



Università  
di Catania



# Corso di Distributed Systems and Big Data a.a. 2021/22

Vincenzo Pluchino 1000023070, Philip Tambè 1000015375

## Relazione progetto: piattaforma per la vendita di film online

### Sommario

1	Introduzione.....	2
2	Architettura dell'applicazione e scelte implementative.....	2
3	Descrizione dei servizi del sistema distribuito + tools .....	6
3.1	Movie service .....	6
3.2	Common DTO.....	9
3.3	Order Service .....	10
3.4	Payment Service .....	15
3.5	API Gateway .....	18
3.6	Kafka e Saga (Choreography Pattern) .....	19
4	Deployment su Docker .....	27

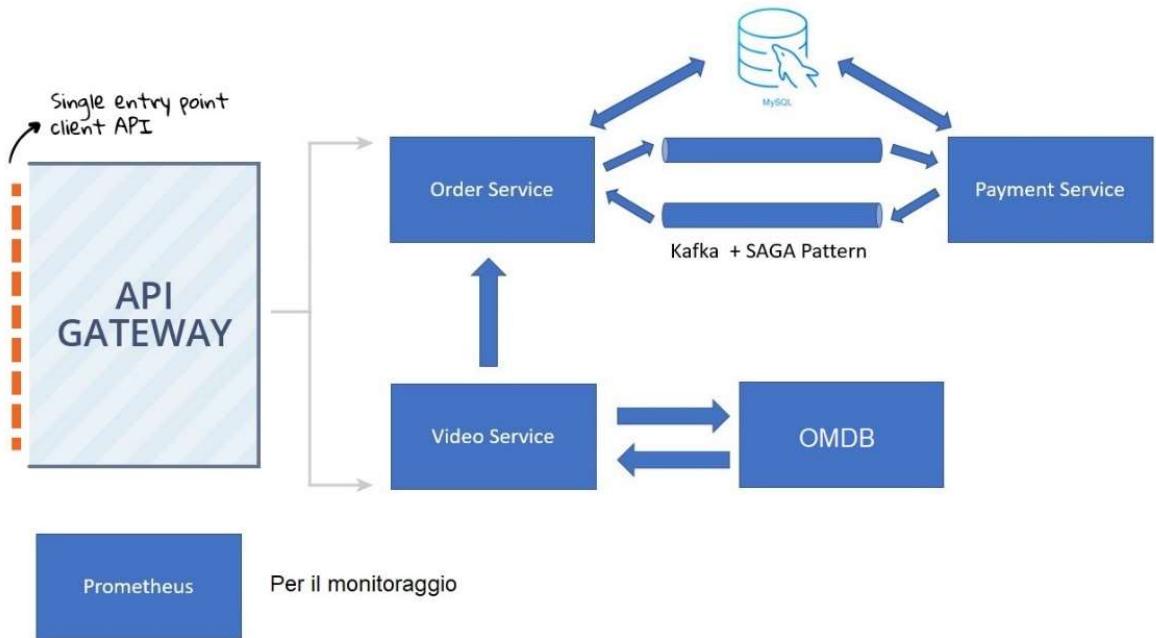
5	Deployment su Kubernetes .....	30
6	Monitoraggio con Prometheus.....	34
7	Testing e risultati.....	35

## 1 Introduzione

Lo scopo dell'elaborato è lo sviluppo di una piattaforma che permetta l'acquisto di film (in formato digitale, no copie fisiche). L'utente ha a disposizione una funzione di ricerca che gli consente di trovare tutti i film disponibili tramite la ricerca del titolo o dell'id (qualora ne fosse a conoscenza tramite siti web esterni) del film; prima di poter effettuare la ricerca, è importante che inserisca il suo identificativo per poter capire a quale utente ci si sta riferendo.

Una volta trovato il film che si intende acquistare, l'utente potrà quindi procedere ad effettuare l'ordine: a quel punto, verrà creato un ordine con stato di pagamento pendente. I pagamenti nella piattaforma avvengono tramite la decurtazione del credito: ogni utente, infatti, avrà a disposizione un credito che, una volta terminato, dovrà ricaricare (presumibilmente con un servizio esterno che va al di fuori dello scopo di questo elaborato). Lo stato di pagamento dell'ordine potrà quindi avere due stati: "completato" o "fallito". Nel primo caso, l'utente ha il giusto quantitativo di credito sulla piattaforma per poter effettuare l'acquisto e quindi il tutto è andato a buon fine (con conseguente diminuzione del credito); nel secondo, l'utente non ha sufficiente credito per acquistare il film scelto e quindi il pagamento fallisce e non avviene nessuna decurtazione del credito residuo dell'utente.

## 2 Architettura dell'applicazione e scelte implementative



*Architettura di riferimento*

Entrando più nel dettaglio, è possibile andare a visualizzare (tramite la figura sopra) i microservizi creati per il funzionamento della piattaforma. Essi, infatti, sono:

- 1) Video Service
- 2) Order Service
- 3) Payment Service

Video Service (o anche Movie Service) è il microservizio che si occupa della ricerca del film tramite il titolo o tramite l'identificativo. La sua funzione principale è quella di interfacciarsi con un servizio esterno che permette la ricerca effettiva dei film, chiamato OMDb. The Open Movie Database è un servizio web RESTful che mette a disposizione delle API che consentono di ottenere molteplici informazioni su un film specifico: il vastissimo database da cui attinge OMDb gli permette inoltre di possedere informazioni su quasi qualsiasi film conosciuto, a prescindere dall'anno di uscita o della popolarità.

Tramite queste API, il microservizio svolge così il suo compito primario. Le altre funzioni sono quelle di verificare l'identificativo immesso dall'utente (la funzione di ricerca è disabilitata sin quando non si immette un id utente valido) e inviare le informazioni dei film a un altro microservizio che verrà introdotto subito dopo, Order Service.

*Nota implementativa importante:* essendo un progetto a fini universitari, si è deciso di facilitare alcuni aspetti come, ad esempio, la disponibilità dei film all'interno della piattaforma: si è ipotizzato che, per lo scopo di questo elaborato, tutti i film

ricercabili attraverso le API di OMDb siano altrettanto disponibili per l'acquisto nella piattaforma.

Il secondo microservizio è Order Service, colui che si occupa di creare l'ordine di acquisto qualora l'utente avesse trovato il film da lui desiderato. Insieme al terzo microservizio, è collegato in modo diretto a un database di tipo MySQL in cui è presente una tabella che tiene traccia degli ordini effettuati dai vari utenti e che si occupa di aggiornare con delle query opportune.

La funzione principale è quella di settare correttamente i campi dell'ordine grazie alle informazioni del film inviatogli da Movie Service e pubblicare poi nel topic Kafka queste informazioni tramite un evento. Questi eventi saranno "consumati" da Payment Service che si occuperà del pagamento. In particolare, l'ordine sarà evaso solo se il prezzo del prodotto rientra nel credito dell'utente.

Una volta controllato lo stato del pagamento, lo stato dell'ordine verrà aggiornato di conseguenza (completato se è andato a buon fine, cancellato se il pagamento ha avuto problemi facendo il rollback delle modifiche).

Payment Service è l'ultimo microservizio progettato per questa piattaforma che, come si intuisce dal nome, ha l'obiettivo primario di gestire il pagamento dell'ordine del film effettuato dall'utente. Come già scritto qualche riga prima, anche questo servizio è collegato al db MySQL in quanto si occupa di tenere traccia delle transazioni e di monitorare la tabella degli utenti registrati in piattaforma per poter effettuare il controllo sul loro credito disponibile in caso di acquisto.

Attraverso la ricezione di un evento Ordine tramite Kafka raccoglie le informazioni utili (come l'id dell'utente, il costo del film...) e setta lo stato del pagamento a seconda se esso sia andato a buon fine oppure no, per poi inviare l'informazione a Order Service che setterà di conseguenza lo stato dell'ordine.

*Altra nota implementativa:* non essendo implementati in quest'applicazione i meccanismi di registrazione (e anche di login), sono stati creati 5 utenti memorizzati nel database, ognuno con un saldo differente, ai fini di testing.

Sempre in riferimento all'architettura, si noti l'utilizzo del **pattern Saga** che coinvolge due microservizi: lo scopo per cui è stato utilizzato è garantire l'ordine di esecuzione delle transazioni che fanno parte della stessa saga, ovvero che devono essere eseguite in un ordine prestabilito e il cui effetto non è compromesso da eventuali transazioni intermedie appartenenti a saghe diverse. In particolare, il pattern Saga implementato si basa su un **approccio coreografico** che è stato preferito all'approccio ad orchestrazione poiché nel nostro specifico caso, avendo un numero limitato di entità che si scambiano eventi all'interno della saga (solo due)

questo diminuisce di molto la complessità delle transazioni e quindi degli scambi di eventi che deve essere effettuati per mantenere coerente lo stato. Infatti, nel caso in cui avessimo usato il pattern orchestrator avremmo dovuto inserire un ulteriore servizio che fungesse da orchestrator appunto per tutti gli scambi di eventi all'interno della saga.

È presente inoltre un **API Gateway**, che prende tutte le chiamate API dai clienti e le instrada al giusto microservice usando il routing della richiesta. Così facendo, lato client si utilizza solamente un unico punto di ingresso piuttosto che effettuare richieste API a diversi url in base al microservizio che si intende utilizzare e così si ha pure un controllo all'accesso ai sistemi e servizi back-end andando ad ottimizzare la comunicazione tra i clienti esterni e i servizi back-end, dando nel frattempo un'esperienza fluida.

Infine, l'elaborato conterebbe anche dell'utilizzo di **Prometheus**: si tratta di un tool di monitoraggio e alerting che archivia metriche in un database proprietario di timeseries, dove una timeseries (serie temporale) non è altro che una serie di data point indicizzati in ordine temporale, ovvero una sequenza presa in punti successivi ugualmente distanziati nel tempo. In particolare, la scelta è stata orientata verso il blackbox monitoring che consiste nel monitorare le risorse computazionali (es. CPU, ram, spazio di archiviazione) utilizzate dagli ambienti di esecuzione ed opzionalmente quelle di rete.

Per quanto riguarda **Docker** e **Kubernetes**, finora non menzionati neanche nell'immagine, si discuterà successivamente nei prossimi paragrafi; entrambi comunque sono stati correttamente utilizzati per la realizzazione di questo sistema distribuito.

Oltre ai requisiti finora descritti e opportunamente concordati con gli insegnanti del corso, ne sono stati aggiunti ulteriormente altri due che hanno reso l'elaborato più completo e ottimizzato:

- Apache **Kafka**, tool essenziale in combinazione con Saga, che permette di gestire flussi di dati provenienti da più sorgenti distribuendoli a più consumatori consentendo di spostare grandi quantità di dati da un punto qualsiasi a un altro nello stesso momento.

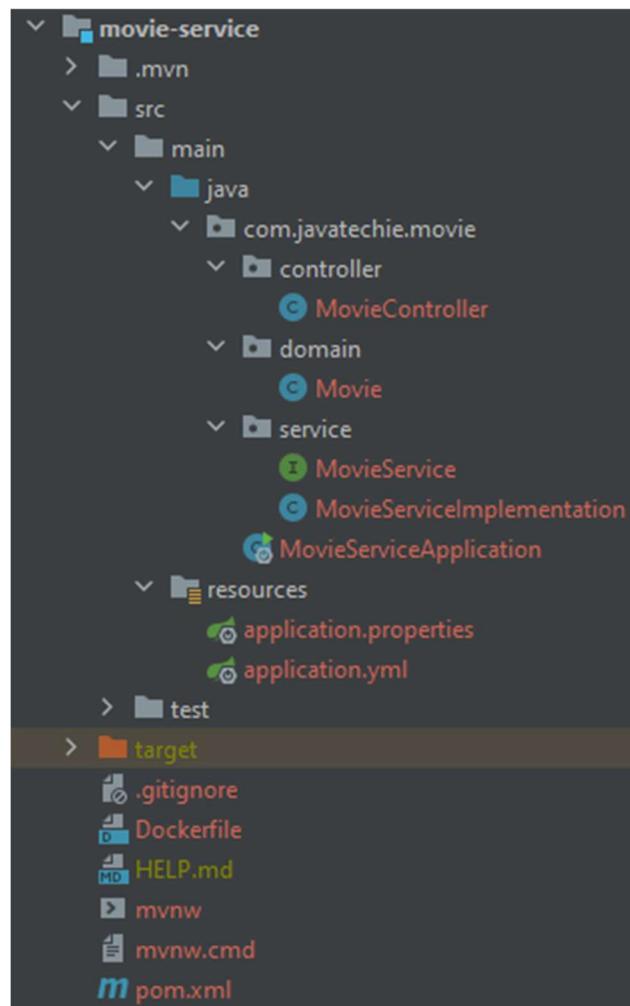
- **Reactive programming**, paradigma di programmazione che consente agli sviluppatori di creare applicazioni asincrone non bloccanti in grado di gestire il controllo del flusso di dati.

*N.B.* I requisiti scelti da noi per quest'elaborato e concordati dai docenti erano solamente i seguenti: blackbox monitoring (Prometheus), API Gateway, 2 MS business + Saga, Docker e K8S.

### 3 Descrizione dei servizi del sistema distribuito + tools

Come IDE per lo sviluppo del codice e per l'organizzazione dei package è stato utilizzato IntelliJ, in particolare Java con l'uso di Spring Boot insieme al plugin Maven.

#### 3.1 Movie service



*Organizzazione dei package e dei file del microservizio*

Per questo microservizio sono state create 3 classi e un'interfaccia, ognuna con un proprio scopo.

La classe "Movie" dentro il package Domain definisce come debbano essere gli oggetti Film che verranno poi utilizzati nella ricerca e nell'invio per creare l'ordine,

impostando tutti gli attributi (titolo, anno, genere, tipo...) e i getter & setter. L'annotazione @JsonProperty permette di rinominare le variabili per un corretto parse quando si invieranno/riceveranno richieste API contenenti oggetti Movie. L'interfaccia MovieService definisce i metodi di ricerca del film (più il controllo dell'identificativo dell'utente) che vengono poi di fatto implementati dalla classe MovieServiceImplementation.

```

@Override
public Mono<Movie> searchMovieByTitle(String apiKey, String title) {
    Mono<Movie> mov= webClient.post() WebClient.RequestBodyUriSpec
        .uri( uri: "/?apikey="+apiKey+"&t="+title) WebClient.RequestBodySpec
        .retrieve() WebClient.ResponseSpec
        .bodyToMono(Movie.class);
    mov.subscribe(s -> sendMovieToOrder(s));
    return mov;
}

@Override
public Mono<Movie> searchMovieById(String apiKey, String imdbId) {
    Mono<Movie> mov= webClient.post() WebClient.RequestBodyUriSpec
        .uri( uri: "/?apikey="+apiKey+"&t="+imdbId) WebClient.RequestBodySpec
        .retrieve() WebClient.ResponseSpec
        .bodyToMono(Movie.class);
    mov.subscribe(s -> sendMovieToOrder(s));
    return mov;
}

public void sendMovieToOrder(Movie mov){
    System.out.println(MovieController.userID);
    template.postForObject( url: "http://order-service:8081/order/userId", MovieController.userID, MovieController
    template.postForObject( url: "http://order-service:8081/order/film", mov, Movie.class);
}

```

*Alcuni dei metodi sviluppati in MovieServiceImplementation*

I metodi searchMovie utilizzano un oggetto WebClient per creare le richieste API a OMDb, andando nel frattempo a recuperare la risposta ricevuta convertendola a un oggetto di tipo Movie. Il metodo subscribe consente poi di utilizzare l'oggetto in altri metodi della stessa classe, come ad esempio sendMovieToOrder (che ha lo scopo di inviare il film al microservizio OrderService). Si può inoltre notare l'utilizzo di Mono, componente di WebFlux, che è un publisher in grado di restituire un singolo valore oppure zero; può essere considerato come la controparte “reactive” di restituire un semplice oggetto come accade in Java con Optional.

```

@GetMapping("/user/{id}")
public String saveUserId(@PathVariable String id) {
    if(movieService.controlloId(id) == true) {
        userID = id;
        return id;
    }
    else {
        return "L'id utente non esiste, per favore inserisci un id valido";
    }
}

@GetMapping("/movies/title/{name}")
public Mono<Movie> getMovieByTitle(@PathVariable String name) {
    Movie film= new Movie();
    if (userID == null) {
        film.setMovieTitle("Devi inserire l'ID utente prima di poter utilizzare i servizi");
        Mono<Movie> mov= Mono.just(film);
        return mov;
    }
    String apiKey = env.getProperty("app.api.key");
    return movieService.searchMovieByTitle(apiKey, name);
}

@GetMapping("/movies/id/{imdbId}")
public Mono<Movie> getMovieById(@PathVariable String imdbId) {
    Movie film= new Movie();
}

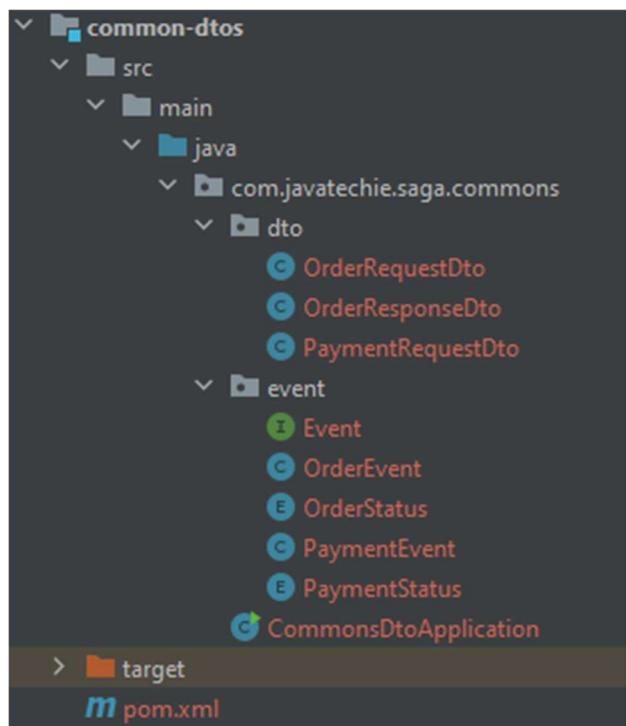
```

*Definizione dei path nella classe MovieController*

In MovieController, attraverso le annotazioni @GetMapping oppure @PostMapping, si definiscono i path su cui è possibile contattare il microservizio; nell’immagine, ad esempio, vengono sviluppati la ricerca del film per titolo (controllando prima che sia stato inserito nel patch /api/user/ un id valido) che consiste nell’andare a richiamare i metodi implementati nella classe Service (già visti nell’immagine precedente). @PathVariable sta ad indicare che, essendo una richiesta di tipo GET, name deve essere di tipo string e deve essere presente quando si effettua la richiesta API.

Infine, nella cartella resource, i file di configurazione “application.properties” e “application.yml” hanno solamente lo scopo di configurare la porta del servizio e l’API key necessaria per usufruire delle API di OMDb.

## 3.2 Common DTO



*Organizzazione dei file nella cartella Common-dtos*

Common DTO (Data Transfer Object) è una cartella del progetto che non implementa alcun microservizio, ma su cui sono presenti diverse classi utili alla creazione degli eventi di tipo Ordine e Pagamento che verranno scambiati tra i due microservizi attraverso Kafka.

In particolare, nella sottocartella dto sono presenti le classi che, insieme agli enum OrderStatus e PaymentStatus, consentono di creare gli eventi tipo Order e di tipo Payment (in quanto sono richiesti nel costruttore).

L'utilizzo di questi elementi DTO è adeguato negli scenari in cui si debbano scambiare dati di continuo tra sottosistemi di un applicativo software (in questo caso tra i microservizi del sistema distribuito).

```
import com.javatechie.saga.commons.dto.OrderRequestDto;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;
import java.util.UUID;
@Data
@NoArgsConstructor
public class OrderEvent implements Event{

    private UUID eventId=UUID.randomUUID();
    private Date eventDate=new Date();
    private OrderRequestDto orderRequestDto;
    private OrderStatus orderStatus;

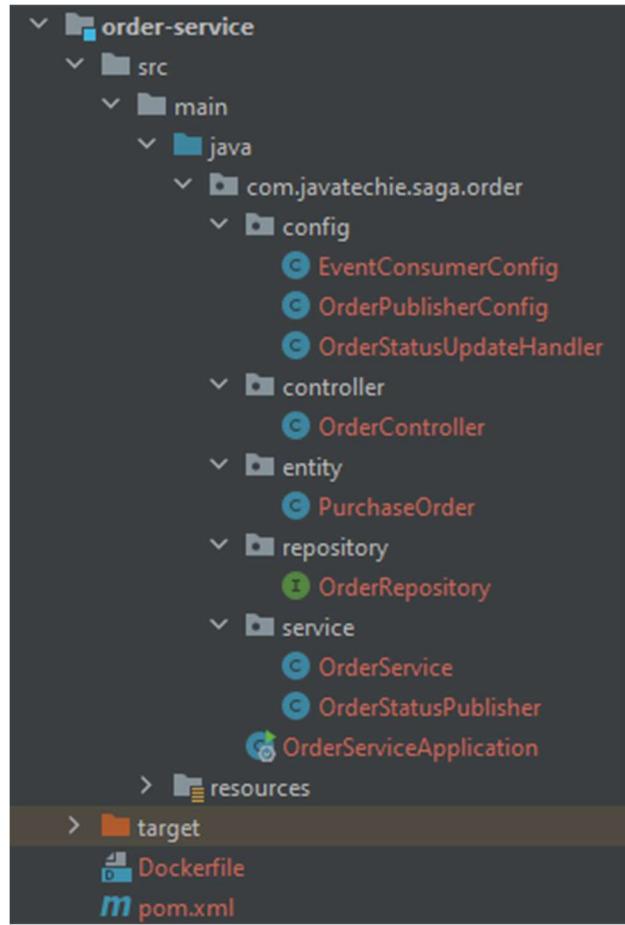
    @Override
    public UUID getEventId() { return eventId; }

    @Override
    public Date getDate() { return eventDate; }

    public OrderEvent(OrderRequestDto orderRequestDto, OrderStatus orderStatus) {
        this.orderRequestDto = orderRequestDto;
        this.orderStatus = orderStatus;
    }
}
```

*Classe OrderEvent*

### 3.3 Order Service



*Organizzazione dei packages e file del servizio degli Ordini*

Anche qui, come in MovieService e PaymentService, la disposizione dei packages consente di distinguere i controller dalle entità e dai “servizi” che offre. Salta subito all’occhio un altro package in più, config, le cui classi al suo interno hanno uno scopo ben preciso: interfacciarsi con Kafka (e i relativi topic) per poter sia produrre messaggi che anche consumarli.

N.B. Il meccanismo di funzionamento di Kafka e i ruoli dei microservizi di Payment e Order all’interno di Saga verranno descritti più in dettaglio in uno dei paragrafi successivi.

OrderStatusUpdateHandler è il gestore dell’aggiornamento dello stato degli ordini: questa classe si occupa di visualizzare gli stati dei pagamenti dei relativi ordini e aggiornare così di conseguenza lo stato dell’ordine. Ha inoltre un metodo che gli consente di convertire un oggetto di tipo PurchaseOrder in un oggetto OrderRequestDTO (vedere paragrafo precedente).

```

@Autowired
private OrderStatusPublisher publisher;

@Transactional
public void updateOrder(int id, Consumer<PurchaseOrder> consumer) {
    repository.findById(id).ifPresent(consumer.andThen(this::updateOrder));
}

private void updateOrder(PurchaseOrder purchaseOrder) {
    boolean isPaymentComplete = PaymentStatus.PAYMENT_COMPLETED.equals(purchaseOrder.getPaymentStatus());
    OrderStatus orderStatus = isPaymentComplete ? OrderStatus.ORDER_COMPLETED : OrderStatus.ORDER_CANCELLED;
    purchaseOrder.setOrderStatus(orderStatus);
    if (!isPaymentComplete) {
        publisher.publishOrderEvent(convertEntityToDto(purchaseOrder), orderStatus);
    }
}

public OrderRequestDto convertEntityToDto(PurchaseOrder purchaseOrder) {
    OrderRequestDto orderRequestDto = new OrderRequestDto();
    orderRequestDto.setOrderId(purchaseOrder.getId());
    orderRequestDto.setUserId(purchaseOrder.getUserId());
    orderRequestDto.setAmount(purchaseOrder.getPrice());
    orderRequestDto.setProductId(purchaseOrder.getProductId());
    return orderRequestDto;
}

```

#### *Metodi della classe OrderStatusUpdateHandler*

Come già preannunciato nel capitolo 2, Order Service utilizza il database MySQL e lo si può facilmente intuire guardando la classe PurchaseOrder e l’interfaccia OrderRepository; nella prima, tramite alcune annotazioni come @Entity, @Table o anche @Id e @GeneratedValue, viene impostata la classe per farla diventare a tutti gli effetti una tabella all’interno del database della piattaforma.

L’interfaccia invece, utilizzando l’estensione JpaRepository, permette di utilizzare JPA (Java Persistence API): un’interfaccia applicativa utile a costruire sistemi ORM. Con l’**Object-Relational Mapping** si è in grado di fare un vero e proprio mapping (un’associazione) tra le tabelle del DB e oggetti Java del dominio applicativo.

Nel package dei servizi sono presenti le classi OrderService e OrderStatusPublisher. In OrderService, sono presenti i metodi sia per creare l’evento (pubblicandolo anche su Kafka tramite la classe OrderStatusPublisher e memorizzando l’ordine sul DB) che per convertire OrderRequestDTO a un oggetto di tipo PurchaseOrder.

```

public void createOrderRequest(Movie mov, String userId) {

    MovieController.userID = "300";
    // System.out.println(MovieController.userID);
    orderRequestDto = new OrderRequestDto();
    String filmID= mov.getImdbID().replace( target: "tt", replacement: "" );
    orderRequestDto.setProductId(Integer.parseInt(filmID));
    orderRequestDto.setUserId((Integer.parseInt(userId)));
    try {
        if (Integer.parseInt(mov.getReleaseYear()) >= 2021) {
            orderRequestDto.setAmount(15);
        } else if (Integer.parseInt(mov.getReleaseYear()) >= 2010) {
            orderRequestDto.setAmount(10);
        } else {
            orderRequestDto.setAmount(5);
        }
    } catch (Exception except) {
        // se l'anno del film dovesse risultare non preciso (es. una serie iniziata nel 2018 e ancora non conclusa)
        // darà come anno di uscita la stringa "2018-" che non può quindi essere convertita in integer
        orderRequestDto.setAmount(5);
    }
}

```

*Metodo per creare un oggetto OrderRequest nella classe OrderService*

Il metodo visualizzato nell’immagine ha lo scopo di prendere l’oggetto Movie inviato dal microservizio MovieService e trasformarlo in OrderRequest, così che possa essere adeguatamente trattato dagli altri metodi descritti qualche riga sopra. Da notare come in questa fase venga anche definito il costo del film (sempre per il discorso di alleggerire alcuni aspetti implementativi, si è deciso di impostare il prezzo del film solamente in base all’anno di uscita).

L’altra classe del package Service (OrderStatusPublisher) consta di un solo metodo il cui scopo è pubblicare l’evento dell’Ordine nel topic di Kafka, così che l’altro servizio possa leggerlo e utilizzarlo. Qui è presente un altro elemento di Reactive, Sinks, che permette il push di segnali di tipo “Reactive Streams” (ovvero Mono o Flux).

```
package com.javatechie.saga.order.service;

import com.javatechie.saga.commons.dto.OrderRequestDto;
import com.javatechie.saga.commons.event.OrderEvent;
import com.javatechie.saga.commons.event.OrderStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Sinks;

@Service
public class OrderStatusPublisher {

    @Autowired
    private Sinks.Many<OrderEvent> orderSinks;

    public void publishOrderEvent(OrderRequestDto orderRequestDto, OrderStatus orderStatus){
        OrderEvent orderEvent=new OrderEvent(orderRequestDto,orderStatus);
        orderSinks.tryEmitNext(orderEvent);
    }
}
```

*Classe OrderStatusPublisher*

L'ultima classe non ancora discussa è quella del Controller, di cui sotto è possibile osservare lo screen. Gli elementi sono simili a quelli descritti sul controllore del microservizio di Movie: notare come l'annotazione @RequestMapping richieda che OGNI path di questo servizio debba iniziare con /order per poter correttamente effettuare una richiesta API a esso.

```

@RestController
@RequestMapping("/order")
public class OrderController {

    private String userId;
    @Autowired
    private OrderService orderService;

    @GetMapping("/create")
    public PurchaseOrder createOrder() { return orderService.createOrder(); }

    @PostMapping("/film")
    public void createRequest(@RequestBody Movie movie) { orderService.createOrderRequest(movie, userId); }

    @PostMapping("/userId")
    public void passingIdUser(@RequestBody String identifier) {
        userId = identifier;
    }

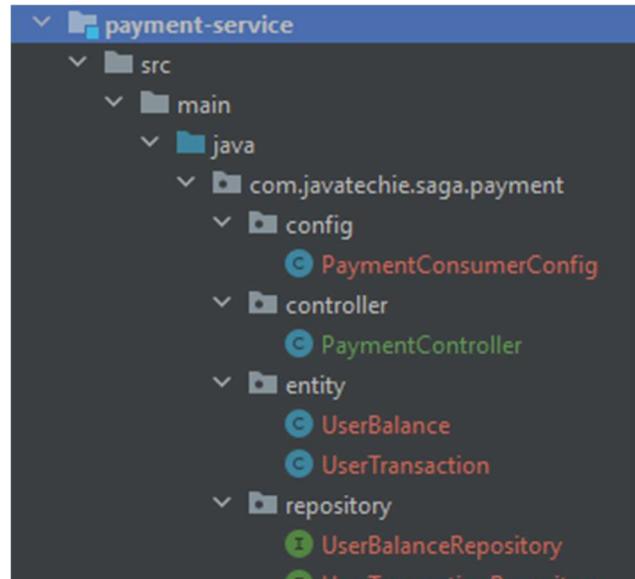
    @GetMapping
    public List<PurchaseOrder> getOrders() { return orderService.getAllOrders(); }
}

```

*Classe OrderController con le implementazioni di cosa succede per ogni path*

Discutendo di alcune configurazioni, impostate nei file application.properties e application.yml (si trovano all'interno della cartella resources), i dettagli importanti da sottolineare sono la definizione (e il relativo binding) del microservizio ai topic Kafka su cui dovrà leggere/scrivere, la porta su cui girerà il servizio e le modalità di accesso al DB MySQL.

## 3.4 Payment Service



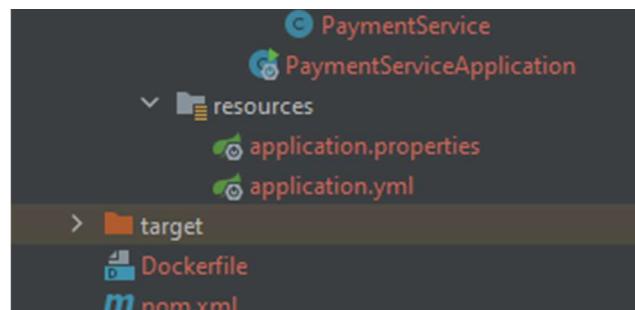
*Organizzazione di file e packages del microservizio relativo ai pagamenti*

Le funzionalità e delle classi di piuttosto simili a Order, per questa sezione ci concentreremo principalmente spicco.

repository, UserBalance e UserTransaction (con le corrispondenti interfacce) rappresentano due classi mappate nel database di cui una tiene conto delle generalità degli utenti con il credito e l'altra delle transazioni di pagamento effettuate dagli utenti; anche qui ricorre di nuovo il concetto di JPA e di ORM già descritto prima.

Il controller invece è piuttosto semplice e consta di un solo path che permette di verificare l'identificativo dell'utente (è l'API principalmente utilizzata da MovieService).

Riguardo le classi di configurazione, PaymentConsumerConfig ha lo scopo di recuperare gli eventi Ordine da Kafka e processare il pagamento appoggiandosi ai metodi implementati da Service.



gli elementi principali Payment Service sono quelli del servizio di questo motivo in si concentrerà sugli elementi più di Rriguardo le entità e le

```

@Configuration
public class PaymentConsumerConfig {

    @Autowired
    private PaymentService paymentService;

    @Bean
    public Function<Flux<OrderEvent>, Flux<PaymentEvent>> paymentProcessor() {
        return orderEventFlux -> orderEventFlux.flatMap(this::processPayment);
    }

    private Mono<PaymentEvent> processPayment(OrderEvent orderEvent) {
        // get the user id
        // check the balance availability
        // if balance sufficient -> Payment completed and deduct amount from DB
        // if payment not sufficient -> cancel order event and update the amount in DB
        if(OrderStatus.ORDER_CREATED.equals(orderEvent.getOrderStatus())){
            return Mono.fromSupplier(()->this.paymentService.newOrderEvent(orderEvent));
        }else{
            return Mono.fromRunnable(()->this.paymentService.cancelOrderEvent(orderEvent));
        }
    }
}

```

*Classe di configurazione di Payment: le righe di codice commentate riassumono come avviene il pagamento*

La classe PaymentService implementa i due metodi più importanti di questo microservizio. Da sottolineare che il tipo di ritorno del primo metodo è un PaymentEvent, così che la classe richiamante questo metodo (quella di configurazione) possa produrre nel topic l'evento che sarà poi utilizzato da OrderService. È inoltre presente un altro metodo, non visualizzabile nello screenshot sottostante, che ha lo scopo di inizializzare il database con alcuni utenti di prova per poter poi effettuare dei test sulla corretta funzionalità del sistema distribuito (il motivo è stato già esplicato nel capitolo 2 dove si introduce il microservizio Payment Service).

```

public PaymentEvent newOrderEvent(OrderEvent orderEvent) {
    OrderRequestDto orderRequestDto = orderEvent.getOrderRequestDto();

    PaymentRequestDto paymentRequestDto = new PaymentRequestDto(orderRequestDto.getOrderId(),
        orderRequestDto.getUserId(), orderRequestDto.getAmount());

    return userBalanceRepository.findById(orderRequestDto.getUserId())
        .filter(ub -> ub.getPrice() > orderRequestDto.getAmount())
        .map(ub -> {
            ub.setPrice(ub.getPrice() - orderRequestDto.getAmount());
            userTransactionRepository.save(new UserTransaction(orderRequestDto.getOrderId(), orderRequestDto.getUserId()));
            return new PaymentEvent(paymentRequestDto, PaymentStatus.PAYMENT_COMPLETED);
        }).orElse(new PaymentEvent(paymentRequestDto, PaymentStatus.PAYMENT_FAILED));
}

@Transactional
public void cancelOrderEvent(OrderEvent orderEvent) {

    userTransactionRepository.findById(orderEvent.getOrderRequestDto().getOrderId())
        .ifPresent(ut->{
            userTransactionRepository.delete(ut);
            userTransactionRepository.findById(ut.getUserId())
                .ifPresent(ub->ub.setAmount(ub.getAmount()+ut.getAmount()));
        });
}

```

*I due importanti metodi presenti nella classe PaymentService*

I file di configurazione (application.properties e application.yml) hanno gli stessi scopi già descritti per l'altro servizio: topic di Kafka, impostazioni di accesso al database, porta su cui girerà tutto il microservizio Payment.

### 3.5 API Gateway

Un piccolo modulo all'interno del progetto è stato destinato per la creazione di un gateway per le API, affinché l'utente possa utilizzare solamente una singola porta per poter effettuare le richieste e utilizzare i servizi senza doversi preoccupare di cambiare porta a seconda del microservizio a cui si sta interfacciando.

Di seguito l'implementazione, piuttosto intuitiva.

```
application:
  name: Gateway-service
cloud:
  gateway:
    routes:
      - id: order-service
        uri: http://order-service:8081
        predicates:
          - Path=/order/**
      - id: payment-service
        uri: http://payment-service:8082
        predicates:
          - Path=/payment/**
      - id: movie-service
        uri: http://movie-service:8083
        predicates:
          - Path=/api/**

server:
  port: 8085
```

Ogni path all'interno della voce "predicates" viene instradato dal gateway verso il corrispettivo url del servizio. Il tutto avviene all'insaputa dell'utente, a cui basta collegarsi sull'url dell'applicazione sulla porta 8085

### 3.6 Kafka e Saga (Choreography Pattern)

Come discusso in precedenza la nostra applicazione sfrutta diversi servizi per il completamento dell'ordine, diventa quindi di fondamentale importanza coordinare i diversi microservizi per mantenere lo stato coerente all'interno dei database.

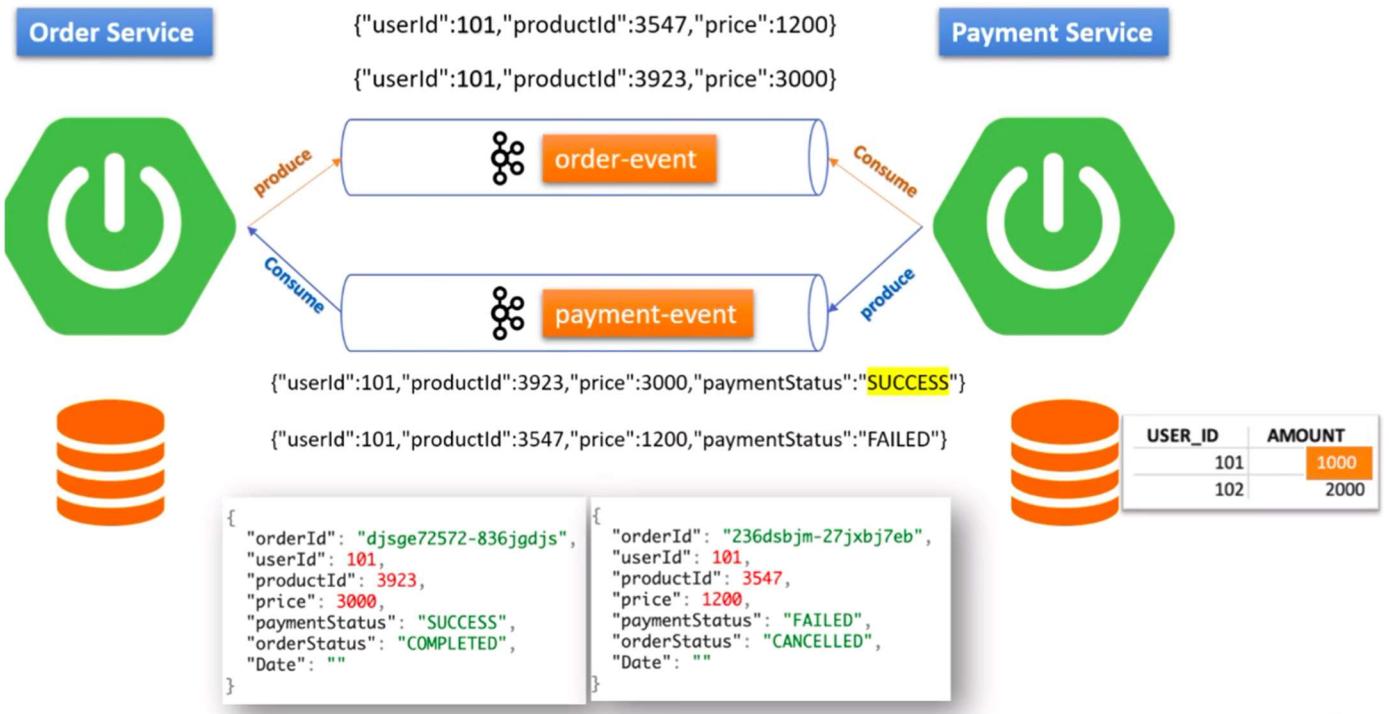
Per fare ciò si utilizza il pattern Saga Choreography che ci consente di implementare un meccanismo di transazione ACID attraverso la trasmissione di eventi. Nello specifico abbiamo scelto l'approccio basato su coreografia in cui ogni entità che effettua una transazione locale pubblica direttamente gli eventi di dominio sulle altre entità/servizi della saga. Questo meccanismo si adattava bene al nostro sistema nel quale gli eventi di dominio vengono scambiati solo tra due entità, in questo caso puntare su un approccio a gestione centralizzata degli eventi come l'orchestrazione avrebbe aumentato la complessità introducendo un ulteriore servizio necessario per l'orchestrazione degli eventi appunto, soluzione che si addice nel caso in cui le entità che devono ricevere gli eventi sono più numerose.

All'interno della nostra applicazione abbiamo un Order Service con un proprio database che ha la responsabilità della gestione degli ordini. Allo stesso modo abbiamo anche il Payment Service che è responsabile della gestione dei pagamenti. Quindi il Order Service riceve la richiesta e per verificare se l'ordine può essere completato, per verifica ciò crea e inoltra un evento (order-event) da pubblicare sul topic kafka per il quale Payment Service sarà subscriber.

Payment Service consumerà l'evento order-event dal topic e verificherà se l'utente ha il saldo disponibile per effettuare il pagamento dell'ordine:

- se l'ordine potrà essere evaso allora Payment Service produrrà un payment-event di risposta per comunicare a Order Service attraverso un topic differente dal precedente (come si vede in figura) impostando il capo paymentStatus a "SUCCESS". Questo evento verrà consumato da Order Service che aggiornerà lo stato dell'order come "COMPLETED".
- Nel caso in cui l'utente non abbia il saldo sufficiente per evadere l'ordine allora il Payment Service produrrà un payment-event di risposta con paymentStatus "FAILED" in modo da informare il consumatore Order Service che il pagamento ha avuto esito negativo.

Payment Service in questo caso si preoccuperà anche di effettuare il rollback delle operazioni che aveva effettuato riportando tutto allo stato precedente in maniera coerente al pagamento fallito.



Struttura di comunicazione tra i servizi Order Service e Payment Service

In altre parole, per effettuare un ordine vengono eseguiti i seguenti passaggi:

1. Order Service riceve la richiesta POST order/create e crea un oggetto Order e inoltra un evento attraverso il metodo publishOrderEvent() sull'oggetto orderStatusPublisher inserendo come attributo l'oggetto orderRequestDto e l'orderStatus che corrisponderà a " ORDER CREATED".

Come si può notare essendo questa una classe metodo che costituisce la transazione del quale si vuole fare il roll-back nel caso in cui qualcosa vada storto si utilizza la notazione **@Transactional**.

Quest'ultima ha lo scopo di definire una serie di metadati di default come: il tipo di propagazione della transazione, il livello di isolamento e le regole di roll-back. Per impostazione predefinita verrà attivato il meccanismo di roll-back nel caso in cui si verifica un'eccezione nel codice.

```

@Transaction
public PurchaseOrder createOrder() {
    PurchaseOrder order = orderRepository.save(convertDtoToEntity(orderRequestDto));
    orderRequestDto.setOrderId(order.getId());
    //produce kafka event with status ORDER_CREATED
    orderStatusPublisher.publishOrderEvent(orderRequestDto, OrderStatus.ORDER_CREATED);
    return order;
}
  
```

metodo di creazione di un nuovo ordine

2. La classe `orderStatusPublisher` sfrutta un oggetto `orderSink` per inoltrare l'evento `orderEvent` attraverso il metodo `tryEmitNext()` questo è un metodo di reactive programming che ci consente di inoltrare sul topic un oggetto `orderEvent` (controllando che sia diverso da null) e generando un segnale di tipo `onNext` nel caso in cui l'evento viene inoltrato correttamente.

I metodi `tryEmit` restituiscono un `Sinks.EmitResult` che altro non è che un enum che porta al fallimento a

In particolare l'oggetto `Sinks.Many<OrderEvent>` utilizza il costrutto `Slnk` della reactive programming il quale consente di pushare eventi attraverso uno stream di tipo `Flux` o un `Mono`, questi sono entrambi meccanismi per la trasmissione asincrona di eventi che si differenziano in quanto:

- `Flux`: consente l'emissione da 0 a N elementi, la trasmissione potrà concludersi con successo nel caso in cui vengono ricevuti tutti gli N eventi in ordine o generare un errore nel caso in cui non si riesce a ricevere uno tra gli N eventi. `Flux` emetterà gli eventi solo quando è presente un subscriber.
- `Mono`: completa con successo la trasmissione quando emette al massimo un elemento/evento e poi termina con un segnale **onComplete** che rappresenta il caso di successo o **onError** in caso di errore .

```
@Service
public class OrderStatusPublisher {

    @Autowired
    private Sinks.Many<OrderEvent> orderSinks;

    public void publishOrderEvent(OrderRequestDto orderRequestDto, OrderStatus orderStatus){
        OrderEvent orderEvent=new OrderEvent(orderRequestDto,orderStatus);
        orderSinks.tryEmitNext(orderEvent);
    }
}
```

*metodo per la pubblicazione di un evento OrderEvent sul topic.*

Nel nostro caso lo stream nel quale si pubblica l'evento `order-event` è definito all'interno della classe `OrderPublisherConfig` ed in particolare viene definito come:

```

@Configuration
public class OrderPublisherConfig {

    @Bean
    public Sinks.Many<OrderEvent> orderSinks(){return Sinks.many().multicast().onBackpressureBuffer();}

    @Bean
    public Supplier<Flux<OrderEvent>> orderSupplier(Sinks.Many<OrderEvent> sinks) { return sinks::asFlux; }
}

```

*configurazione degli stream di dati per la pubblicazione e consumazione degli eventi.*

Il tipo di ritorno Sinks.many().multicast().onBackpressureBuffer() ci indica che il canale di streaming dal quale si recupera il segnale è di tipo multicast che quindi può supportare diversi subscriber agendo in modalità “Backpressure” per ciascun subscriber ovvero è possibile emettere/pushare qualunque quantità di dati e questi verranno memorizzati in un buffer finché il subscriber non pronto a consumarli.

Per impostazione predefinita, se tutti i suoi subscriber cancellano la loro sottoscrizione al topic, viene cancellato anche il suo buffer interno e smette di accettare nuovi subscriber.

3. Analogamente a quanto visto in Order Service che trametteva l'order-event in Payment Service abbiamo la contro parte per catturare/consumare l'evento.

Payment Service si aspetta quindi di ricevere da uno stream Flux di eventi di tipo OrderEvent, in particolare richiamiamo il metodo flatMap(this::processPayment) che consente di mappare gli eventi ricevuti da un Publisher (dato che possiamo ricevere diversi tipi di eventi) e restituendo una nuova sequenza di uno specifico tipo di evento.

flatMap() quindi pusha gli eventi ricevuti in un publisher “figlio” dove è possibile applicare la subscription per un tipo di evento piuttosto che l'intera sequenza eterogenea di eventi che viene ricevuta. A questo punto la gestione dipenderà dello stato di OrderEvent:

- Se lo stato di OrderEvent è “ORDER CREATE” allora si richiama il metodo newOrderEvent(orderEvent) che gestirà il controllo del credito e il pagamento dell'ordine.
- Se lo stato di OrderEvent è diverso da “ORDER CREATE” allora viene richiamato il metodo cancelOrderEvent(orderEvent) per fare il rollback di tutte le operazioni che erano state fatte precedentemente nella gestione di un nuovo ordine che però non è evidentemente andato a buon fine.

```

@Configuration
public class PaymentConsumerConfig {

    @Autowired
    private PaymentService paymentService;

    @Bean
    public Function<Flux<OrderEvent>, Flux<PaymentEvent>> paymentProcessor() {
        return orderEventFlux -> orderEventFlux.flatMap(this::processPayment);
    }

    private Mono<PaymentEvent> processPayment(OrderEvent orderEvent) {
        // get the user id
        // check the balance availability
        // if balance sufficient -> Payment completed and deduct amount from DB
        // if payment not sufficient -> cancel order event and update the amount in DB
        if(OrderStatus.ORDER_CREATED.equals(orderEvent.getOrderStatus())){
            return Mono.fromSupplier(()->this.paymentService.newOrderEvent(orderEvent));
        }else{
            return Mono.fromRunnable(()->this.paymentService.cancelOrderEvent(orderEvent));
        }
    }
}

```

*Consumazione da parte di Payment Service degli eventi Order Event*

4. A questo punto Payment Service attraverso il metodo newOrderEvent() verifica nel proprio database MySQL se l'utente che intende acquistare il film possiede il credito necessario per concludere l'ordine. Tutto questo avviene recuperando le informazioni dell'utente attraverso il suo id ( userBalanceRepository.findById(orderRequestDto.getUserId()) ) che era stato incorporato precedentemente al momento della creazione dell'evento order-event da parte di OrderService.
- Recuperate le informazioni dell'utente compreso il suo credito disponibile al quale viene sottratto il costo del film da pagare ( ub.setPrice( ub.getPrice() - orderRequestDto.getAmount() ).
- Quindi se questa sottrazione che rappresenta il pagamento andrà a buon fine il Payment Service in questa funzione restituirà un oggetto di tipo PaymentEvent con status “PAYMENT COMPLETED”, altrimenti si restituirà un evento PaymentEvent con status “PAYMENT FAILED”.

```

@Transactional
public PaymentEvent newOrderEvent(OrderEvent orderEvent) {
    OrderRequestDto orderRequestDto = orderEvent.getOrderRequestDto();

    PaymentRequestDto paymentRequestDto = new PaymentRequestDto(orderRequestDto.getOrderId(),
        orderRequestDto.getUserId(), orderRequestDto.getAmount());

    return userBalanceRepository.findById(orderRequestDto.getUserId())
        .filter(ub -> ub.getPrice() > orderRequestDto.getAmount())
        .map(ub -> {
            ub.setPrice(ub.getPrice() - orderRequestDto.getAmount());
            userTransactionRepository.save(new UserTransaction(orderRequestDto.getOrderId(), orderRequestDto.getUserId(), orderRequestDto.getAmount()));
            return new PaymentEvent(paymentRequestDto, PaymentStatus.PAYMENT_COMPLETED);
        }).orElse(new PaymentEvent(paymentRequestDto, PaymentStatus.PAYMENT_FAILED));
}

```

*metodo per il pagamento di un ordine*

5. Questi eventi di tipo PaymentEvent verranno inoltrati su uno stream mono in particolare

l'evento viene pushato nello stream Mono.fromSupplier() poiché lo stato con cui era stato ricevuto OrderEvent era "ORDER CREATE". Se lo stato di OrderEvent quando è stato ricevuto fosse stato diverso da "ORDER CREATE" allora avremmo fatto il rollback di tutte le operazioni che erano state fatte precedentemente nella gestione di un nuovo ordine che però non è evidentemente andato a buon fine. In questo caso l'evento viene pushato nello stream Mono.fromRunnable().

6. Payment Service consumando l'evento di Payment Service ne verifica lo stato e se l'evento PaymentEvent ha status "PAYMENT COMPLETED" allora lo status du purchaseOrder viene impostato a "ORDER COMPLETED" e l'ordine è stato completato con successo.

Altimenti se lo status di PaymentEvent è "PAYMENT FAILED" l'oggetto purchaseOrder viene convertito in un OrderEvent e il servizio Order Service riprova a effettuare il pagameto.

```
@Transactional
public void updateOrder(int id, Consumer<PurchaseOrder> consumer) {
    repository.findById(id).ifPresent(consumer.andThen(this::updateOrder));
}

private void updateOrder(PurchaseOrder purchaseOrder) {
    boolean isPaymentComplete = PaymentStatus.PAYMENT_COMPLETED.equals(purchaseOrder.getPaymentStatus());
    OrderStatus orderStatus = isPaymentComplete ? OrderStatus.ORDER_COMPLETED : OrderStatus.ORDER_CANCELLED;
    purchaseOrder.setOrderStatus(orderStatus);
    if (!isPaymentComplete) {
        publisher.publishOrderEvent(convertEntityToDto(purchaseOrder), orderStatus);
    }
}
```

metodo per il completamento dell'ordine dopo la ricezione dell'evento Payment Event.

Entrambi i servizi per poter comunicare attraverso kafka setteranno nei propri file application.yml le specifiche opportune per indicare l'uso dei rispettivi topic dai quali consumeranno e pubblicheranno gli eventi.

```
spring:
  cloud:
    stream:
      function:
        definition : orderSupplier;paymentEventConsumer
      bindings:
        orderSupplier-out-0:
          destination: order-event
        paymentEventConsumer-in-0 :
          destination: payment-event
    application:
      name: Order-service

server:
  port: 8081
```

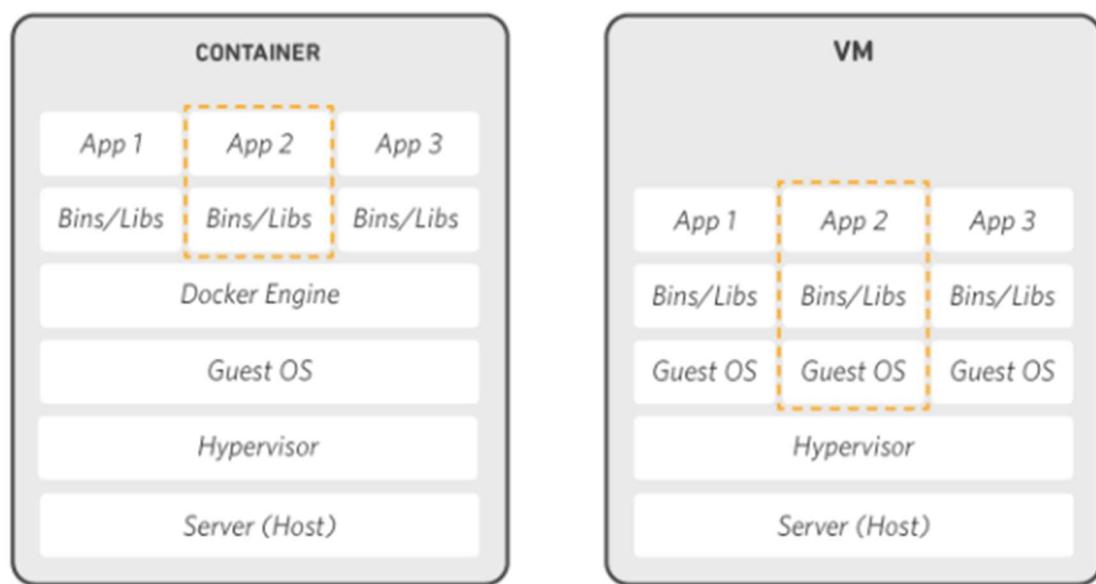
File *application.yml* di Order Service

## 4 Deployment su Docker

Fino a questo momento, tutti i servizi di questo sistema distribuito potevano essere lanciati solamente su un'unica macchina (in locale). Ma in questo modo, l'applicazione di “distribuito” avrebbe avuto ben poco...

Per questo motivo, si è passati al deployment su Docker attraverso i container. Docker è un software che permette di creare, testare e distribuire applicazioni in modo molto rapido; raccoglie il software in unità standardizzate chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime. È possibile, inoltre, distribuire e ricalibrare le risorse per un'applicazione in qualsiasi ambiente, tenendo sempre sotto controllo il codice eseguito.

Il funzionamento è simile a quello della macchina virtuale: così come la VM virtualizza i server hardware (ovvero elimina la necessità di gestirli direttamente), i container virtualizzano il sistema operativo di un server. Docker è installato su ogni server e fornisce semplici comandi con cui creare, avviare o interrompere i container.



Docker è quindi ottimale nel semplificare, come nel caso di questo elaborato, la creazione e l'esecuzione di architetture di microservizi distribuite, la distribuzione di codice distribuzione standardizzate continue, la creazione di sistemi di elaborazione dati altamente scalabili e la creazione di piattaforme completamente gestite per gli sviluppatori.

Per poter “containerizzare” i microservizi del progetto è stato innanzitutto necessaria la creazione di un Dockerfile per ognuno di essi. Questo file è essenziale per poter buildare (costruire) l’immagine di partenza del container che ospiterà quel determinato servizio.

```
▶ FROM openjdk:17
  ADD target/movie-service-exec.jar movie-service.jar
  EXPOSE 8083
  ENTRYPOINT ["java", "-jar", "movie-service.jar"]
```

*Esempio di dockerfile: la struttura è uguale per tutti i servizi*

Prima di poter costruire le immagini è necessario creare i file jar di ogni microservizio: per fare ciò, viene in aiuto il plugin di Maven che tramite un semplice comando consente di farlo. È importante tenere d’occhio le dipendenze (espresso nei file pom.xml) che i servizi hanno tra loro o con plugin/librerie esterne. Una volta che i jar sono tutti pronti, tramite il comando “`docker build /pathdockerfile -t nomeimmagine`” si vanno a creare tutte le immagini necessarie.

```
PS E:\Progetto DSDB\project-dsdb\saga-choreography-pattern> docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
order-service       latest   eb5eede10921  4 days ago   531MB
payment-service     latest   cbbc2c9bedc0  4 days ago   531MB
movie-service       latest   54fe2a575cc0  10 days ago  491MB
api-gateway        latest   c1b71055ea08  10 days ago  503MB
wurstmeister/kafka latest   2dd91ce2efe1  4 weeks ago  508MB
nginx              alpine   6a03ac0c1437  2 months ago  23.2MB
nginx              latest   04661cdce581  2 months ago  141MB
mysql              5.6     f3b364958c23  3 months ago  303MB
gcr.io/k8s-minikube/kicbase v0.0.28 e2a6c047bedd  4 months ago  1.08GB
hello-world        latest   feb5d9fea6a5  4 months ago  13.3kB
alpine             latest   14119a10abf4  5 months ago  5.6MB
wurstmeister/zookeeper latest   3f43f72cb283  3 years ago  510MB
```

Per Kafka, Zookeeper e mysql non c’è bisogno di alcun Dockerfile. In quel caso, le immagini verranno scaricate da Docker Hub, repository online ufficiale di Docker

Il prossimo step è quello di creare i container. Un tool molto efficace nel caso di sistemi distribuiti è **Docker Compose**, che definisce ed esegue applicazioni Docker multi-contenitore: tramite un file Compose (in questo elaborato chiamato [docker-compose](#)) è possibile utilizzare un singolo comando per creare e avviare tutti i servizi della propria configurazione; stessa cosa nel caso in cui si debbano fermare ed eliminare i servizi creati con Compose.

```
gateway-service:
  image: api-gateway
  container_name: gateway-service-container
  ports:
    - '8085:8085'
movie-service:
  image: movie-service
  container_name: movie-service-container
  ports:
    - '8083:8083'
payment-service:
  image: payment-service
  container_name: payment-service-container
  depends_on:
    - kafka
    - mysql-db
  environment:
    MYSQL_DATABASE: 'dsbs'
    MYSQL_USER: 'yins'
    MYSQL_PASSWORD: 'sniv'
    MYSQL_ROOT_PASSWORD: 'root'
  ports:
    - '8082:8082'
```

Alcune linee di codice del docker-compose. È importante sottolineare che "container\_name" è ormai deprecato ma viene utilizzato per comodità in questo progetto per avere i nomi dei container più specifici e facilmente identificabili

Dall'immagine sopra, si può notare come è possibile configurare diverse impostazioni per ognuno dei container, come ad esempio le dipendenze con altri servizi o le variabili d'ambiente. Con la voce ports, inoltre, con una dicitura del tipo '8082:8082' viene fatta una mappatura tra la porta 8082 del container e la porta 8082 dell'host locale (il proprio pc in questo caso).

Con il comando "docker-compose up" il tool, tramite Docker, creerà i container e li farà mandare in esecuzione.

CONTAINER ID	IMAGE	NAMES	COMMAND	CREATED	STATUS	PORTS
3991d8b57f1d	order-service	order-service-containe... order-service	"java -jar order-ser..."	3 minutes ago	Up 3 minutes	0.0.0.0:8081->8081/tcp
942a0dce1dff	payment-service	payment-service-containe... payment-service	"java -jar payment-s..."	3 minutes ago	Up 3 minutes	0.0.0.0:8082->8082/tcp
df4ddf4faf7b	wurstmeister/kafka	kafka	"start-kafka.sh"	3 minutes ago	Up 3 minutes	0.0.0.0:9092->9092/tcp
48cfb4d5ff27	wurstmeister/zookeeper	zookeeper	"/bin/sh -c '/usr/sb..."	3 minutes ago	Up 3 minutes	22/tcp, 2888/tcp, 3888/t... cp, 0.0.0.0:2181->2181/tcp
ac77025ac937	api-gateway	gateway-service-containe... movie-service	"java -jar api-gatew..."	3 minutes ago	Up 3 minutes	0.0.0.0:8085->8085/tcp
8b298759a9b5	movie-service	movie-service-containe... sql_database	"java -jar movie-ser..."	3 minutes ago	Up 3 minutes	0.0.0.0:8083->8083/tcp
3cd1239029a7	mysql:5.6	sql_database	"docker-entrypoint.s..."	3 minutes ago	Up 3 minutes	0.0.0.0:3306->3306/tcp

*La lista dei container creati e attualmente in esecuzione*

Se si desidera avere più dettagli o controllare eventuali errori, si possono stampare su console i logs di ognuno dei container oppure entrare direttamente all'interno del container, ad esempio aprendo un terminale bash. Si noti che montano tutti SO di tipo Linux/Unix.

## 5 Deployment su Kubernetes

In generale, i container sono un buon modo per distribuire ed eseguire le applicazioni. Quello che succede però in uno scenario reale, come ad esempio un sistema distribuito, è quello di avere la necessità di gestire i container che eseguono le applicazioni e garantire che non si verifichino interruzioni dei servizi. Per esempio, se un container si interrompe, è necessario avviare un nuovo container. È possibile per ovviare a questo problema utilizzare un sistema di orchestrazione e gestione dei container come Kubernetes: esso fornisce un framework per far funzionare i sistemi distribuiti in modo resiliente e si occupa della scalabilità, failover e distribuzione delle applicazioni. Sono anche disponibili altre funzionalità molto utili, come:

- Scoperta dei servizi e bilanciamento del carico
- Orchestrazione dello storage e ottimizzazione dei carichi
- Rollout/rollback automatizzati e self-healing

Per poter effettuare il passaggio dei container a Kubernetes, è necessario configurare un file yml (o più di uno se si desidera) descrivendo i deployment e i

servizi da creare nel nodo.

Nel caso di questo progetto, avendo già creato precedentemente un docker-compose, è possibile utilizzare un comodissimo tool chiamato kompose che permette di convertire un file Compose in un file di container orchestrator.

Così facendo si avrà a disposizione un file di configurazione yml di base che, con opportune modifiche qualora necessarie, permetterà la creazione dei pods con i conseguenti deployment e service.

```
kind: Deployment
metadata:
  annotations:
    prometheus.io/scrape: "true"
    meta.helm.sh/release-name: prometheus
    meta.helm.sh/release-namespace: default
    prometheus.io/path: /metrics
    prometheus.io/port: "30003"
    kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f docker-compose.yml
      -o kubem.yml
    kompose.version: 1.21.0 (992df58d8)
  creationTimestamp: null
  labels:
    release: prometheus
    app: prometheus-node-exporter
    #app: rpc-app
    io.kompose.service: order-service
  name: order-service
spec:
  replicas: 1
  selector:
    matchLabels:
      release: prometheus
      app: prometheus-node-exporter
```

*Alcune righe del file kubem.yml, convertito dal docker-compose*

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/scrape: "true"
    meta.helm.sh/release-name: prometheus
    meta.helm.sh/release-namespace: default
    prometheus.io/path: /metrics
    prometheus.io/port: "30002"
    kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f docker-compose.yml
      -o kubem.yml
    kompose.version: 1.21.0 (992df58d8)
  creationTimestamp: null
  labels:
    release: prometheus
    app: prometheus-node-exporter
    #app: rpc-app
    io.kompose.service: order-service
  name: order-service
spec:
  ports:
    - name: "8081"
      prometheus.io/port: "30002"
      kompose.cmd: /snap/kompose/19/kompose-linux-amd64 convert -f docker-compose.yml
        -o kubem.yml
      kompose.version: 1.21.0 (992df58d8)
  creationTimestamp: null
  labels:
    release: prometheus
    app: prometheus-node-exporter
    #app: rpc-app
    io.kompose.service: order-service
  name: order-service
spec:
  ports:
    - name: "8081"
      port: 8081
      targetPort: 8081
      nodePort: 30002
      type: NodePort
  selector:
    release: prometheus
    app: prometheus-node-exporter
    #app: rpc-app
    io.kompose.service: order-service
```

Per poter utilizzare i comandi di Kubernetes si è usufruito dello strumento opensource **minikube**, compatibile su tutti i SO commerciali e che permette di eseguire un cluster a nodo singolo all'interno di una macchina virtuale sulla macchina locale.

Non appena avviato minikube, tramite il comando “kubectl apply -f nomefile.yml” verranno creati tutti gli elementi sul cluster (pods, service, ecc...). È importante sottolineare che, per il progetto, le immagini dei container vengano prelevate da una repository privata di Docker Hub (per ulteriori informazioni vedere il file kube.yml sulla repository GitHub); l'alternativa sarebbe quella di prelevare le immagini direttamente in locale da quelle già create precedentemente su Docker.

```
PS E:\Progetto DSDB\project-dsdb\saga-choreography-pattern> kubectl get po
NAME                               READY   STATUS    RESTARTS   AGE
gateway-service-5b59d7d6dc-m64c8   1/1     Running   4 (2d2h ago)   4d1h
kafka-84b7f9fb49-smtv8           1/1     Running   4 (13m ago)   4d1h
movie-service-7f7559bbf8-bbq9s   1/1     Running   4 (2d2h ago)   4d1h
mysql-db-8b6b58466-wqxnw        1/1     Running   11 (6m45s ago)  4d1h
order-service-6595db75d8-qtxfz   1/1     Running   6 (4m22s ago)  4d1h
payment-service-fc8c9d569-cfbj5  1/1     Running   6                   4d1h
zookeeper-65d9656b77-2d5dd     1/1     Running   4 (13m ago)   4d1h
PS E:\Progetto DSDB\project-dsdb\saga-choreography-pattern> kubectl get service
NAME         TYPE      CLUSTER-IP   EXTERNAL-IP  PORT(S)   AGE
gateway-service  NodePort  10.111.8.20 <none>       8085:30085/TCP  4d1h
kafka-server    ClusterIP  10.99.75.52 <none>       29092/TCP   4d1h
kubernetes     ClusterIP  10.96.0.1  <none>       443/TCP    4d2h
movie-service   NodePort  10.104.18.126 <none>      8083:30083/TCP  4d1h
mysql-db        ClusterIP  10.111.199.184 <none>      3306/TCP   4d1h
order-service   NodePort  10.96.56.217  <none>      8081:30081/TCP  4d1h
payment-service NodePort  10.110.73.11  <none>      8082:30082/TCP  4d1h
zookeeper      ClusterIP  10.111.151.150 <none>      2181/TCP   4d1h
PS E:\Progetto DSDB\project-dsdb\saga-choreography-pattern>
```

*Lista dei pods e dei service della piattaforma*

Anche qui, come su Docker, è possibile stampare i logs per ogni pods ed eventualmente accedere ai container aprendo un terminale bash.

N.B. un pod potrebbe contenere anche più di un container ma, per questo sistema distribuito, è stato deciso di averne uno per pod.

NAMESPACE	NAME	TARGET PORT	URL
default	alertmanager-operated	No node port	
default	gateway-service	8085/8085	<a href="http://192.168.49.2:30000">http://192.168.49.2:30000</a>
default	kafka-server	No node port	
default	kubernetes	No node port	
default	movie-service	8083/8083	<a href="http://192.168.49.2:30001">http://192.168.49.2:30001</a>
default	mysql-db	No node port	
default	order-service	8081/8081	<a href="http://192.168.49.2:30002">http://192.168.49.2:30002</a>
default	payment-service	8082/8082	<a href="http://192.168.49.2:30003">http://192.168.49.2:30003</a>
default	sentinel-service	No node port	

*Lista delle porte e degli URL configurati con NodePort*

Per poter effettuare delle prove e testare la piattaforma, per alcuni servizi è stata utilizzata la configurazione degli IP e delle porte del tipo NodePort. Quest'ultimo infatti consente di esporre il servizio sull'IP di ciascun nodo su una porta statica (NodePort); viene anche creato automaticamente un servizio ClusterIP a cui indirizzerà il servizio NodePort. Questo permetterà di contattare il servizio NodePort, dall'esterno del cluster, richiedendo <NodeIP>:<NodePort>.

In questo modo, ad esempio, tramite l'url <http://192.168.49.2:30000> è possibile con il proprio host locale (completamente esterno al nodo Kubernetes) di accedere al microservizio del gateway api.

## 6 Monitoraggio con Prometheus

Come componente del progetto, la scelta per monitorare la piattaforma creata è stata indirizzata su Prometheus, in particolare il blackbox monitoring. Nel capitolo 2 è stato già introdotto di che tool si tratti e in cosa consiste il monitoraggio di tipo blackbox.

Dopo l'installazione del kube-prometheus-stack su Kubernetes, per poter far rilevare a Prometheus i pods e i relativi container da monitorare è necessario modificare il file yml (quello convertito dal docker-compose, vedi capitolo 5) per far coincidere i labels dei servizi offerti da Prometheus con quelli dei deployment e dei servizi dell'elaborato.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.17.0.10:9100/metrics	DOWN	endpoint="http-metrics" instance="172.17.0.10:9100" job="node-exporter" namespace="default" pod="payment-service-794fdb988c-h2mcx" service="prometheus-prometheus-node-exporter"	4.967s ago	0.628ms	Get "http://172.17.0.10:9100/metrics": dial tcp 17.2.17.0.10:9100: connect: connection refused
http://172.17.0.15:9100/metrics	DOWN	endpoint="http-metrics" instance="172.17.0.15:9100" job="node-exporter" namespace="default" pod="movie-service-8b644ddfd-6stfl" service="prometheus-prometheus-node-exporter"	1.320s ago	0.398ms	Get "http://172.17.0.15:9100/metrics": dial tcp 17.2.17.0.15:9100: connect: connection refused
http://172.17.0.17:9100/metrics	DOWN	endpoint="http-metrics" instance="172.17.0.17:9100" job="node-exporter" namespace="default" pod="gateway-service-7b54dbd447-4qxv9" service="prometheus-prometheus-node-exporter"	6.959s ago	0.460ms	Get "http://172.17.0.17:9100/metrics": dial tcp 17.2.17.0.17:9100: connect: connection refused
http://172.17.0.2:9100/metrics	DOWN	endpoint="http-metrics" instance="172.17.0.2:9100" job="node-exporter" namespace="default" pod="order-service-5cf55789fd-b8q4m" service="prometheus-prometheus-node-exporter"	21.316s ago	0.837ms	Get "http://172.17.0.2:9100/metrics": dial tcp 17.2.17.0.2:9100: connect: connection refused

Nonostante Prometheus riesca a rilevare i servizi da monitorare non si connette (generando un errore visualizzabile nell'immagine), rendendo così impossibile vedere le risorse computazionali utilizzate. Neanche tramite il tool Grafana è possibile visualizzare nulla, presumibilmente per lo stesso motivo.

## 7 Testing e risultati

Quest'ultimo capitolo è incentrato sui risultati ottenuti dalle prove effettuate per testare il funzionamento di questo sistema distribuito.

Si faccia attenzione sul fatto che è stato testato con l'ultimo deployment, ovvero quello con Kubernetes, per cui è stato utilizzato l'endpoints dell'api gateway visualizzabile nell'ultima immagine del capitolo 5 (dove si parla dei Node Port).

Come software per testare le API dei microservizi è stato utilizzato Postman.

GET <http://192.168.49.2:30000/api/movies/title/Avengers> Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (2) Test Results Status: 200 OK Time: 2.01 s Size: 196 B Save Response

Pretty Raw Preview Visualize JSON

```

1 "Title": "Devi inserire l'ID utente prima di poter utilizzare i servizi",
2 "Year": null,
3 "Type": null,
4 "Genre": null,
5 "imdbID": null
6
7
  
```

*Ricerca film senza essersi identificati*

Il gateway (che si trova alla porta 30000 su quell'url) reindirizza correttamente la richiesta al microservizio Movie Service, che attualmente gira in uno dei pod del cluster. Come risposta, viene generato un messaggio d'errore in quanto non è possibile effettuare la ricerca di un film se non ci si è identificati con id utente.

GET <http://192.168.49.2:30000/api/user/9999> Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (2) Test Results Status: 200 OK Time: 127 ms Size: 136 B Save Response

Pretty Raw Preview Visualize Text

```

1 L'id utente non esiste, per favore inserisci un id valido
  
```

*Inserimento di un id non valido*

Anche in questo caso, l'utente si sta interfacciando con Movie Service. Dopo un controllo dell'identificativo, viene verificato che l'id immesso non esiste nel database degli utenti registrati alla piattaforma e per questo motivo viene generato dal servizio una risposta di errore.

GET  http://192.168.49.2:30000/api/user/104

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (2) Test Results Status: 200 OK Time: 658 ms Size: 81 B Save Response ▾

Pretty Raw Preview Visualize Text ▾  □

```
1 104
```

*Inserimento di un id corretto*

Se l'identificativo controllato risulta presente in db, come risposta positiva il servizio ritornerà l'id immesso dall'utente. A questo punto si potrà procedere alla ricerca del film.

GET  http://192.168.49.2:30000/api/movies/title/Spiderman

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (2) Test Results Status: 200 OK Time: 9.43 s Size: 158 B Save Response ▾

Pretty Raw Preview Visualize JSON ▾  □

```
1
2   "Title": "Spiderman",
3   "Year": "1990",
4   "Type": "movie",
5   "Genre": "Short",
6   "imdbID": "tt0100669"
7
```

*Ricerca di un film*

Il film è stato trovato, di conseguenza verranno tornate come risposta alcune informazioni relative al film. È possibile a questo punto cercare ulteriore film oppure acquistare l'ultimo cercato (Spiderman in questo caso).

Body Cookies Headers (2) Test Results

Status: 200 OK Time: 1031 ms Size: 173 B Save Response

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

```

1
2   "id": 1,
3   "userId": 104,
4   "productId": 100669,
5   "price": 5,
6   "orderStatus": "ORDER_CREATED",
7   "paymentStatus": null
8

```

*Acquisto di un film e conseguente creazione di un ordine*

Il gateway adesso ha inoltrato la richiesta a Order Service, che ha permesso di creare l'ordine e iniziare l'acquisto del film. Come risposta vengono ritornate le informazioni principali dell'ordine; paymentStatus è ancora nulla in quanto verrà processato a momenti.

Diamo un'occhiata a Kafka per capire cos'è successo ai messaggi e agli eventi consumati/prodotti:

```

Terminal: Local (5) > Local > Local (2) > + <
bash-5.1# cat 00000000000000000000.log
{"eventId": "01304df6-e8b4-4258-a5a5-29187f478255", "eventDate": "2022-01-31T17:56:02.919+00:00", "orderRequestDto": {"userId": 104, "productId": 100669, "amount": 5, "orderId": 1}, "orderStatus": "ORDER_CREATED", "date": "2022-01-31T17:56:02.919+00:00"}contentType application/json@spring_json_header_types0{"contentType": "java.lang.String"}bash-5.1#

```

*Order Event: i dati corrispondono alla risposta ricevuta dall'utente che aveva utilizzato l'API*

```

bash-5.1# cat 00000000000000000000.log
{"eventId": "464ce981-0297-48fb-baff-258fe98c91d8", "eventDate": "2022-01-31T17:56:09.992+00:00", "paymentRequestDto": {"orderId": 1, "userId": 104, "amount": 5}, "paymentStatus": "PAYMENT_COMPLETED", "date": "2022-01-31T17:56:09.992+00:00"}contentType application/json@spring_json_header_types0{"contentType": "java.lang.String"}bash-5.1#

```

*Payment Event: il pagamento dell'ordine è andato a buon fine. Tramite id dell'ordine e dell'utente si riesce a comprendere di che ordine si tratta*

L'utente con id “104” aveva abbastanza credito da poter acquistare il film di Spiderman e quindi il pagamento è andato a buon fine.

Per poter controllare se lo stato dell'ordine (visto che lo stato di pagamento è

passato a COMPLETED) è stato aggiornato di conseguenza, bisogna controllare direttamente nel database.

```
mysql> select * from purchase_order_tbl;
+----+-----+-----+-----+-----+
| id | order_status | payment_status | price | product_id | user_id |
+----+-----+-----+-----+-----+
| 1 | ORDER_COMPLETED | PAYMENT_COMPLETED | 5 | 100669 | 104 |
+----+-----+-----+-----+-----+
1 row in set (0.01 sec)

mysql>
```

*Lo stato dell'ordine nella tabella presente nel DB è stato correttamente aggiornato*

```
mysql> select * from user_transaction;
+-----+-----+-----+
| order_id | amount | user_id |
+-----+-----+-----+
| 1 | 5 | 104 |
+-----+-----+
1 row in set (0.02 sec)

mysql>
```

*Visto che il pagamento è andato a buon fine, anche la corrispettiva transazione è stata memorizzata in DB in un'altra tabella chiamata user transaction*

Si supponga adesso il caso in cui l'utente con id “105”, che ha a disposizione solo 5 crediti, voglia comprare due film con costo 5 cadauno. Al secondo acquisto, inizialmente l'ordine viene creato, ma cosa succede al pagamento e quindi al conseguente PaymentEvent creato?

```
a.lang.String}swHb0-0002-0002000000000000{"eventId":"b07efab4-4f5f-4586-b649-63a08f8f92d5","eventDate":"2022-01-31T20:23:55.965+00:00","paymentRequestDto":{orderId":2,"userId":105,"amount":5}, "paymentStatus":"PAYMENT_FAILED","date":"2022-01-31T20:23:55.965+00:00"}contentType application/json@spring_json_header_types0{"contentType":"java.lang.String"}bash-5.1#
```

*Il pagamento non va a buon fine: lo stato del pagamento di questo PaymentEvent viene settato su FAILED*

Controlliamo nel database se l'ordine creato (ma con pagamento fallito) ha aggiornato il proprio stato

```
mysql> select * from purchase_order_tbl;
+----+-----+-----+-----+
| id | order_status | payment_status | price | product_id | user_id |
+----+-----+-----+-----+
| 2 | ORDER_CANCELLED | PAYMENT_FAILED | 5 | 100669 | 105 |
|
+----+-----+-----+-----+
2 rows in set (0.02 sec)

mysql>
```

*L'ordine è stato correttamente cancellato, cambiando lo status da "CREATED" a "CANCELLED"*

Il credito residuo dell'utente è stato comunque diminuito? Guardiamo la tabella user\_balance

```
mysql> select * from user_balance;
+-----+-----+
| user_id | price |
+-----+-----+
| 101 | 100 |
| 102 | 30 |
| 103 | 15 |
| 104 | 50 |
| 105 | 5 |
+
5 rows in set (0.03 sec)
```

*L'utente con id 105, non essendo riuscito ad acquistare il film, si troverà col proprio credito invariato*

Per maggiori informazioni, è possibile consultare l'intero codice sviluppato al seguente link:

<https://github.com/PhilipTamb/DSBD>