Quinto Sprint

Nel panorama moderno dello sviluppo software, il modo in cui costruiamo e distribuiamo le applicazioni ha subito trasformazioni radicali. Una delle evoluzioni più notevoli è stata la **transizione** da architetture **monolitiche** a quelle basate su **microservizi**.

In un sistema monolitico, l'intera applicazione è costruita come un **singola unità** coesa. Tutte le funzionalità, dal gestire gli accessi degli utenti alla gestione dei dati, sono racchiuse in una singola codebase e vengono eseguite come un'unica entità. Sebbene questo approccio offra una semplicità iniziale, porta con sé una serie di sfide:

- **Scalabilità**: Scalare un'applicazione monolitica richiede di replicare l'intero monolite, anche se solo una piccola porzione dell'applicazione sta vivendo un carico pesante.

- **Manutenibilità**: Con il tempo, il codice può diventare ingombrante e difficile da gestire, rendendo complesse anche modifiche apparentemente semplici.

- **Tempi di Rilascio**: Il rilascio di nuove funzionalità può richiedere il deploy dell'intera applicazione, aumentando i rischi e le complicazioni.

Per superare questi ostacoli, l'architettura a **microservizi** è emersa come una soluzione promettente. In questo modello, un'applicazione è suddivisa in piccole unità o "servizi", ognuno dei quali ha **una funzione specifica** e opera in modo **indipendente** dagli altri. Questi servizi comunicano tra loro attraverso API leggere e protocolli standardizzati.

Le architetture basate su microservizi offrono diversi vantaggi:

- **Resilienza**: Un singolo servizio che fallisce non porta necessariamente all'indisponibilità dell'intera applicazione.

- **Scalabilità Granulare**: È possibile scalare solo i servizi che necessitano di più risorse, piuttosto che l'intera applicazione.

- **Rilasci Indipendenti**: I servizi possono essere sviluppati, testati e rilasciati indipendentemente l'uno dall'altro, accelerando il ciclo di sviluppo e riducendo i rischi.

- **Poliglottismo**: Ogni microservizio può essere scritto in un linguaggio di programmazione differente, in base alle esigenze specifiche del servizio.

- **Database separati**: Ogni microservizio può possedere il proprio database ed è responsabile dei propri dati e della propria logica di business.

Struttura Microservizi Netflix

Netflix è stata una delle prime aziende ad adottare la soluzione dei microservizi. La loro storia, infatti, rappresenta una vera e propria evoluzione nell'**approccio architetturale**. All'inizio, Netflix operava come un servizio monolitico, ma con l'impetuosa crescita del numero di utenti e l'espansione globale, è emersa la necessità di una soluzione più **scalabile**, **resiliente** e **veloce**. Questa necessità ha portato alla decisione di adottare un'architettura a microservizi.

Nel tentativo di massimizzare l'efficienza, Netflix ha scomposto le sue vaste funzionalità in **centinaia di microservizi indipendenti**. Questo ha permesso a ogni microservizio di concentrarsi su funzioni specifiche, che spaziano dalla gestione degli utenti alle raccomandazioni di contenuti, passando per la gestione dei pagamenti. Questa divisione granulare ha portato anche a una notevole **autonomia tra i team di sviluppo**. Ogni team, ora, ha la libertà di selezionare le piattaforme e le tecnologie che ritiene più adatte alle proprie esigenze, generando così un ecosistema tecnologico incredibilmente variegato.

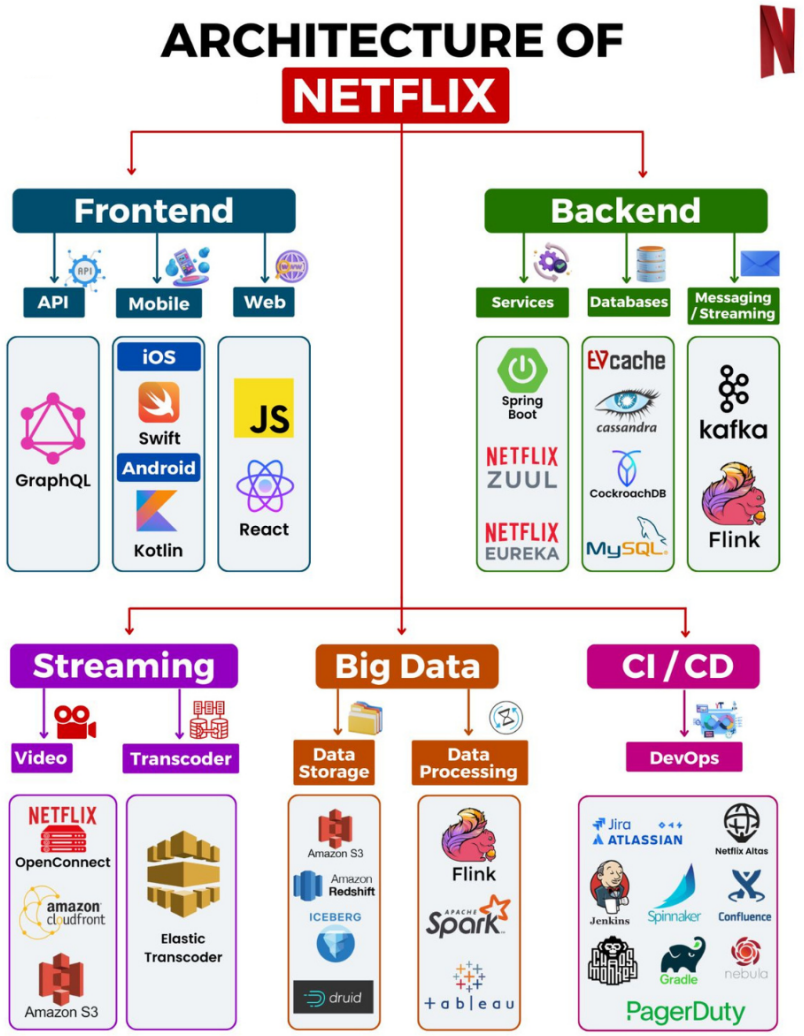
Una parte fondamentale dell'approccio ai microservizi di Netflix è la sua dedizione ai **database indipendenti.** Ogni servizio ha un datastore dedicato, ottimizzato per le sue particolari esigenze. Netflix ha inoltre introdotto un proprio API **Gateway**, denominato "**Zuul**", per smistare le richieste ai microservizi appropriati, garantendo così un efficiente flusso di comunicazione.

La **resilienza**, ovvero la capacità di un sistema di recuperare rapidamente da difficoltà, è al cuore delle preoccupazioni di Netflix. Per garantire la massima disponibilità, hanno creato strumenti innovativi come "Chaos Monkey", che introduce volutamente guasti nel sistema per testarne la robustezza.

Con centinaia di servizi in esecuzione, il monitoraggio e il tracciamento diventano essenziali. Netflix ha risposto a questa esigenza sviluppando strumenti come "Atlas" per il monitoraggio in tempo reale. Ogni microservizio può essere **sviluppato, testato e rilasciato** in modo indipendente, permettendo a Netflix di innovare costantemente.

Ma questa transizione non riguarda solo la tecnologia. Alla base del successo di Netflix c'è una cultura aziendale solida, fondata sulla "libertà e responsabilità". I team sono incoraggiati a prendere iniziative, ma allo stesso tempo sono responsabili delle conseguenze delle loro decisioni.

Questa è solo un'analisi superficiale dell'architettura di Netflix, ma serve a comprendere quali possono essere i vantaggi e le difficoltà durante una transizione da un'architettura monolitica ad una a microservizi.



Service Discovery

Il Service Discovery gioca un ruolo fondamentale nel tenere traccia della posizione e dello stato dei vari microservizi in un sistema distribuito. Ma non si tratta solo di "memorizzare" la posizione, si tratta di fornire un sistema dinamico che può adattarsi alle variazioni in tempo reale. Ecco un riassunto delle sue funzioni principali:

- **Registro Dinamico**: Quando un microservizio viene avviato (spesso su un contenitore o un'istanza di cloud che potrebbe avere un indirizzo IP dinamico), registra se stesso presso un sistema di Service Discovery, indicando dove può **essere raggiunto** (indirizzo IP, porta, ecc.) e potenzialmente altri metadati, come la versione del servizio o informazioni sulla salute.

- **Scoperta in Tempo Reale**: I servizi che hanno bisogno di comunicare con altri servizi possono **interrogare** il Service Discovery per trovare la **posizione** corrente di un servizio. Dato che le istanze dei servizi possono venire avviate o fermate dinamicamente, o potrebbero subire guasti, la posizione e la disponibilità di un servizio possono cambiare frequentemente.

- **Gestione della Salute**: Molti sistemi di Service Discovery includono anche meccanismi di controllo dello stato di salute (**health checks**). Questi controlli permettono di verificare periodicamente se un servizio è operativo e reattivo. Se un servizio fallisce un controllo di salute, può essere automaticamente rimosso dal registro, prevenendo così che altre parti del sistema tentino di inviare richieste a un servizio non funzionante.

- **Load Balancing**: Alcune soluzioni di Service Discovery offrono, o possono essere integrate con, funzionalità di **bilanciamento** del carico. Quando un servizio chiede la posizione di un altro servizio, il sistema di Service Discovery potrebbe fornire l'indirizzo di un'istanza particolare in base a vari criteri, come il bilanciamento del carico o la vicinanza geografica.

Gateway

Il Gateway nei microservizi è un componente essenziale che funge da **punto di ingresso** nell'ecosistema dei microservizi. Riceve le richieste esterne e le indirizza ai servizi appropriati. Ecco una spiegazione dettagliata del Gateway in un'architettura basata su microservizi:

- **Punto di Ingresso Unico**: Il Gateway fornisce un singolo punto di ingresso per tutte le richieste esterne che arrivano all'applicazione. Questo **centralizza** la gestione delle richieste, rendendo più semplice l'implementazione di preoccupazioni cross-cutting come l'autenticazione, la registrazione e il bilanciamento del carico.

- **Routing delle Richieste**: Una delle funzioni principali del Gateway è **dirigere le richieste** ai microservizi appropriati. Quando una richiesta arriva, il Gateway decide quale servizio deve gestire quella richiesta basandosi su informazioni come l'URL, il metodo HTTP e potenzialmente l'header o il corpo della richiesta.

- **Bilanciamento del Carico**: Il Gateway può **bilanciare** il carico tra diverse istanze di un microservizio. Ad esempio, se ci sono tre istanze del servizio "utenti", il Gateway può distribuire le richieste tra queste istanze in modo equilibrato o basato su altri criteri.

- **Autenticazione e Autorizzazione**: Prima che una richiesta raggiunga un microservizio, il Gateway può verificarla per assicurarsi che l'utente o il sistema che fa la richiesta abbia le credenziali appropriate. Se una richiesta non è autenticata o autorizzata, il Gateway può rifiutarla prima che raggiunga qualsiasi microservizio.

- **Aggregazione delle Risposte**: In alcune situazioni, una singola richiesta client potrebbe richiedere dati da più microservizi. Il Gateway può aggregare le risposte da vari servizi e fornire un'unica risposta al client.

- **Limitazione delle Richieste (Rate Limiting)**: Per evitare il sovraccarico dei servizi a causa di un volume elevato di richieste, il Gateway può implementare la limitazione delle richieste, garantendo che un particolare client o servizio possa fare solo un certo numero di richieste in un determinato periodo.

- **Protezione dalle Minacce**: Il Gateway può anche fungere da una sorta di firewall, proteggendo i microservizi da minacce come attacchi DDoS, attacchi di iniezione e altro.

- **Monitoraggio e Logging**: Essendo il punto di ingresso per tutte le richieste, il Gateway è in una posizione ideale per monitorare il traffico e registrare informazioni utili, come tempi di risposta, origine delle richieste e tipi di errore.

- **Trasformazione delle Richieste/Risposte**: In alcuni scenari, potrebbe essere necessario trasformare una richiesta o una risposta prima che raggiunga il client o un servizio. Il Gateway può modificarle, ad esempio, convertendo da XML a JSON o viceversa.

- **Versionamento**: Se un'organizzazione ha più versioni di un servizio in produzione, il Gateway può indirizzare le richieste alla versione appropriata in base a informazioni come headers o l'URL della richiesta.

In sintesi, il Gateway nei microservizi agisce come un mediatore tra i client esterni e i servizi interni, semplificando le interazioni, migliorando la sicurezza e offrendo una serie di funzionalità che migliorano la scalabilità e la manutenibilità dell'intero sistema.

Comunicazione tra Microservizi

Quando si lavora con un'architettura a Microservizi potremo avere la necessità di **comunicare** da un microservizio ad un altro. Ad esempio supponiamo di avere due microservizi:

- **Servizio Ristorante**: Incaricato di gestire gli articoli serviti da un ristorante, elaborare ordini e simili.

- **Servizio Utente**: Incaricato della gestione dei dettagli degli utenti.

Immaginiamo ora che un utente visiti la pagina di un ristorante, selezioni una cucina di sua scelta e faccia clic sul pulsante "Effettua Ordine". Quando questa richiesta arriva al Servizio Ristorante, il servizio dovrà validare i dettagli dell'utente prima di elaborare l'ordine. Tuttavia, i dettagli del nostro utente sono gestiti dal Servizio Utente. Ecco quindi l'esigenza di comunicare dal Servizio Ristorante al Servizio Utente

La comunicazione può essere di due tipi **sincrona** o **asincrona**, vediamo le differenze principali tra questi 2 metodi.

Comunicazione Sincrona:

- Quando un servizio (ad esempio, il Servizio A) fa una chiamata HTTP ad un altro servizio (Servizio B), **attende** una risposta da quel servizio prima di proseguire.
- Durante questo tempo, il Servizio A potrebbe non essere in grado di **gestire altre richieste** o potrebbe farlo in modo limitato, a seconda della sua configurazione e dell'infrastruttura sottostante.
- Una volta ricevuta la risposta dal Servizio B, il Servizio A può **procedere** con la sua elaborazione.
- Questo tipo di comunicazione è **diretto e sequenziale**. La comunicazione avviene tipicamente tramite chiamate RESTful.

Comunicazione Asincrona:

- In una comunicazione asincrona, quando il Servizio A invia un messaggio o una richiesta al Servizio B, **non attende** direttamente una risposta da esso. Invece, il Servizio A può continuare a gestire altre richieste o elaborare altre attività.
- Il Servizio B, **dopo aver ricevuto ed elaborato** il messaggio, può inviare una risposta o un avviso al Servizio A attraverso un altro canale o meccanismo (ad esempio, **code di messaggi**, **eventi**, ecc.).
- Anche se le chiamate HTTP possono essere utilizzate per iniziare una comunicazione asincrona, spesso si utilizzano sistemi di **messaggistica** come RabbitMQ, Kafka o SQS di Amazon per gestire e orchestrare tali comunicazioni.
- Questo tipo di comunicazione è particolarmente utile in scenari dove le operazioni possono richiedere **molto tempo**, e non ha senso o è inefficiente attendere una risposta immediata.

In questa sezione andremo ad analizzare solamente un metodo di comunicazione sincrona, ovvero il **Feign Client.**

Feign Client

Feign Client è un framework Java per la creazione di **client HTTP dichiarativi**, che semplifica il processo di connessione e di comunicazione tra microservizi. È particolarmente popolare in applicazioni basate su Spring Cloud, grazie alla sua integrazione nativa. Feign fornisce un modo più pulito e più semplice di scrivere client API rispetto all'utilizzo diretto di RestTemplate in Spring.

Caratteristiche principali:

- **Interfaccia Dichiarativa**: Con Feign, definiamo un'interfaccia Java annotando i suoi metodi con le annotazioni Spring MVC (@RequestMapping, @PathVariable, ecc.). Non c'è bisogno di implementare manualmente questa interfaccia.

- **Integrazione con Ribbon**: Feign è integrato con Ribbon per fornire **bilanciamento del carico client-side**. Ciò significa che se hai più istanze di un microservizio, Feign e Ribbon possono distribuire le richieste tra di esse.

- **Decoder e Encoder Personalizzabili**: Puoi personalizzare le modalità con cui Feign **serializza** le richieste e deserializza le risposte, utilizzando librerie come Jackson o Gson.

- **Interceptor**: Feign supporta l'**intercettazione delle richieste**, il che è utile per casi d'uso come l'aggiunta di intestazioni personalizzate a tutte le richieste.

Implementazione di Feign Client

Per iniziare aggiungiamo la dipendenza **spring-cloud-starter-openfeign** al file pom.xml. Successivamente nella classe principale Spring Boot (quella annotata con @SpringBootApplication), aggiungiamo l'annotazione **@EnableFeignClients**.
A questo punto creiamo **un'interfaccia** Java per rappresentare il microservizio che vogliamo chiamare.
Ad esempio:

```
@FeignClient(name = "user-service", url =
"http://localhost:8080")
public interface UserServiceClient {
    @RequestMapping(method = RequestMethod.GET, value =
"/users/{userId}")
    User getUser(@PathVariable("userId") String userId);
```

```
}
```

In questo esempio, abbiamo definito un client Feign per un servizio utente con'URL "http://localhost:8080". Questa classe è come se rappresentasse il controller User che risiede nell'altro microservizio senza però implementarne i metodi.

Ora è possibile iniettare UserServiceClient in qualsiasi componente Spring e chiamare il suo metodo getUser per effettuare una chiamata HTTP al servizio utente:

```
@Service
public class SomeService {
    private final UserServiceClient userServiceClient;

    @Autowired
    public SomeService(UserServiceClient
userServiceClient) {
        this.userServiceClient = userServiceClient;
    }

    public User doSomethingWithUser(String userId) {
        return userServiceClient.getUser(userId);
    }
}
```

Gestione Database

La gestione dei database in un'architettura a microservizi è una delle sfide più complesse da affrontare. Nelle architetture monolitiche, è comune avere un unico database che serve l'intera applicazione. Tuttavia, nei microservizi, una pratica comune e consigliata è che ogni microservizio abbia il **proprio database privato**, o persino un **tipo di database** completamente diverso, a seconda delle proprie esigenze specifiche. Ciò garantisce che il microservizio sia del tutto **disaccoppiato** dagli altri, rendendo possibile modificarlo, scalare e distribuire in modo indipendente.

Vantaggi:

- **Isolamento**: Se un servizio fallisce, non influisce sugli altri.
- **Flessibilità Tecnologica**: Ogni microservizio può utilizzare il tipo di database che meglio si adatta alle sue esigenze. Ad esempio, un servizio potrebbe richiedere un database relazionale, mentre un altro potrebbe beneficiare di un NoSQL.

- **Scalabilità**: Ogni database può essere scalato indipendentemente in base alle esigenze del servizio specifico.

Sfide:

- **Consistenza**: Mantenere la coerenza dei dati tra più database è una sfida. Il concetto di transazione distribuita, in cui una singola operazione potrebbe coinvolgere più servizi e database, diventa complesso e spesso viene evitato in favore di pattern come la Saga.

- **Complessità**: Gestire più database e garantire la loro disponibilità, backup e manutenzione può diventare complicato.

Docker

Docker è una piattaforma software progettata per facilitare la creazione, il testing e la distribuzione di applicazioni all'interno di **Container**. Questi contenitori rappresentano ambienti isolati e leggeri, fornendo tutto il necessario per **l'esecuzione di un'applicazione**, inclusi codice, runtime e librerie. Una delle principali forze di Docker è la sua capacità di assicurare coerenza: un'applicazione contenuta in un contenitore Docker funzionerà in modo **uniforme** in diversi ambienti, nonostante le variazioni dei sistemi sottostanti. Nel corso di questo Sprint, adotteremo Docker come strumento chiave per il deploy della nostra applicazione sul server. Questo sarà realizzato attraverso la creazione di contenitori specifici, ciascuno dei quali avrà il compito di **ospitare i diversi microservizi** che compongono l'applicazione.
Andiamo ad analizzare i principali componenti di Docker:

Docker Client:

Il Docker Client è il primo componente di Docker e rappresenta l'interfaccia principale per gli utenti per interagire con Docker. Grazie alla sua **architettura client-server**, Docker può connettersi sia localmente che da remoto all'host. Quando un utente invia un comando a Docker, il client trasmette il comando all'host, che lo esegue attraverso l'API di Docker. Il client può comunicare con più host contemporaneamente senza problemi.

Docker Image:

Le immagini Docker sono modelli binari in sola lettura, scritti in YAML, utilizzati per **costruire i contenitori**. Contengono metadati che definiscono le capacità del contenitore. Ogni immagine è composta da diversi strati, dove ogni strato dipende da quello sottostante. Lo strato di base contiene il sistema operativo e l'immagine principale. Gli strati successivi contengono le dipendenze e le istruzioni, definite nel **Dockerfile**. Le immagini possono essere condivise all'interno di un'organizzazione

attraverso un registro privato o, se necessario, attraverso un registro pubblico come **Docker Hub**.
Pensiamo a un videogioco da installare sul computer. L'immagine Docker sarebbe come il file di installazione del gioco, che contiene tutte le istruzioni e i file necessari per far funzionare il gioco. Quando effettivamente installiamo e avviamo il gioco, stiamo creando una "istanza" del gioco, simile a un contenitore Docker che esegue un'applicazione.

In sintesi, un'immagine Docker è come **una ricetta o un modello** che contiene tutto ciò di cui un'applicazione ha bisogno per funzionare. Quando eseguiamo l'applicazione, creiamo un "contenitore" da quell'immagine.

Docker Daemon:

Il Docker Daemon è il componente che gestisce direttamente **le operazioni relative ai contenitori.** Funziona come un processo in background e gestisce reti Docker, volumi di archiviazione, contenitori e immagini. Quando viene dato un comando come docker run, il client lo traduce in una chiamata API HTTP e lo invia al daemon, che poi esegue la richiesta e interagisce con il sistema operativo. Il daemon risponde solo alle richieste dell'API di Docker.

Docker Networking:

Docker Networking permette la **comunicazione** tra contenitori. Docker offre cinque principali tipi di driver di rete:

- **None**: Disabilita completamente la rete, impedendo la comunicazione tra contenitori.
- **Bridge**: È il driver di rete predefinito utilizzato quando più contenitori devono comunicare sullo stesso host Docker.
- **Host**: Usato quando non è necessario l'isolamento tra un contenitore e l'host.
- **Overlay**: Permette la comunicazione tra diversi servizi in un swarm quando i contenitori sono su host diversi.
- **macvlan**: Assegna un indirizzo MAC ai contenitori, permettendo la comunicazione tra di loro attraverso questo indirizzo.

Docker Registry:

Il Docker Registry è il luogo dove vengono **conservate** le immagini Docker. Docker Hub è il registro predefinito per le immagini pubbliche. Tuttavia, i registri possono essere sia privati che pubblici. Con il comando docker pull, le immagini vengono scaricate dal registro, mentre con docker push vengono caricate.

Docker Container:

Un contenitore Docker è un'**istanza eseguibile** di un'immagine. I contenitori sono metodi leggeri e indipendenti per eseguire applicazioni. Possono essere connessi a una o più reti, creare nuove immagini basate sul loro stato attuale e vengono isolati l'uno dall'altro. Se un contenitore viene eliminato, qualsiasi applicazione o dato al suo interno viene anche rimosso.

DockerFile:

Quando si crea un'immagine Docker, è essenziale definire **l'ambiente** che accoglierà la nostra applicazione. La configurazione di tale ambiente dipende dalla natura dell'applicazione che intendiamo inserire. Il Dockerfile è un documento testuale che elenca una **serie di istruzioni** che Docker utilizza per costruire un'immagine. Queste direttive stabiliscono come allestire l'ambiente, quali file incorporare e quali comandi avviare. Dopo averlo redatto e impostato, è possibile utilizzare il Dockerfile per generare un'immagine Docker mediante il comando **docker build**.

Vediamo un esempio di DockerFile per un'applicazione SpringBoot:

```
# Utilizza l'immagine ufficiale di Java come base
FROM openjdk:11-jre-slim

# Crea una directory nel contenitore per ospitare l'applicazione
WORKDIR /app

# Copia il file JAR dell'applicazione Spring Boot (ad es. myapp.jar) nella directory /app del contenitore
COPY target/myapp.jar /app/myapp.jar

# Espone la porta 8080 per permettere la comunicazione con l'applicazione
EXPOSE 8080

# Comando da eseguire quando il contenitore viene avviato
ENTRYPOINT ["java", "-jar", "/app/myapp.jar"]
```

- **FROM openjdk:11-jre-slim**:Questa istruzione indica a Docker di utilizzare l'immagine ufficiale openjdk:11-jre-slim come **base** per la nuova immagine. Questa è una versione leggera di Java 11, ideale per eseguire applicazioni Java senza la necessità di strumenti di sviluppo aggiuntivi.

- **WORKDIR /app**: Questa istruzione imposta /app come **directory** di lavoro all'interno del contenitore. Tutte le successive operazioni (come COPY o ENTRYPOINT) verranno eseguite in questa directory a meno che non venga specificato diversamente.

- **COPY target/myapp.jar /app/myapp.jar**: Questa istruzione copia il file **myapp.jar** dalla directory target del tuo host alla directory /app del contenitore. myapp.jar è l'artefatto compilato dell'applicazione Spring Boot.

- **EXPOSE 8080**: Questa istruzione informa Docker che l'applicazione all'interno del contenitore **ascolterà sulla porta 8080**.

- **ENTRYPOINT ["java", "-jar", "/app/myapp.jar"]**: Questa istruzione specifica il comando che verrà eseguito ogni volta che il contenitore viene **avviato**. In questo caso, avvierà l'applicazione Spring Boot utilizzando il comando **java -jar**.

Docker Compose:

Docker Compose è uno strumento per definire e gestire applicazioni Docker **multi-contenitore**. Permette agli sviluppatori di definire un ambiente composto da più servizi in un unico file, chiamato **docker-compose.yml**, e di avviare questi servizi con un singolo comando, **docker-compose up**.

Caratteristiche principali di Docker Compose:

- **Definizione di servizi multi-contenitore**: Con Docker Compose, puoi definire diversi servizi che rappresentano contenitori diversi, come un'applicazione web e un database, e farli interagire tra loro.

- **Configurazione in un unico file**: Tutti i servizi, le reti e i volumi sono definiti in un unico file docker-compose.yml, rendendo facile la gestione e la distribuzione dell'ambiente.

- **Facilità d'uso**: Con un singolo comando, docker-compose up, puoi avviare tutti i servizi definiti nel file docker-compose.yml. Allo stesso modo, con docker-compose down, puoi fermare e rimuovere tutti i servizi.

- **Isolamento dell'ambiente**: Ogni progetto ha il proprio ambiente isolato, il che significa che puoi avere configurazioni diverse per diversi progetti senza conflitti.

Vediamo un esempio:

```
version: '3'

services:
  springboot-app:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://mysql-db:3306/mydatabase
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=my-secret-pw
    ports:
      - "8080:8080"
    depends_on:
      - mysql-db

  mysql-db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: my-secret-pw
      MYSQL_DATABASE: mydatabase
    volumes:
      - mysql_data:/var/lib/mysql

volumes:
  mysql_data:
```

Spiegazione:

- **springboot-app:** Questo servizio rappresenta la tua applicazione Spring Boot. Utilizza il Dockerfile nella directory corrente per costruire l'immagine. Le variabili d'ambiente configurano la connessione al database MySQL.

- **mysql-db**: Questo servizio utilizza l'immagine ufficiale di MySQL. L'ambiente del database è configurato tramite variabili d'ambiente. I dati del database sono persistenti grazie al volume mysql_data.

- **depends_on**: Indica che l'applicazione Spring Boot dipende dal servizio mysql-db, garantendo che mysql-db venga **avviato prima** di springboot-app.

- **volumes**: Definisce un volume chiamato mysql_data per la persistenza dei dati del database.

Autenticazione

Nell'era digitale di oggi, la **sicurezza** delle informazioni è diventata una priorità fondamentale. Ogni giorno, milioni di persone accedono a siti web, applicazioni e piattaforme online per svolgere una vasta gamma di attività, dallo shopping online alla gestione delle finanze personali. In questo contesto, garantire che solo gli utenti **autorizzati** possano accedere a determinate risorse o informazioni è di vitale importanza.
Ecco dove entra in gioco l'autenticazione.

L'autenticazione è il processo attraverso il quale un sistema verifica l'identità di un utente, assicurandosi che sia effettivamente chi dice di essere. Questo si traduce spesso nell'inserimento di credenziali come un nome utente e una password, ma può anche includere metodi più avanzati come l'autenticazione a due fattori, la scansione delle impronte digitali o il riconoscimento facciale.

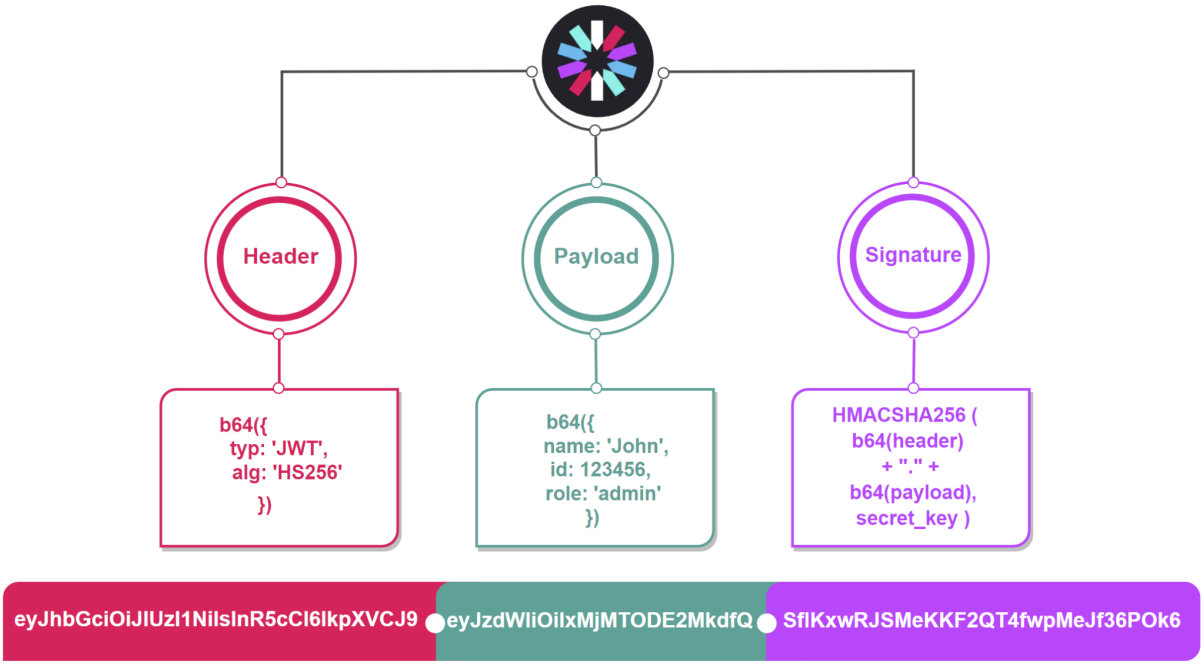
Senza un sistema di autenticazione efficace, le **informazioni sensibili** potrebbero cadere nelle mani sbagliate. Questo non solo potrebbe compromettere la privacy degli utenti, ma potrebbe anche portare a gravi danni finanziari o alla perdita di dati cruciali. Inoltre, un sistema vulnerabile potrebbe essere sfruttato da attori malintenzionati per lanciare attacchi o diffondere malware.

In questo capitolo, esploreremo i concetti fondamentali dell'autenticazione in un'applicazione web, concentrandoci in particolare sull'utilizzo di **JWT**.

JWT

Un **JWT(Json Web Token)** è una stringa codificata che rappresenta un set di informazioni (chiamate "**claims**") che possono essere utilizzate per **autenticare** un utente o trasmettere informazioni in maniera sicura tra un client e un server. La stringa è divisa in tre parti: **header, payload e signature**.

- **Header**: Contiene metadati sul token, come il tipo di token e l'algoritmo di firma utilizzato.
- **Payload**: Contiene i "claims", che sono informazioni sull'utente o altre informazioni pertinenti.
- **Signature**: È una firma digitale che garantisce l'integrità del token e verifica che non sia stato alterato durante il trasporto.



Processo di autenticazione

- **Generazione del Token**: Una volta che l'utente ha fornito le credenziali **valide** (ad esempio, nome utente e password), il backend Spring utilizza una libreria come "Spring Security" insieme a "jjwt" (Java JWT) per **generare** un JWT. Questo token conterrà i reclami necessari, come l'ID dell'utente, ruoli, e altre informazioni personalizzate.

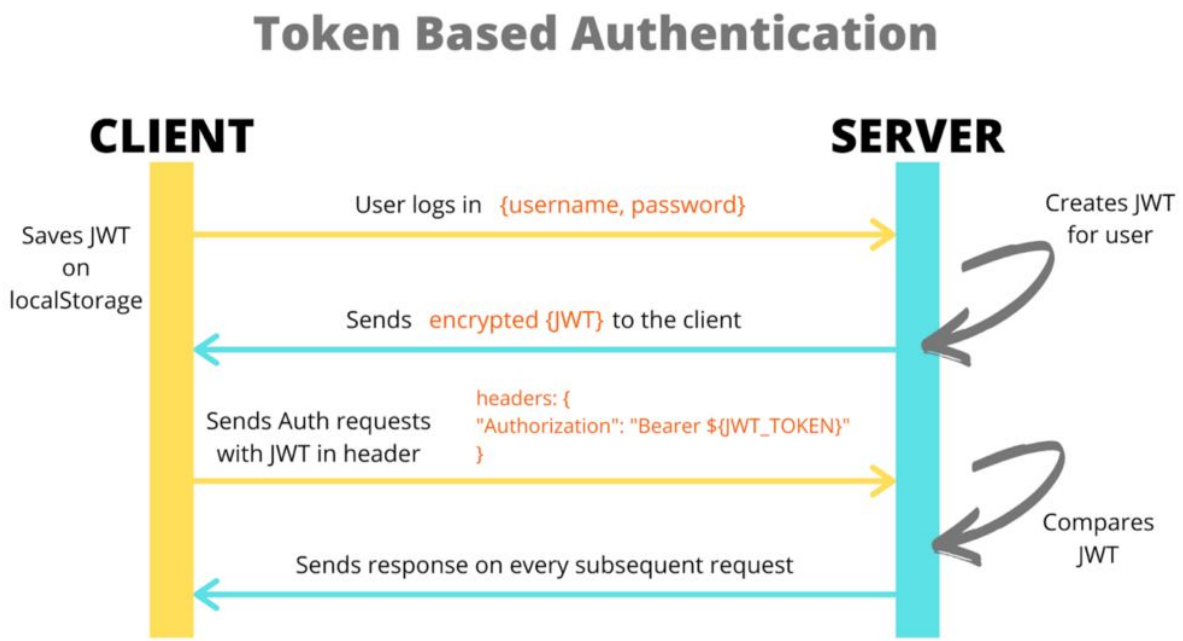
- **Conservazione del Token**: Una volta effettuato l'accesso e ricevuto il JWT dal backend come parte della risposta, il frontend Angular lo **conserva** per **future richieste**. Il local storage è un'opzione comune per conservare il JWT, ma ci sono considerazioni di sicurezza da tenere a mente. Un'altra opzione è utilizzare i cookies HttpOnly, che sono protetti contro gli attacchi XSS.

- **Uso del Token**: Per ogni richiesta successiva al backend, Angular allega il JWT **nell'header** della richiesta, spesso utilizzando l'header "**Authorization**" con il prefisso "Bearer", ad esempio: **Authorization: Bearer <token>**.

- **Verifica del Token**: Una volta ricevuto il JWT, il backend tramite **Spring Security**, estrae il JWT dall'header della richiesta per ogni richiesta entrante che richiede autenticazione e lo **verifica**. Questo processo include la decodifica del JWT, la verifica della **firma(secret key)** per assicurarsi che non sia stato alterato, e la verifica della validità del token (ad esempio, controllando la data di scadenza). Se il token è valido, la richiesta viene elaborata. Se non è valido o

è scaduto, la richiesta viene respinta e viene inviata una risposta di errore al frontend.

- **Caso Unauthorized**: Se il backend restituisce un **errore di autenticazione** (ad esempio, a causa di un token scaduto o non valido), il frontend Angular può gestire l'errore, ad esempio reindirizzando l'utente alla pagina di login o mostrando un messaggio di errore.



Vantaggi nell'utilizzo di JWT:

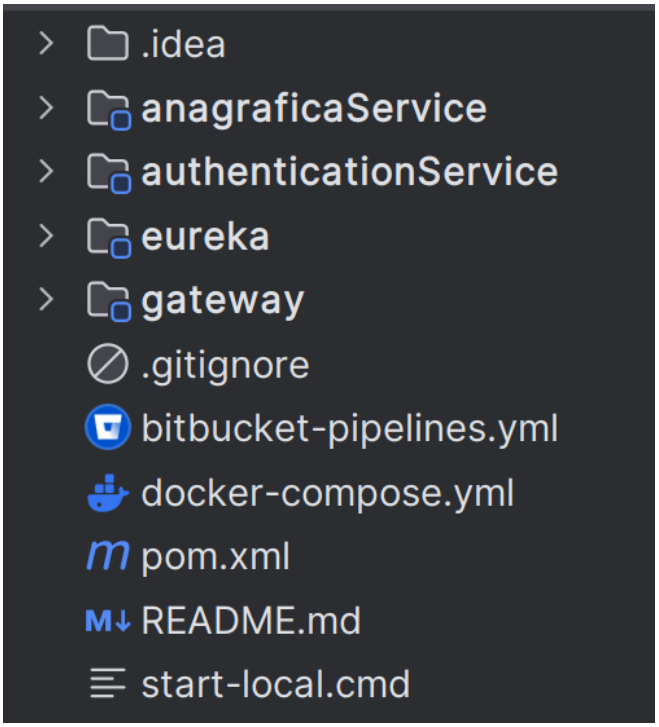
- **Stateless**: I JWT sono stateless, il che significa che ogni token contiene tutte le informazioni necessarie per l'autenticazione. Ciò elimina la necessità di **conservare sessioni** lato server.

- **Scalabilità**: Poiché non è necessario conservare sessioni lato server, le applicazioni basate su JWT sono facilmente scalabili.

- **Mobile Friendly**: I JWT sono ideali per applicazioni mobile, dove la gestione delle sessioni può essere complicata.

Analisi del codice

In questo capitolo esploreremo in dettaglio il codice e la struttura del template del quinto sprint, focalizzandoci sulla struttura del progetto, le configurazioni dei microservizi e l'impiego di Docker per la loro containerizzazione.
Vorrei sottolineare il fatto che il template in esame è contenuto in una singola repository, il che significa che avremo una directory principale con 4 sottomoduli rappresentanti i vari microservizi (ai quali dovrete aggiungere i vostri). Questa scelta facilita la gestione del deploy. Tuttavia, è fondamentale comprendere che, in un progetto reale, ogni microservizio avrà la sua repository e verrà deployato separatamente.
Fatta questa premessa possiamo iniziare ad analizzare la struttura del nostro progetto:



Come precedentemente menzionato, la directory principale contiene 4 sottomoduli, ciascuno rappresentante un microservizio:

- **anagraficaService**: Gestisce i dati anagrafici degli utenti.
- **authenticationService**: Responsabile dell'autenticazione. Include metodi come login e signup e genera token JWT per gli utenti autenticati.
- **eureka**: Servizio di Discovery per registrare gli indirizzi dei diversi microservizi.
- **gateway**: Funziona come servizio di Gateway, instradando le richieste al microservizio appropriato.

Oltre a questi sottomoduli, che esamineremo più approfonditamente, ci sono altri tre file di rilievo:

- **bitbucket-pipelines.yml**: Contiene una pipeline, una sequenza di istruzioni che useremo per il deploy dell'applicazione. Effettuando una push sul branch master, si attiverà questa pipeline, eseguendo una serie di comandi per il deploy.
- **docker-compose.yml**: Attraverso questo file, definiremo e gestiremo il nostro ambiente, rappresentato dall'insieme dei container Docker che costituiscono la nostra applicazione.
- **pom.xml**: Si tratta del pom padre. Al suo interno, sono elencate proprietà e dipendenze condivise tra tutti i sottomoduli, come la versione di Java e le dipendenze di Spring Boot, tra le altre.

Eureka

Partiamo con l'analizzare il Discovery Service. Per definire un microservizio come discovery sarà necessario importare la seguente dipendenza nel pom

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

E successivamente inserire l'annotation **@EnableEurekaServer** nel main dell'applicazione

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {

public static void main(String[] args) {

      SpringApplication.run(EurekaApplication.class, args);
      }

}
```

Fatto questo non ci resterà che definire delle configurazioni all'interno del application.yml:

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0
```

- **server**:
    - **port: 8761**: Questa configurazione imposta la porta su cui il servizio Eureka sarà in ascolto. La porta 8761 è la porta predefinita per un server Eureka.

- **eureka**:
    - **client**: Questa sezione contiene le configurazioni relative al comportamento del client Eureka.
    - **registerWithEureka: false**: Questa configurazione indica che il servizio Eureka non dovrebbe registrarsi con se stesso. Questo ha senso poiché, in molti casi, si desidera che solo altri microservizi si registrino con Eureka e non che Eureka si registri con se stesso.
    - **fetchRegistry: false**: Questa configurazione indica che il servizio Eureka non dovrebbe tentare di recuperare il registro dei servizi da un altro server Eureka. Di nuovo, ciò ha senso per un server Eureka standalone che non fa parte di un cluster.
    - **waitTimeInMsWhenSyncEmpty: 0**: Questa configurazione determina il tempo di attesa (in millisecondi) quando la sincronizzazione del registro dei servizi è vuota. Impostando questo valore a 0, si indica che non ci dovrebbe essere alcun ritardo.

Definite queste configurazioni abbiamo terminato con il microservizio Eureka, ora passiamo ad analizzare il gateway.

Gateway

Come prima cosa andiamo a vedere le dipendenze necessarie:

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

In questo caso nel main dovremo inserire l'annotation **@EnableDiscoveryClient**, una volta annotata una Spring Boot Application con @EnableDiscoveryClient, essa può sia registrarsi con un servizio di discovery (se supportato e configurato) sia interrogare il servizio di discovery per trovare altri servizi.

Successivamente come per il microservizio Eureka ci basterà definire ulteriori configurazioni nell'application.yml e avremo terminato di definire il nostro gateway.

In particolare all'interno dell'application.yml andremo a definire le varie rotte per i nostri microservizi. Analizziamo una in dettaglio:

```
- id: authservice
  uri: lb://authservice
  predicates:
    - Path=/auth/**
```

Con **id** andiamo a definire in modo univoco la **rotta**, non è strettamente necessario che corrisponda al nome dell'applicazione spring a cui stiamo facendo riferimento ma sarebbe meglio utilizzare lo stesso nome. **uri: lb://authservice** stiamo dicendo a spring di gestire le richieste indirizzate a questo URI tramite il suo meccanismo di bilanciamento del carico integrato (**lb sta per "load-balanced"**) per distribuire le richieste tra le diverse istanze disponibili del **servizio di destinazione** (in questo caso authservice è **strettamente legato al nome dell'applicazione Spring** che rappresenta il servizio) . La sezione predicates definisce le condizioni per **l'instradamento**, in questo caso tutte le richieste il cui path inizia con "/auth" verranno instradate verso il servizio **authservice**.

Oltre la definizione delle rotte troviamo le configurazioni CORS:

```
globalcors:
  cors-configurations:
    '[/**]':
      allowedOrigins: "*"
      allowedMethods:
        - GET
        - POST
        - PUT
        - DELETE
      allowedHeaders:
        - "*"
```

Con queste configurazioni stiamo **consentendo** richieste da **qualsiasi origine** (allowedOrigins: "*") con i metodi **specificati** e con qualsiasi **header** (allowedHeaders:"*").

Infine con:

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

Andiamo a definire la **defaultZone**, ovvero dove si troverà, il nostro servizio di Discovery. Ovviamente quando andremo a deployare l'applicazione dovremo andare a

modificare questa proprietà poichè eureka non sarà in esecuzione su localhost ma su un **server**.

Authentication Service

Nell'analisi di questo servizio, non approfondirò l'intero codice. Piuttosto, focalizzerò l'attenzione sulla generazione del token JWT e sull'impiego di FeignClient per interagire con altri microservizi.

Come discusso in precedenza nella sezione teorica, il processo di autenticazione prende avvio quando l'utente inserisce credenziali valide. Di conseguenza, il servizio di login nel backend si incarica della generazione del token JWT. Vediamo come:

```
public LoggedUserDTO login(LoginDTO loginDTO) {
    Authentication authentication = authenticationManager
        .authenticate(new UsernamePasswordAuthenticationToken(loginDTO.getUsername(), loginDTO.getPassword()));

    SecurityContextHolder.getContext().setAuthentication(authentication);

    UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();

    String jwt = jwtUtils.generateJwt(userDetails);

    List<String> roles = userDetails.getAuthorities().stream() Stream<capture of extends GrantedAuthority>
        .map(GrantedAuthority::getAuthority) Stream<String>
        .collect(Collectors.toList());

    return new LoggedUserDTO(userDetails.getId(),
        userDetails.getUsername(),
        userDetails.getEmail(),
        jwt,
        roles);
}
```

Come prima cosa proviamo ad autenticare l'utente con le credenziali fornite:

```
Authentication authentication = authenticationManager
    .authenticate(new UsernamePasswordAuthenticationToken(loginDTO.getUsername(), loginDTO.getPassword()));
```

Per autenticare un utente è necessario definire nelle nostre configurazioni di sicurezza, in questo caso parliamo della classe **WebSecurityConfig**, un bean **AuthenticationManager**

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration authConfig) throws Exception {
    return authConfig.getAuthenticationManager();
}
```

Questo bean tramite il metodo **authenticate** si occuperà di recuperare dal database lo user corrispondente allo username inviato dall'utente per **confrontare le**

credenziali con quelle inviate dall'utente. Lo user verrà recuperato tramite il metodo **loadByUsername** della classe **UserDetailsService**. Nel caso in cui le credenziali fossero corrette ci verrà restituito un oggetto authentication che conterrà, tra le varie informazioni, il "**Principal**" ovvero l'utente autenticato.
Successivamente passeremo l'utente autenticato al metodo **jwtUtils.generateJwt** che tramite la libreria Jwts andrà a generare un token jwt che utilizzeremo per autenticare tutte le richieste.
Analizziamo più in dettaglio il metodo per generare il token:

```
public String generateJwt(UserDetailsImpl userPrincipal) {
    List<String> authorities = userPrincipal.getAuthorities() Collection<capt
        .stream() Stream<capture of extends GrantedAuthority>
        .map(GrantedAuthority::getAuthority) Stream<String>
        .collect(Collectors.toList());
    String roles = String.join( delimiter: ",", authorities);
    return Jwts.builder()
        .setSubject(userPrincipal.getUsername())
        .setIssuedAt(new Date())
        .setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))
        .claim( s: "roles", roles)
        .signWith(key(), SignatureAlgorithm.HS256)
        .compact();
}
```

Iniziamo estraendo le "**GrantedAuthorities**" dell'utente e convertendole in una lista di stringhe utilizzando le funzionalità delle stream di Java. Dopo aver ottenuto questa lista, la combiniamo in una singola stringa, dove ogni autorità è separata da una virgola.

Una volta preparate le autorizzazioni, utilizziamo il metodo builder di Jwts per costruire il token JWT. Durante questa costruzione, impostiamo diverse informazioni:

- **Subject**: Questo rappresenta lo username dell'utente autenticato.
- **Date**: Definiamo sia la data di creazione (**issuedAt**) che la data di scadenza (**expiration**) del token.
- **Claim**: Questi sono dati aggiuntivi associati all'utente autenticato e vengono salvati nel token come coppie chiave-valore. Nel nostro caso, associamo la chiave "roles" alla stringa delle autorizzazioni che abbiamo preparato in precedenza.

Concludiamo il processo di creazione del token firmandolo con una **chiave segreta**, che abbiamo precedentemente configurato nelle proprietà della nostra applicazione.

Questa firma è cruciale: garantisce che ogni token prodotto dalla nostra applicazione sia **autentico** e firmato con la nostra chiave segreta, assicurando così la sua **validità**.

Una volta terminato di creare il token possiamo concludere il nostro metodo di login generando un DTO con le informazioni dell'utente loggato e con il token appena creato. Questo token verrà poi salvato dal front end che dovrà aggiungerlo ad ogni richiesta che richiede l'autorizzazione nell'header "Authorization".

Terminato di analizzare come avviene l'autenticazione e la creazione di un token Jwt vorrei soffermarmi sul metodo registerUser. Questo metodo si occupa di registrare le informazioni di un nuovo utente. Oltre alle credenziali dovrà occuparsi anche di registrare le informazioni anagrafiche. Come abbiamo visto prima però abbiamo un microservizio apposito che si occupa di gestire le informazioni anagrafiche dei vari utenti, dunque dovremo richiamare quest'ultimo per registrare le informazioni anagrafiche. Questa comunicazione verrà effettuata tramite il FeignClient.

FeignClient

Feign è una libreria sviluppata da Netflix e integrata nel progetto Spring Cloud, che permette di semplificare la scrittura di client HTTP. È particolarmente utile nelle architetture basate su microservizi, dove i servizi devono frequentemente comunicare tra loro attraverso chiamate HTTP. Invece di utilizzare metodi tradizionali per creare client HTTP, come RestTemplate, con Feign puoi definire un client utilizzando interfacce annotate in modo simile a come si definiscono i controller in Spring MVC.
Nel nostro caso abbiamo definito un client per il microservizio di anagrafica in modo da poter chiamare senza problemi il metodo di registrazione anagrafica direttamente dal microservizio authentication. Andiamo a vedere come configurare un Feign Client:

```
@FeignClient(name = "anagservice", configuration = FeigClientConfig.class)
public interface AnagraficaFeignClient {

    1 implementation  fmotz
    @PostMapping("/anag/registerAnagrafica")
    AnagraficaDTO register(@RequestBody AnagraficaDTO anagraficaDTO);
}
```

Come potete osservare, la struttura ricorda quella di un tipico controller Spring, ma presenta alcune differenze chiave. Innanzitutto, utilizziamo l'annotazione **@FeignClient** invece di **@RestController**. All'interno di @FeignClient, l'attributo **name** indica l'ID con cui il microservizio target si è registrato presso il service discovery Eureka. Inoltre, c'è un attributo opzionale, **configuration**, che permette di specificare una classe di configurazione personalizzata, argomento che approfondiremo in seguito.

Un'altra distinzione fondamentale è che, mentre con i controller Spring definiamo classi, con Feign definiamo **interfacce**. Questo perché non c'è bisogno di fornire implementazioni per i metodi; ci basta definire le informazioni necessarie per invocarli. Ad esempio, per il metodo register, abbiamo solo specificato il mapping (**@PostMapping("/anag/registerAnagrafica")**), il tipo di body da inviare (**AnagraficaDTO**) e il tipo di oggetto che ci aspettiamo in risposta (anch'esso **AnagraficaDTO**).

Dopo aver configurato il nostro FeignClient, per utilizzarlo basta semplicemente iniettarlo nelle classi dove ne abbiamo bisogno e invocare il metodo desiderato.

```
@Autowired
private AnagraficaFeignClient anagraficaFeignClient;
```

```
try {
    signUpRequest.getAnagrafica().setUserId(savedUser.getId());
    anagraficaFeignClient.register(signUpRequest.getAnagrafica());
} catch (FeignException e) {
    throw new CustomFeignException("Errore durante la registrazione dell'anagrafica", e);
}
```

In questo caso nel momento in cui andiamo richiamare il metodo register stiamo effettuando una chiamata al microservizio anagrafica passando come oggetto un DTO contenente le informazioni anagrafiche.
Come dobbiamo comportarci però nel caso in cui il metodo che andiamo a richiamare richieda l'autenticazione? In questo caso come per le chiamate effettuate dal front end dovremo andare ad aggiungere un token jwt valido alla richiesta. Vediamo come fare:

```
User savedUser = userRepository.save(User.builder()
    .email(signUpRequest.getEmail())
    .username(signUpRequest.getUsername())
    .password(encoder.encode(signUpRequest.getPassword()))
    .roles(authService.createRoles(signUpRequest.getRoles()))
    .build());

UserDetailsImpl userDetails = (UserDetailsImpl) userDetailsService.loadUserByUsername(savedUser.getUsername());

jwtUtils.setRequestJwt(jwtUtils.generateJwt(userDetails));
```

Inizialmente, dobbiamo salvare il nuovo utente nel database. Per fare ciò, facciamo riferimento alla **userRepository** per memorizzare le **credenziali** e i **ruoli** associati al

nuovo utente. Dato che l'utente è appena stato creato, non è necessario invocare il metodo authenticate come faremmo durante il processo di login. Invece, possiamo direttamente ottenere l'oggetto **UserDetails** utilizzando il metodo **loadByUsername** della classe **userDetailsService**. Questo è lo stesso metodo che authenticate utilizza per **estrarre le informazioni** dell'utente dal database. Una volta ottenuto l'oggetto userDetails, siamo in grado di generare il token JWT corrispondente. Questo token viene poi associato alla richiesta corrente attraverso il metodo **jwtUtils.setRequestJwt**.

```
2 usages  fmotz
public void setRequestJwt(String jwt) {
    ServletRequestAttributes requestAttributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
    if(requestAttributes != null) {
        requestAttributes.getRequest().getSession().setAttribute( s: "Authorization",  o: "Bearer " + jwt);
    }
}
```

Questo metodo si occupa solamente di aggiungere alla richiesta corrente il token jwt generato con le informazioni del nuovo user. In questo modo potremo tramite un interceptor configurato per le chiamate del FeignClient recuperare il token dalla richiesta corrente e aggiungerlo alla richiesta per il microservizio anagrafica. Vediamo come configurare l'interceptor:

```
@Component
public class FeignClientInterceptor implements RequestInterceptor {

    2 usages
    private static final String AUTHORIZATION_HEADER = "Authorization";

    fmotz
    @Override
    public void apply(RequestTemplate requestTemplate) {
        ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
        if (attributes != null) {
            HttpServletRequest request = attributes.getRequest();
            String jwt = (String) request.getSession().getAttribute(AUTHORIZATION_HEADER);
            if (jwt != null) {
                requestTemplate.header(AUTHORIZATION_HEADER, jwt);
            }
        }
    }
}
```

Per realizzare l'Interceptor, dobbiamo implementare l'interfaccia **RequestInterceptor** e, di conseguenza, il suo metodo **apply**. Quest'ultimo viene invocato ogni volta che effettuiamo una chiamata tramite il **FeignClient**. Il suo compito principale è recuperare il token JWT dalla **richiesta corrente**, che abbiamo precedentemente impostato con jwtUtils.setRequestJwt, e allegarlo alla **richiesta diretta al FeignClient**.

Potrebbe emergere una certa confusione riguardo alle "richieste" in gioco. È essenziale comprendere che stiamo trattando due richieste distinte:

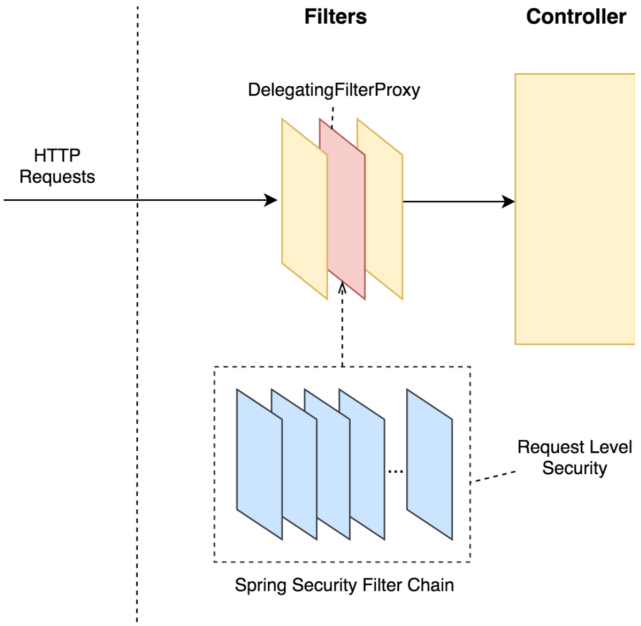
- **Richiesta dal Frontend al Controller Auth**: Quando il frontend invia una richiesta al controller Auth, in particolare al metodo register, dopo aver registrato l'utente, generiamo il JWT e lo associamo al **contesto di tale richiesta**. Questo ci permette di accedere al token in altre parti dell'applicazione.

- **Richiesta tramite FeignClient al Servizio Anagrafica**: Successivamente, quando inviamo una richiesta al servizio Anagrafica tramite FeignClient, dobbiamo includere il token per **l'autenticazione**. L'Interceptor entra in gioco qui: recupera il token dalla prima richiesta e lo allega alla **seconda**, garantendo che il servizio Anagrafica riceva il token necessario per autenticare la richiesta e salvare i dati anagrafici.

In sintesi, l'Interceptor funge da ponte, assicurando che il token JWT generato in risposta alla prima richiesta venga correttamente trasmesso nella seconda richiesta al servizio Anagrafica.

Anagrafica Service

Nel capitolo teorico sull'autenticazione, abbiamo discusso come, una volta generato il token e inviato al front end, questo debba includere il JWT nell'header di tutte le richieste che necessitano di autenticazione. Adesso esploreremo come realizzare la verifica del token sul backend per autenticare tali richieste. Prima di procedere, mi piacerebbe approfondire il funzionamento di Spring Security.



Dall'analisi dell'immagine, emerge chiaramente che l'elemento centrale di Spring Security è rappresentato dalla catena di filtri, conosciuta come "**Filter Chain**". Questa catena ha il compito di **intercettare** ogni richiesta HTTP prima che essa raggiunga i Controller dell'applicazione. La logica dietro questo meccanismo è quella di fornire un punto di controllo centralizzato per la sicurezza, permettendo di eseguire diverse operazioni di verifica e filtraggio prima che la richiesta proceda ulteriormente.

Nel contesto dell'autenticazione basata su token, uno di questi filtri avrà la responsabilità di controllare e verificare la **validità** del token fornito nella richiesta. Se il token è valido, la richiesta verrà autenticata e potrà accedere ai metodi del controller che richiedono autenticazione..

Ora, vediamo come possiamo implementare questa verifica del token all'interno della catena di filtri di Spring Security:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(AbstractHttpConfigurer::disable)
        .exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth ->
            auth.anyRequest().authenticated()
        );

    http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

Iniziamo definendo le configurazioni della nostra FilterChain all'interno della classe WebSecurityConfig. Durante questa definizione, specificheremo diversi aspetti:

- **Configurazioni CSRF**: Utilizzando **csrf(AbstractHttpConfigurer::disable)**, stiamo disabilitando le configurazioni CSRF, poiché nel nostro contesto non sono necessarie.

- **Gestione delle eccezioni**: Con **exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))**, stiamo specificando la classe di gestione delle eccezioni. Questa verrà invocata, ad esempio, quando si riceve una richiesta non autenticata.

- **Autorizzazione delle richieste HTTP**: Mediante **authorizeHttpRequests(auth -> auth.anyRequest().authenticated())**, stiamo definendo i percorsi delle chiamate che necessitano di autenticazione. In questo scenario, tutte le chiamate necessitano di autenticazione, dato che non abbiamo definito eccezioni o percorsi esclusi dal processo di autenticazione.

- **Aggiunta del filtro personalizzato**: Con **addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class)**, stiamo integrando nella Filter Chain il nostro filtro personalizzato. Questo filtro gestirà il processo di autenticazione basato su JWT. La filter chain include alcuni filtri predefiniti, tra cui il UsernamePasswordAuthenticationFilter. Anche se al momento non è essenziale approfondire l'implementazione di questi filtri, è importante notare che il nostro filtro personalizzato dovrà essere posizionato prima del UsernamePasswordAuthenticationFilter.

Finito di configurare la FilterChain andiamo a costruire il nostro filtro per l'autenticazione del token:

```
public class AuthTokenFilter extends OncePerRequestFilter {

    1 usage
    private static final Logger logger = LoggerFactory.getLogger(AuthTokenFilter.class);

    @Autowired
    private JwtUtils jwtUtils;

    no usages  fmotz
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        try {
            String jwt = jwtUtils.getJwtFromHeader(request);
            if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
                String username = jwtUtils.getUserNameFromJwtToken(jwt);

                List<String> roles = jwtUtils.getRolesFromJwtToken(jwt);

                List<GrantedAuthority> authorities = roles.stream() Stream<String>
                    .map(SimpleGrantedAuthority::new) Stream<SimpleGrantedAuthority>
                    .collect(Collectors.toList());

                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(username,  credentials: null, authorities);

                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (Exception e) {
            logger.error("Cannot set user authentication: {}", e.getMessage());
        }

        filterChain.doFilter(request, response);
    }

}
```

Come prima cosa notiamo che la classe AuthTokenFilter estende **OncePerRequestFilter**, che è una classe di Spring Security progettata per garantire che un filtro venga eseguito una sola volta per ogni richiesta. Successivamente

andiamo ad approfondire il metodo principale doFilterInternal sovrascritto dalla classe madre:

- **Estrazione del JWT**: Inizialmente, il metodo tenta di estrarre il JWT dall'header della richiesta HTTP utilizzando il metodo **getJwtFromHeader** di jwtUtils.

- **Validazione del JWT**: Se il JWT non è nullo, viene validato utilizzando il metodo **validateJwtToken** di jwtUtils.

- **Estrazione delle informazioni dal JWT**: Se il JWT è valido, vengono estratte ulteriori informazioni dal token: Il nome utente viene estratto con **getUserNameFromJwtToken**. I ruoli associati all'utente vengono estratti con **getRolesFromJwtToken**.

- **Creazione delle Autorità**: I ruoli estratti vengono quindi trasformati in una lista di **GrantedAuthority**, che rappresenta le autorizzazioni o i ruoli assegnati a un utente in Spring Security.

- **Autenticazione**: Viene creato un nuovo oggetto **UsernamePasswordAuthenticationToken** con il nome utente, nessuna password (dato che stiamo utilizzando JWT) e le autorità. Questo oggetto rappresenta **un'entità autenticata**. Successivamente, l'oggetto di autenticazione viene impostato nel **SecurityContextHolder**, che è un componente di Spring Security che detiene i dettagli dell'entità autenticata.

- **Gestione delle eccezioni**: Se si verifica un'eccezione durante uno dei passaggi precedenti, viene registrato un messaggio di errore utilizzando il logger.

- **Proseguimento della catena di filtri**: Infine, indipendentemente dal fatto che l'autenticazione sia stata eseguita con successo o meno, la richiesta e la risposta vengono passate al prossimo filtro nella catena utilizzando **filterChain.doFilter(request, response)**.

La classe AuthTokenFilter è un filtro personalizzato che estrae un JWT dall'header di una richiesta HTTP, lo valida e, se valido, estrae le informazioni dell'utente e le autorità dal token. Successivamente, imposta l'entità autenticata nel contesto di sicurezza di Spring Security. Questo filtro garantisce che le successive componenti o servizi in un'applicazione Spring possano accedere ai dettagli dell'entità autenticata.

Ora che abbiamo terminato di analizzare singolarmente le caratteristiche dei diversi microservizi possiamo approfondire il discorso Docker.

Docker

Durante questo sottocapitolo andremo a vedere come è possibile containerizzare un'applicazione Spring Boot tramite Docker. Vediamo i vantaggi principali nel creare dei Docker Container:

- **Isolamento**: Docker permette di eseguire l'applicazione in un ambiente isolato, garantendo che funzioni allo stesso modo indipendentemente dalla macchina su cui viene eseguita.

- **Consistenza tra ambienti**: Con Docker, è possibile garantire che l'ambiente di sviluppo, test, staging e produzione siano identici. Questo riduce notevolmente i problemi legati alle differenze tra gli ambienti.

- **Riduzione della complessità**: Le applicazioni Spring Boot possono avere molte dipendenze, come database, broker di messaggi, cache, ecc. Utilizzando Docker, è possibile definire e gestire queste dipendenze in modo coerente e riproducibile.

- **Scalabilità**: Con Docker e orchestratori come Kubernetes, è possibile scalare facilmente le applicazioni Spring Boot in base alle esigenze di carico.

- **Integrazione continua e consegna continua (CI/CD)**: Docker si integra bene con gli strumenti di CI/CD, permettendo di automatizzare il processo di build, test e distribuzione delle applicazioni.

- **Risparmio di risorse**: Piuttosto che eseguire molteplici VM (Virtual Machines) complete per ogni applicazione o servizio, Docker consente di eseguire molteplici container sullo stesso host, condividendo lo stesso kernel del sistema operativo, risparmiando così risorse.

- **Portabilità**: Una volta creato un container Docker, può essere facilmente trasferito e eseguito su qualsiasi sistema che supporti Docker, indipendentemente dalle specifiche configurazioni o dipendenze del sistema sottostante.

Prima di esplorare i dettagli sulla creazione e gestione dei container, è fondamentale comprendere la distinzione tra i tre ambienti principali nei quali opereremo:

- **Ambiente Locale**: Questo è l'ambiente con cui molti sviluppatori sono familiari. Si tratta del contesto in cui si esegue un'applicazione direttamente sulla propria macchina, spesso avviandola dall'IDE (Integrated Development Environment). In questo scenario, l'applicazione viene eseguita in "localhost", e non si fa uso di container Docker. È l'ambiente ideale per lo sviluppo iniziale e i test rapidi,

poiché offre tempi di avvio brevi e un accesso diretto al codice e alle risorse del sistema.

- **Container Locali**: In questo ambiente, si inizia a sperimentare con Docker. Le applicazioni vengono "**containerizzate**", ovvero vengono incapsulate all'interno di container Docker. Questi container vengono poi eseguiti localmente sulla macchina dello sviluppatore. Questo passaggio consente di testare l'applicazione in un **ambiente** che simula più da vicino una situazione di **produzione**, garantendo che tutte le dipendenze e le configurazioni siano correttamente incapsulate all'interno del container.

- **Container Deployati**: Questo è l'ambiente finale e rappresenta la fase in cui i container Docker vengono **distribuiti su un server** o una **piattaforma di hosting**, rendendoli accessibili via rete. Questo potrebbe essere un server fisico, un cluster di server, o piattaforme cloud come AWS, Azure o Google Cloud. In questo contesto, l'applicazione è pronta per essere utilizzata da utenti reali e deve garantire prestazioni, sicurezza e disponibilità. Gli strumenti di orchestrazione, come Kubernetes, possono entrare in gioco in questa fase per gestire, scalare e monitorare i container in produzione.

Comprendere questi tre ambienti è cruciale, poiché ciascuno ha delle specificità e delle esigenze diverse. Mentre l'ambiente locale è ottimizzato per la velocità e l'agilità nello sviluppo, gli ambienti basati su container sono focalizzati sulla riproducibilità, l'isolamento e la scalabilità dell'applicazione.

Durante questa formazione non vedremo Kubernetes ma vorrei fornirvi una breve spiegazione sulle funzionalità di questo servizio.

TODO: FARE KUBERNETES

Ora che abbiamo esplorato in dettaglio i diversi ambienti di sviluppo e le loro peculiarità, possiamo focalizzarci sulla creazione effettiva dei container Docker. Prima di procedere, è importante chiarire un concetto: i container Docker non sono esattamente delle "Macchine Virtuali". Mentre le macchine virtuali includono un sistema operativo completo, i container Docker condividono il kernel del sistema operativo host e includono solo le librerie e le impostazioni necessarie per eseguire l'applicazione specifica. Questo li rende molto più **leggeri** e **veloci** rispetto alle tradizionali VM.

Per creare un container Docker, ci sono diverse strade da percorrere. Ecco due delle più comuni:

- **Dockerfile**: Questa è la modalità tradizionale e più flessibile per creare immagini Docker. Un Dockerfile è uno script che contiene una serie di istruzioni

per costruire un'immagine Docker. Attraverso il Dockerfile, si ha un controllo granulare su come l'immagine viene costruita, quali file vengono inclusi, come vengono gestite le dipendenze e come viene configurato l'ambiente di esecuzione. Una volta scritto il Dockerfile, si utilizza il comando docker build per creare l'immagine.

- **Jib**: Si tratta di un plugin di Maven sviluppato da Google che semplifica la creazione di immagini Docker per applicazioni Java. Con Jib, non è necessario scrivere un Dockerfile o installare Docker sul proprio sistema. Il plugin si occupa di costruire un'immagine ottimizzata per le applicazioni Java, sfruttando le best practices. Jib separa le dipendenze del progetto Java dalle classi, permettendo a Docker di riutilizzare i layer delle dipendenze tra diverse build, accelerando così il processo.

Entrambi gli approcci hanno i loro vantaggi. Mentre Jib offre una soluzione rapida e senza problemi per gli sviluppatori Java, il Dockerfile fornisce una maggiore flessibilità e controllo sulla creazione dell'immagine.

DOCKERFILE:
Come abbiamo visto nella parte teorica il Dockerfile è un documento testuale che elenca una **serie di istruzioni** che Docker utilizza per costruire un'immagine.
Prendiamo come esempio il Dockerfile del servizio di autenticazione:

```
# Usa l'immagine ufficiale di Java come base
FROM openjdk:11

# Copia il file JAR del microservizio nel container
COPY /authenticationService/target/authenticationService.jar app.jar

# Espone la porta su cui il microservizio sarà in ascolto
EXPOSE 8081

# Comando per eseguire l'applicazione
CMD ["java","-jar","/app.jar"]
```

Analizziamo le istruzioni passo passo:

- **FROM openjdk:11** -> Con questa istruzione andiamo a definire l'immagine base da cui partire. OpenJDK è un'immagine Docker ufficiale disponibile su Docker Hub che contiene l'Open Java Development Kit (OpenJDK). In questo modo

non dovremo andare a installare e configurare Java sul nostro container, ma partiamo a costruire la nostra immagine da una di default che contiene Java;

- **COPY /authenticationService/target/authenticationService.jar app.jar** -> Questa istruzione non fa altro che andare a copiare il JAR del nostro progetto in un file all'interno del container, in questo caso chiamato app.jar. E' importante sottolineare che per eseguire questo passaggio è fondamentale aver creato il jar della nostra applicazione. Dunque prima di creare il container dovremo eseguire il comando **mvn clean package** sul nostro microservizio. Questo comando si occupa di creare il file JAR della nostra applicazione:

```
<build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <excludes>
                    <exclude>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                    </exclude>
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Nel pom.xml andremo a specificare il nome che avrà il nostro JAR una volta costruito tramite il tag **<finalName>**.

- **EXPOSE 8081** -> L'istruzione EXPOSE in un Dockerfile ha lo scopo di informare Docker che il container ascolterà su una specifica porta di rete quando verrà eseguito. **Importante:** Anche se EXPOSE indica che una porta sarà utilizzata, non rende effettivamente quella porta accessibile dall'esterno del container. Serve principalmente come una forma di documentazione per chi utilizza l'immagine, indicando quale porta l'applicazione all'interno del container utilizzerà.

- **CMD ["java","-jar","/app.jar"]** -> L'istruzione CMD in un Dockerfile specifica il comando che verrà eseguito di default quando il container viene avviato. Esso definisce il comportamento di default del container in assenza di un comando specificato al momento dell'esecuzione del container. In questo caso definiamo l'istruzione per eseguire il JAR della nostra applicazione copiato in precedenza.

Questo è un esempio di Dockerfile classico per la costruzione di un container per un'applicazione Spring Boot, dopo vedremo come andare a richiamare questo file.

Ora vediamo come buildare un'immagine con JIB.

JIB:
Come abbiamo detto precedentemente in questo caso non avremo bisogno di un Dockerfile ma andremo a definire tutto nel pom del servizio, in particolare all'interno del tag <build> definiremo le specifiche del plugin Jib.

```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>3.2.1</version>
    <configuration>
        <from>
            <image>${jib.base.image}</image>
        </from>
        <to>
            <image>${DOCKER_REPOSITORY}/formazione/microservizi/micro-anagrafica</image>
            <tags>v${project.version},latest</tags>
            <auth>
                <username>${DOCKER_USERNAME}</username>
                <password>${DOCKER_PASSWORD}</password>
            </auth>
        </to>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>${DOCKER_GOAL}</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Vediamo le varie configurazioni in dettaglio:

- **<from><image></from>**: Questo tag definisce **l'immagine base** da cui l'immagine Docker sarà costruita. In questo contesto, l'immagine base è specificata come **openjdk:11** attraverso il tag <jib.base.image>;

- **<to></to>**: Questi tag contengono le specifiche relative all'immagine di destinazione:
    - **Nome dell'immagine**: La variabile **DOCKER_REPOSITORY** è preimpostata con il valore "local". Durante il deploy, verrà sostituita con il nome della repository del server.

    - **Tags** dell'immagine: Questi rappresentano la versione del progetto e "**latest**", che indica l'ultima immagine creata.
    - **Credenziali Docker**: In ambiente locale, le credenziali utilizzate saranno quelle con cui si è effettuato l'accesso a Docker Desktop.

- **<execution>** : All'interno di questo tag indichiamo in quale **fase** di maven verrà attivato il plugin jib (dovremo impostare "package") e il **goal** definito dalla variabile DOCKER_GOAL, che assumerà due valori:
    - **dockerBuild**, questo goal costruisce un'immagine Docker e la carica sul Docker daemon locale, particolarmente utile quando si desidera costruire un'immagine e testarla localmente sulla propria macchina;
    - **build**, questo goal costruisce un'immagine Docker e la spinge direttamente a un registro Docker remoto. È utile quando si desidera distribuire o condividere l'immagine con altri, o quando si desidera utilizzare l'immagine in un ambiente di produzione o di staging. Richiede credenziali di autenticazione valide per il registro Docker remoto, come specificato nella configurazione del plugin.

Ora che abbiamo visto le due modalità per costruire le immagini vediamo come configurare un file docker-compose.

DOCKER-COMPOSE:
Come abbiamo visto precedentemente analizzando la struttura del progetto, è presente un file chiamato docker-compose.yml nella directory principale. Questo file ci servirà a creare un ambiente multi-contenitore ed avviare insieme i nostri servizi tramite il comando docker-compose up.
Analizziamo passo passo questo file:

```
version: '3.5'

x-common-variables: &env
  SPRING_PROFILES_ACTIVE: ${ENVIRONMENT}
  EUREKA_CLIENT_SERVICE-URL_DEFAULTZONE: http://eureka:8761/eureka/
```

Come prima cosa andiamo a definire la versione del docker compose e delle variabili. Queste ci aiuteranno a modificare alcune proprietà dei nostri microservizi tra ambiente locale e di produzione.

Successivamente nelle sezione "services" andiamo appunto a definire i nostri servizi:

```yaml
services:
  eureka:
    build:
      context: ./
      dockerfile: ./eureka/Dockerfile
    image: ${DOCKER_REPOSITORY:-local}/formazione/microservizi/eureka:latest
    restart: always
    ports:
      - "8772:8761"

  gateway:
    build:
      context: ./
      dockerfile: ./gateway/Dockerfile
    image: ${DOCKER_REPOSITORY:-local}/formazione/microservizi/gateway:latest
    restart: always
    depends_on:
      - eureka
    ports:
      - "8094:8080"
    environment:
      <<: *env

  micro-authentication:
    build:
      context: ./
      dockerfile: ./authenticationService/Dockerfile
    image: ${DOCKER_REPOSITORY:-local}/formazione/microservizi/micro-authentication:latest
    restart: always
    depends_on:
      - db-authentication
      - eureka
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://db-authentication:3306/authentication?createDatab
      SPRING_DATASOURCE_USERNAME: root
      SPRING_DATASOURCE_PASSWORD: root
      <<: *env
```

Prendiamo come esempio “**micro-authentication**” e andiamo ad analizzare le diverse sezioni:

- **build**, qua andiamo a definire il percorso che ospita il Dockerfile del servizio di autenticazione, questo file verrà richiamato nel momento in cui sarà necessario costruire l’immagine;
- **image**, viene definito il nome dell’immagine;
- **restart**, qua diciamo al microservizio di provare a restartare continuamente nel caso di build fallita (ad esempio nel caso in cui il servizio eureka non sia ancora

startato avremo un errore durante la build e dovremo fare un tentativo per far ripartire il microservizio);

- **depends_on**, nel contesto di un file docker-compose, la direttiva depends_on indica le dipendenze tra i servizi. Essa specifica l'ordine in cui i servizi devono essere avviati all'interno di un ambiente Docker Compose. È importante notare che depends_on garantisce solo l'ordine di avvio dei servizi, ma non garantisce che un servizio sia "pronto" prima che un altro servizio inizi.
- **enviroment**, qua andiamo a definire delle proprietà che verranno applicate al file properties del microservizio nel momento della creazione del container. La notazione "<<: *env" serve ad aggiungere le variabili dichiarate precedentemente.

Come possiamo notare nella sezione depends_on oltre al servizio eureka abbiamo inserito il servizio db-authentication, questo perchè le nostre applicazioni Spring utilizzano un database e dunque dovremo realizzare un'immagine anche per questo. Vediamo come fare:

```
db-authentication:
  image: mysql:8.0.28
  ports:
    - "3319:3306"
  restart: always
  environment:
    MYSQL_DATABASE: dbauth
    MYSQL_PASSWORD: root
    MYSQL_ROOT_PASSWORD: root
  command: mysqld --lower_case_table_names=1 --skip-ssl --character_set_server=utf8mb4 --explicit_defaults_for_timestamp
  volumes:
    - db-authentication-data:/var/lib/mysql
```

In questo caso utilizzeremo un'immagine di default per i db MYSQL ovvero **mysql:8.0.28**.

La sezione **ports** in un file docker-compose specifica una mappatura tra **le porte del container e le porte dell'host** (la macchina su cui viene eseguito Docker). Questo permette di accedere ai servizi all'interno del container attraverso specifiche porte sulla macchina host.
3319:3306: Questa mappatura indica che la porta 3306 all'interno del container dovrebbe essere mappata sulla porta 3319 dell'host.
In pratica, ciò significa che se hai un'applicazione o un servizio sulla tua macchina host (o su altre macchine che possono accedere alla tua macchina host) e vuoi connetterti a MySQL all'interno di questo container, utilizzeresti la porta 3319 invece della tradizionale porta 3306.

La sezione **volumes** in un file docker-compose è utilizzata per montare percorsi del disco o volumi specifici nei container. Questo permette di **persistere i dati tra le esecuzioni dei container o di condividere file e directory** tra l'host e il container.

db-authentication-data:/var/lib/mysql: Questa mappatura indica che il volume chiamato **db-authentication-data** (che è un volume Docker) dovrebbe essere montato nel container al percorso **/var/lib/mysql**:

- **db-authentication-data**: Questo è un volume Docker. I volumi Docker sono spazi di storage gestiti da **Docker** che possono essere utilizzati dai container. Sono **indipendenti** dal ciclo di vita dei container, il che significa che i dati in un volume persistono anche dopo che un container è stato fermato o eliminato. Questo è particolarmente utile per i database, poiché garantisce che i dati non vengano persi quando il container del database viene ricreato o aggiornato.

- **/var/lib/mysql**: Questo è il percorso **standard** in cui MySQL all'interno del container **memorizza** i suoi dati. Montando il volume db-authentication-data a questo percorso, ci assicuriamo che **tutti i dati del database siano scritti** in questo volume e, quindi, persistano tra le diverse esecuzioni del container.

Più sotto nel docker-compose sarà necessario andare a definire i volumi presenti nell'ambiente che stiamo creando. Nel nostro caso utilizzeremo 2 volumi corrispondenti ai dati dei 2 database.

```
# Volumes
volumes:
  db-anagrafica-data:
  db-authentication-data:
```

Ultima cosa che volevo far notare del docker-compose è la costruzione del servizio anagrafica:

```
micro-anagrafica:
  image: ${DOCKER_REPOSITORY:-local}/formazione/microservizi/micro-anagrafica:latest
  restart: always
  depends_on:
    - db-anagrafica
    - eureka
  environment:
    SPRING_DATASOURCE_URL: jdbc:mysql://db-anagrafica:3306/anagrafica?createDatabaseIfNotExi
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: root
    <<: *env
```

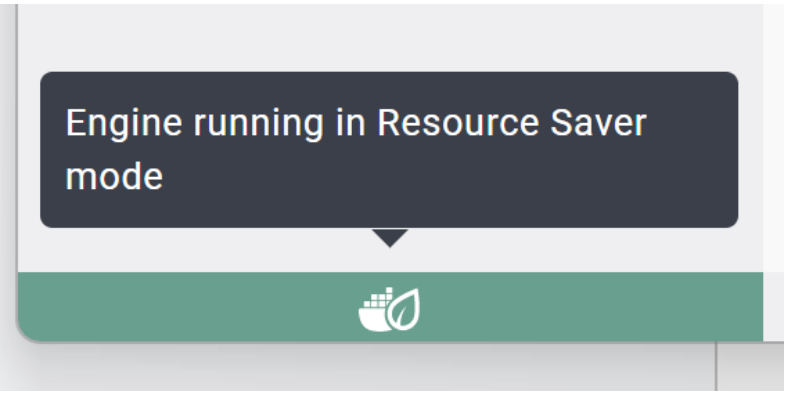
Come potete notare in questo caso non è presente la sezione **build.** Questa scelta è stata fatta perché abbiamo delegato al plugin **Jib** la responsabilità di costruire l'immagine Docker. Pertanto, non c'è bisogno di fare riferimento a un Dockerfile all'interno del docker-compose. Infatti, durante la fase di build del progetto, Jib si occuperà **autonomamente** di generare e costruire l'immagine Docker.

Ora che abbiamo capito come costruire i nostri container Docker, vediamo come possiamo runnarli ed eseguirli, prima localmente e successivamente su un server.

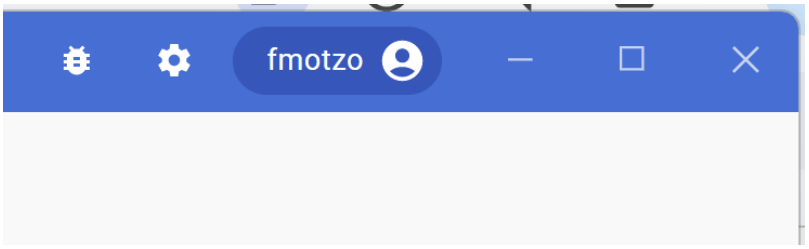
LOCALE:
Come prima cosa per runnare in locale sarà necessario registrarsi e scaricare **Docker Desktop**. Una volta che avremo scaricato e configurato Docker Desktop siamo pronti a far runnare i nostri container. Se notate nella directory principale del progetto è presente un file chiamato "**start-local.cmd**", si tratta di uno script che essenzialmente richiama due istruzioni:

- **mvn clean package**, serve a creare i file JAR dei nostri microservizi;
- **docker-compose up**, richiama il file docker-compose per creare e runnare i nostri container in locale;

Se tutto è andato come previsto su Docker Desktop verrà creato il container che ospiterà le immagini relative ai nostri servizi.
Se doveste avere dei problemi durante l'esecuzione dei container verificate di avere Docker Desktop attivo.



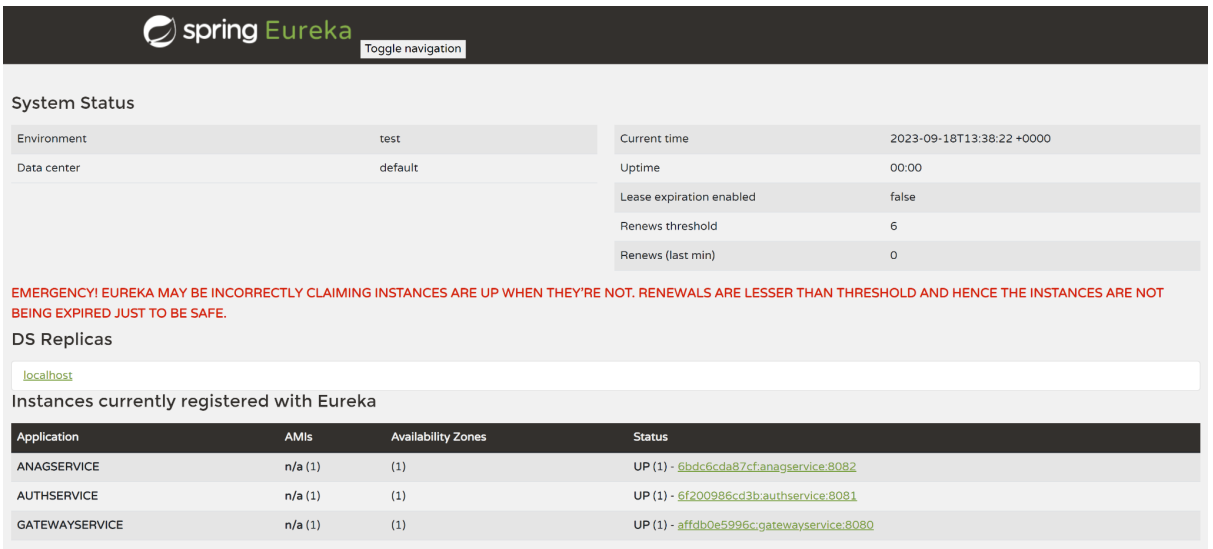
Potreste avere dei problemi per quanto riguarda le credenziali e l'autenticazione. In questo verificate prima di essere loggati su Docker Desktop.



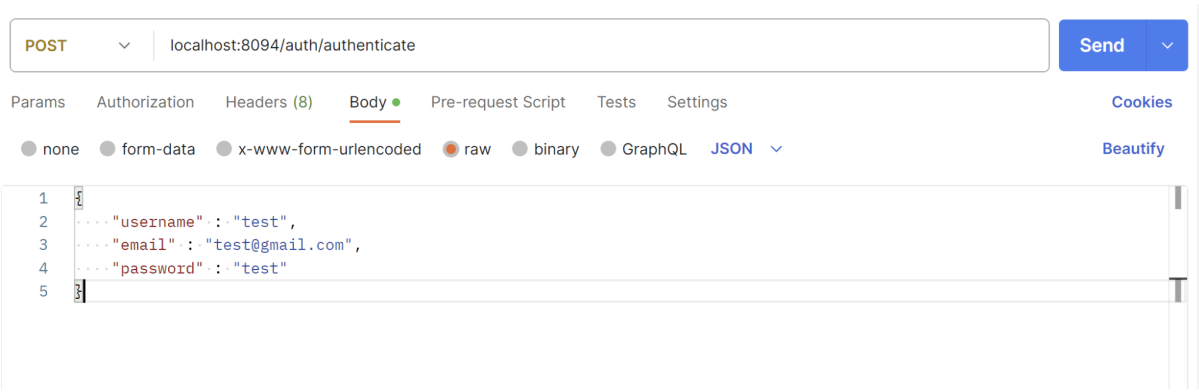
In caso foste già loggati ma continuiate ad avere errore di autenticazione sulle credenziali provate ad eseguire i seguenti passaggi:

1. Fate Logout su Docker Desktop;
2. Andate sulla barra di ricerca di Windows-> Gestione Credenziali -> Credenziali Windows ed eliminate tutte le credenziali relative a Docker;
3. Recatevi nella cartella al path C:\Users\vostroNomeUtente\.docker ed eliminate il file config.json;
4. A questo punto tornate sul terminale di Intellij e prima di runnare nuovamente lo script “start-local.cmd” eseguite il comando “docker login” ed effettuate l’accesso con le vostre credenziali di Docker Desktop;
5. Infine runnate nuovamente lo script e verificate che la creazione dei container vada a buon fine;

Se tutto ha funzionato correttamente avrete i vostri container che runnano su Docker locale, per verificare che i vari microservizi siano partiti correttamente possiamo recarci sul nostro browser all’indirizzo http://localhost:8772/ e verificare che i microservizi si siano registrati correttamente (prima di disperarsi attendete un pò, potrebbero volerci 2 o 3 minuti prima che si registrino tutti i servizi).



A questo punto possiamo testare i nostri endpoint su Postman puntando all'indirizzo del gateway ovvero localhost:8094 come abbiamo specificato nel docker-compose.



REMOTO:
In merito al deploy sul server, sarà gestito attraverso la pipeline di BitBucket, la cui configurazione non sarà di vostra competenza. Al completamento del progetto, vi basterà effettuare un push sul branch master. Questa azione attiverà automaticamente la pipeline, che seguirà una serie di istruzioni predefinite per costruire e implementare i container Docker sul server.