Distributed Systems and Big Data

Documentazione Progetto 9 A

Shipping System

A cura di:
Dilonardo Matteo - 1000014339
Valerio Labbia Remo - 1000012407

Sommario

Indice delle figure

Introduzione

La seguente trattazione descrive il progetto da implementare scelto, ossia il 9 variante “A”. Sono stati usati fondamentalmente:

- Spring MVC
- JPA
- MySQL

Per lo sviluppo, oltre a Docker e Kafka, si è usato GitHub (link al repository: https://github.com/GerteN/shippingsystem).

Il progetto, facendo riferimento a un’architettura composta da un’applicazione client che comunica con un API Gateway che fa da reverse proxy per i vari microservizi, ha come obiettivo la realizzazione del microservizio di “Shipping” per la gestione delle spedizioni.
Questo microservizio è associato a un database, offre delle API REST e in generale può comunicare in modo asincrono con gli altri servizi tramite Kafka.

Implementazione API REST

Una parte importante della gestione delle spedizioni è data dalla creazione di API adeguate, che forniscono degli endpoint per la visualizzazione di una spedizione o di tutte quelle effettuate da un utente. Nel caso in cui l'utente risulti l'amministratore del sistema, è previsto che possa visualizzare tutte quelle presenti nel sistema.

Le API necessarie per fornire un servizio di gestione delle spedizioni sono state realizzate esponendo i seguenti endpoint:

- "GET /shipping/{id}", dove "{id}" è il codice univoco identificativo della spedizione.
- "GET /shippings/", in cui è prevista l'aggiunta dei due parametri "per_page" e "page" nella richiesta.

Con entrambi occorre passare anche un parametro "X-User-ID", all'interno dell'header HTTP, per specificare l'utente che sta usando l'endpoint; se questo ha valore 0, allora è l'amministratore Dunque, nel caso corretto, fondamentalmente restituiranno la rappresentazione JSON della/e spedizione/i associate al valore utente passato; oppure, se quell'utente è l'amministratore, ogni spedizione cercata esistente.

Per esporre tali endpoint è stata predisposta una classe "ShippingController" che, tramite l'annotazione "@RequestMapping(path="")", definisce il path in cui esporli, spostando tuttavia la logica di controllo e di gestione delle spedizioni in una classe apposita, chiamata "ShippingService", seguendo quindi la logica di MVC. Tale classe è stata dunque iniettata all'interno di "ShippingController" per permetterne l'utilizzo.

Come indicato sopra le due richieste devono essere entrambe di tipo GET, quindi è stata utilizzata anche l'annotazione "@GetMapping" per rifiutare le richieste che non siano di tipo GET. Una volta ricevuta una richiesta da uno degli endpoint specificati, la responsabilità viene spostata a "ShippingService" che si occupa di elaborare tale richiesta.

All’interno del service sono definiti sia i metodi relativi alle API offerte, sia quelli relativi alla gestione dei messaggi Kafka. Per il primo endpoint è stata semplicemente implementata la richiesta attraverso l’uso di una Repository, “ShippingRepository”, ovvero un’interfaccia per la gestione dei dati presenti nel database, che effettua una richiesta all'interno del db sfruttando l’estensione “PagingAndSortingRepository” di “CrudRepository”. Vengono inoltre effettuati i controlli relativi alla presenza della spedizione richiesta e alla corrispondenza tra l’identificativo utente fornito tramite header e quello associato alla spedizione richiesta, nonché alla presenza dello X-User-ID stesso, generando degli errori dove necessario.

Il secondo endpoint è stato realizzato usando la stessa logica implementativa della prima, ma prevedendo nella richiesta l’uso della paginazione (che Spring mette a disposizione) per gestire un numero massimo di spedizioni visualizzabili per pagina; i due parametri “page” e “size” sono stati definiti nel file di “application.properties”, indicandone l’associazione con “page” e “per_page” ricevuti tramite l’header. In particolare viene usata una classe “Pageable” che permette anche di ridefinire la richiesta effettuata nel database, tramite Repository, restituendo non una lista di spedizioni ma un tipo “Page<Shipping>”, che contiene al suo interno sia i dati relativi alle spedizioni che quelli relativi alla paginazione.

Shipping

Come detto nell'introduzione, il microservizio da noi sviluppato è incentrato sulle spedizioni. L'entità Shipping definita si compone di:

- shippingId = chiave primaria, un codice numerico per indicare l'ID di una spedizione.
- orderId = l'ID di un ordine associato ad una spedizione. Abbiamo considerato che ogni ordine possa far riferimento a una sola spedizione, e che una spedizione possa essere riferita a un solo ordine.
- userId = Integer per indicare di quale utente sia la specifica spedizione. Un utente può avere associate più spedizioni, e una specifica spedizione (quindi anche l'ordine) può appartenere a un solo utente; questo è stato possibile attraverso l'assegnazione del valore true a "unique" in @Column.
- shippingAddress = l'indirizzo di spedizione.
- products = questa mappa <Integer,Integer> ha come chiave l'ID di un prodotto e come valore la relativa quantità. Specifica i prodotti e le quantità facenti parte di una spedizione.
- status = stringa contenente lo stato della spedizione. La macchina a stati per esso prevede: "Default initial" appena una nuova spedizione è creata, "Abort" se lo status_code ricevuto (relativo a un ordine, e quindi a una spedizione) dal messaggio Kafka con key "order_validation" non è 0, "Ok" se l'appena citato status_code è 0, "TODO" se la spedizione associata alla coppia (orderId, userId), passata tramite il messaggio Kafka con key "order_paid", esiste.
- DDT = numero incrementale, generato non al momento della creazione della spedizione, bensì quando la spedizione viene impostata a TODO. Per ottenere un corretto continuo incremento indipendente, senza riassegnazione di valori precedentemente usati per spedizioni che magari poi sono eliminate, è stata creata un'entità di appoggio "DDT" contenente, oltre alla chiave primaria obbligatoria, un

Integer “seq”. Praticamente, al momento dell’aggiornamento di una spedizione con stato TODO, da questa tabella DDT in db viene recuperato il valore di seq (presupponendolo relativo alla chiave primaria con valore 1), incrementato di 1, e impostato come valore dell’attributo DDT della spedizione.

Per fare in modo di non avere attributi NULL (DDT a parte) alla creazione di una spedizione in db, per ognuno di questi è stata usata l’annotazione “@Column(nullable = false)”.

Diagramma delle classi: Shipping e DDT

Sono di seguito riportate le due entità di cui si è parlato precedentemente, attraverso un diagramma delle classi:

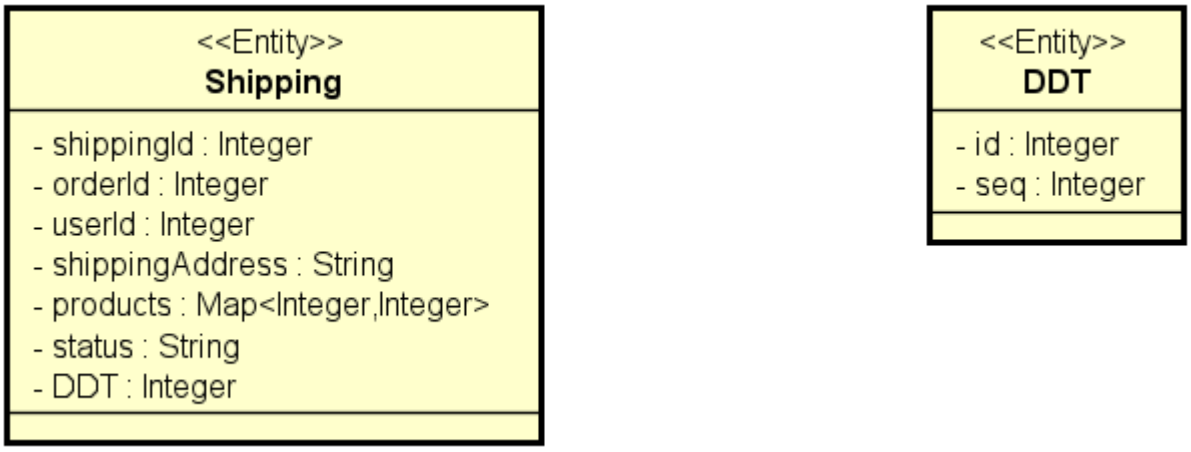

Figura 1: DCD delle entità Shipping e DDT

Oltre ad esse, di fondamentale importanza sono state le seguenti classi e interfacce per costruire Controller, Service e Repository:

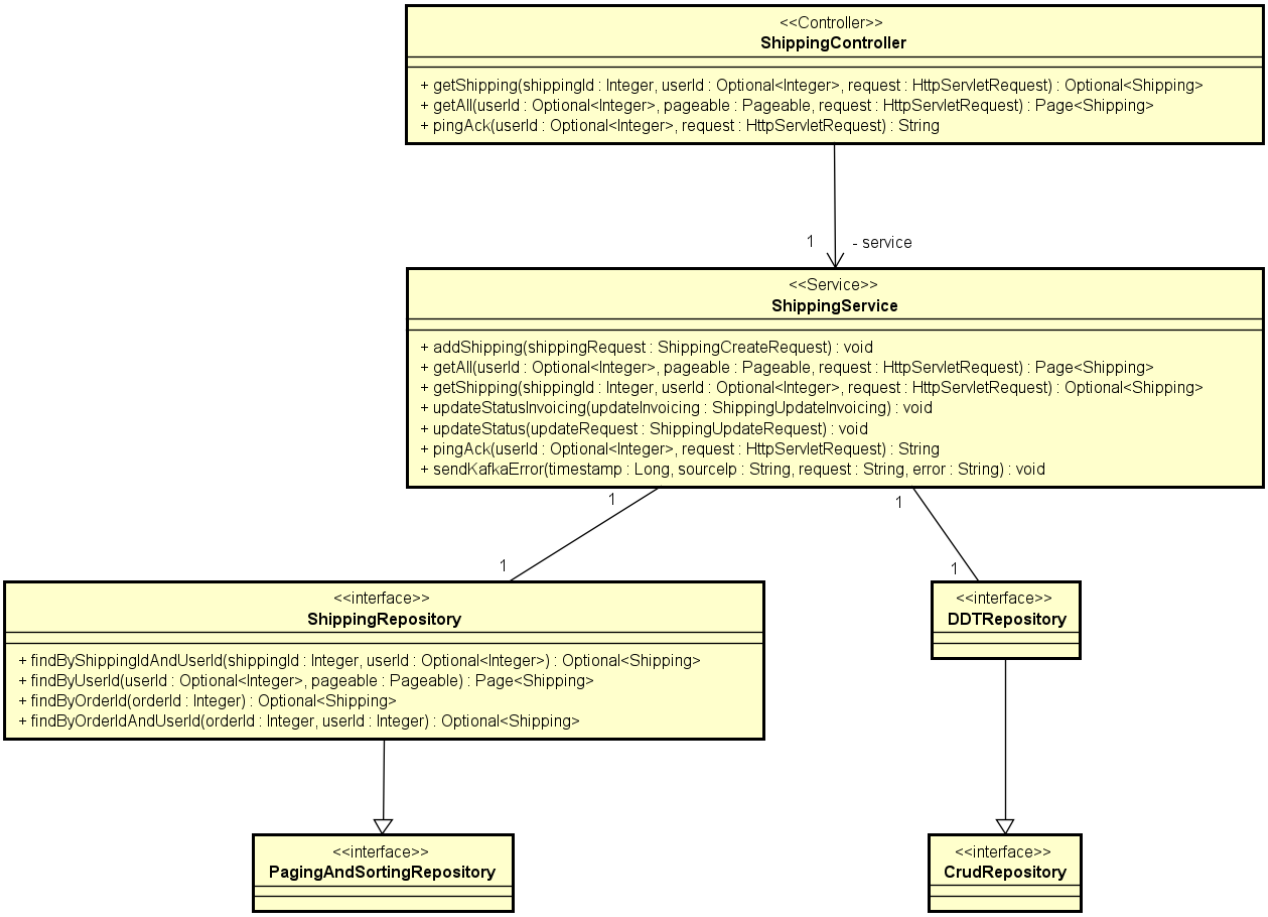

Figura 2: DCD Controller, Service e Repository di Shipping e DDT

Error Handling

Per la gestione degli errori, sia lato client che lato server, sono stati implementati due meccanismi differenti. Gli errori generati dal client sono gestiti tramite alcuni controlli effettuati su determinate parti di codice che nel caso di uso scorretto, malizioso o meno, delle API fornite dal Service, genera un errore adeguato. Gli errori HTTP usati sono i seguenti:

- 400 Bad Request = usato quando la richiesta non contiene nell'header il campo X-User-ID o se questo non ha un valore.
- 403 Forbidden = questo codice di errore viene restituito unicamente nella strategia di health-check, ping-ack mode, quando ad effettuare la richiesta di visualizzazione dello stato generale del servizio non è l'admin.
- 404 Not Found = usato sia per identificare una richiesta di spedizione/i il cui risultato è nullo, e sia per identificarne una non autorizzata ma il cui risultato non è nullo. Questo meccanismo viene utilizzato per evitare la divulgazione di informazioni riguardo la presenza o meno dei dati richiesti a un potenziale attaccante.

Gli errori relativi a eccezioni generate dal server vengono catturate e gestite da un handler apposito, "ShippingErrorHandling", che estende la classe "AbstractHandlerExceptionResolver". Secondo le specifiche del progetto, appena si verifica un errore esso è gestito da questa classe attraverso la creazione di un messaggio Kafka con il timestamp, il source ip, indicazione implicita del servizio che lo sta generando, la richiesta che ha portato alla generazione dell'errore (con indicati path e method) e lo stack trace dell'errore. Il messaggio appena creato viene infine inviato nel topic "logging" con key "http_errors", ed è automaticamente generato un HTTP error code 500. Ciò avviene anche grazie alla classe di configurazione "KafkaProducerConfig" per produrre questi messaggi.

Messaggi Kafka

Inizialmente abbiamo configurato le impostazioni (nel file con annotazione "@Configuration" in "KafkaConsumerConfig") per consumare i messaggi, diversificando quelli per il topic "orders", con key "order_completed" o "order_validation", da quelli per il topic "invoicing" con key "order_paid".

Successivamente, dato che Shipping è consumatore di questi messaggi, nella classe "KafkaConsumerShipping" abbiamo programmato quale metodo di ShippingService chiamare in base al topic e alla chiave del messaggio Kafka ricevuto.

Per quanto riguarda la ricezione di un messaggio sul topic orders, se la chiave è:

- order_completed = è passata una nuova istanza di "ShippingCreateRequest", creata dal value ricevuto, alla funzione "addShipping" del servizio per le spedizioni; questa creerà una nuova spedizione in db, non settando un valore per il campo DDT, e impostando lo status a "Default initial".
- order_validation = questa volta sarà usato il metodo "updateStatus" di ShippingService, a cui si passa una nuova istanza di "ShippingUpdateRequest"; attraverso l'orderId e lo status contenuti nel value del messaggio Kafka, usati per questa istanza passata, sarà determinato se aggiornare lo stato della spedizione corrispondente da "Abort" a "Ok".

Invece per la ricezione di un messaggio sul topic invoicing, con chiave:

- order_paid = intanto è istanziata, facendo uso dei valori del value ricevuto, una "ShippingUpdateInvoicing" contenente soprattutto gli Integer userId e orderId. Questa istanza sarà passata al metodo "updateStatusInvoicing" di ShippingService, che se non esiste una spedizione associata a (orderId, userId) allora manderà un messaggio nel topic "logging" con key "shipping_unavailable" aggiungendo il timestamp; se esiste, imposterà lo status a "TODO" e aggiungerà il numero di DDT incrementale alla spedizione.

Per inviare i vari messaggi di errore (di cui si è parlato anche nel capitolo precedente, tralasciando ShippingErrorHandling), nel topic logging di fondamentale importanza è anche il metodo “sendKafkaError” usato in ShippingService. Attraverso questo, è istanziata una “ShippingHttpErrors” in cui sono settati i vari dettagli (timestamp, request, ecc) dell’errore; qui l’oggetto Java, contenente i vari parametri dell’errore, viene poi convertito in JSON e inviato al topic logging.

Di seguito sono mostrate le classi ShippingCreateRequest, ShippingUpdateRequest e ShippingUpdateInvoicing, attraverso il diagramma delle classi:

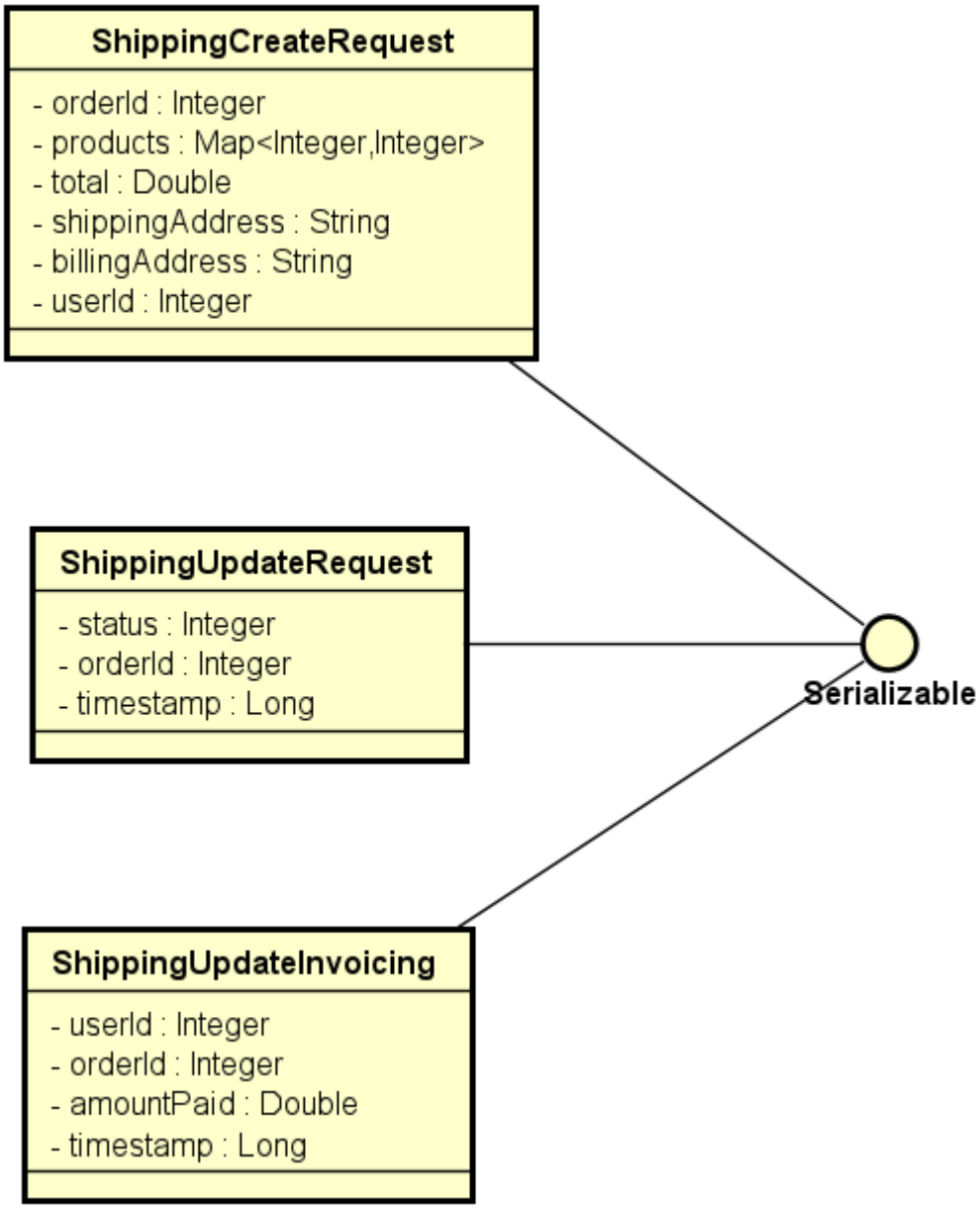

Figura 3: DCD ShippingCreateRequest, ShippingUpdateRequest e ShippingUpdateInvoicing

Docker

Per il deployment del servizio è stato utilizzato Docker, creando un “docker-compose.yml” nella directory del progetto. All’interno sono definite delle variabili relative all’environment di Kafka e altre relative alla connessione con il database. Insieme alla descrizione del servizio di Shipping, in cui viene specificato il percorso del Dockerfile, esposte le porte e inserite le variabili necessarie, vengono deployati anche i seguenti servizi:

- MySQL, per la gestione del database.
- Kafka, per la gestione dei messaggi.
- ZooKeeper, per la gestione del broker Kafka.

Per l’utilizzo di MySQL viene utilizzata l’immagine “mysql” con i dati di accesso precedentemente definiti (come quelli nel .env: nome del db = ship, nome utente = springant e password = DSB2020!).

Kafka è anch’esso deployato usando le variabili relative al Broker ID, alla porta, ai topic da creare e infine all’endpoint del listener.

Nel Dockerfile, presente all’interno del modulo “shipping”, vengono inviati i comandi relativi all’installazione delle dipendenze maven, presenti nel pom.xml, viene creato l’eseguibile “shipping.jar” e infine avviato tramite il comando “java -jar shipping.jar” all’interno del container.