



UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI INGEGNERIA ELETTRICA ELETTRONICA E
INFORMATICA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Andrea Giuseppe Cicero
Mario Roberto Nuñez Pereira

PROGETTO DSBD 1-B

RELAZIONE

Sommario

1 – Introduzione	3
2 – Architettura dell'applicazione.....	3
2.1 – MongoDB	4
2.2 – Product Manager	4
2.3 – Zookeeper	5
2.4 – Kafka	5
3 – Struttura del microservizio Product Manager	6
4 – Entità database.....	7
4.1 – Product.....	7
4.2 – Category	7
4.3 – DatabaseSequence.....	8
5 – Controllers, Services e Repositories.....	8
5.1 – Repositories	9
5.2 – Controllers.....	9
5.3 – Services	10
6 – Gestione degli errori.....	11
7 – Kafka	12

1 – Introduzione

In questa relazione tratteremo la realizzazione di un microservizio, da noi arbitrariamente denominato *Product manager*, per la gestione dei prodotti di un e-commerce. In particolare le interazioni sul microservizio riguardano lo sviluppo di interfacce REST, l'implementazione di Kafka per produrre e consumare dei messaggi su di uno specifico topic, e l'utilizzo di *MongoDB* come DBMS non relazionale. Il microservizio prevede un handler delle eccezioni, molto utile per catturare e ottenere una corretta gestione di un eventuale errore commesso dall'utilizzatore ed avere delle informazioni tecniche su di esso.

2 – Architettura dell'applicazione

Per permettere il corretto funzionamento dell'applicazione secondo le specifiche ricevute nella consegna sono stati implementati dei container docker:

- *productmanager*: container su cui gira l'applicazione effettiva che si occupa di fornire le funzionalità richieste;
- *mongo*: container su cui vengono resi persistenti i dati inseriti e/o modificati tramite le funzioni offerte dal servizio *productmanager*;
- *zookeeper*: container che fornisce un server per l'invio dei messaggi tra processi distribuiti;
- *kafka*: container che funge da broker per l'invio e la ricezione dei messaggi scambiati tra produttori e consumatori sui vari topic.

I servizi citati sopra sono stati definiti ed eseguiti all'interno di un sistema multi-container di Docker. Nello specifico, nella root del microservizio si ha un file *docker-compose.yml*, dove vengono definiti e configurati i servizi dell'applicazione. Insieme al *docker-compose.yml*, nella root di riferimento si presenta anche un Dockerfile relativo al microservizio *productmanager*, per definire attraverso un insieme di istruzioni, un template opportuno da cui partire per creare una immagine su misura. Di seguito sono descritte alcune specifiche di configurazione e creazione dei servizi utilizzati, presenti all'interno del *docker-compose.yml*.

2.1 – MongoDB

Al momento della creazione del container, Mongo necessita di essere configurato:

- Le variabili d'ambiente relative ad username e password, per la configurazione iniziale dell'utente amministratore, che sono le medesime utilizzate dal Product manager per le successive configurazioni.
- Le porte alla quale si espone il servizio sulla rete, che mappa la porta del container allo stesso numero di quella dell'host 27017;
- Il volume alla quale dovranno essere memorizzati i dati del DBMS.

2.2 – Product Manager

Al momento della creazione del container, il Product Manager riceve:

- le variabili d'ambiente relative ad username e password per accedere al nostro DBMS;
- Le porte alla quale si espone il servizio sulla rete, che mappa la porta del container allo stesso numero di quella dell'host 3333.

Inoltre, come citato in precedenza, è stato necessario definire un Dockerfile per il servizio *productmanager*, in cui vengono specificati i comandi per automatizzare la creazione personalizzata di una immagine del servizio stesso sul container. Per iniziare, all'interno del Dockerfile, si definisce una *build in più fasi*: la prima fase della build prevede l'uso dell'immagine di base ufficiale di Maven per la compilazione del codice sorgente dell'applicazione; nella seconda fase, è possibile accedere all'output della build precedente in quella corrente, in cui il jar costruito viene assemblato nell'immagine di output finale. Per la build in più fasi è necessario:

- L'uso di più istruzioni FROM nel Dockerfile per definire le *base image* da cui partire per derivare la nostra immagine personalizzata (nel nostro caso, maven:3-jdk-8 e java:8-alpine).
- Specificare la work directory all'interno del container su cui avranno effetto tutte le successive istruzioni, tra queste compaiono la copia della directory del progetto e l'eseguibile dell'applicazione *productmanager.jar*.

2.3 – Zookeeper

Al momento della creazione del container, viene eseguito Zookeeper, un servizio centralizzato in grado di fornire una sincronizzazione robusta e flessibile tra i nodi Kafka, e tiene traccia dello stato dei nodi del cluster Kafka, dei topic e delle partizioni. Pertanto, per usufruire di Kafka è necessario eseguire prima Zookeeper.

2.4 – Kafka

Al momento della creazione del container, kafka ha bisogno di essere configurato. Citiamo di seguito alcune delle variabili di ambiente utilizzate relative alla configurazione di Kafka:

- **KAFKA_BROKER_ID**, per configurare l'ID del broker;
- **KAFKA_CREATE_TOPICS**, per creare dei nuovi topic, esplicitando per ciascuno di essi le relative partizioni e repliche. I topic creati sono *orders*, *logging* e *notifications*, hanno tutti una partizione.
- **KAFKA_ADVERTISED_HOST_NAME**, per pubblicizzare il nome dell'host;
- **KAFKA_ADVERTISED_PORT**, con valore pari a 9092, in modo che i client kafka possano connettersi correttamente dopo il rilevamento;
- **KAFKA_LISTENERS**, per indicare l'host e la porta a cui Kafka si collega per l'ascolto;
- **KAFKA_ZOOKEEPER_CONNECT**, per indicare a Kafka come entrare in contatto con ZooKeeper.

Inoltre, come accennato sul service di Zookeeper, è necessario esprimere un ordine di partenza dei servizi. Infatti, utilizziamo `depends_on` su Kafka per indicare che l'avvio di Kafka segue quello di Zookeeper.

3 – Struttura del microservizio Product Manager

Il microservizio Product Manager è realizzato in Spring Boot con il supporto del plug-in Apache Maven capace di compilare, eseguire e gestire tramite un tool di *build automation* le dipendenze. Con l'aiuto di Spring Initializr generiamo la struttura del progetto in Spring Boot, dopodiché definiamo le dipendenze Maven necessarie dell'applicazione. L'elenco delle dipendenze Maven per lo sviluppo del microservizio è definito nel file *pom.xml*. In particolare, tra le dipendenze adottate è incluso:

- il modulo *spring-boot-starter-web*, per l'iniezione delle dipendenze e le applicazioni RESTful utilizzando Spring MVC;
- il modulo *spring-boot-starter-validation* per il supporto alla convalida dei bean di Spring Boot;
- il modulo *spring-boot-starter-data-mongodb*, per le API di Spring Data MongoDB.

(Ulteriori precisazioni in merito alle dipendenze utilizzate sono riportate nel paragrafo 7 – *Kafka*).

Il passo successivo è stato specificare su *application.properties* del progetto le proprietà di Spring Boot e i parametri di connessione ascrivibili al database MongoDB, come ad esempio:

```
spring.data.mongodb.database=${MONGO_DB_NAME}
spring.data.mongodb.host=${MONGO_HOST}
spring.data.mongodb.username=${MONGO_USER}
spring.data.mongodb.password=${MONGO_PASS}
spring.data.mongodb.port=${MONGO_PORT}
```

Da notare, la possibilità di gestire file di property con variabili, denotate dal delimitatore *\${..}*, dove per agevolezza vengono valorizzate sul file *docker-compose.yml* e richiamate dove occorrono. Infine, il microservizio Product Manager al suo interno contiene tutte le classi, illustrate nei paragrafi successivi, necessarie per il funzionamento desiderato descritto dalla consegna assegnataci, accuratamente racchiuse in appositi package distinti per tipologia di classi contenenti.

4 – Entità database

La creazione dei documenti in formato BSON del database gestito tramite MongoDB, contenenti i dati da mantenere in modo persistente per la corretta esecuzione delle API REST, viene realizzata automaticamente dalla nostra applicazione grazie all'uso dell'annotazione `@Document`. Tale annotazione contrassegna le classi **Product**, **Category** e **DatabaseSequence**, ognuna delle quali contiene come attributi privati i dati che verranno memorizzati sul database o recuperati da esso a seguito di una richiesta. Queste classi, inoltre, hanno in comune la presenza di un attributo annotato con `@Id`, che rappresenta l'identificativo univoco di ogni record salvato all'interno del rispettivo documento BSON. Di seguito vengono descritte nello specifico le varie classi rappresentanti le entità del database, le loro peculiarità e l'uso che ne viene fatto.

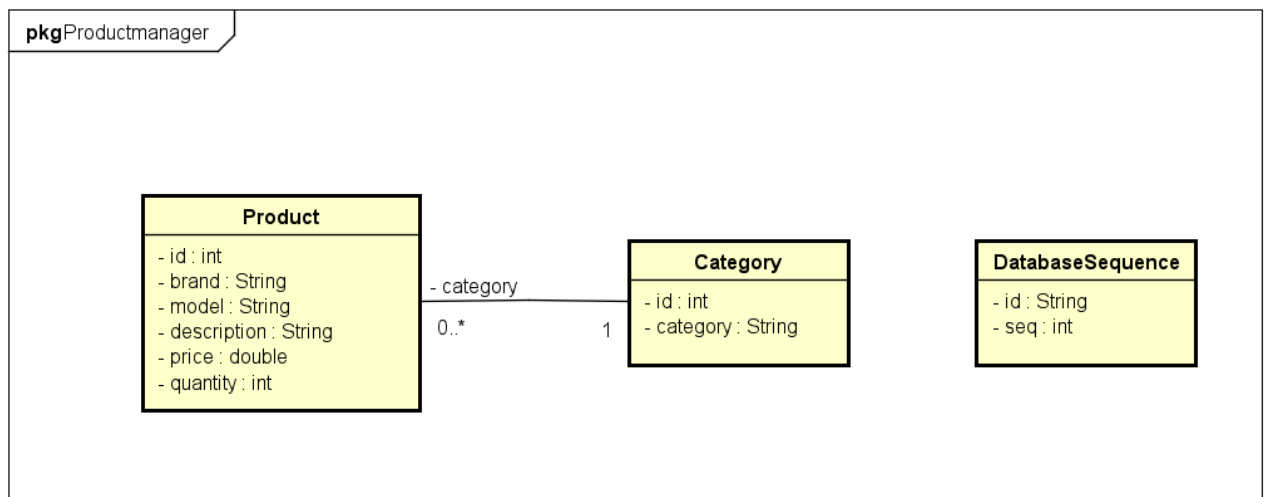


Diagramma UML classi entità database

4.1 – Product

La classe **Product**, oltre ai normali attributi riguardanti i dati relativi ad uno specifico prodotto (come, ad esempio, *brand* o *model*), possiede l'attributo *category* che viene annotato con `@DBRef` per caratterizzarlo come riferimento ad una sola categoria tra quelle presenti nell'entità **Category** del database, contenente le varie categorie di prodotto tra cui scegliere durante l'inserimento dello stesso.

4.2 – Category

La classe **Category** viene utilizzata per rappresentare le singole categorie che verranno assegnate ai prodotti durante la loro aggiunta sul database, e perciò, in aggiunta all'attributo comune dell'id, possiede un unico attributo *category* per specificare il nome della categoria.

4.3 – DatabaseSequence

La classe **DatabaseSequence**, al contrario di **Product** e **Category**, non rappresenta un'entità nel database su cui è possibile interagire direttamente tramite le API REST, ma è una classe atta a creare un documento BSON utile per tenere traccia delle sequenze numeriche che verranno utilizzate per l'auto generazione ed incremento degli id numerici per ogni nuovo record salvato nel database.

5 – Controllers, Services e Repositories

Per il corretto funzionamento della nostra applicazione, essa prevede l'utilizzo di tre categorie principali di componenti, ovvero *Controllers*, *Services* e *Repositories*:

- **Controller**: si occupano di gestire le interazioni tra utenti e logica di *business*. Ricevono comandi ed espongono le informazioni in modo da essere comprensibili per l'utente o utilizzabili da altri sistemi. L'annotazione che identifica questo tipo di componente è *@Controller*.
- **Service**: questa tipologia di componenti è identificata dall'annotazione *@Service* ed ha la funzione di elaborare i dati e di fornirli ai Controller per essere esposti verso il client. Allo stesso tempo, le informazioni da salvare vengono inviate allo strato di accesso ai dati.
- **Repository**: si tratta di interfacce dedicate a specifiche entità che forniscono un insieme di metodi utili al recupero dei dati da una generica sorgente (nel nostro caso *MongoDB*).

Quindi, a seguito della creazione delle classi **Product** e **Category**, sono stati creati anche i relativi repositories (**ProductRepository**, **CategoryRepository**), services (**ProductService**, **CategoryService**) e controllers (**ProductController**, **CategoryController**), mentre per la classe **DatabaseSequence**, essendo essa una classe di appoggio necessaria per l'autogenerazione degli id, è stato creato il solo Service **NextSequenceService**.

5.1 – Repositories

L'interfaccia **ProductRepository** estende l'interfaccia **MongoRepository<T, ID>** (dove **T** e **ID**, in questo caso, assumono rispettivamente i valori di **Product** ed **Integer**). Tramite questa estensione possiamo usufruire delle varie funzioni messe a disposizione per il recupero e la memorizzazione su database *Mongo* delle informazioni relative ai prodotti del nostro ipotetico e-commerce, comprese le funzioni necessarie per la loro paginazione. A queste funzioni, per nostro uso futuro, aggiungiamo la signature del metodo **findByModel**, con cui sarà possibile recuperare uno specifico prodotto attraverso l'inserimento del modello dello stesso. Tale metodo sarà usato per effettuare dei controlli sui dati inseriti o ricercati dall'utente che effettua la richiesta.

Per l'interfaccia **CategoryRepository** vale tutto ciò che è stato precedentemente detto per **ProductRepository**, con la differenza che il valore assunto da **T** nell'estensione del **MongoRepository** è pari a **Category** ed al posto di avere il metodo **findByModel** si ha **findByCategory** con lo stesso principio di funzionamento ma applicato alle categorie, anch'esso usato per i controlli sui dati inseriti.

5.2 – Controllers

I controllers **ProductController** e **CategoryController** sono le classi contenenti le API REST da noi sviluppate, mappate secondo le specifiche richieste nella consegna assegnataci attraverso le quali è possibile interagire con il database operante sul container *mongo* descritto in precedenza. Tali API sono uguali per entrambe le classi ma operanti su entità diverse. Nello specifico, si hanno 4 API principali per ogni controller:

- una POST per l'aggiunta di un/una prodotto/categoria sul database;
- una PUT per la modifica di uno/a specifico/a prodotto/categoria sul database tramite l'inserimento dell'id univoco nel path della richiesta;
- una GET per il recupero di tutte le informazioni relative ad uno/a specifico/a prodotto/categoria sul database tramite l'inserimento dell'id univoco nel path della richiesta;
- una GET per il recupero di un elenco contenente tutti i dati su tutti i/le prodotti/categorie visualizzati secondo i criteri di paginazione indicatici;

Le istruzioni per l'effettiva esecuzione di queste API sono demandate ai metodi omonimi presenti sulle rispettive classi Service, **ProductService** e **CategoryService**.

5.3 – Services

I services **ProductService** e **CategoryService** sono le classi al cui interno risiede tutta la logica per l'esecuzione delle API sopra descritte. Per rendere atomiche le operazioni svolte tramite i metodi che risiedono in queste classi, entrambe le classi sono state annotate con *@Transactional*. Ogni metodo definito in queste classi rappresenta la logica di una specifica API, che opera sulla relativa entità tramite i collegamenti ai rispettivi repositories creati tramite l'annotazione *@Autowired*, ed al loro interno sono previsti i controlli per verificare il corretto inserimento o l'effettiva presenza dei dati che si intende memorizzare, modificare o recuperare sul database tramite la richiesta (è ad esempio vietato inserire una categoria dal nome identico ad un'altra o un prodotto dello stesso modello di un altro già presente). Al fallimento di uno di questi controlli verrà lanciato un tipo di eccezione diversa (la cui gestione è spiegata nel paragrafo 6 – *Gestione degli errori*) in base al tipo di errore commesso dall'utente.

Le meccaniche all'interno di questi service sono pressoché identiche anche se applicate su entità diverse, ma è comunque necessario sottolineare alcune cose. Per prima cosa, come scelta implementativa per rendere meno tediosa all'utente la modifica dei dati di un prodotto, il metodo **updateProduct** (situato in **ProductService**) permette modificare anche un solo campo del prodotto con l'id indicato nel path, e quindi non è necessario inserire ad ogni richiesta tutti i campi di un prodotto per mandare a buon fine la richiesta. Questo è possibile grazie all'uso di un metodo privato del service, **getRightParam**, richiamato all'interno della chiamata ai metodi set che permettono la modifica vera e propria dei dati del prodotto, a cui vengono passati come parametri il valore del dato inserito nel body della richiesta e quello posseduto in precedenza dal prodotto. Tale metodo ritornerà il valore inserito nel body solo se questo è non nullo (cosa possibile solo se l'utente ha fatto un inserimento corretto) oppure il valore già posseduto dal prodotto. Invece, per quanto riguarda il recupero e la visualizzazione di tutti/e i/le prodotti/categorie, è importante evidenziare che nelle rispettive API GET sono stati assegnati dei valori di default ai parametri interi *per_page* e *page*, utilizzati per effettuare la paginazione degli elementi recuperati. In questa maniera è possibile fare la

richiesta di visualizzazione anche con il solo path */categories*, senza che sia necessario specificare ogni volta i parametri di paginazione in forma esplicita tramite query.

Il service **NextSequenceService** è una classe che contiene un solo metodo, *getNextSequence*, con cui è possibile recuperare ed incrementare di uno automaticamente il numero degli id presenti nel database riferiti da una specifica entità, in modo tale da assegnare il valore incrementato all'id del prodotto o della categoria che si sta inserendo in quel momento tramite l'API POST */products* o POST */categories*. Per l'esattezza, il metodo ritorna un contatore con il valore incrementato che è istanza della classe **DatabaseSequence**, di cui abbiamo parlato in precedenza. Questo service è necessario poiché utilizzando *MongoDB* non è possibile sfruttare l'annotazione *@GeneratedValue(strategy = GenerationType.AUTO)* posta sull'attributo che s'intende usare come id.

6 – Gestione degli errori

Per ottenere una corretta gestione degli errori all'interno dell'applicazione che ci è stato assegnato, è stato fatto uso dell'annotazione *@Valid*, per assicurare un corretto inserimento dei dati nelle richieste di aggiunta prodotto e/o categoria, unitamente alla classe **RestExceptionHandler**, da noi appositamente creata, contrassegnata dall'annotazione *@ControllerAdvice*. Tale annotazione offre un supporto per la gestione unificata delle eccezioni in un'intera applicazione, abilitando un meccanismo che si stacca dal vecchio modello MVC e fa uso del tipo oggetto *ResponseEntity* insieme alla flessibilità dell'annotazione *@ExceptionHandler*. L'annotazione *@ControllerAdvice*, ci consente di centralizzare la gestione di tutte le eccezioni sollevate dall'applicazione in un unico componente globale. In questo modo, otteniamo un meccanismo estremamente semplice ma anche molto flessibile, poiché:

- permette il pieno controllo sul body della risposta e sul codice di stato;
- Fornisce la mappatura di diverse eccezioni sullo stesso metodo ed ottenendo un comportamento comune per esse.

Per una migliore differenziazione e identificazione degli errori, inoltre, sono state create delle classi eccezione apposite, tutte quante estendenti la classe **RuntimeException**, personalizzate per restituire ognuna un diverso messaggio a seconda della tipologia d'errore commesso da chi effettua la richiesta tramite API REST. La classe **RestExceptionHandler**, nello specifico,

prevede l'utilizzo di un solo metodo principale, denominato *handleException*, annotato con *@ExceptionHandler(Exception.class)* per la gestione di qualsiasi eccezione generata dall'applicazione, la cui logica interna prevede l'assegnazione dei valori esatti ai campi necessari per la corretta generazione del messaggio d'errore da inviare tramite Kafka sul topic logging. Ciò avviene tramite dei controlli per il riconoscimento dell'eccezione lanciata, al cui interno vi è l'assegnazione del codice d'errore o di una stringa contenente lo stack trace dell'eccezione generata e l'istanziamento della *ResponseEntity* da restituire, che esplicherà l'errore commesso nel body della risposta. Per concludere, l'annotazione *@Valid* fa sì che venga lanciata l'eccezione *MethodArgumentNotValidException* ogni qual volta uno dei valori inseriti per l'aggiunta di un prodotto o di una categoria è nullo (e di conseguenza si ha un errore nell'inserimento). Tale eccezione, non essendo una di quelle da noi create bensì presente tra le eccezioni del framework di Spring, per essere gestita necessita di un override del metodo *handleMethodArgumentNotValid*, il quale si occuperà di richiamare il sopracitato metodo *handleException*, passandogli come parametri l'eccezione stessa, la *WebRequest* e l'*HttpServletRequest*. Quest'ultimo parametro contiene le informazioni di nostro interesse riguardanti la richiesta che è stata effettuata con un errore d'inserimento nel path o nel body, e viene recuperato tramite il metodo *getCurrentHttpRequest*, anch'esso appositamente scritto.

7 – Kafka

Come già accennato, si hanno i servizi kafka e Zookeeper nei container di Docker. Affinché questi servizi abbiano un corretto funzionamento, all'interno del Product manager service è necessario introdurre:

- la dipendenza Maven *spring-kafka*;
- le classi di configurazione del servizio kafka;
- definire i topic alla quale si è produttori e consumatori;
- la business logic relativa alla messaggistica di Kafka.

L'implementazione di Kafka si trova all'interno del package *kafka*, precisamente nella Source root *dsbd2020/project/productmanager/kafka*, dove all'interno sono presenti le classi di

configurazione **KafkaProducerConfig** e **KafkaConsumerConfig**. Sempre all'interno del package *kafka*, abbiamo anche la classe **KafkaOrder**, dove viene definita la logica applicativa relativa alla messaggistica di kafka.

Riguardo alla prima classe di configurazione, **KafkaProducerConfig**, tramite l'utilizzo delle annotazioni *@Configuration* e *@Bean*, è possibile:

- La creazione di una mappa, attraverso il metodo ***producerConfigs***, in possesso dei parametri di configurazione per ogni istanza del client kafka Producer. Nello specifico, tali parametri definiscono alcune proprietà come:
 - ***BOOTSTRAP_SERVERS_CONFIG***: Host e porta su cui è in esecuzione Kafka;
 - ***KEY_SERIALIZER_CLASS_CONFIG***: Classe di serializzazione da utilizzare per la chiave;
 - ***VALUE_SERIALIZER_CLASS_CONFIG***: Classe di serializzazione da utilizzare per il valore.
- Definire un metodo per l'ottenimento di una *ProducerFactory* al fine di realizzare una strategia di creazione delle istanze di kafka Producer.
- Definire un metodo per la creazione di un *KafkaTemplate*, che consentirà l'invio di messaggi sui topic.
- Uno o più Bean responsabili della creazione di nuovi topic sul nostro broker.

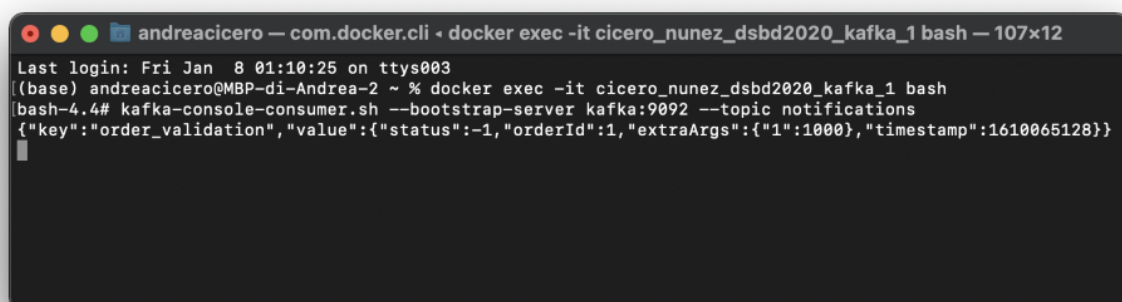
Come scelta implementativa efficiente abbiamo un'unica istanza di Producer in grado di produrre messaggi su più topic.

Invece, per quanto riguarda la seconda classe di configurazione, **KafkaConsumerConfig**, tramite l'utilizzo delle stesse annotazioni, è possibile:

- la creazione di una mappa con i parametri di configurazione per ogni istanza del client kafka Consumer, in cui però, a differenza del Producer, si effettua la de-serializzazione di chiavi e valori;
- definire una *ConsumerFactory* per la strategia di creazione di una nuova istanza di Kafka Consumer;

Come scelta implementativa abbiamo un'unica istanza di Consumer in grado di ascoltare messaggi sul topic *orders*.

Come menzionato prima, abbiamo anche la classe **kafkaOrder**, dove viene definita la logica applicativa della messaggistica. All'interno della classe si nota come il Product Manager Service si comporta sia da Consumer kafka del topic *orders* che da Producer dei topic *logging*, *notifications* e *orders*. Sul codice è stato necessario l'utilizzo dell'annotazione *@KafkaListener*, che rende il metodo della classe un ascoltatore sul topic *orders* del messaggio *order_completed*. Quando l'ascoltatore riceve i messaggi, deserializza una stringa che rappresenta un Gson della richiesta ad uno oggetto della classe **TopicOrderCompleted**. Come da traccia progettuale, l'oggetto ricevuto di cui siamo consumatori subisce delle verifiche sulla validità della richiesta, che conducono verso un responso. L'elaborazione prevede la verifica della disponibilità della quantità di ciascun prodotto e la validità dell'importo ricevuto dell'ordine sia corretto. Sulla base dei risultati disponibili creiamo e settiamo un oggetto della classe **TopicOrderValidation**, della quale il Product manager Service è anche un Producer sui topic *logging*, *notifications* e *orders*. Per l'invio degli oggetti Java su un topic, è necessario un *KafkaTemplate* per invocare il metodo **send** passando come parametri il nome del topic e il messaggio. Quando si parla di messaggio, ci riferiamo all'oggetto *TopicOrderValidation* in Gson serializzato in Stringa. Inoltre, per semplificare l'organizzazione abbiamo creato un package *messageKafka* con all'interno le relative classi che rappresentano i messaggi scambiati tra il Consumer e Producer Kafka. Infine, di seguito viene mostrato uno screenshot di una finestra CLI in cui viene messo in ascolto un consumatore kafka predefinito sul topic *notifications*, si può notare la ricezione di un messaggio di tipo **TopicOrderValidation** prodotto dalla nostra applicazione dopo aver ricevuto un messaggio sul topic *orders*.



```
andreacicero — com.docker.cli • docker exec -it cicero_nunez_dsbd2020_kafka_1 bash — 107x12
Last login: Fri Jan  8 01:10:25 on ttys003
(base) andreacicero@MBP-di-Andrea-2 ~ % docker exec -it cicero_nunez_dsbd2020_kafka_1 bash
bash-4.4# kafka-console-consumer.sh --bootstrap-server kafka:9092 --topic notifications
{"key":"order_validation","value":{"status":-1,"orderId":1,"extraArgs":{"1":1000},"timestamp":1610065128}}
```