Università degli Studi di Catania

Dipartimento di Ingegneria Elettrica Elettronica e Informatica

Corso di Laurea Magistrale in Ingegneria Informatica (LM-32)

Cavallaro Salvatore - 1000008690

Patanè Rosario - 1000013474

Relazione finale progetto e-commerce

Progetto 2 Variante A

Corso di *Distributed Systems and Big Data*

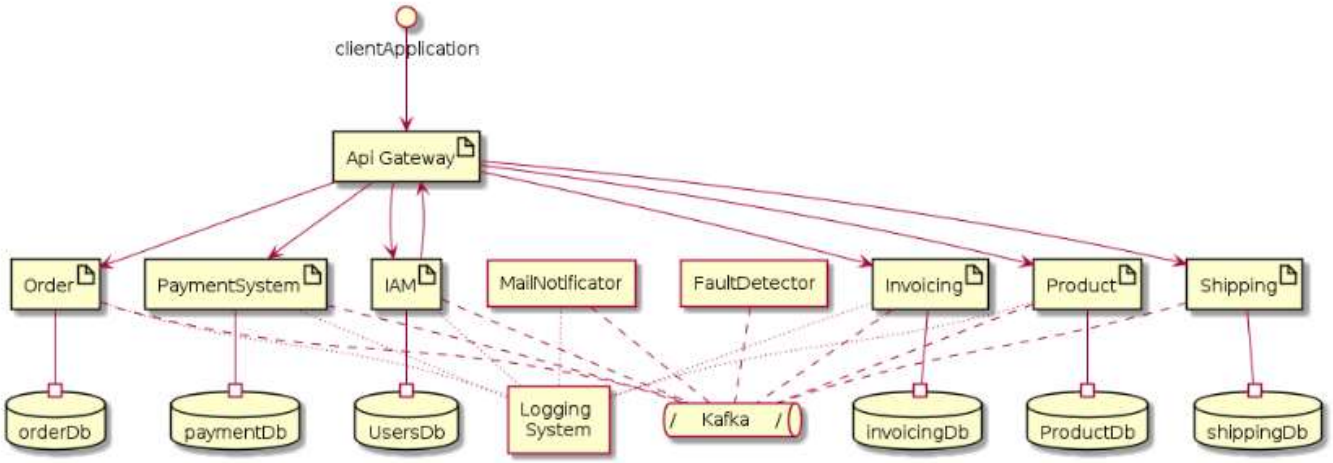
Prof.ssa Antonella Di Stefano

ANNO ACCADEMICO 2020/2021

Introduzione

La presente relazione descrive l'implementazione di un microservizio parte di un sistema più ampio mirato alla realizzazione di funzionalità per un'applicazione in ambito e-commerce. Nello specifico si è implementato il componente per la gestione degli ordini, al quale, d'ora in avanti, ci si riferirà come *OrderManager*.

Lo sviluppo del componente, una volta ultimato, verrà integrato ai restanti microservizi, sviluppati da altri team, che costituiscono il sistema complessivo. Qui di seguito una rappresentazione architetturale dell'applicazione:



Lo sviluppo del nostro microservizio, in accordo alla rappresentazione sopra mostrata, prevede l'implementazione dell'OrderManager, l'uso e la configurazione dei seguenti altri componenti su appositi microservizi:

- MySQL Database (per la persistenza dei dati)
- Apache Kafka (per il messaging asincrono) e Zookeeper
- Api Gateway
- Eureka Discovery Service

Dei vari componenti, quindi, viene effettuato il deployment su container Docker (connessi mediante una rete interna realizzata in fase di building a partire dal file .yaml); tutto ciò al fine di realizzare un'applicazione a microservizi secondo i pattern e i principi studiati nell'ambito dei Sistemi Distribuiti. L'applicazione viene, inoltre, sviluppata in Java mediante l'impiego del framework Spring.

OrderManager

L'OrderManager è il microservizio il cui sviluppo è stato affidato al nostro team. In modo specifico, la variante richiesta è la A, ovvero quella dove si deve utilizzare Java Spring MVC, JPA e MySQL. Il microservizio rappresenta l'elemento centrale del progetto sviluppato dal team e si occupa essenzialmente di realizzare funzionalità per la gestione degli ordini. In particolare, l'aggiunta di un ordine al database mediante una POST all'endpoint /orders, la lettura di un ordine dell'utente dal

database mediante id con GET all'endpoint /orders/{id} e la lettura di tutti gli ordini dell'utente con GET all'endpoint /orders/.

Una delle specifiche richieste rappresenta il livello di sicurezza nell'effettuare le GET/POST, dove quest'ultima viene fatta correttamente solo se fornito lo user id dell'utente mediante l'Header HTTP "X-User-ID". Questo viene catturato grazie ad una apposita annotazione di Spring: @RequestHeader("X-User-ID").

Il microservizio viene implementato tramite una suddivisione in due parti: la prima, che si occupa del controller, la seconda, che si occupa di realizzare i servizi associati. Vengono, infatti, create due classi, rispettivamente OrderController e OrderService, opportunamente annotate. Nel primo caso, per mappare le GET/POST ai relativi metodi, nel secondo, per definire il comportamento dei vari metodi che eseguono operazioni all'interno del nostro microservizio in modo transazionale (@Transactional).

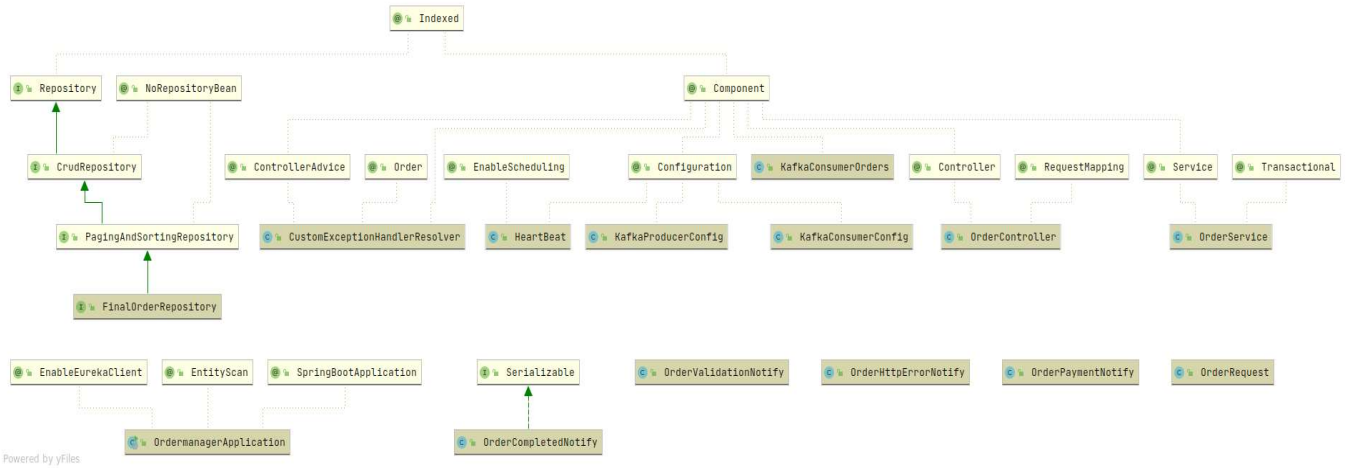
Il microservizio esegue inoltre sulla porta 4444 ed ha una dipendenza da altri due microservizi su altrettanti container Docker: MySQL database e il registration server. Inoltre, OrderManager comunica con ulteriori container essenziali per poter esplicare correttamente le sue funzionalità. Questi contengono microservizi quali quelli per la gestione degli utenti e per la gestione dei prodotti, sviluppati da altri team. Al fine di poter realizzare e testare correttamente il progetto, sono state implementate versioni minimali di tali microservizi, ovvero UserManager e ProductManager che eseguono sulle porte 2222 e 3333.

Un'altra delle specifiche richieste è la capacità del microservizio di effettuare in modo periodico richieste POST all'endpoint */ping* di un apposito microservizio sviluppato da un altro team, ovvero l'*Heart-Beat Fault Detector*. Questa funzionalità mira a ottenere informazioni in merito allo stato di connessione al proprio database del singolo microservizio e al suo stato.

In modo specifico, tale caratteristica viene realizzata mediante una apposita classe annotata come *@Configuration* e come *@EnableScheduling*, al fine di rendere periodico il comportamento dei metodi contenuti al suo interno. Infatti, il metodo che esegue l'Heart-Beat check viene annotato come *@Scheduled* (con parametro impostato su file .env) ed esegue un ciclo dove periodicamente vengono effettuate un numero di POST */ping* pari al numero di microservizi che si vogliono diagnosticare. Nel nostro caso, i microservizi di cui necessita l'OrderManager, ovvero UserManager e ProductManager.

Si noti che anche in questo caso al fine di testare correttamente tale funzionalità si è sviluppato su container un microservizio minimale di *Heart-Beat Fault Detector* che banalmente restituisce un messaggio prestabilito. Il messaggio ricevuto, come risultato del check ai microservizi, si sceglie semplicemente di stamparlo su terminale. Tale microservizio esegue sulla porta 8888 e dipende solo dal registration server.

Qui di seguito il *diagramma delle classi* del microservizio OrderManager:



Ultima nota viene fatta in merito alla gestione dei casi in cui la chiamata delle REST API definite nel microservizio avvenga con valori non corretti (scenari alternativi) e la relativa gestione in termini di eccezioni sollevate e relative risposte HTTP.

In *OrderService* si effettua un controllo in merito all'esistenza di utenti e prodotti, laddove richiesto e, qualora non presenti, si lancia una eccezione che scatena una risposta HTTP di tipo 40x. Inoltre, si suppone che il controllo di un campo vuoto e la correttezza di una chiave nei campi del messaggio JSON della POST avvenga ad opera del client, per cui si è ritenuto opportuno non gestire tali eccezioni.

Eureka Discovery Service

Uno degli ulteriori servizi che si è scelto di implementare è il *Discovery Service Eureka* mediante un apposito microservizio su container Docker che agisce da *Registration Server* e che ha il compito di registrare e tenere traccia dello stato dei singoli *client Eureka*. Il registration server esegue sulla porta 1111 che viene esposta al fine di poter accedere all'apposito servizio da interfaccia web.

Il Registration Server viene quindi interpellato dall'OrderManager (client Eureka) ogni qualvolta si necessita di effettuare richieste verso altri microservizi, ovvero altri client Eureka (es. POST /ping, GET product/id/{id}, ecc). Si ricorda che l'OrderManager si registra al registration server mediante annotazione *@EnableEurekaClient*.

API Gateway

Un altro dei microservizi richiesti è l'API Gateway, che nel caso del presente progetto viene sviluppato mediante *Spring Cloud Gateway*. Tale microservizio esegue sulla porta 9999 che viene esposta proprio al fine di realizzare la funzionalità per la quale tale componente viene impiegato.

Il microservizio, infatti, ha come compito quello di mappare le richieste del client verso i vari componenti che costituiscono l'applicazione complessiva, in modo tale da nascondere completamente quella che è la suddivisione a microservizi del sistema. In modo specifico, la configurazione viene fatta mediante un metodo, annotato come *@Bean*, che restituisce un *RouteLocator* configurato al fine di mappare microservizi diversi a URI diversi. Il microservizio

viene configurato inoltre come *client Eureka*, in modo che il registration server possa tenerne traccia costantemente.

Apache Kafka

Una delle altre specifiche del sistema è la capacità di implementare dei meccanismi di messaging asincrono attraverso l'uso di *Apache Kafka* e *Zookeeper*. Questi ultimi vengono eseguiti come

microservizi all'interno di container Docker di cui, nel caso di Kafka, viene esposta la porta 9092 sulla quale è in esecuzione per permettere di pubblicare/consumare messaggi sui vari *topic* come processo di test mediante gli script di fake producer/consumer eseguiti sul S.O. host.

Nel caso specifico dell'OrderManager, questo microservizio deve essere sia produttore che consumatore di messaggi Kafka. Esso viene quindi configurato mediante due classi: *KafkaConsumerConfig* e *KafkaProducerConfig*, in modo tale da definire i parametri e i topic su cui mettere/prendere i messaggi.

Inoltre, per quanto riguarda le operazioni di produzione di messaggi Kafka si definisce un metodo *sendMessage* utilizzato nella classe di *OrderService* per produrre il messaggio da inviare in corrispondenza di un ordine completato correttamente con chiave *order_completed* e sui topic *orders* e *notification*. Allo stesso modo, vengono prodotti messaggi al fallimento di una richiesta HTTP con errori di tipo 40x e 50x. In modo specifico, ciò viene fatto all'interno di una classe *CustomExceptionHandlerResolver* che annotata con *@ControllerAdvice*, *@Order* permette di gestire dell'eccezioni con precedenza definendo un comportamento personalizzato che comprende, appunto, la generazione di messaggi da inviare sul topic *logging* di Kafka.

Infine, *OrderManager* consuma messaggi Kafka secondo quanto definito all'interno della classe *KafkaConsumerOrders* verificando i campi interni del messaggio JSON ricevuto e intraprendendo le opportune azioni definite dalle specifiche di progetto. Nel caso del consumo di messaggi con chiave *order_validation* setta semplicemente un apposito campo *status* (nel database) in corrispondenza dell'ordine con orderId specificato. Per quanto riguarda i messaggi con chiave *order_paid* setta invece lo stesso campo *status* e produce un messaggio Kafka in accordo con i valori letti dal messaggio JSON ricevuto.