UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA TRIENNALE IN
INGEGNERIA INFORMATICA

Utilizzo del framework Spring e del broker di messaggi RabbitMQ in una applicazione web per il corporate social networking

Relatore:
PROF. FANTOZZI CARLO

Laureando:
LUISETTO FEDERICO
1187521

Anno Accademico 2021/2022
Data di laurea 07/03/2022

Sommario

Il presente documento descrive il mio lavoro svolto durante il periodo di tirocinio, della durata di circa duecentocinquanta ore, presso l'azienda Sync Lab. Il lavoro di tirocinio si inserisce in un progetto, denominato Challenginator, che consiste nello sviluppo di un'applicazione web per la gestione di sfide tra colleghi.

Gli obiettivi di questo tirocinio erano molteplici. In primo luogo era richiesto lo studio teorico delle applicazioni a microservizi, del framework Spring e del broker RabbitMQ. In secondo luogo era richiesta l'implementazione del servizio di notifica tramite e-mail di Challenginator. Infine era richiesto un meccanismo di log per l'applicazione web, così da facilitare il debugging in caso di future modifiche a quest'ultima.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco del codice

Capitolo 1

Introduzione

1.1 L'azienda

Figura 1.1: Logo azienda Sync Lab. Fonte: SyncLab

Sync Lab [13] nasce come *software house* a Napoli nel 2002. Rapidamente cresciuta nel mercato dell' ICT ora è un *system integrator* che conta 6 sedi in Italia e più di 300 dipendenti.
L'azienda propone innovativi prodotti software che spaziano in diversi settori: GDPR, Big Data, Cloud Computing, IoT, Mobile e Cyber Security. Grazie alla vasta gamma di soluzioni offerte Sync Lab lavora in diversi mercati, quali Sanità, Industria, Energia, Finanza e Trasporti & Logistica.

1.2 Challenginator

Challenginator 4 è un progetto interno all'azienda ideato per permettere ai propri dipendenti di sfidarsi l'un l'altro in attività inerenti alla vita lavorativa. L'intento è quello di

contribuire alla crescita personale del singolo, cercando di migliorare le dinamiche lavorative.

L'idea di base è semplice: un collega ne sfida un altro in un'attività (ad esempio timbrare in orario, ordinare quotidianamente la scrivania etc.) e un superiore supervisionerà l'operato, giudicando lo sfidato.

Per l'implementazione è stato utilizzato il linguaggio Spring 2.2 data la sua leggerezza e modularità; grazie a quest'ultima è stato possibile selezionare solo le componenti necessarie per il progetto. Tra queste troviamo *Data JPA* per la comunicazione con il database, *Data REST* per lo scambio di dati mediante chiamate REST e *Cloud* per aggiungere un servizio atto all'individuazione delle componenti (microservizi) dell'applicazione.

Il Database Management System (DBMS) scelto per salvare in modo persistente i dati è PostgreSQL, dato che si è lavorato con database relazionali.

L'invio delle email ed il servizio di log sono affidati al broker RabbitMQ, preferito a Kafka per i motivi descritti nella Sezione 2.3.

Per la scrittura, la manutenzione ed il testing del progetto è stato utilizzato dell'altro software, descritto nello specifico nella Sezione 2.4.

Il mio contributo a questo progetto è costituito dall'implementazione, assieme ad un collega, del servizio di notifica e l'implementazione, autonoma, del servizio di logging.

Capitolo 2

Metodologie e tecnologie

In questo capitolo descriverò gli strumenti utilizzati per la realizzazione ed il testing del progetto. Tra di essi mi soffermerò in particolare sul *framework* Spring (e relativi moduli derivati), il broker RabbitMQ e l'architettura a microservizi dato che sono stati i principali ambiti di studio nel periodo di tirocinio.

2.1 Architettura a Microservizi

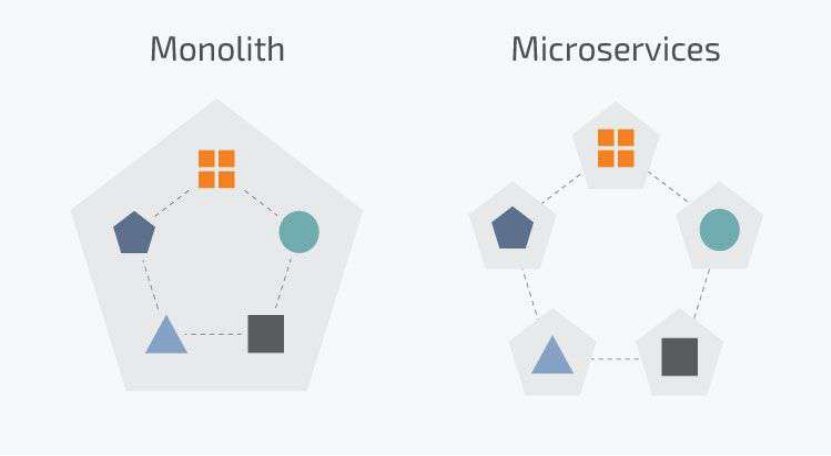

Figura 2.1: Architettura monolitica vs microservizi. Fonte: N-iX

Nel mondo della programmazione una delle nuove tendenze è l'adozione dell'architettura a microservizi per la creazione di applicazioni [8]. L'approccio a microservizi offre infatti vantaggi tangibili, tra cui un aumento della scalabilità, della flessibilità e dell'agi-

lità. Molti leader tecnologici hanno effettuato con successo il passaggio dall'architettura monolitica ai microservizi, e ad ora molte aziende considerano di seguire questo esempio come il modo più efficiente per la crescita del business.
Dal canto suo l'approccio monolitico è un modello predefinito per la creazione di un'applicazione software. Tuttavia la sua popolarità sta diminuendo dato che questo tipo di architettura pone una serie di sfide associate, quali una grossa mole di codice, la difficoltà nell'adozione di una nuova tecnologia e dell'implementazione di modifiche.

2.1.1 Monolitica vs. Microservizi

Architettura Monolitica

L'architettura monolitica è considerata la via tradizionale nella costruzione delle applicazioni. Un'applicazione monolitica è costituita da una singola ed indivisibile unità dove vengono gestite tutte le operazioni. Tra queste possiamo trovare l'interfaccia grafica atta alla comunicazione con l'utente e la comunicazione/gestione del database. Di prassi l'architettura monolitica è caratterizzata da una grande mole di codice e da mancanza di modularità. Se è necessario effettuare un aggiornamento o cambiare qualcosa, i programmatori dovranno accedere allo stesso codice, modificando l'intera applicazione.

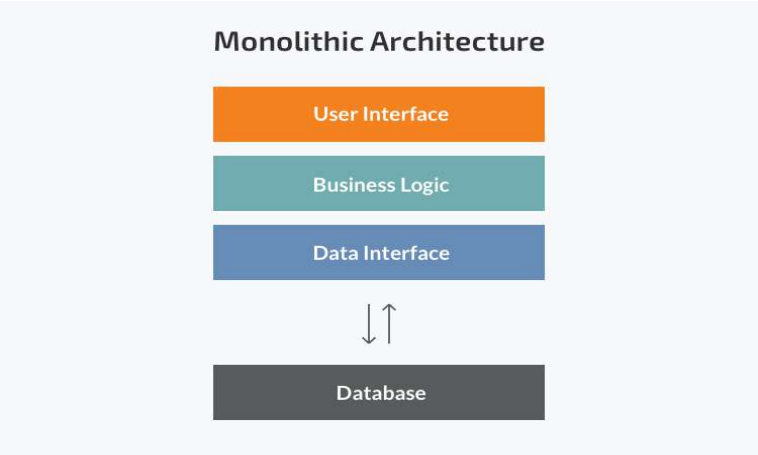

Figura 2.2: Architettura monolitica. Fonte: N-iX

I punti di forza di questo tipo di architettura sono la semplicità di sviluppo, derivante dal fatto che il modello monolitico è uno standard per creare applicazioni, e di distribuzione,

la riduzione dei problemi trasversali quali il *logging*, il *caching* ed il monitoraggio delle prestazioni. Infine il *debugging* ed il *testing* sono più semplici dato che riguardano una singola unità indivisibile.
I punti di debolezza sono principalmente legati alla gestione del codice nel tempo. Mano a mano che l'applicazione si arricchisce di nuove funzionalità diventa difficile la comprensione d'insieme.

Pro	Contro
Meno problemi trasversali	Difficoltà di comprensione
Semplicità debugging e testing	Difficoltà nella manutenzione
Semplicità di sviluppo	Nessuna scalabilità di componenti
Semplicità di distribuzione	Problematica nuove tecnologie

Tabella 2.1: Pro e contro di una architettura monolitica

Architettura a Microservizi

Mentre un'applicazione monolitica è una singola indivisibile unità, l'architettura a microservizi suddivide l'applicazione in una raccolta di unità indipedenti più piccole. Queste unità eseguono ogni processo dell'applicazione come un servizio separato, così ognuno di questi servizi può avere una propria logica ed un proprio database.
All'interno di un'architettura a microservizi l'intero applicativo è suddiviso in moduli distribuiti in modo indipendente che comunicano tra loro attraverso le Application Programming Interface (API). Ogni servizio può essere aggiornato, distribuito e ridimensionato in modo indipendente.

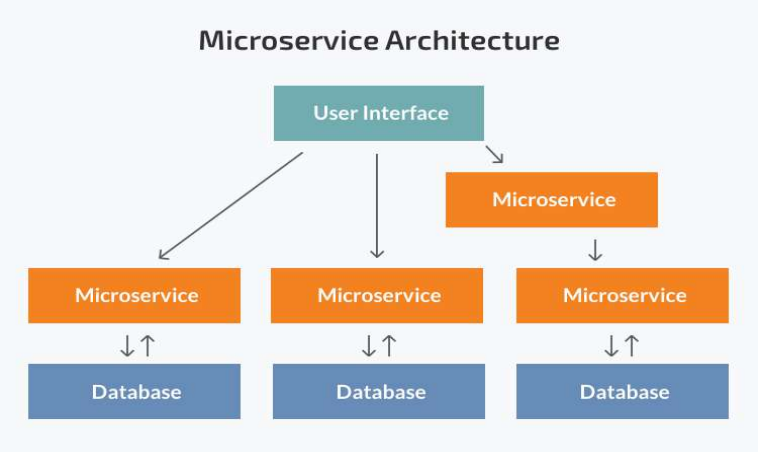

Figura 2.3: Architettura a microservizi. Fonte: N-iX

I punti di forza di questo tipo di architettura sono l'indipendenza delle componenti, che permette l'implementazione e l'aggiornamento del singolo modulo; da questo deriva anche una migliore scalabilità ed il confinamento dei bug, dato che il malfunzionamento di un servizio non comporta l'interruzione di tutta l'applicazione. La divisione dei vari servizi permette una migliore e più semplice comprensione dell'applicazione ed una maggiore flessibilità nella scelta delle tecnologie.
I punti di debolezza riguardano la complessità di realizzazione e di *testing* di un'applicazione con questa architettura, l'incremento di problemi trasversali nell'implementazione e la difficoltà di distribuzione, data l'interconnessione di più moduli indipendenti.

Pro	Contro
Indipendenza delle componenti	Complessità aggiuntiva
Facilità di comprensione	Difficoltà di distribuzione
Migliore scalabilità	Problematiche trasversali
Flessibilità nelle tecnologie	Testing difficoltoso

Tabella 2.2: Pro e contro di una architettura a microservizi

2.1.2 Componenti microservizi

API Gateway

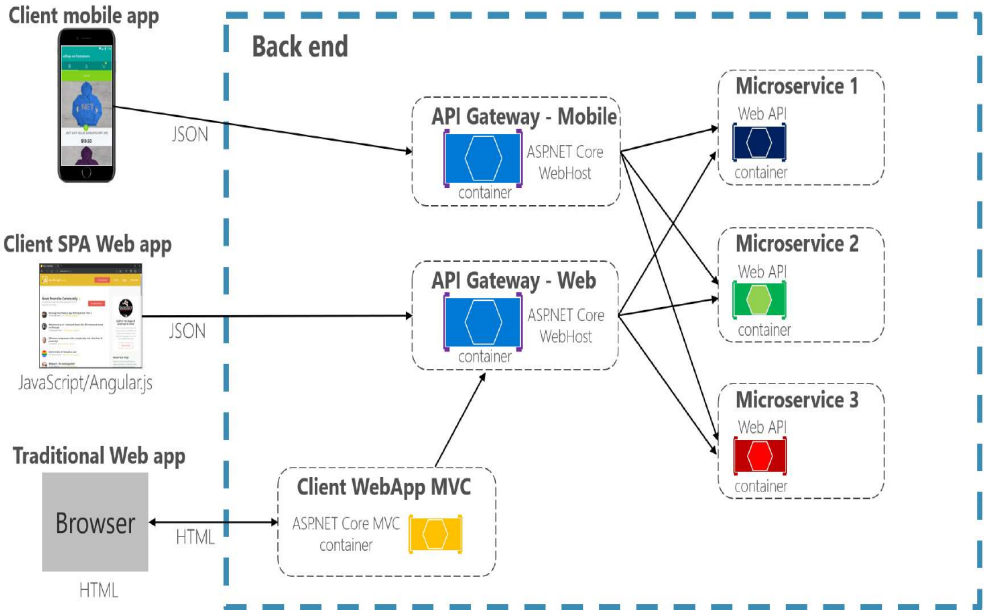

Figura 2.4: Servizio API Gateway. Fonte: Microsoft

Circuit Breaker

Il *circuit breaker* è un modello di progettazione utilizzato nell'architettura a microservizi. I vari servizi per interagire tra loro si scambiano dati mediante richieste attraverso la rete. Durante questo scambio di informazioni è possibile che un servizio sia inattivo, per una connessione lenta, un timeout o un'indisponibilità temporanea. La soluzione migliore è far richiesta nuovamente a quel servizio, però se il problema è grave e quest'ultimo non è a disposizione per lungo tempo si può incorrere nell'eventualità in cui le chiamate a quel servizio saturino la rete, compromettendo le prestazioni e l'esperienza utente. Per ovviare a queste problematiche si utilizza il Design Pattern *circuit breaker*: quando il numero di guasti di un servizio supera una determinata soglia l'interruttore va in stato *open* per un periodo di *timeout*, ritornando un errore al servizio chiamate ad ogni nuova chiamata. Trascorso questo lasso temporale l'interruttore va in stato *half-open*, consentendo il passaggio di un limitato numero di richieste di richieste di test. Se queste ultime avranno esito positivo l'interruttore andrà in stato *closed* e riprende il normale funzionamento. In caso contrario il periodo di *timeout* ricomincia.

Un API Gateway funge da "porta d'ingresso" per accedere ai vari microservizi da un singolo punto di accesso ed anche da *proxy* inverso dato che indirizza le richieste dai client ai servizi. Questo servizio sta nel mezzo tra le applicazioni ed i microservizi così non

è necessario esporre i singoli applicativi, aumentando così la sicurezza. In figura 2.4 si nota che solitamente si utilizza più di un API Gateway così da non avere un aggregatore monolitico (che violerebbe l'autonomia dei microservizi accoppiandoli tutti tra loro).

Service Discovery e Service Registry

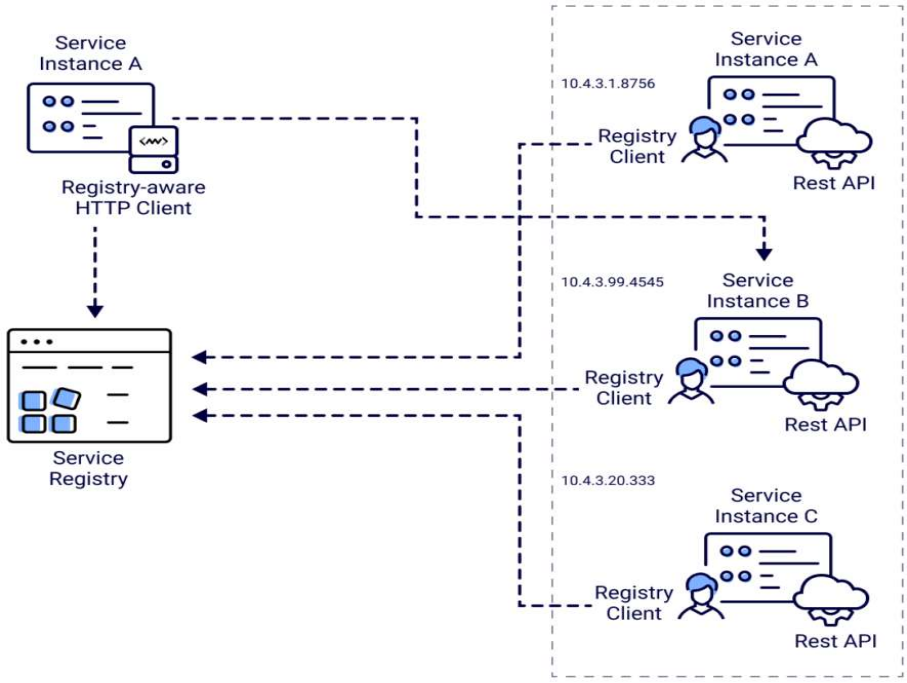

Figura 2.5: Schema del Service Discovery. Fonte: Middleware

Il *service discovery* è un protocollo di rete utilizzato per la rilevazione automatica di servizi e dispositivi in rete. In figura 2.5 è rappresentato un *service discovery* lato client come quello utilizzato nel progetto; in questo tipo di servizio il client deve ricercare il *service registry* per poter individuare un servizio. Quindi il client seleziona un'istanza del servizio libera mediante un algoritmo di bilanciamento del carico.

Il *service registry* è una componente fondamentale del *service discovery*. Esso è costituito da un database contenente le locazioni di tutte le istanze dei servizi disponibili. I client possono salvare le locazioni ottenute dal *service registry* nella cache, ma non devono appoggiarsi ai dati ottenuti per troppo tempo dato che quelle informazioni possono diventare velocemente obsolete.

2.2 Spring

Spring è un framework open source per lo sviluppo di applicazioni su piattaforma Java. Ad esso sono associati tanti altri progetti, modulari, che ne completano le funzionalità. Questi hanno nomi composti, come *Spring Boot*, *Spring Data* o *Spring Security*. Seguirà ora un elenco dei componenti di questo framework da me utilizzati.

- **Spring Boot**: questo modulo permette la creazione di una applicazione utilizzante il *framework* Spring pronta all'uso. Mediante *Spring Inizializr*, *Spring Boot* imposta i parametri dell'applicativo da creare (quali nome del progetto, il linguaggio utilizzato, i metadata del progetto ed il packaging) ed aiuta ad aggiungere le dipendenze necessarie tramite un checkbox.

- **Spring Data REST**: questo modulo permette la gestione dell'applicazione mediante chiamate REST e la navigazione attraverso link.

- **Spring Data JPA**: questo modulo semplifica la comunicazione col database da parte dell'applicazione mediante Java Persistence API (JPA). Permette l'esecuzione di query, nonché l'impaginazione ed il controllo dei dati.

- **Spring REST Docs**: questo modulo è utile per la scrittura della documentazione progettuale e durante la fase testing. Utilizzando Postman 2.4, mediante la chiamata GET `http://localhost:808x/v3/api-docs` (**x** indica un numero da 0 a 5, in base al microservizio da testare), si ha a disposizione una descrizione dei campi necessari per poter effettuare una chiamata POST o SET a quel microservizio.

- **Spring AMQP**: questo modulo è necessario per la corretta comunicazione dei microservizi col broker. Permette di inviare e ricevere messaggi e la dichiarazione di nuove *queue*, *exchange* e *bindings*.

- **Spring Cloud Circuit Breaker**: questo modulo fornisce un'astrazione tra le diverse implementazioni dei *circuit breaker*.

- **Spring Cloud Gateway**: questo modulo permette la creazione di un API Gateway che fornisce un semplice ma efficace modo per instradare le chiamate dell'utente ai vari microservizi.

- **Spring Cloud Netflix**: questo modulo introduce nel progetto il *service discovery* Eureka

- **Spring Security**: questo modulo è un framework di autenticazione e controllo accessi potente e altamente personalizzabile. È lo standard de facto per la protezione delle applicazioni basate su Spring.

2.3 RabbitMQ

RabbitMQ è un broker di messaggi che utilizza il protocollo AMQP per svolgere le sue funzioni. AMQP è uno standard aperto che definisce un protocollo a livello applicativo per il *message-oriented middleware*; questo standard è definito in modo tale da garantire funzionalità di messaggistica, accodamento e *routing*, affidabilità e sicurezza [12]. Un broker di messaggi è una tecnologia che si fa carico della convalida, della trasformazione e del corretto indirizzamento dei messaggi. Questo strumento media la comunicazione tra le applicazioni permettendo la comunicazione attraverso lo scambio di messaggi, che possono includere qualsiasi tipo di informazione.
Le componenti di RabbitMQ sono le seguenti

- *Publisher*: entità che comunica col broker per inviare un messaggio.

- *Consumer*: entità che riceve un messaggio dal broker.

- *Queue*: entità dove i messaggi sono custoditi prima dell'invio al consumer. Il legame tra *queue* ed *exchange* è chiamato *binding*.

- *Exchange*: entità che riceve i messaggi dal *publisher* e li accoda in una o più *queue*. Ci sono più tipi di *exchange*.

  - *Fanout* accoda tutti i messaggi in tutte le *queue* che conosce.

  - *Direct* accoda il messaggio alle code la cui chiave di associazione corrisponde esattamente alla *routing key* del messaggio.

  - *Topic* è un insieme dei precedenti. In questo modello si instradano i messaggi verso una o più *queue* in base al modello utilizzato per associare una *queue*

all'*exchange*. Questo modello lascia maggiore libertà nello specificare le *routing key* permettendo di utilizzare le *wildcards* * per sostituire una qualsiasi parola e # per sostituire 0 o più parole.

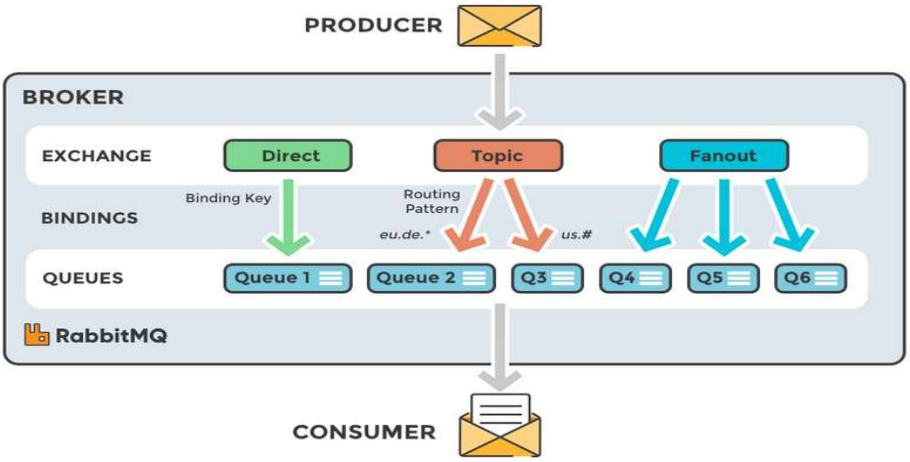

Figura 2.6: Componenti del broker RabbitMQ. Fonte: RabbitMQ

Rispetto ad altri broker RabbitMQ introduce l'*queue* tra le sue componenti, e grazie a questa modifica implementa fedelmente il protocollo AMQP. Precedentemente vi erano solo i primi tre elementi sopracitati, con coda singola e metodo FIFO.
L'aspetto che contraddistingue RabbitMQ rispetto ad altri broker (quali Kafka) è la capacità di lavorare molto velocemente quando le code sono quasi vuote: questo scenario è perfetto per la comunicazione tra microservizi. Inoltre la gestione delle code è più semplice, dato che il messaggio viene rimosso dalla coda una volta letto. Kafka invece è stato progettato per salvare i messaggi che arrivano per poterli leggere ed analizzare (tipicamente per il tracking degli utenti, logging) o per processarli in real-time.

2.4 Altri software utilizzati

Nell'implementazione di questo progetto sono state impiegati vari strumenti software:

- **Visual Studio Code:** Visual Studio Code è un editor di codice sorgente leggero ma potente che viene eseguito sul desktop ed è disponibile per Windows, macOS e Linux. Viene fornito con il supporto integrato per JavaScript, TypeScript e Node.js e

ha un ricco ecosistema di estensioni per altri linguaggi (come C++, Java, Python, PHP, Go) e runtime (come .NET e Unity) [15]. Le estensioni utilizzate nel progetto sono state *Extension Pack for Java*, *Spring Boot Extension Pack*, *Lombok Annotations Support for VS Code*, *Docker*, *Remote - Containers* e *GitHub Pull Requests and Issues*.

- **Git:** Git è un sistema di controllo versione distribuito gratuito e open source progettato per gestire progetti sia piccoli che molto grandi con velocità ed efficienza [7]. Nel progetto è stato adottato cosicché più programmatori potessero cooperare nella scrittura e nella risoluzione dei problemi.

- **Docker:** Docker è una piattaforma per lo sviluppo, la distribuzione e l'esecuzione di applicazioni. Consente di separare le applicazioni dall'infrastruttura in modo da poter distribuire rapidamente il software. Sfruttando le metodologie Docker si può ridurre significativamente il tempo intercorso tra la scrittura del codice e l'esecuzione in produzione [6]. Questo software è stato utilizzato per l'esecuzione di PostgreSQL e RabbitMQ e per la creazione delle immagini dei microservizi.

- **Postman:** Postman è una piattaforma per la creazione e l'utilizzo di API. Postman semplifica ogni fase del ciclo di vita dell'API e ottimizza la collaborazione in modo da poter creare API migliori, più velocemente. La piattaforma include un set completo di strumenti che aiutano ad accelerare il ciclo di vita delle API, dalla progettazione, test, documentazione e derisione alla condivisione e rilevabilità delle API [11].

- **DBeaver:** DBeaver è uno strumento di gestione del database universale per tutti coloro che hanno bisogno di lavorare con i dati in modo professionale. Con DBeaver si possono manipolare i dati, ad esempio in un normale foglio di calcolo, creare report analitici basati su record provenienti da diversi archivi di dati ed esportare le informazioni in un formato appropriato. Per gli utenti avanzati di database, DBeaver fornisce un potente editor SQL, numerose funzionalità di amministrazione, capacità di migrazione di dati e schemi, monitoraggio delle sessioni di connessione al database [5].

- **PostgreSQL:** PostgreSQL è un DBMS ad oggetti. Il suo utilizzo nel progetto è frutto della necessità di rendere persistenti i dati, come utenti, password, lo storico delle sfide e log di sistema.

- **Angular:** Angular è una piattaforma per lo sviluppo di applicazioni web, basata su TypeScript. Come piattaforma, Angular include un framework basato su componenti per la creazione di applicazioni Web scalabili, una raccolta di librerie ben integrate che coprono un'ampia varietà di funzionalità, tra cui routing, gestione dei moduli, comunicazione client-server e una suite di strumenti per sviluppatori per aiutare a sviluppare, costruire, testare e aggiornare il codice.[1]

- **Maven**: Apache Maven è uno strumento di comprensione e gestione dei progetti software. Basato sul concetto di un modello a oggetti di progetto (*POM*), Maven può gestire la creazione, il reporting e la documentazione di un progetto da un'informazione centrale [3]. Un *POM* fornisce tutte le configurazioni di un singolo progetto, come per esempio il nome del progetto, la versione, il proprietario, dipendenze da altri progetti e le fasi di *build* dell'applicazione.

Oltre agli strumenti sopraelencati nel progetto sono adottati anche diversi Design Pattern:

- **Microservizi:** pattern opposto allo sviluppo monolitico in quanto ogni servizio deve poter essere sviluppato e distribuito in maniera indipendente. La comunicazione tra le varie componenti è solitamente basata su HTTP tramite chiamate REST.

- **Dependency injection:** Le due principali tecnologie impiegate nello sviluppo del prodotto, ovvero Angular e Spring, permettono entrambe di utilizzare il pattern *dependecy injection* ovvero un *design pattern* che permette di attuare l'*inversion of control*. Le applicazioni Spring vengono istanziate all'interno di uno speciale container denominato *IoC container* il quale si occupa di istanziare gli oggetti (*beans*) dichiarati nel progetto e di reprire e iniettare tutte le dipendenze ad essi associate.

- **Lazy loading:** *design pattern* utilizzato in *Angular* che permette di caricare i moduli solo nel momento in cui effettivamente se ne ha bisogno ottenendo un guadagno in prestazioni di caricamento dell'applicazione.

Capitolo 3

Progetto personale

Dopo una prima fase puramente teorica il tirocinio è proseguito con un progetto personale per poter testare le conoscenze apprese. Quest'ultimo doveva includere il *framework* Spring, il *broker* RabbitMQ e permettere il salvataggio persistente dei dati su database PostgreSQL. Il progetto [14] scelto aveva il compito di gestire le ordinazioni provenienti da più ristoranti. La struttura che lo compone è riportata di seguito:

```
springbootrabbitmqexample
├── config
│   └── MessagingConfig.java
├── consumer
│   └── User.java
├── dto
│   ├── Order.java
│   ├── OrderRepository.java
│   └── OrderStatus.java
├── publisher
│   ├── OrderPublisher.java
│   └── OrdersID.java
└── SpringbootRabbitmqExampleApplication.java
```

3.1 Gestione salvataggio ordini

Per poter permettere il salvataggio persistente degli ordini ho utilizzato il DBMS PostgreSQL. Il database è composto da una singola tabella `orders` e gli ordini sono distinti univocamente dalla coppia (`orderId`,`restaurant`).

orders
orderId
restaurant
name
qty
price

Tabella 3.1: Tabella `orders`

La tabella 3.1 viene generata dal codice contenuto nel file `Order`.

Listing 3.1: Frammento codice tabella `orders`

```
1  @Entity
2  @Table(name = "orders")
3  @IdClass(OrdersID.class)
4  public class Order {
5      @Id
6      private String orderId;
7      @Id
8      private String restaurant;
9      private String name;
10     private int qty;
11     private double price;
12 }
```

Nell'ordine, le annotazioni utilizzate servono per

1. Indicare questa classe come modello per la generazione della tabella
2. Specificare il nome della tabella
3. Utilizzare una *primary key* composta, creando anche una classe che racchiuda gli oggetti che compongono la *primary key* stessa

Listing 3.2: Classe per utilizzare *primary key* composta

```
1  public class OrdersID implements Serializable {
2      private String orderId;
3      private String restaurant;
4  }
```

3.2 Gestione di invio e ricezione degli ordini

Per poter permettere lo scambio di informazioni nell'applicazione ho configurato il broker di messaggi *RabbitMQ* 2.3 nella classe `MessagingConfig` specificando i nomi di *queue* ed *exchange*.
L'invio degli ordini avviene mediante chiamate REST utilizzando Postman per l'invio della *request* e la visualizzazione della *response*. Una volta inviato l'ordine `OrderPublisher` ha il compito di ricevere queste chiamate ed estrarre le informazioni utili: nel body (in formato JSON) della chiamata vi è l'ordine mentre nel *path* di quest'ultima vi è il nome del ristorante. Esempio di chiamata REST:
`http://localhost:9292/order/lanterna`

Listing 3.3: Esempio contenuto del body

```
{
    "orderId": 1,
    "name": "spaghetti",
    "qty": 4,
    "price": 50
}
```

Dopo aver elaborato i dati ricevuti la classe `OrderPublisher` li invia al broker utilizzando l'oggetto `template` e successivamente li salva in modo persistente richiamando il metodo `save` dell'oggetto `orderRepo`:

Listing 3.4: Frammento del codice della classe `OrderPublisher` e relativo metodo `bookOrder`

```
@Autowired
private RabbitTemplate template;
@Autowired
private OrderRepository orderRepo;
...
template.convertAndSend(MessagingConfig.EXCHANGE,
    MessagingConfig.ROUTING_KEY, orderStatus);
orderRepo.save(order);
...
```

Per finire nel progetto è stato inserito anche una classe `User` il cui compito è quello di ricevere i messaggi dal broker, leggerli e stampare a schermo i dettagli dell'ordinazione appena effettuata.

Listing 3.5: Frammento del codice della classe `User`

```
1 @RabbitListener(queues = MessagingConfig.QUEUE)
2 public void consumeMessageFromQueue(OrderStatus orderStatus) {
3     System.out.println("Message recieved: " + orderStatus +
4     "\nOrder details: " + orderStatus.getOrder().toString());
5 }
```

Capitolo 4

Challenginator

L'analisi che segue è stata svolta da persone terze quando il progetto è stato avviato [4]. *Challenginator* è una *web application* che permette l'invio, la ricezione ed il tracciamento di sfide tra i membri di un team. Mediante la *user interface* ogni utente può effettuare la registrazione e, dopo il processo di autenticazione tramite login, l'accesso alla propria area personale. Da quest'ultima l'utente ha accesso a:

- Home: pagina riepilogativa
- Dashboard: pagina con accesso rapido alle sfide passate, alla procedura di lancio di una nuova sfida ed elencazione di sfide in cui si è stati sfidati, si sfida o si fa da valutatore
- Preferenze di notifica: pagina dove l'utente decide quali tipologie di notifica ricevere
- Storico: pagina contenente lo storico delle sfide
- Nuova Challenge: pagina dove è possibile lanciare una nuova sfida
- Logout

Ogni utente può sfidare un collega attraverso l'apposito form di inserimento di una nuova sfida ed in modo automatico verrà assegnato un valutatore. Quest'ultimo è il soggetto, gerarchicamente a livello superiore, che decreterà la riuscita od il fallimento della sfida una volta completata da parte dello sfidato.

4.1 Analisi dei requisiti

Lo scopo del prodotto è stato riportato nella sezione precedente. Durante l'intero flusso dell'applicazione gli attori del sistema sono:

- **Utente non autenticato:** identifica un utente che non ha ancora effettuato l'accesso al rispettivo account
- **Utente autenticato:** identifica un utente che ha effettuato correttamente il login e può essere identificato a sua volta come:
  - **Sfidante:** utente che ha inserito una sfida contro un utente terzo
  - **Sfidato:** utente che ha ricevuto una sfida da un utente terzo
  - **Valutatore:** utente che deve decretare la riuscita o meno di una sfida

4.1.1 Casi d'uso

I casi d'uso individuati sono elencati di seguito.

Attore primario	utente non autenticato
Descrizione	autenticazione utente
Precondizioni	l'utente non si è ancora autenticato nell'applicazione
Input	l'utente inserisce ed invia i dati per il login
Postcondizioni	il cliente è autenticato

Tabella 4.1: UC-1: Login utente

Attore primario	utente non autenticato
Descrizione	registrazione nuovo utente
Precondizioni	l'utente non si è ancora autenticato nell'applicazione
Input	l'utente inserisce ed invia i dati per la creazione del proprio account
Postcondizioni	il cliente è autenticato

Tabella 4.2: UC-2: Registrazione utente

Attore primario	utente autenticato (sfidante, sfidato, valutatore)
Descrizione	l'utente vuole visualizzare le sfide in cui è coinvolto come sfidato, sfidante o valutatore
Precondizioni	l'utente ha effettuato il login
Input	l'utente clicca sulla voce di menù dashboard, se non la sta già visualizzando a seguito del login
Postcondizioni	l'utente visualizza la *dashboard* contenente tutte le sfide in cui è coinvolto a vario titolo

Tabella 4.3: UC-3: Visualizzazione lista sfide (*dashboard*)

Attore primario	utente autenticato - sfidante
Descrizione	l'utente che ha lanciato la sfida vuole cancellarla
Precondizioni	l'utente sta visualizzando la dashboard riepilogativa ed è l'utente sfidante
Input	l'utente clicca sul pulsante per cancellare la sfida
Postcondizioni	l'utente ha cancellato la sfida

Tabella 4.4: UC-3.1: Cancellazione challenge

Attore primario	utente autenticato - sfidato
Descrizione	l'utente vuole accettare una sfida che gli viene proposta
Precondizioni	l'utente sta visualizzando la *dashboard* riepilogativa, e una sfida in cui è identificato come sfidato è in attesa
Input	l'utente clicca sul pulsante per accettare la sfida
Postcondizioni	l'utente ha accettato la sfida

Tabella 4.5: UC-3.2: Accettazione challenge

Attore primario	utente autenticato - sfidato
Descrizione	l'utente vuole rifiutare una sfida che gli viene proposta
Precondizioni	l'utente sta visualizzando la *dashboard* riepilogativa, e una sfida in cui è identificato come sfidato è in attesa
Input	l'utente clicca sul pulsante per rifiutare la sfida
Postcondizioni	l'utente ha rifiutato la sfida

Tabella 4.6: UC-3.3: Rifiuto challenge

Attore primario	utente autenticato - sfidato
Descrizione	l'utente vuole dichiarare la sfida in cui è sfidato come completata
Precondizioni	l'utente sta visualizzando la *dashboard* riepilogativa, e vuole dichiarare la sfida (in cui risulta sfidato) come completata
Input	l'utente clicca sul pulsante per completare la sfida
Postcondizioni	l'utente ha modificato lo stato della sfida

Tabella 4.7: UC-3.4: Dichiarare la sfida come completata

Attore primario	utente autenticato - sfidato
Descrizione	l'utente vuole arrendersi in una sfida
Precondizioni	l'utente sta visualizzando la *dashboard* riepilogativa, e vuole arrendersi in una sfida
Input	l'utente clicca sul pulsante per arrendersi
Postcondizioni	l'utente ha modificato lo stato della sfida

Tabella 4.8: UC-3.5: Arrendersi in una sfida

Attore primario	utente autenticato
Descrizione	l'utente vuole visualizzare il dettaglio di una sfida
Precondizioni	l'utente sta visualizzando la *dashboard* riepilogativa
Input	l'utente clicca sul pulsante per visualizzare i dettagli della sfida, quali inizio e fine, tempo rimanente, la descrizione completa...
Postcondizioni	l'utente visualizza la pagina di dettaglio della sfida

Tabella 4.9: UC-4: Visualizzazione dettaglio sfida

Attore primario	utente autenticato
Descrizione	l'utente vuole inserire una nuova sfida
Precondizioni	l'utente clicca il pulsante di creazione nuova sfida
Input	l'utente inserisce i dati richiesti per lanciare una sfida
Postcondizioni	l'utente ha lanciato una sfida

Tabella 4.10: UC-5: Inserimento nuova sfida

Attore primario	utente autenticato
Descrizione	l'utente vuole visualizzare il dettaglio di una sfida
Precondizioni	l'utente sta visualizzando la *dashboard* riepilogativa
Input	l'utente clicca il pulsante per visualizzare il dettaglio della sfida
Postcondizioni	l'utente visualizza la pagina di dettaglio della sfida

Tabella 4.11: UC-6: Visualizzazione storico sfide

Attore primario	utente autenticato
Descrizione	l'utente vuole decidere quali notifiche ricevere
Precondizioni	l'utente seleziona le *checkbox*
Input	l'utente clicca il pulsante di selezione preferenze di notifica, come in figura 5.1
Postcondizioni	l'utente salva le sue preferenze di notifica

Tabella 4.12: UC-7: Scelta preferenze di notifica

4.1.2 Tracciamento dei requisiti

Per elencare i requisiti individuati correlati ai casi d'uso utilizzo la seguente codifica:

`R[Importanza][Tipologia]-[Codice]`

- **Importanza:** indica l'importanza di tale requisito attraverso i valori
  - **1** requisito obbligatorio
  - **2** requisito desiderabile
  - **3** requisito opzionale
- **Tipologia**
  - **V** requisito di vincolo
  - **F** requisito funzionale
  - **Q** requisito di qualità
- **Codice:** identificatore univoco in forma gerarchica padre/figlio

[CodiceBase](.[CodiceSottoCaso])*

Il CodiceBase identifica il caso d'uso generico. Il CodiceSottoCaso (opzionale) identifica i sottocasi.

Tabella 4.13: Tabella tracciamento dei requisiti funzionali

Requisito	**Descrizione**
R1F-1	L'utente deve poter fare il login per accedere al sito
R1F-1.2	Il sistema deve fare il display dell'errore di autenticazione
R1F-2	L'utente non autenticato deve potersi registrare
R1F-2.1	Il sistema deve fare il display dell'errore di registrazione
R1F-3	L'utente deve poter visualizzare la dashboard
R1F-3.1	L'utente deve poter visualizzare le sfide in cui è coinvolto come sfidato
R1F-3.1	L'utente deve poter visualizzare le sfide in cui è coinvolto come sfidante
R1F-3.1	L'utente deve poter visualizzare le sfide in cui è coinvolto come valutatore
R1F-3.2	L'utente deve poter accettare una sfida che gli viene proposta
R1F-3.3	L'utente deve poter cancellare una sfida lanciata
R1F-3.4	L'utente deve poter segnare come completata una sfida
R1F-3.5	L'utente deve poter arrendersi in una sfida
R1F-3.6	L'utente valutatore deve poter valutare con succeso una sfida
R1F-3.7	L'utente valutatore deve poter valutare il fallimento di una sfida
R1F-4	L'utente deve poter inserire una nuova sfida
R1F-4.1	L'utente deve poter selezionare chi sfidare
R1F-4.2	L'utente deve poter specificare il titolo della sfida
R1F-4.3	L'utente deve poter specificare il dettaglio della sfida
R1F-4.4	L'utente deve poter inserire il termine per completare la sfida
R1F-5	L'utente deve poter visualizzare lo storico delle sfide
R1F-6	L'utente deve poter visualizzare il dettaglio di una sfida
R1F-6.1	L'utente deve poter visualizzare il dettaglio cronologico della sfida
R1F-7	L'utente deve poter decidere quali notifiche ricevere

Tabella 4.14: Tabella del tracciamento dei requisiti di vincolo

Requisito	**Descrizione**

R1V-1	Il backend deve essere realizzato tramite il framework Spring
R1V-2	Il frontend deve essere realizzato tramite il framework Angular
R1V-3	Il backend deve adottare un'architettura a microservizi
R1V-3	La persistenza dei dati deve essere garantita con un database PostgreSQL
R1V-4	La GUI deve essere realizzata mediante l'utilizzo della libreria *Bootstrap*

4.1.3 Problematiche riscontrate

Dall'analisi preventiva dei rischi sono state individuate alcune possibili problematiche a cui si potrà andare incontro:

1. **Conoscenza delle tecnologie**
    - **Descrizione:** possibile rallentamento nell'attività di sviluppo a causa dell'utilizzo delle nuove tecnologie acquisite nella fasi di studio individuale 2
    - **Soluzione:** autoverifica periodica delle conoscenze acquisite, testandole con piccoli progetti
2. **Inclusione di nuovi requisiti**
    - **Descrizione:** l'utilizzo di un metodo di sviluppo *agile* potrebbe complicare l'aggiunta di nuovi requisiti da soddisfare in corso d'opera
    - **Soluzione:** coinvolgere il committente, in questo caso il tutor, in una ciclica analisi dei requisiti, così da intervenire solamente per piccoli aggiustamenti

4.2 Progettazione

L'architettura della componente *backend* del progetto *Challenginator* è realizzata con i microservizi, ovvero servizi di piccole dimensioni che per poter comunicare tra loro utilizzano API ben definite. La comunicazione con il lato *frontend* avviene utilizzando chiamate all'API Gateway, microservizio che si interpone tra *frontend* e *backend*. La comunicazione tra microservizi avviene mediante il broker RabbitMQ.
Il progetto è composto dai seguenti microservizi:

- api-gateway
- challenge-service
- eurekaserver
- logger-service
- notification-service
- scheduler-service
- user-service

Il database che permette il salvataggio permanente dei dati è composto dalle seguenti tabelle:
Tabella `app_user` per salvare i dati riguardanti gli utenti. Le password prima di esser salvate saranno crittografate.

app_user
id
app_user_role
boss_id
email
enabled
locked
name
password
score
surname

Tabella 4.15: Tabella `app_user` del database `challenginator`

Tabella `log` per salvare i log dei vari microservizi.

log
id
level
log_message
service_name
time

Tabella 4.16: Tabella `log` del database `challenginator`

Tabella `challenge` per salvare i dettagli delle singole sfide.

challenge
id
challenged
challenger
deadline
description
evaluator
result
status
timestamp_acceptance
timestamp_creation
title

Tabella 4.17: Tabella `challenge` del database `challenginator`

Tabella `user_preference` per salvare le preferenze di notifica di ogni utente. I campi di questa tabella (escluso `id`) sono di tipo `boolean`.

user_preference
userid
challenge_accepted
challenge_completed
challenge_created
challenge_deleted
challenge_giveup
challenge_refused
challenge_terminated
challenge_updated

Tabella 4.18: Tabella `user_preference` del database `challenginator`

Capitolo 5

Servizio di notifica

Il servizio di notifica implementato nell'applicazione utilizza RabbitMQ per raccogliere le informazione da trasmettere, genera il messaggio e poi lo spedisce mediante e-mail. La struttura del microservizio è riportata di seguito:

```
notificationservice
├── challenge
│   ├── Challenge.java
│   ├── ChallengeEventType.java
│   ├── ChallengeResult.java
│   └── ChallengeStatus.java
├── mail
│   ├── EmailService.java
│   ├── EmailServiceConfig.java
│   └── SimpleMailMessageBuilder.java
├── messaging
│   └── Receiver.java
├── user
│   ├── AppUser.java
│   ├── UserPreference.java
│   ├── UserPreferenceRepository.java
│   └── UserPreferenceService.java
├── AppConfig.java
├── Controller.java
└── NotificationServiceApplication.java
```

5.1 Preferenze notifiche

Nella web app vi è un'apposita sezione per le preferenze di notifica per avere un controllo granulare su quest'ultime. Come si può notare dalla figura 5.1 vi sono vari campi dotati di check-box per poter selezionare quali notifiche ricevere.

Figura 5.1: Preferenze di notifica webapp

Al primo accesso si spuntano i check-box interessati e si salvano le proprie preferenze. Nel salvataggio viene creata una nuova istanza nella tabella `user_preference`, contenente i campi scelti dall'utente, così da salvare il tutto permanentemente.

Listing 5.1: Frammento codice preferenze utente

```
1  @Entity
2  public class UserPreference {
3      @Id
4      private Long userID;
5      private boolean CHALLENGE_CREATED;
6      private boolean CHALLENGE_ACCEPTED;
7      private boolean CHALLENGE_REFUSED;
8      private boolean CHALLENGE_COMPLETED;
9      private boolean CHALLENGE_UPDATED;
10     private boolean CHALLENGE_DELETED;
11     private boolean CHALLENGE_GIVEUP;
12     private boolean CHALLENGE_TERMINATED;
13 }
```

Come si può notare dal precedente frammento di codice, le istanze nel database hanno come *primary key* l'id utente, mentre i rimanenti campi sono `boolean` dato che il dato contenuto è la preferenza dei check-box.
Dopo l'operazione di aggiornamento delle preferenze, durante il flusso dell'applicazione (utenti si sfidano, utenti completano le sfide, ..), verranno inviate ai rispettivi utenti le notifiche richieste: il mittente è `rabbitmailtest@gmail.com`, indirizzo creato appositamente per questa parte del progetto, l'oggetto sarà l'evento che ha scatenato l'invio dell'email ed il corpo del messaggio conterrà le informazioni specifiche, quali sfidato, sfidante, giudice, tempo di completamento, titolo e descrizione della sfida.

Figura 5.2: Esempio mail

5.2 Generazione e-mail

La generazione e l'invio delle e-mail sono gestiti dalla classe `EmailService`, appartenente alla sezione *mail* del servizio. Quest'ultima delega la configurazione del servizio mail (protocollo, porta, nome utente, password, ...) a `EmailServiceConfig` e la costruzione della mail stessa (mittente, ricevitore, oggetto e corpo) a `SimpleMailMessageBuilder`

Listing 5.2: Generazione corpo email

```
1 public void process(Challenge challenge, ChallengeEventType eventType)
2 {
3     try {
4         AppUser challenger = getUser(challenge.getChallenger());
5         AppUser challenged = getUser(challenge.getChallenged());
6         AppUser evaluator = getUser(challenge.getEvaluator());
```

```
7
8           SimpleMailMessageBuilder mailBuilder =
9               new SimpleMailMessageBuilder(
10              challenge,
11              eventType,
12              emailServiceConfig.username,
13              challenger,
14              challenged,
15              evaluator);
16          SimpleMailMessage message = mailBuilder.build();
17          emailSender.send(message);
18      } catch (Exception e) {
19          e.printStackTrace();
20      }
21  }
```

L'intero meccanismo di invio è messo in moto dalla classe `Receiver`, appartenente alla sezione *messaging*, che mediante RabbitMQ si mette in ascolto dei nuovi messaggi nella coda `challengeQueue`. All'arrivo di un nuovo messaggio nella coda viene richiamato il metodo della classe `EmailService` sopra riportato.

Listing 5.3: Ricezione update ed invio notifica

```
1  @EnableRabbit
2  @Service
3  public class Receiver {
4      @Autowired
5      EmailService emailService;
6      static final public String queueName = "challengeQueue";
7      @RabbitListener(queues = queueName)
8      public void receiveMessage(byte[] newChallenge,
9          @Header(name="event_type") String eventType) {
10     ObjectMapper mapper = new ObjectMapper();
11     Challenge challenge = null;
12     try {
13       challenge = mapper.readValue(new String(newChallenge),
14         Challenge.class);
15     } catch(JsonProcessingException e) {
16       e.printStackTrace();
```

```
17          }
18          emailService.process(challenge,
19                  ChallengeEventType.valueOf(eventType));
20      }
21  }
```

Capitolo 6

Logger

6.1 Log4j

In prima analisi per il progetto si era pensato di utilizzare *Log4j*, dato il larghissimo impiego, per il logging degli eventi da salvare poi su un file. Per poterlo utilizzare all'interno di Challenginator si sono aggiungente le dovute dipendenze, mediante *Maven*, e un file `log4j2-spring.xml` dove venivano specificate le impostazioni del logger per ogni microservizio coinvolto.

Nel contempo però è emersa una vulnerabilità che ha scosso l'intero mondo informatico: si è scoperta la falla *Log4Shell* (`CVE-2021-44228`) che permetteva ad un utente malintenzionato di controllare i messaggi di registro o i parametri dei messaggi di registro. In questa maniera era possibile eseguire codice arbitrario caricato dai server LDAP quando è abilitata la sostituzione della ricerca dei messaggi.

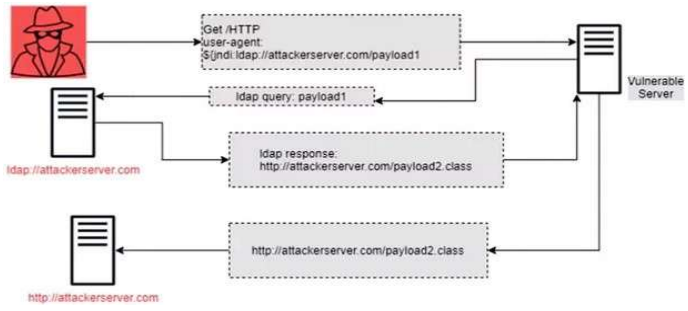

Figura 6.1: Funzionamento attacco mediante Log4Shell. Fonte: LFFL

Il CISA ha fornito una chiara rappresentazione, mediante uno schema chart, del grado di vulnerabilità del proprio sistema se si utilizza *Log4j*. [2]

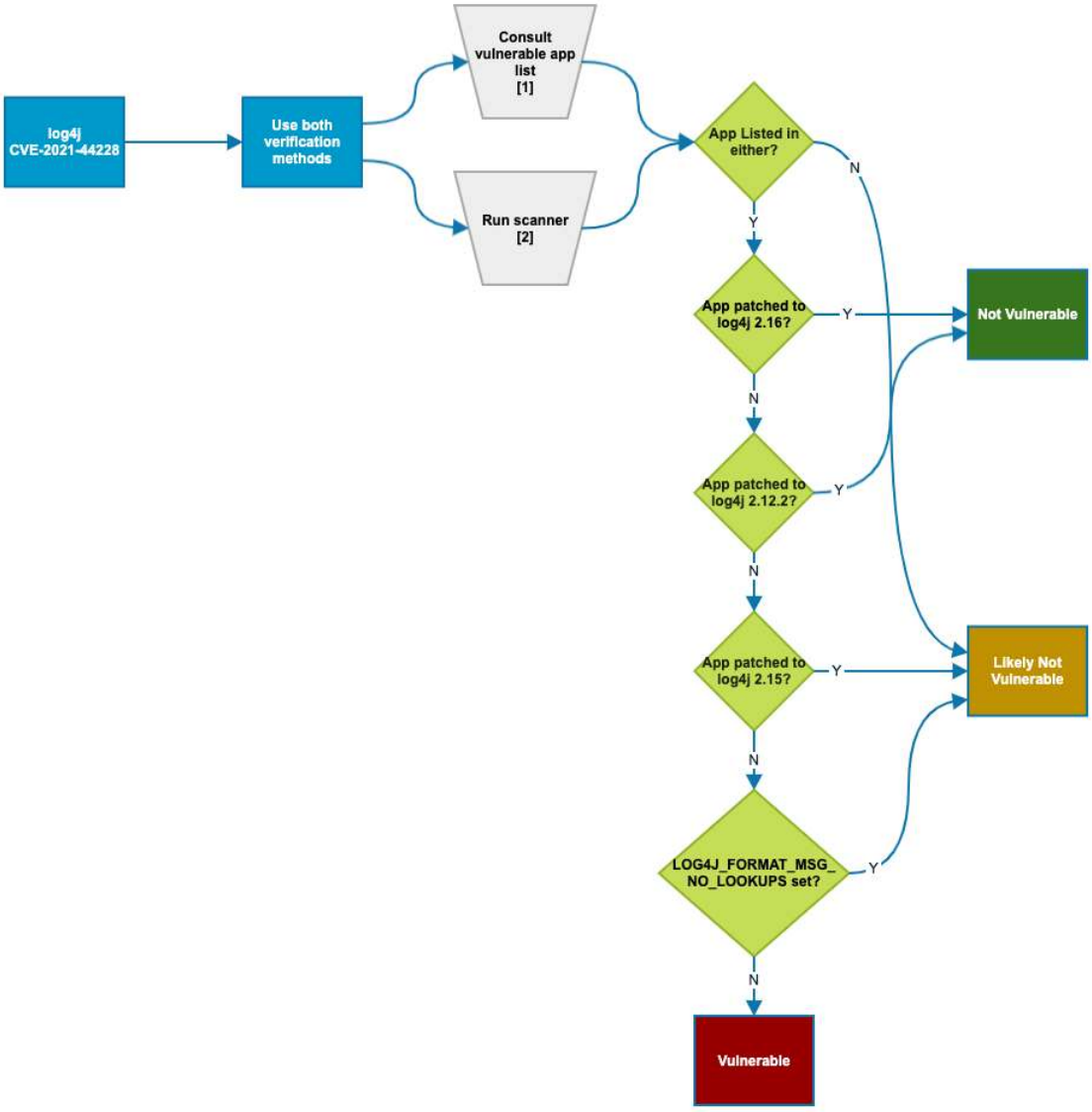

Figura 6.2: Grafico sulla vulnerabilità di prodotti che utilizzano Log4j. Fonte: CISA

Dopo questa vulnerabilità abbiamo deciso di affidarci ad un altro progetto per il logging: Logback.

6.2 Logback

Logback [10] è uno dei più utilizzati framework per il logging nella community Java dato che, rispetto a Log4j, offre una più veloce implementazione, più opzioni di configurazione e maggiore flessibilità nell'archiviazione di vecchi file.

Per inserire questo sistema di log all'interno del progetto è necessario inserire le relative dipendenze maven nel file `pom.xml`:

Listing 6.1: Dipendenze Maven Logback

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.2.10</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.10</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.32</version>
</dependency>
```

Per la configurazione è necessario creare un file di nome `logback-spring.xml` ed inserirlo nel progetto. In questo file si possono configurare gli *appender*, all'interno dei quali Logback scriverà i messaggi di log; nel progetto inizialmente i log erano riportati nel terminale dell'applicazione e salvati in un file, successivamente con l'adozione di Docker venivano inviati direttamente ad una tabella del database.

Per poter registrare gli eventi bisogna, all'interno delle singole classi, dichiarare una variabile `Logger` ed inviare i singoli messaggi.

Listing 6.2: Esempio Logger Logback

```
public class Example {
    private static final Logger logger
```

```
3         = LoggerFactory.getLogger(Example.class);
4
5     public static void main(String[] args) {
6         logger.info("Example log {}",Example.class.getSimpleName());
7     }
8 }
```

[9]. Le funzioni da chiamare dipendono dal livello di importanza dell'informazione da trasmettere:

- `.error` per il livello ERROR. Indica un errore di esecuzione o una condizione improvvisa.
- `.warn` per il livello WARN. Indica una condizione inaspettata o anomala di esecuzione, che però non necessariamente ha comportato un errore.
- `.info` per il livello INFO. Usato per segnalare gli eventi di esecuzione.
- `.debug` per il livello DEBUG. Usato nella fase di debug dell'applicazione.

Questa soluzione sembrava quella ottimale, ma sfortunatamente la comunicazione tra l'applicazione e il database non si instaurava e abbiamo dovuto intraprendere una nuova strada, una soluzione custom per il logging.

6.3 Logger Custom

La configurazione personalizzata è composta da un nuovo servizio denominato *logger_service* che raccoglie tutti i messaggi di log dai vari microservizi e li invia al database ed una nuova componente in ogni microservizio atta all'invio dei messaggi di log al servizio dedicato. La struttura del microservizio è riportata di seguito:

```
logger_service
├── controller
│   └── Controller.java
├── log
│   ├── Log.java
│   ├── LogLevel.java
│   └── LogRepository.java
└── messaging
```

```
    ├── LogReceiver.java
    └── MessagingConfig.java
├── LogService.java
└── LoggerServiceApplication.java
```

6.3.1 Logger service

Questo nuovo servizio è composto da principalmente tre parti:

- *controller*: riceve delle *request* tramite URL e genera le *response* corrispondenti
- *log*: genera la classe `Log`, crea la relativa tabella nel database e si occupa delle Java Persistence API
- *messaging*: configura RabbitMQ per questo servizio, riceve le informazioni sui log dai vari microservizi e richiede il salvataggio persistente di questi ultimi

Per la creazione del *controller* si utilizzano le potenzialità del framework Spring. Mediante l'annotazione `RestController` (include le annotazioni `Controller` e `ResponseBody`) si ottiene un *controller* specializzato, capace di gestire le *request* mediante metodi in grado di trasformare automaticamente gli oggetti ritornati in *HttpResponse*. L'annotazione `GetMapping` gestisce le chiamate GET al link specificato; tali percorsi sono esposti alla porta 8080 mediante l'*API gateway*.

Listing 6.3: Controller logger service

```
1  @RestController
2  public class Controller {
3      @Autowired
4      LogService logService;
5      // restituisce tutti i log
6      @GetMapping("/logs/getalllogs")
7      public List<Log> getController() {
8          return logService.getAll();
9      }
10     // controller di prova, aggiunge log fasullo
11     @GetMapping("/logs/add")
12     public String addLog() {
13         logService.save(
```

```
14                  new Log(null,"now",LogLevel.WARN,"service","msg"));
15              return "";
16          }
17      }
```

L'oggetto di tipo `LogService` richiama l'omonima classe, preceduta dall'annotazione `Service` per indicare che internamente è contenuta la logica.

Listing 6.4: Frammento della classe LogService

```
1  @Service
2  public class LogService {
3      @Autowired
4      private LogRepository logRepository;
5      public void save(Log Log) {
6          logRepository.save(Log);
7      }
8      public List<Log> getAll() {
9          return logRepository.findAll();
10     }
11 }
```

L'oggetto di tipo `LogRepository`, che richiama anch'esso l'omonima classe, è utilizzato per la comunicazione con il DBMS. Fa parte della sezione *log* di questo servizio, atta alla creazione della classe `Log` e della scrittura persistente.
La classe `Log`, mediante l'annotazione `Entity` indica sé stessa come modello della tabella da creare nel DBMS. Le prime tre annotazioni permettono all'oggetto `id` di essere una *primary key* generata automaticamente in modo incrementale, mentre `Enumerated` permette a `level` di convertire un tipo `Enum` in stringa.

Listing 6.5: Frammento della classe Log

```
1  @Entity
2  public class Log {
3      @Id   //PK
4      @SequenceGenerator(
5              name = "log_sequence",
6              sequenceName = "log_sequence",
7              allocationSize = 1)
```

```
8       @GeneratedValue(strategy = GenerationType.SEQUENCE,
9       generator = "log_sequence")
10      private Long id;
11      private String time;
12      @Enumerated(EnumType.STRING)
13      private LogLevel level;
14      private String service_name;
15      private String log_message;
16  }
```

Infine la sezione *messaging* è composta da un file di configurazione per RabbitMQ in cui vengono specificati il nome della coda e dell'*exchange*, e da una classe `LogReceiver` che mediante il tag `@RabbitListener` si mette in ascolto nella coda apposita per poter inviare i messaggi di log a `LogService` mediante l'annotazione `RabbitListener`

Listing 6.6: Classe LogReceiver

```
1  @Component
2  public class LogReceiver {
3      @Autowired
4      LogService logService;
5      @RabbitListener(queues = MessagingConfig.QUEUE)
6      public void consumeMessageFromQueue(Log log) {
7          System.out.println("\nMsg from: " + log.getService_name()
8              + " with msg: \n" + log.getLog_message());
9          logService.save(log);
10     }
11 }
```

6.3.2 Componente microservizi

Nei vari microservizi che utilizzano questo nuovo sistema di log si sono dovute aggiungere le relative dipendenze per interfacciarsi al Database Management System, una sezione *messaging* ed un oggetto di tipo `LogService` per poter inviare i messaggi di log. Di seguito la struttura della sezione *messaging*:

```
messaging
└── Log.java
```

```
├── LogLevel.java
├── LogService.java
└── MessagingConfig.java
```

Listing 6.7: Invio dati di log

```
1 logService.log("Token di autenticazione creato",LogLevel.INFO);
```

La differenza tra questo servizio di log e quello precedente è che quest'ultimo si preoccupa di ricevere i dati dei log e di inviarli mediante RabbitMQ.

Listing 6.8: Frammento della classe LogService - microservizi

```
1  @Service
2  public class LogService {
3      @Autowired
4      private RabbitTemplate template;
5      @Value("${spring.application.name}")
6      private String appName;
7      DateTimeFormatter formatter =
8          DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
9      public Log log(String msg, LogLevel ll) {
10         Log log = new Log(null,
11             LocalDateTime.now().format(formatter).toString(),
12             ll,appName,msg);
13         template.convertAndSend(MessagingConfig.EXCHANGE,
14             MessagingConfig.QUEUE, log);
15         return log;
16     }
17 }
```

Capitolo 7

Conclusioni

In questo capitolo sono riportate le considerazioni riguardanti l'attività di stage.

7.1 Raggiungimento degli obiettivi

Lo stage si è svolto rispettando i tempi prefissati e tutti gli obiettivi obbligatori, desiderabili e facoltativi sono stati terminati seguendo il piano di lavoro redatto prima dell'inizio dello stage.

- Obbligatori:
  - Acquisizione delle competenze sulle tematiche dell'architettura a microservizi, del framework Spring e del broker RabbitMQ
  - Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma
  - Portare a termine le implementazioni previste con una percentuale di superamento pari al 80%
- Desiderabili:
  - Portare a termine le implementazioni previste con una percentuale di superamento pari al 100%
- Facoltativi:

- Dockerizzare le componenti su container
- Creare un sistema di log

7.2 Conoscenze acquisite

Nella sfera delle conoscenze acquisite sono molto soddisfatto di essermi potuto avvicinare allo sviluppo software lato *back-end*. Il progetto mi ha permesso di studiare a fondo il *framework* Spring di cui avevo solamente sentito parlare e di capire cos'è e come funziona un broker di messaggi. Il periodo di tirocinio mi ha anche permesso di dare uno sguardo al mondo del lavoro inserendomi nelle dinamiche aziendali e permettendomi di collaborare con colleghi più esperti nel settore, i quali mi hanno dato utili consigli.

Acronimi

AMQP Advanced Message Queuing Protocol. 10, 11

API Application Programming Interface. 5, 9, 12

DBMS Database Management System. 2, 13, 15, 42, 43

FIFO First In First Out. 11

GDPR General data protection regulation. 1

ICT Information and Comunication Technology. 1

IoT Internet of Things. 1

JPA Java Persistence API. 9, 41

REST Representational state transfer. 2, 9

Glossario

Design Pattern in informatica e specialmente nell'ambito dell'ingegneria del software, è un concetto che può essere definito "una soluzione progettuale generale ad un problema ricorrente". 7, 13

PostgreSQL è un completo DBMS ad oggetti rilasciato con licenza libera (stile Licenza BSD). Spesso abbreviato come "Postgres", sebbene questo sia un nome vecchio dello stesso progetto, è una reale alternativa sia rispetto ad altri prodotti liberi come MySQL, Firebird SQL e MaxDB che a quelli a codice chiuso come Oracle, IBM Informix o DB2 ed offre caratteristiche uniche nel suo genere che lo pongono per alcuni aspetti all'avanguardia nel settore delle basi di dati. 2, 13

RabbitMQ è un broker di messaggi che implementa il protocollo AMQP. 2, 3, 10–12, 26

Spring è un *framework open-source* per lo sviluppo di applicazioni su piattaforma *Java*. 2, 3, 9

Sitografia

[1] *Angular*. URL: https://angular.io.

[2] *Apache Log4j*. URL: https://logging.apache.org/log4j/2.x.

[3] *Apache Maven*. URL: https://maven.apache.org/.

[4] Marco Canovese. "Sviluppo full stack di una web app per la gestione di task aziendali".

[5] *DBeaver Community*. URL: https://dbeaver.io.

[6] *Docker*. URL: https://www.docker.com.

[7] *Git*. URL: https://git-scm.com.

[8] Romana Gnatyk. *Microservices vs Monolith: which architecture is the best choice for your business?* URL: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business.

[9] Eric Goebelbecker. *A Guide to Logback*. URL: https://www.baeldung.com/logback.

[10] *Logback*. URL: https://logback.qos.ch.

[11] *Postman*. URL: https://www.postman.com.

[12] *Protocollo AMQP*. URL: https://it.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol.

[13] *Sync Lab*. URL: https://www.synclab.it.

[14] Java Techie. *Spring Boot RabbitMQ — Publisher & Subscriber Example — AMQP*. URL: https://www.youtube.com/watch?v=o4qCdBR4gUM.

[15] *Visual Studio Code IDE*. URL: https://code.visualstudio.com.