutturate).

**GESTIONE DEI DATI DECENTRALIZZATA** Ogni microservizio gestisce i dati di cui è responsabile. Per risolvere i problemi legati alla possibile inconsistenza dei dati viene adottato il saga pattern (si veda la sezione [3.4.5 a pagina 22](#)), oppure accettando uno stato di eventuale consistenza dei dati.

**PROGETTAZIONE PER IL FALLIMENTO** I microservizi devono avere sistemi per conoscere lo stato del sistema e per fornire dati relativi alle operazioni che accadono (logging). Viene adottato in questo caso il circuit breaker pattern (si veda la sezione [3.4.3 a pagina 20](#)).

## 3.2 COMPARAZIONE CON L'APPROCCIO MONOLITICO

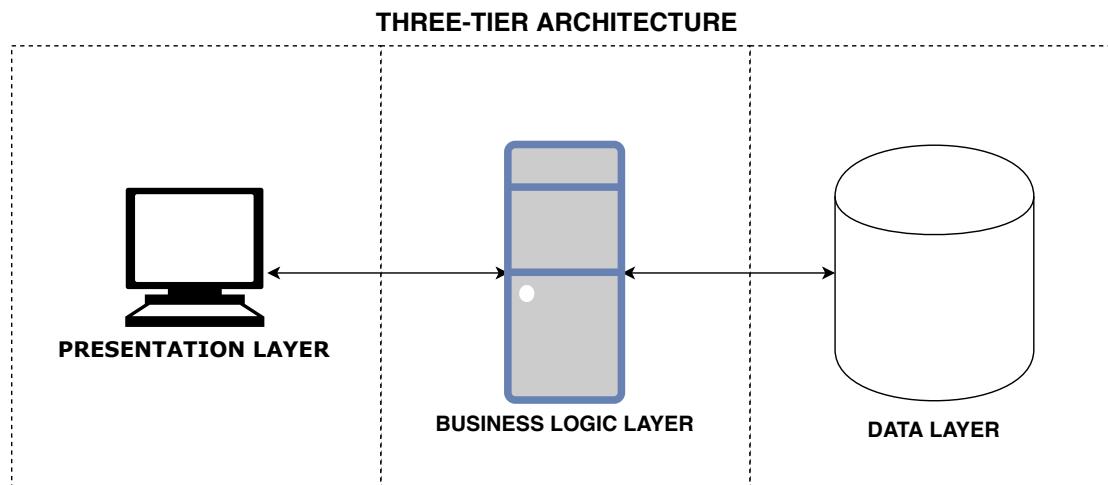


Figura 16: Architettura monolitica

L’architettura monolitica, anche detta multi-tier, suddivide le funzionalità di un software in più strati logici divisi e indipendenti tra loro [Wikipedia 2019]. Solitamente questi strati logici in un’architettura three-tier sono:

- presentation layer;
- business logic layer;
- data layer.

In tale architettura non vi è una netta suddivisione di servizi, bensì un’organizzazione monolitica suddivisa per funzione.

Caratteristica	Architettura monolitica	Architettura a microservizi
<b>Sviluppo</b>	Facile, è tutto in un blocco unico.	Facile nel microservizio. Difficile nell'architettura.
<b>Rilascio</b>	Difficile, bisogna rilasciare il monolite ogni volta.	Facile, ogni microservizio è rilasciabile indipendentemente dagli altri.
<b>Tecnologie</b>	Difficile, bisogna integrare nuove tecnologie con altre già in uso, non è sempre possibile.	Facile, essendo ogni microservizio indipendente si possono integrare molte tecnologie.
<b>Distribuzione</b>	Essendo un monolite, non vi è distribuzione.	È difficile orchestrare i diversi microservizi in modo che cooperino.
<b>Consistenza</b>	Facile, la consistenza dei dati è sempre garantita.	Non è possibile garantire che i dati siano sempre consistenti tra tutti i microservizi.
<b>Testabilità</b>	Facile, si ha il controllo sull'intera architettura.	Facile nel singolo microservizio, difficile per i test di integrazione.
<b>Scalabilità</b>	Difficile, bisogna replicare il monolite e usare un load balancer per gestire le richieste.	Facile, è possibile replicare solamente i microservizi che vengono più utilizzati.

Tabella 1: Confronto tra monolite e microservizi [Fowler 2015]

### 3.3 COMPARAZIONE CON LE ARCHITETTURE ORIENTATE AI SERVIZI

Nell'architettura orientata ai servizi vi è una suddivisione delle componenti simile a quella usata nell'architettura a microservizi.

[...]Essa è sempre incentrata sul business dei servizi. La differenza principale è che mentre nell'architettura a microservizi le componenti sono il più separate possibile, nell'architettura orientata ai servizi le componenti condividono quanta più logica possibile, quindi mette in risalto l'astrazione e il riuso delle funzionalità già implementate. [Richards 2016] [...]

In questa architettura è presente un componente che funge da orchestratore, un Enterprise Service Bus (ESB, infrastruttura software che fornisce servizi di supporto ad architetture Service-Oriented Architecture (SOA) complesse, agendo come una dorsale attraverso la quale viaggiano servizi software e componenti applicativi [*Enterprise service bus 2019*]), che serve per pubblicare i servizi, gestire la sicurezza, trasformare i dati tra i servizi e supportare le regole di business. La differenza principale tra le due architetture risiede nell'Enterprise Service Bus (ESB). Pur consentendo il disaccoppiamento dei contratti tra i servizi e l'interoperabilità tra servizi eterogenei, l'ESB rischia di diventare un componente molto complesso che di fatto nasconde quel che si tentava di eliminare dalle architetture monolitiche.

Architettura orientata ai servizi	Architettura a microservizi
Condivisione massima delle funzionalità	Condivisione minima delle funzionalità
Controllo e standard in comune	Collaborazione delle persone, libertà per altri aspetti
Uso di ESB	Sistema di messaggistica
Protocolli di comunicazione eterogenei	Protocolli di comunicazione leggeri (REST)
Focus sul riuso	Focus sull'indipendenza
Un cambiamento importante potrebbe richiedere la modifica di codice condiviso o dell'ESB	Un cambiamento importante non ha ripercussioni sugli altri microservizi

Tabella 2: Confronto tra servizi e microservizi [Despodovski 2017]

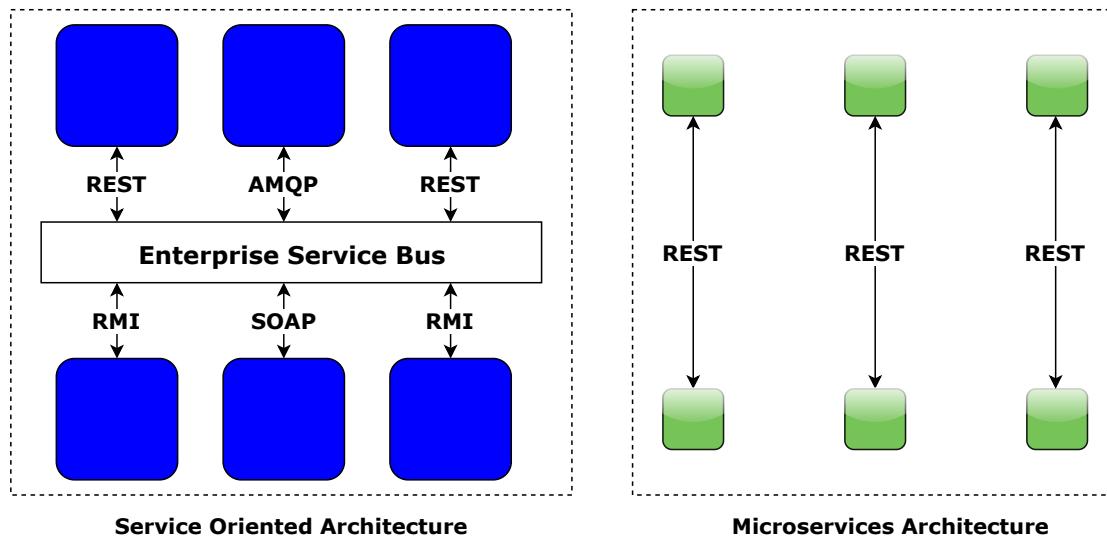


Figura 17: Confronto tra architettura SOA e architettura a microservizi per protocolli di comunicazione



- (a) I servizi in un'architettura SOA sono come i musicisti di un'orchestra, indipendenti ma coordinati da un orchestratore
- (b) I microservizi invece sono come le ballerine di una coreografia, che si coordinano senza aiuti esterni

Figura 18: Confronto tra architettura SOA e architettura a microservizi per metodo di coordinamento

### 3.4 PATTERN UTILIZZATI

Per progettare un sistema con un'architettura a microservizi è utile usare a proprio vantaggio le best practice attualmente più diffuse, perché raccolgono l'esperienza e la conoscenza raccolte nel corso degli anni. In questo modo si evita di progettare un'architettura scadente e di reinventare principi e tecniche già note. Per raggiungere questo obiettivo nel progetto ho preso in studio i seguenti design pattern per architetture a microservizi.

#### 3.4.1 API Gateway

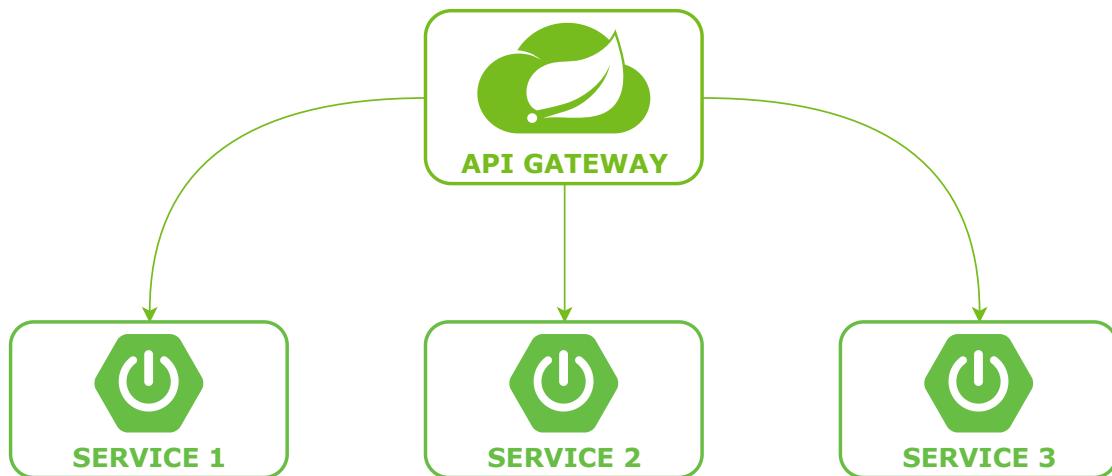


Figura 19: Esempio di API gateway attraverso cui chiamare diversi microservizi

Questo pattern viene adottato per uniformare il punto di ingresso verso i vari microservizi all'intera applicazione e all'esterno. Un API gateway è quindi un componente che serve per esporre un'interfaccia comune a tutti per raggiungere i diversi microservizi offerti dall'architettura. Le funzionalità offerte sono descritte in seguito [Bhojwani 2018].

**ROUTING DELLE RICHIESTE** Un API gateway serve per inoltrare le richieste verso il microservizio corretto, fungendo da router.

**COMPOSIZIONE DELLE API** A fronte di una richiesta, un API gateway può sfruttare diversi microservizi per portarla a termine.

**TRADUZIONE DI PROTOCOLLO** Se un microservizio non riconosce richieste REST, è possibile che l'API gateway fornisca una traduzione verso il protocollo di comunicazione usato dal microservizio.

**API IN BASE AL CLIENT** Un API gateway può fornire diverse interfacce in base al dispositivo o all'applicazione da cui riceve la richiesta.

È necessario mantenere l'API gateway il più leggero possibile, in modo da non renderlo un sostituto di un ESB.

### 3.4.2 Service Registry/Discovery

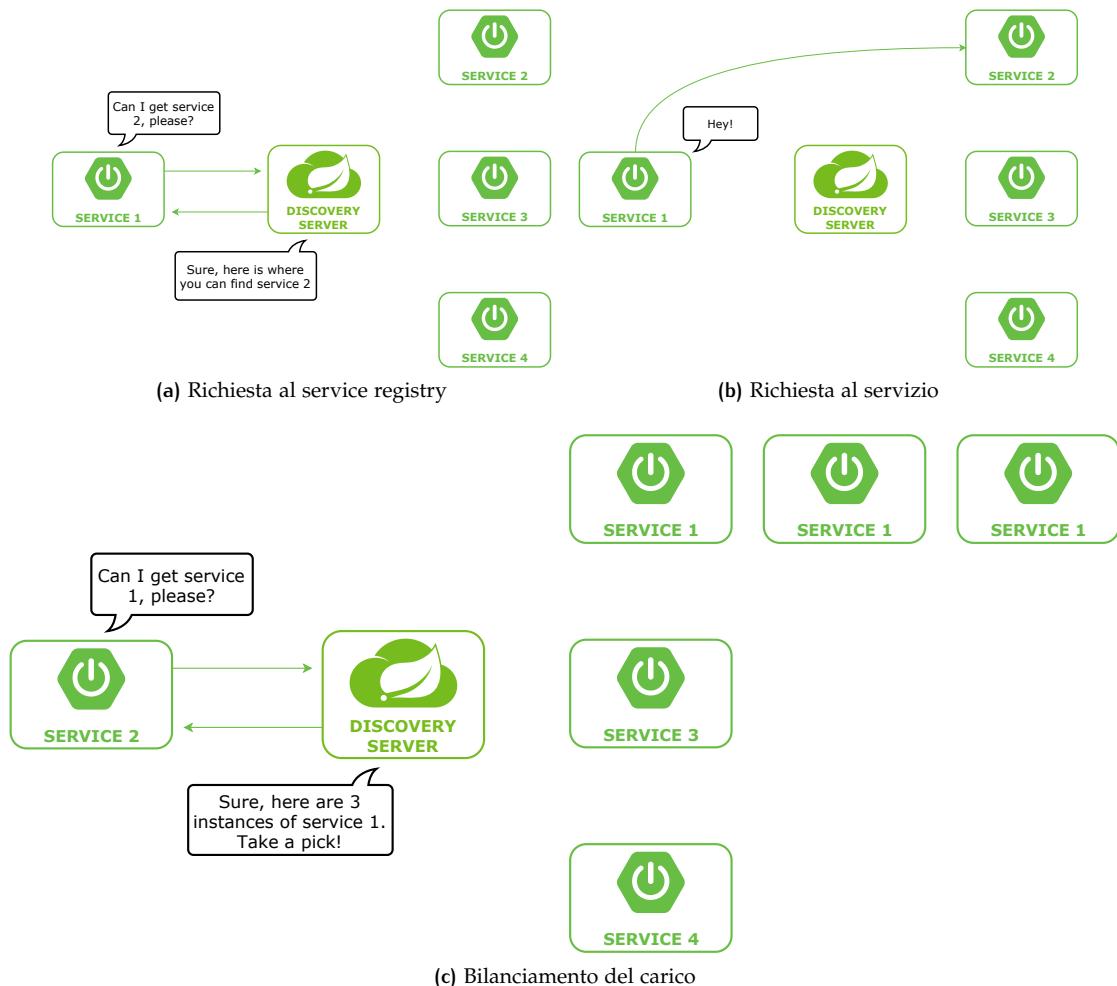


Figura 20: Come funziona il service discovery

Con questo pattern si vuole rendere l'architettura robusta rispetto a possibili modifiche della rete in cui interagiscono i microservizi. Ogni istanza di ogni microservizio è caratterizzata da un *socket* (insieme di indirizzo IP e numero di porta per identificare un'istanza del microservizio) diverso, inoltre trattandosi di un'architettura a microservizi è possibile che vi siano molteplici istanze di un microservizio e che alcuni microservizi vengano spenti o incorrano in errori. Quindi un service registry consente di avere sempre sotto controllo la rete dei microservizi, indirizzando le richieste verso l'istanza corretta.

Poiché non vi è modo di sapere quante istanze e quali microservizi saranno presenti nell'applicazione, viene usato il self registration pattern, in cui un microservizio conosce la locazione del service registry e provvede a registrarsi in esso. Successivamente la discovery dei servizi può avvenire in due modalità:

**CLIENT SIDE DISCOVERY** Un microservizio riceve una lista con le istanze del microservizio che deve usare, poi sceglie quale usare adottando una distribuzione del carico tra le varie istanze.

**SERVER SIDE DISCOVERY** Un microservizio manda la richiesta del microservizio che deve usare a un server che è responsabile per la scelta del microservizio corretto.

### 3.4.3 Circuit Breaker



**Figura 21:** Un circuit breaker è come un interruttore

Un’architettura a microservizi deve essere robusta rispetto a errori che possono occorrere in uno dei suoi componenti. Il problema di tale architettura è che essendo formata da componenti che comunicano in rete potrebbe non rilevare un errore dovuto a un rallentamento, o a una mancata risposta. Un circuit breaker, così come un temporizzatore in un circuito elettrico, interviene dopo un certo lasso di tempo o dopo un certo numero di richieste non andate a buon fine, in modo da evitare la creazione di uno stallo in tutta l’applicazione e se possibile apporre le opportune correzioni [Bhojwani 2018]. Le funzionalità offerte sono:

**INTERRUZIONE DEL CIRCUITO** Viene interrotto il tentativo di portare a termine un’azione senza successo e viene segnalato il fallimento.

**RIPRISTINO DELLO STATO DEL SISTEMA** Vengono iniziate le azioni necessarie per riportare il sistema in uno stato coerente.

#### 3.4.4 Database per service

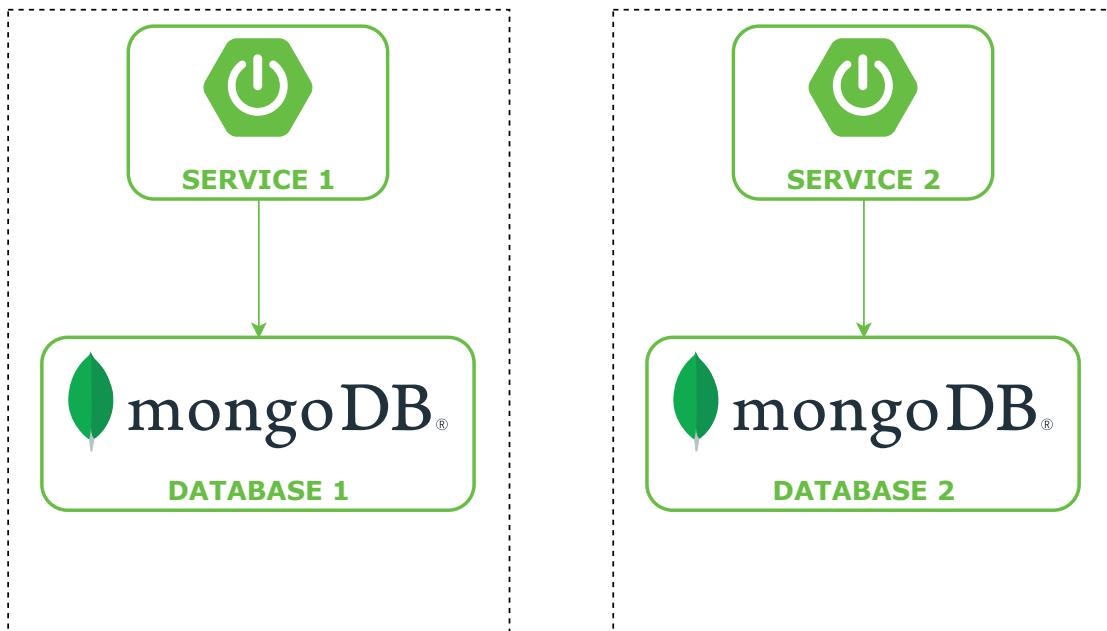


Figura 22: Esempio di database per service

Come illustrato sopra, i microservizi devono condividere meno risorse possibili. Considerando che ogni microservizio deve amministrare dati di business diversi e che ciascuno può avere esigenze di scalabilità diverse, è utile applicare questo design pattern, che prevede l'utilizzo di singoli database gestiti all'interno dei microservizi [Richardson 2019a]. Ogni microservizio potrà accedere ai dati di un altro solamente sfruttando le funzionalità da esso esposte, senza poter interagire con la sua base di dati. Questo permette inoltre di adottare tecnologie diverse per gestire i dati di ogni singolo microservizio.

### 3.4.5 Saga

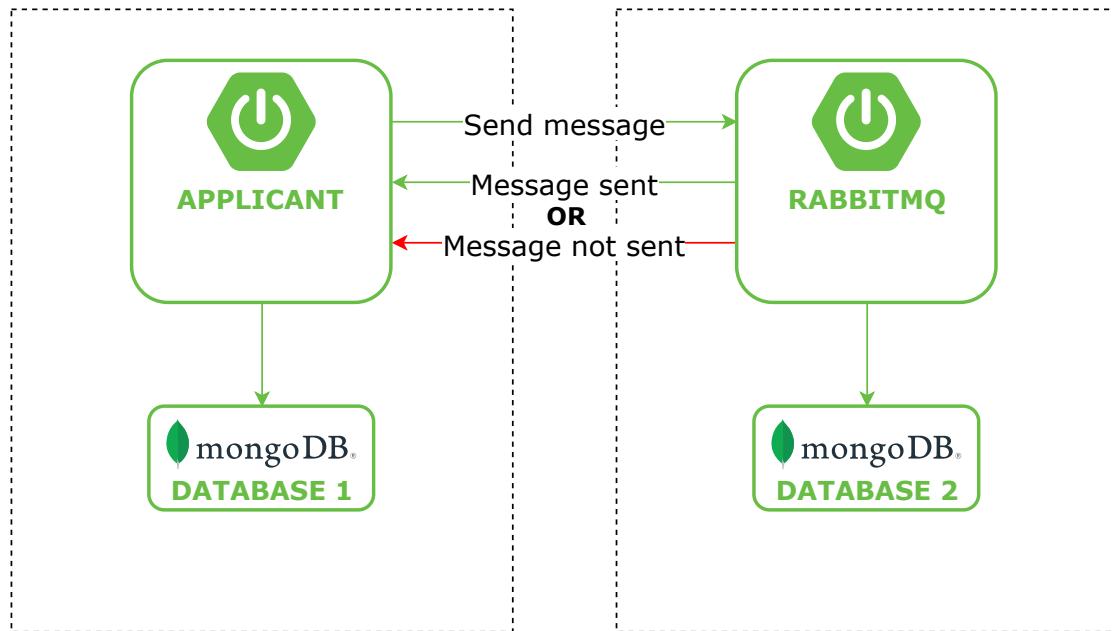


Figura 23: Esempio di saga pattern con coreografia

Questo pattern viene introdotto come conseguenza del database per service pattern. Poiché ogni microservizio gestisce il proprio database, risulta impossibile garantire le proprietà Atomicity, Consistency, Isolation, Durability (ACID) tutte insieme, solitamente presenti in un database centralizzato.

Tali proprietà sono:

**ATOMICITY** Una transazione è indivisibile nella sua esecuzione e la sua esecuzione deve essere o totale o nulla.

**CONSISTENCY** Quando inizia una transazione il database si trova in uno stato coerente e quando la transazione termina il database deve essere in un altro stato coerente (non devono verificarsi contraddizioni tra dati memorizzati nel database).

**ISOLATION** Ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione.

**DURABILITY** I cambiamenti apportati da una transazione non dovranno essere più persi.

Le proprietà ACID rimangono rispettate a livello di microservizio, ma non è possibile mantenerle tutte in un sistema distribuito, se non attraverso meccanismi complessi e non sempre supportati.

Per garantire la consistenza dei dati tra i vari microservizi il saga pattern offre una soluzione semplice, ovvero usa delle transazioni di compensazione per una o più transazioni eseguite, in modo da poter tornare a uno stato precedente del sistema [Richardson 2019b].

Vi sono due metodi per coordinare un pattern di tipo saga:

**COREOGRAPHY** Le decisioni vengono prese dai microservizi, attraverso lo scambio di messaggi, in modo da tornare a uno stato precedente nel caso il contesto lo richieda.

**ORCHESTRATION** Un orchestratore manda i messaggi ai partecipanti del saga pattern, indicando quali operazioni effettuare.

Un buon modello di saga pattern può essere progettato facendo uso di una macchina a stati, in modo da capire quali transazioni sono effettivamente necessarie per tornare a uno stato consistente.



# 4 | PROGETTAZIONE

*S*uccessivamente allo studio delle tecnologie e dei pattern ho cercato di progettare al meglio i microservizi da implementare. In seguito fornisco alcuni dettagli riguardo alla progettazione.

## 4.1 DESCRIZIONE DELLA PROGETTAZIONE

Prima di procedere con l'implementazione ho ritenuto opportuno progettare l'architettura sia a livello di singolo microservizio che a livello di comunicazione tra i microservizi che la compongono.

Per i singoli microservizi ho adottato i seguenti package ove necessario:

**CONFIG** Contiene le classi relative alle configurazioni e ai *bean* (oggetti istanziati e amministrati dall'inversione di controllo di Spring) usati da Spring;

**CONTROLLER** Contiene le classi controller gestite da Spring MVC;

**LOGGER** Contiene le classi per gestire il logging locale e remoto;

**MODEL** Contiene i modelli dei dati presenti nei database dei vari microservizi;

**REPOSITORY** Contiene i repository gestiti da Spring Data REST per fornire un'API REST a tali dati;

**RESOURCE** Contiene le classi per convertire un modello dati in una risorsa REST;

**SERVICE** Contiene servizi di utilità che possono essere usati da più classi dello stesso microservizio.

Per il microservizio rabbitmq ho adottato inoltre i seguenti package:

**EXCEPTION** Contiene i gestori delle eccezioni personalizzati per RabbitMQ;

**RECEIVER** Contiene gli scodatori di RabbitMQ;

**SENDER** Contiene gli incodatori di RabbitMQ.

Dopo aver studiato le diverse tecnologie menzionate nella descrizione del progetto, mi sono reso conto di quanto i diversi moduli di Spring potessero semplificare moltissimo la progettazione dell'intera architettura. Questo è stato possibile per i seguenti motivi:

- Le funzionalità esposte da un microservizio attraverso un'API REST sono anche quelle che generalmente devono fare qualcosa, quindi vanno implementate.
- Spring Data REST fornisce un'API REST basata su un modello dati già implementata, quindi non richiede sviluppo aggiuntivo. La progettazione si ritrova nel modello dati da usare.

- Spring MVC consente di usare il pattern MVC, in particolar modo consente di definire dei controller che possono sopperire alle funzionalità che non siano già implementate automaticamente attraverso Spring Data REST.
- Spring Data consente di abilitare controlli di base sui valori dei modelli che descrivono la struttura dati usata in MongoDB.

Dati questi assunti la progettazione dei microservizi di documenti e di logging è stata incentrata solamente nella descrizione del modello dati.

Anche la progettazione dei restanti microservizi si è ridotta notevolmente al seguito dello studio delle tecnologie offerte da Spring. Questo perché essendo i pattern service registry/discovery e API gateway molto diffusi ed essendo RabbitMQ molto utilizzato come broker per rendere asincrone le comunicazioni tra microservizi il framework Spring prevede dei moduli già progettati e implementati, che richiedono solamente l'impostazione di alcune configurazioni per poter essere usati.

# 5 | IMPLEMENTAZIONE

In questo capitolo parlo dell'implementazione dei microservizi, in particolare dei problemi e delle soluzioni apportate durante il lavoro e degli esiti del collaudo.

## 5.1 DESCRIZIONE DELL'IMPLEMENTAZIONE

In tutti i microservizi sviluppati ho applicato le seguenti tecnologie:

- Spring Boot per usare le configurazioni di default usate da Spring e per abilitare la scansione automatica dei componenti di Spring.
- Log4J per gestire il log delle azioni in locale nel caso in cui il microservizio logging non fosse disponibile.
- Resilience4J per applicare il pattern circuit breaker, quindi per reagire alle eccezioni dovute agli altri microservizi scrivendo nel log e restituendo un messaggio di errore appropriato.
- Spring Eureka Client per registrare i microservizi automaticamente presso il service registry.

### 5.1.1 Microservizio document

In questo microservizio ho applicato, oltre alle tecnologie comuni, le seguenti tecnologie:

- Spring Data REST per definire le repository per le richieste REST dei documenti gestiti dal microservizio.
- Spring MVC per definire i controller usati per gestire upload e restituzione dei documenti convertendo da *base64* (sistema di codifica che consente la traduzione di dati binari in stringhe di testo ASCII, rappresentando i dati sulla base di 64 caratteri ASCII diversi) a file decodificato e viceversa.
- Spring Data per gestire i vincoli sui dati da memorizzare in MongoDB.

### 5.1.2 Microservizio rabbitmq

In questo microservizio ho applicato, oltre alle tecnologie comuni, le seguenti tecnologie:

- Spring AMQP per interagire con RabbitMQ gestendo le code, gli incodatori e gli scodatori dei messaggi e le possibili eccezioni lanciate da questi ultimi, in modo da non tentare inutilmente l'invio ripetuto di messaggi verso microservizi non raggiungibili.

### 5.1.3 Microservizio logging

In questo microservizio ho applicato, oltre alle tecnologie comuni, le seguenti tecnologie:

- Spring Data REST per rendere accessibili i log attraverso un'API REST.
- Spring MVC per poter scrivere i log dei vari microservizi attraverso Log4J.
- Log4J per gestire i log dei vari microservizi.

### 5.1.4 Microservizio eureka

In questo microservizio ho usato solamente la tecnologia Spring Eureka Server. Questo microservizio funge da service registry, in modo che tutte le istanze dei microservizi siano registrate in esso e che siano rese disponibili all'API gateway.

### 5.1.5 Microservizio apigateway

In questo microservizio ho usato solamente la tecnologia Spring Cloud Gateway. Infatti l'unica funzionalità resa disponibile dal microservizio è fornire un punto d'accesso agli altri microservizi registrati nel service registry i cui nomi vengono tracciati in un file di configurazione. Questo consente ai microservizi di slegarsi dal concetto di socket e usare solamente il nome del microservizio di cui necessitano per ottenerne un'istanza usabile.

## 5.2 PROBLEMI E SOLUZIONI

### 5.2.1 Più servizi, meno DRY

Durante lo sviluppo dei primi due microservizi (document e rabbitmq) mi sono reso conto che alcune funzionalità progettate per essere usate all'interno di un singolo controller, ad esempio per gestire i log e le chiamate verso API di microservizi esterni, si ripetevano secondo modelli uguali. A questo punto ho deciso di riprogettare una parte dei microservizi, astraendo le parti di codice ripetute e ponendole all'interno del package service e del package logger.

In questo modo, pur venendo meno al principio Don't Repeat Yourself (DRY, principio di progettazione e sviluppo secondo cui andrebbe evitata ogni forma di ripetizione e ridondanza logica nell'implementazione di un sistema software), è stato possibile riusare molto più codice tra i vari microservizi, semplicemente ripetendo il codice di tali servizi, senza cambiarvi nulla. I servizi ripetuti nei vari microservizi sono descritti in seguito.

**APICALLER** È un gestore delle chiamate dai microservizi verso microservizi esterni. Permette di gestire chiamate attraverso il protocollo HTTP quali get, post, put e delete.

**SYNCLOGGER** È un servizio per gestire le chiamate di logging interne al microservizio. Quando viene invocato, inoltra le richieste al microservizio logging. Se questo non è disponibile, si attiva il circuit breaker e viene usato il logging in locale.

### 5.2.2 Controlli sui punti critici

Inizialmente prevedevo l'uso del circuit breaker pattern per prevenire errori interni alle componenti dei microservizi senza fare distinzioni. Dopo aver infranto il principio DRY mi sono accorto che mettere in funzione il circuit breaker pattern ha senso se sono coinvolti più microservizi, altrimenti risulta più utile cercare di intercettare il problema e di risolverlo all'interno del microservizio.

Per avere il minor numero possibile di punti nel codice in cui attivare il circuit breaker, ho deciso di applicarlo sul servizio APICaller, in modo da intercettare qualsiasi chiamata fallita tra due microservizi qualsiasi. In questo modo è stato possibile intercettare molti più errori a fronte di un numero di controlli effettuati nel codice molto ridotto.

## 5.3 VERIFICA

Pur avendo dedicato poco tempo ai test di unità, il disporre di microservizi separati e del supporto di colleghi al lavoro sul frontend dell'applicazione ha permesso di avere ogni microservizio funzionante senza perdere quantità di tempo eccessive in riadattamenti.

Ho sviluppato dei test di unità nel microservizio document per verificare che il nome e il contenuto dei documenti siano salvati correttamente. Nel microservizio logging ho scritto test per accettare che i messaggi e le classi degli oggetti che generano tali messaggi siano corretti.

Essendo Spring già configurato per molti aspetti, ho apportato le maggiori correzioni solamente alle classi dei modelli dati, mentre le altre non sono state toccate.

Purtroppo non ho preso in considerazione lo sviluppo di test di integrazione con il frontend. Per questo motivo durante lo sviluppo del microservizio document ho dovuto apportare delle modifiche al codice che descrive il modello dati, problema che avrei potuto evitare se avessi scritto dei test di integrazione con il frontend prima dello sviluppo.

## 5.4 COLLAUDO

I test di collaudo sono stati ricavati dai requisiti sopra riportati. Tali test sono stati svolti usando l'interfaccia Web desktop sviluppata in Angular da alcuni colleghi. Prossimamente verranno svolti nuovamente usando l'interfaccia mobile sviluppata in Ionic da altri colleghi. In seguito mostro i risultati ottenuti nel collaudo finale.

Test	Descrizione	Esito
TC01	I documenti memorizzati nel servizio document devono avere un nome e il contenuto del file	Superato
TC02	È possibile aggiungere tipi di documenti diversi	Superato
TC03	Il servizio può restituire una lista con tutti i documenti	Superato
TC04	Il servizio permette la ricerca di un documento per id	Superato
TC05	È possibile eliminare un documento per id	Superato
TC06	È possibile modificare il nome e il contenuto di un documento, identificato da un codice univoco	Superato
TC07	Deve essere restituito un errore in caso il documento non possa essere memorizzato	Superato
TC08	Deve essere restituito un errore in caso il documento non possa essere eliminato	Superato
TC09	Deve essere restituito un errore in caso il documento non possa essere modificato	Superato
TC10	Deve essere restituito un errore in caso venga cercato un documento inesistente	Superato
TC11	Il servizio rabbitmq deve gestire una o più code di messaggi	Superato
TC12	Il servizio rabbitmq deve gestire uno o più inviatori di messaggi	Superato
TC13	Il servizio rabbitmq deve gestire uno o più scodatori di messaggi, ognuno iscritto a una coda	Superato
TC14	Deve essere restituito un errore nel caso il messaggio non possa essere inoltrato dagli scodatori	Superato
TC15	Deve essere possibile capire quale microservizio ha effettuato la richiesta di log	Superato
TC16	Il log deve essere tracciabile per tipo di messaggio	Superato
TC17	Il log deve essere tracciabile per tempo di memorizzazione del messaggio di log	Superato
TC18	Ogni microservizio deve poter richiedere la memorizzazione di un messaggio di log	Superato
TC19	I microservizi devono registrarsi sul service registry Eureka	Superato
TC20	I microservizi devono poter essere raggiungibili attraverso il microservizio apigateway	Superato

Tabella 3: Risultati dei test di collaudo

# 6 | CONCLUSIONI

*C*oncludo la tesi con il consuntivo del lavoro svolto e alcune valutazioni personali riguardanti il progetto e lo stage.

## 6.1 CONSUNTIVO FINALE

Come consuntivo riporto in seguito il numero di obiettivi raggiunti durante il progetto e le conoscenze acquisite, a fronte delle ore impiegate per il lavoro. Poiché non vi sono state assenze durante il periodo di stage, salvo un giorno legato alla festa di Sant'Antonio, recuperata a fine stage, le ore effettive di lavoro sono state 304. Considerando che l'attività di stage conta 11 Crediti Formativi Universitari (CFU), corrispondenti a circa 275 ore di impegno effettivo, le ore da me svolte sono in linea con quanto previsto inizialmente.

## 6.2 SODDISFACIMENTO DEGLI OBIETTIVI

Dato che i test di collaudo sono soddisfatti e sono basati sugli obiettivi del piano di lavoro, ne consegue che gli obiettivi previsti sono soddisfatti. Riporto in seguito una tabella con gli obiettivi.

Obiettivo	Esito
Acquisizione competenze sull'architettura a microservizi e sulle tecnologie usate per lo sviluppo del progetto	Superato
Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma	Superato
Portare a termine le funzionalità richieste dal cliente con una percentuale di superamento degli item di collaudo pari al 50%	Superato
Acquisire competenze su Spring Cloud	Superato
Portare a termine le funzionalità richieste dal cliente con una percentuale di superamento degli item di collaudo pari all' 80%	Superato
Implementazione delle funzionalità di Service Registry e Service Discovery mediante Spring Cloud	Superato

Tabella 4: Esito degli obiettivi

Oltre alle funzionalità previste mi sono permesso di aggiungere un'implementazione del pattern circuit breaker (si veda la sezione [3.4.3 a pagina 20](#)) nei microservizi che lo richiedevano, in modo da rendere più robusta l'applicazione rispetto agli errori che potrebbero verificarsi quando un microservizio non viene trovato.

## 6.3 POSSIBILI MIGLIORAMENTI

L'applicazione così come implementata presenta a mio parere alcuni limiti e carenze che è possibile risolvere.

### 6.3.1 Prestazioni e scalabilità

Come ho notato verso la fine dello stage insieme a Fabio Pallaro, i microservizi sviluppati adottando Spring Boot e Spring Cloud hanno un consumo di memoria molto elevato, circa 512 MB di memoria per ogni microservizio. Rilasciandoli su macchine diverse il problema non è enorme, in quanto a oggi è difficile trovare un elaboratore con meno di tale quantitativo di memoria.

Considerando che sono microservizi e che ognuno dovrebbe svolgere un compito più o meno modesto, mi sembra un quantitativo di risorse eccessivo, che spesso neanche un'applicazione monolitica presenta, come nel caso dell'attuale SMS.

Avere un'architettura a microservizi diventa troppo dispendioso nel caso in cui ogni microservizio consumi tante risorse quante ne servirebbero alla stessa applicazione con architettura monolitica.

Fabio suggerisce un'implementazione dei microservizi in Java Spark per risolvere il problema, un framework che secondo alcuni benchmark risulta molto meno esoso di risorse, arrivando fino a un decimo della memoria consumata da un medesimo microservizio implementato con Spring Boot.

### 6.3.2 Misure

Come appreso durante il progetto di Ingegneria del Software, le misure sono importanti. Per mantenere un controllo costante sulle misure relative ai microservizi, Codecentric offre uno strumento per controllare dettagli e misure relativi a stato del microservizio, prestazioni, proprietà, tracciamenti delle richieste e altro. È utile abilitarlo sui microservizi di service registry, quindi Eureka, perché in tali microservizi sono registrati tutti gli altri ed è quindi possibile avere molti dettagli relativi al sistema consultabili da un unico posto.

### 6.3.3 Logging

La soluzione per tenere il log dei microservizi in un unico microservizio separato presenta pregi e difetti. Il pregio migliore è che essendo una soluzione progettata per l'applicazione presenta log molto leggibili e di immediata comprensione rispetto ad altri sistemi. Il difetto peggiore è che raccoglie dati di log solamente nei punti dei microservizi in cui viene fatta un'esplicita richiesta di tracciamento da parte dello sviluppatore. Purtroppo uno sviluppatore non può sapere in anticipo in quali punti potrebbe verificarsi un errore o una situazione imprevista che richiede la generazione di un log.

Per risolvere il problema potrebbe bastare l'uso di una soluzione centralizzata come Spring Cloud Sleuth oppure Graylog [RV 2016]. In tal caso la tracciabilità dei log sarebbe implementata tramite codici invece che da nomi significativi, rendendo tali dati molto più usabili per fini statistici. Inoltre con tali framework l'onere di generazione del log non spetterebbe più allo sviluppatore del microservizio.

## 6.4 CONOSCENZE ACQUISITE

Grazie a questa esperienza ho imparato molte cose su nuove tecnologie e su come progettare un'applicazione a microservizi in autonomia. Malgrado avessi già affrontato problemi simili in contesti diversi, ad esempio durante il progetto di ingegneria del software, credo sia stato uno stage molto costruttivo, perché mi ha permesso di mantenere il focus su un unico progetto e perché mi ha dato l'opportunità di collaborare con una squadra di persone molto in gamba, con punti di vista diversi dal mio.

Oltre alle conoscenze tecniche acquisite, credo sia stato il giusto modo per mettermi alla prova e per capire quanto effettivamente fosse già nel mio bagaglio di conoscenze e quanto avessi ancora da imparare.

Nel complesso posso dire che le conoscenze e competenze acquisite sono le seguenti:

- lavoro autonomo e in team;
- apprendimento di tecnologie sconosciute, in particolare del framework Spring;
- approfondimento di tecnologie in parte conosciute, quindi Java, MongoDB e JSP;
- collaborazione con altre persone;
- organizzazione e gestione del tempo, in modo da rispettare le scadenze concordate.

## 6.5 VALUTAZIONI PERSONALI

Penso che l'esperienza di stage sia stata utile per avere un confronto con una delle diverse realtà che ancora non ho esplorato. Molte delle conoscenze e delle competenze che ho cercato almeno in parte di apprendere in questi tre anni dal corso di laurea triennale in Informatica si sono rivelate utili per il completamento dello stage, mi riferisco in particolar modo a corsi caratterizzanti e corsi con progetti pratici, che mi hanno fornito gli strumenti utili per una formazione personale in crescita continua.

Lo stage da me svolto mi ha fatto capire inoltre quanto sia importante non legarsi a una specifica tecnologia o a una specifica modalità di progettare e concepire un'applicazione. Basti pensare che le architetture a microservizi sono fortemente in crescita solo da qualche anno. Inoltre una delle tecnologie più adottate in Spring per applicare il pattern circuit breaker, ovvero Hystrix, è rimasta priva di aggiornamenti solo da qualche mese.

Ogni esperienza porta a chi la vive nuove conoscenze, permette di aprire nuove porte e di chiuderne altre. Trovo che lo stage sia utile sia per chi deve approcciarsi per la prima volta al mondo del lavoro, sia per chi ha esperienze pregresse in merito, perché costringe a confrontarsi con un ambiente nuovo e persone nuove. È inoltre un buon banco di prova per le proprie capacità e per capire quale strada prendere.

In ogni caso a mio parere non può essere considerata l'unica bussola per orientarsi nel mondo del lavoro, perché un solo stage in una sola realtà fornisce uno dei tanti punti di vista che è possibile adottare.

Concludo dicendo che Sync Lab tra le poche realtà con cui ho avuto modo di confrontarmi mi ha offerto molto, in particolare un buon ambiente di lavoro con colleghi cordiali, un ottimo supporto per lo stage da svolgere e un buon livello di autonomia che mi ha aiutato a mettermi alla prova per risolvere problemi che pensavo non sarei riuscito ad affrontare, facendomi acquisire più fiducia in me stesso.



# BIBLIOGRAFIA

## RIFERIMENTI BIBLIOGRAFICI

Richards, Mark

2016 *Microservices vs. Service-Oriented Architecture*, O'Reilly Media. (Citato a p. 16.)

Richardson, Chris

2019b *Microservices Patterns*, Manning Publications Co. (Citato a p. 22.)

RV, Rajesh

2016 *Spring Microservices*, Packt Publishing. (Citato a p. 32.)

## RIFERIMENTI WEB

Bhojwani, Rajesh

2018 *Design Patterns for Microservices*, ott. 2018, <https://dzone.com/articles/design-patterns-for-microservices>. (Citato alle p. 18, 20.)

Despodovski, Rade

2017 *Microservices vs. SOA – Is There Any Difference at All?*, nov. 2017, <https://dzone.com/articles/microservices-vs-soa-is-there-any-difference-at-al>. (Citato a p. 17.)

*Enterprise service bus*

2019 giu. 2019, [https://it.wikipedia.org/wiki/Enterprise\\_service\\_bus](https://it.wikipedia.org/wiki/Enterprise_service_bus). (Citato a p. 16.)

Fowler, Martin

2011 *TolerantReader*, mag. 2011, <https://www.martinfowler.com/bliki/TolerantReader.html>. (Citato a p. 15.)

2014 *Microservices Guide*, <https://martinfowler.com/microservices/>. (Citato a p. 14.)

2015 *Microservice Trade-Offs*, lug. 2015, <https://martinfowler.com/articles/microservice-trade-offs.html>. (Citato a p. 16.)

Fowler, Martin e James Lewis

2014 *Microservices*, mar. 2014, <https://martinfowler.com/articles/microservices.html>. (Citato a p. 14.)

Richardson, Chris

2019a *Database per service*, <https://microservices.io/patterns/data/database-per-service.html>. (Citato a p. 21.)

Wikipedia

2019 *Architettura multi-tier*, apr. 2019, <https://it.wikipedia.org/wiki/Architettura-multi-tier>. (Citato a p. 15.)