# Progetto DSBD 2020/2021


**Numero progetto: 2**
**Variante: B**
**Strategia health-check: Ping Ack**
**Autori: Irene Baldacchino (1000012344) e Salvatore Gambadoro (1000008650)**

# 1. Introduzione

Lo scopo del progetto è quello di implementare un microservizio per la gestione degli ordini di un sistema “e-commerce” distribuito.
Per l’implementazione si è previsto l’utilizzo di Spring MVC, Spring Data MongoDB e MongoDB.
Le api che verranno fornite sono:

- POST /orders;
- GET /orders/{id};
- GET /orders/.

Inoltre sono stati realizzati dei container docker che prevedono l’esecuzione del microservizio realizzato attraverso Spring Boot.
Il progetto è costituito da due microservizi: il fake_producer e l’orders_management.
Il primo è stato realizzato per la produzione di messaggi kafka per il testing dell’orders_management.
Il secondo è stato realizzato per la gestione degli ordini.

# 2. Struttura delle classi

## 2.1 Model

Le classi utilizzate per la realizzazione dei modelli per la gestione del database sono le seguenti:

1. **Order:** classe realizzata al fine di definire una struttura relativa alla relazione tra il prodotto, la quantità di elementi e il prezzo di cui si vuole effettuare l’acquisto.

```
@Document(collection = "Order")
public class Order {

    private Integer idProduct;
    private Integer quantity;
    private double price;
```

2. **TotalOrder:** classe realizzata al fine di definire una struttura relativa all'insieme di ordini che devono essere acquistati da un singolo utente.

```java
@Document(collection = "TotalOrder")
public class TotalOrder {

    @Id
    private ObjectId _id;

    private String userId;

    private List<Order> orders;

    private String addressShipment;

    private String addressBilling;

    private double amount;

    private String status;
```

## 2.2 Classi di supporto

1. **StatusMicroservice:** classe realizzata al fine di fornire una risposta riguardante lo stato del servizio e del database per gestire il ping.

```java
public class StatusMicroservice {
    private String serviceStatus;
    private String dbStatus;
```

2. **HttpError:** classe realizzata al fine di fornire una struttura per i messaggi di errore che necessitano di essere inviati sul topic *logging* nel momento in cui viene sollevata l'eccezione.

```java
public class HttpError implements Serializable {

    private LocalDateTime timestamp;
    private String sourceIp;
    private String services;
    private String request;
    private Object error;
```

3. **OrderCompleted:** classe realizzata al fine di fornire una struttura dei messaggi per l'invio sul topic *orderupdates* con chiave *order_completed*.

```
public class OrderCompleted implements Serializable {

    private String orderId;
    private List<Order> products;
    private double total;
    private String shippingAddress;
    private String billingAddress;
    private String userId;
    private HashMap<String, String> extraArgs;
```

4. **OrderPaid:** classe realizzata al fine di fornire una struttura dei messaggi per l'invio sul topic *orderupdates* con chiave *order_paid*.

```
public class OrderPaid  implements Serializable {
    private ObjectId orderId;
    private String userId;
    private double amount;
    private HashMap<String, String> extraArgs;
```

5. **OrderValidation:** classe realizzata al fine di fornire una struttura dei messaggi per l'invio sul topic *orderupdates* con chiave *order_validation*.

```
public class OrderValidation implements Serializable {

    private LocalDateTime timestamp;
    private Integer status;
    private ObjectId orderId;
    private List<String> extraArgs;
```

# 3. Orders Management

## 3.1 Spring Boot

L'utilizzo di Spring Boot ha permesso l'implementazione di diversi componenti con la possibilità di interagire mediante il pattern Model-View-Controller.

### 3.1.1 ControllerOrder

Il ControllerOrder è il componente che applica le funzioni del Controller previste dal pattern MVC con il compito di ricevere le richieste provenienti dal client e di reagire eseguendo operazioni sui dati e quindi sul database.

La prima funzione che analizziamo è l'**addOrder**, la quale permette di aggiungere un ordine al database mongo. Questa è raggiungibile attraverso il path "/orders", alla porta 8088, in cui è in ascolto il container. I parametri attesi dalla funzione sono *totalOrder* e *UserId.* La funzione permetterà il calcolo dell'amount, cioè il costo totale dell'ordine, e di aggiungere un nuovo ordine al repository. Tale funzione produrrà dei messaggi di tipo *OrderCompleted* sui topic *orderupdates* e *pushnotifications.*

```
@PostMapping(path="/")
public @ResponseBody String addOrder(@RequestBody TotalOrder totalOrder,
                                     @RequestHeader("X-User-ID") String userId){

        totalOrder.setUserId(userId);
        double amount = 0.0;
        for (int i = 0; i < totalOrder.getOrders().size(); i++) {
            amount = amount + totalOrder.getOrders().get(i).getPrice();
        }
        totalOrder.setAmount(amount);
        repository.save(totalOrder);
        OrderCompleted orderCompleted = new OrderCompleted(totalOrder.get_id(),
                totalOrder.getOrders(), amount, totalOrder.getAddressShipment(),
                totalOrder.getAddressBilling(), userId);
        kafkaTemplate.send(topicNameOrder, key: "order_completed", new Gson().toJson(orderCompleted));
        kafkaTemplate.send(topicNameNotification, key: "order_completed", new Gson().toJson(orderCompleted));
```

La funzione **getOrderById** è raggiungibile attraverso il path "/orders/{id}" (porta 8088) e permette di visualizzare gli ordini presenti nel database relativi ad un determinato id. L'unico parametro atteso dalla funzione è l'*UserId.* Tale funzione controlla se sono presenti ordini che rispettino i criteri indicati in precedenza, in caso contrario ritornerà un'apposità eccezione *OrderNotFoundException* che permetterà di produrre dei messaggi kafka per i topic *logging* e *http_errors* che comunicheranno il verificarsi di un errore 404.

```
@GetMapping(path="/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public @ResponseBody
ResponseEntity<Object> getOrderById(@PathVariable("id") ObjectId id,
                                    @RequestHeader("X-User-ID") String userId){

        TotalOrder order = repository.findBy_id(id);
        if (order != null && (userId.equals("0") || order.getUserId().equals(userId))) {
            return new ResponseEntity<Object>(order, HttpStatus.OK);
        }else{
            throw new OrderNotFoundException();
        }
}
```

La funzione **getOrderByIdUser** è raggiungibile al path "/orders/" (porta 8088) e permette di gestire la paginazione indicando il numero di pagina e di elementi da visualizzare. I parametri attesi sono: *UserId, per_page (indica il numero di elementi per pagina)* e *page (indica la pagina).*
In assenza dei parametri per_page e page, la funzione tornerà gli ordini dell'utente con relativo *UserId* nel caso in cui l'id è 0 ritornerà gli ordini relativi a tutti gli utenti. Anche in questo caso, in assenza di ordini con i criteri richiesti, la funzione genererà un *OrderNotFoundException.*

```
@GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
public @ResponseBody ResponseEntity<Object> getOrdersbyIdUser(@RequestHeader("X-User-ID") String userId,
                                                              @RequestParam(required = false) Integer per_page,
                                                              @RequestParam(required = false) Integer page){
    List<TotalOrder> orders = repository.findByUserId(userId);
    if(orders.size() != 0) {

        switch (userId) {
            case "0":
                if (page != null && per_page != null) {
                    orders = paginationService.getAllOrders(page, per_page);
                } else {
                    orders = repository.findAll();
                }
                break;
            default:
                if (page != null && per_page != null) {
                    orders = paginationService.getOrdersByUserID(page, per_page, userId);
                } else {
                    //nel caso id != 0 e senza pagination
                    orders = repository.findByUserId(userId);
                }
                break;
        }

        return new ResponseEntity<Object>(orders, HttpStatus.OK);

    }else{
        throw new OrderNotFoundException();
    }

}
```

Al fine di fornire il servizio di paginazione è stato realizzato un Service apposito dove sono presenti due diverse funzioni: *getAllOrders* e *getOrdersByUserID.*
In entrambi casi è stato utilizzato un oggetto di tipo Pageable, il quale permette di generare un elemento con un certo numero di oggetti e pagine, la quale inserita come input alle classiche funzioni di find, implementate nella repository, permette di inserirle all'interno di uno Slice. Lo Slice è molto simile a Page, tranne per il fatto che non fornisce il numero di pagine totali nel database. Aiuta a migliorare le prestazioni quando non è necessario visualizzare il numero totale di pagine nell'interfaccia utente.

```java
@Service
public class PaginationService {
    @Autowired
    TotalOrderRepository repository;

    public List<TotalOrder> getAllOrders(Integer page, Integer per_page)
    {
        Pageable paging = PageRequest.of(page, per_page);

        Slice<TotalOrder> sliceResult = repository.findAll(paging);

        if(sliceResult.hasContent()) {
            return sliceResult.getContent();
        } else {
            return new ArrayList<TotalOrder>();
        }
    }

    public List<TotalOrder> getOrdersByUserID(Integer page, Integer per_page, String userId)
    {
        Pageable paging = PageRequest.of(page, per_page);

        Slice<TotalOrder> sliceResult = repository.findByUserId(userId, paging);

        if(sliceResult.hasContent()) {
            return sliceResult.getContent();
        } else {
            return new ArrayList<TotalOrder>();
        }
    }
}
```

La funzione **ping** è raggiugibile al path “/orders/ping” (porta 8088) e permettte di controllare lo stato del microservizio orders_management e del database mongo.
Per il controllo del database verrà inviato un ping a quest’ultimo. In presenza di risposta, il database mongo_db è pienamente funzionante, in caso contrario, l’assenza di risposta, rileva un problema nel database.

```java
@GetMapping(path="/ping")
public @ResponseBody
ResponseEntity<Object> ping(){
        try {
            Document answer = mongoTemplate.getDb().runCommand(new BasicDBObject("ping", "1"));
            return new ResponseEntity<Object>(new StatusMicroservice( serviceStatus: "up",  dbStatus: "up"), HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<Object>(new StatusMicroservice( serviceStatus: "up",  dbStatus: "down"), HttpStatus.OK);
        }
}
```

*Sono stati effettuati dei test stoppando il container relativo a mongo e rieseguendolo: non è stato rilevato alcun problema di funzionamento.*

### 3.1.2 Mongo

Per la realizzazione del database è stato utilizzato mongoDB, il quale non è altro che un database non relazionale.
Per raggiungere questo scopo, è stata creata un'interfaccia *TotalOrderRepository* che estende la *MongoRepository.*
All'interno dell'interfaccia sono stati inseriti i metodi utilizzati al fine di soddisfare le richieste imposte dalla consegna.

```
public interface TotalOrderRepository extends MongoRepository<TotalOrder, ObjectId>{

    public TotalOrder findBy_id(ObjectId _id);
    public List<TotalOrder> findByUserId(String userId);
    public Slice<TotalOrder> findByUserId(String userId, Pageable paging);

    public TotalOrder findTotalOrderBy_idAndUserIdAndAmount(ObjectId orderId, String userId, double amount);
    public TotalOrder findTotalOrderBy_idAndUserId(ObjectId orderId, String userId);
```

Inoltre al fine di risolvere un'eccezione generata da Spring "*Exception opening socket"* è stato necessaria la creazione di un **ApplicationConfiguration** con lo scopo di risolvere l'eccezione e configurare manualmente la connessione mongo.

```
@EnableMongoRepositories
@SpringBootApplication(
        exclude = {
                MongoAutoConfiguration.class,
                MongoDataAutoConfiguration.class,
                MongoReactiveDataAutoConfiguration.class,
                EmbeddedMongoAutoConfiguration.class
        }
)
@AutoConfigureAfter({EmbeddedMongoAutoConfiguration.class, MongoDataAutoConfiguration.class, MongoAutoConfiguration.class})
public class ApplicationConfiguration extends AbstractMongoClientConfiguration {

    @Value(value = "${MONGO_HOST}")
    private String mongoHost;

    /*@Value(value="${MONGO_USER}")
    private String mongoUser;

    @Value(value="${MONGO_PASS}")
    private String mongoPass;

    @Value(value="${MONGO_AUTH_DB}")
    private String mongoAuthDb;*/

    @Value(value="${MONGO_PORT}")
    private String mongoPort;

    @Value(value="${MONGO_DB_NAME}")
    private String mongoDBName;
```

```
public ApplicationConfiguration() {
}

@Override
protected String getDatabaseName() { return this.mongoDBName; }

@Override
@Bean
public MongoClient mongoClient() {
    /*String s = String.format("mongodb://%s:%s@%s:%s/%s?authSource=%s",
            mongoUser, mongoPass, mongoHost, mongoPort, mongoDBName, mongoAuthDb);*/
    String s = String.format("mongodb://%s:%s/%s",
            mongoHost, mongoPort, mongoDBName);
    return MongoClients.create(s);
}
```

In particolare abbiamo deciso di non includere i parametri di autenticazione in quanto mongo non supporta l'inserimento automatico dell'utente all'interno della tabella admin. Per cui, a fini di sviluppo, si è deciso di ometterli.

### 3.1.3 Gestione delle eccezioni

Come scritto nel paragrafo 3.1.1, nel momento in cui un ordine non è associato ad un utente oppure l'ordine non esiste, è necessario generare un errore con status code 404, corrispondente a *NOT_FOUND.*
In questo caso è stata creata un'eccezione personalizzata *OrderNotFoundException* nella quale mediante l'annotation *@ResponseStatus* è stato impostato lo status code.

```
@ResponseStatus(code = HttpStatus.NOT_FOUND)
public class OrderNotFoundException extends RuntimeException{

}
```

Per la gestione sul comportamento dell'eccezione sollevata, si è utilizzato un *@ExceptionHandler* implementato all'interno del ControllerException. Quest'ultimo presenta l'annotation *@ControllerAdvice,* la quale permette di gestire le eccezioni di tutti i controller presenti nell'applicazione.

Inoltre al suo interno, sono stati gestiti anche altre tipologie di errore nel momento in cui una richiesta HTTP possa fallire:

1. Method Not Allowed (Errore 405);

```
//Error 405
@ExceptionHandler(HttpRequestMethodNotSupportedException.class)
public ResponseEntity<Object> methodArgumentNotValidException(HttpServletRequest request,
                                                               HttpRequestMethodNotSupportedException ex) {
    HttpError error = new HttpError(HttpStatus.METHOD_NOT_ALLOWED, request);
    kafkaTemplate.send( topic: "logging",  key: "http_errors", new Gson().toJson(error));
    return new ResponseEntity<>(HttpStatus.METHOD_NOT_ALLOWED);
}
```

2. Bad Request (Errore 400);

```
//Error 400
@ExceptionHandler({MethodArgumentNotValidException.class, MissingServletRequestParameterException.class,
        TypeMismatchException.class})
public ResponseEntity<Object> methodBadRequest(HttpServletRequest request){
    HttpError error = new HttpError(HttpStatus.BAD_REQUEST, request);
    kafkaTemplate.send( topic: "logging",  key: "http_errors", new Gson().toJson(error));
    return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
}
```

3. Internal Server Error (Errore 500);

```
//Error 500
@ExceptionHandler(ConversionNotSupportedException.class)
public ResponseEntity<Object> methodInternalServerError(HttpServletRequest request,
                                                         ConversionNotSupportedException ex){
    HttpError error = new HttpError(request, ex);
    kafkaTemplate.send( topic: "logging",  key: "http_errors", new Gson().toJson(error));
    return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
}
```

4. Service Unavailable (Errore 503);

```
//Error 503
@ExceptionHandler(ServiceUnavailableException.class)
public ResponseEntity<Object> methodServiceUnavaliable(HttpServletRequest request,
                                                        ServiceUnavailableException ex){
    HttpError error = new HttpError(request, ex);
    kafkaTemplate.send( topic: "logging",  key: "http_errors", new Gson().toJson(error));
    return new ResponseEntity<>(HttpStatus.SERVICE_UNAVAILABLE);
}
```

5. Not Found (Errore 404).

```
//Error 404
@ExceptionHandler(OrderNotFoundException.class)
public ResponseEntity<Object> methodNotFound(HttpServletRequest request, OrderNotFoundException ex){
    HttpError error = new HttpError(HttpStatus.NOT_FOUND, request);
    kafkaTemplate.send( topic: "logging",  key: "http_errors", new Gson().toJson(error));
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

In base al codice di errore, se è di tipo 50x andremo ad inserire nel messaggio da inviare al topic *logging* con chiave *http_errors*, lo stack trace dell'exception generata. Se invece il codice di errore è di tipo 40x, verrà inserito, nel messaggio, lo status code in formato numerico.

## 3.2 Kafka

La gestione di Kafka prevede l'uso di un KafkaConsumer e un KafkaProducer.

### 3.2.1 Kafka Producer

I parametri di configurazione per il KafkaProducer sono:

1. BOOTSTRAP_SERVERS_CONFIG: *"kafka:9092"*;
2. KEY_SERIALIZER_CLASS_CONFIG: *StringSerializer.class;*
3. VALUE_SERIALIZER_CLASS_CONFIG: *StringSerializer.class.*

```java
@Configuration
public class KafkaProducerConfiguration {

        @Value("kafka:9092")
        private String bootstrapServers;

        // Bean di creazione di una mappa con i parametri di configurazione per il client kafka
        @Bean
        public Map<String, Object> producerConfigs() {
            Map<String, Object> props = new HashMap<>();
            props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                    bootstrapServers); // Host e porta sulla quale kafka è in ascolto
            props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                    StringSerializer.class); // Classe di serializzazione da utilizzare per serializzare la chiave
            props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                    StringSerializer.class); // Classe di serializzazione da utilizzare per serializzare i valori
            return props;
        }

        @Bean
        public ProducerFactory<String, String> producerFactory() {
            // implementation for a singleton shared Producer instance.
            return new DefaultKafkaProducerFactory<>(producerConfigs());
        }

        // KafkaTemplate fornisce le utility per inviare messaggi a kafka
        @Bean
        public KafkaTemplate<String, String> kafkaTemplate() {
            return new KafkaTemplate<>(producerFactory());
        }
}
```

Un esempio di come vengono prodotti i messaggi da inviare sui topic è possibile visualizzarlo all'interno del ControllerOrder.

Come prima cosa viene instaziato un *KafkaTemplate* che fornisce dei metodi per inviare messaggi ai topics.
Successivamente all'interno di una funzione, ad esempio *addOrder*, nel monento in cui l'inserimento va a buon fine, mediante l'utilizzo del metodo *send* contenuto nel *KafkaTemplate* è possibile inviare il messaggio sul topic designato (ordersupdates), con la chiave corrispondente (order_completed) ed il contenuto in formato JSON (ordercompleted).

```
kafkaTemplate.send(topicNameOrder,  key: "order_completed", new Gson().toJson(orderCompleted));
```

### 3.2.2 Kafka Consumer

Per quanto riguarda il KafkaConsumer i parametri di configurazione sono:

1. BOOTSTRAP_SERVERS_CONFIG: *"kafka:9092"*;
2. GROUP_ID_CONFIG: *"group-consumer"*;
3. KEY_SERIALIZER_CLASS_CONFIG: *StringSerializer.class;*
4. VALUE_SERIALIZER_CLASS_CONFIG: *StringSerializer.class.*

```
@Configuration
public class KafkaConsumerConfiguration {
    @Value("kafka:9092")
    private String bootstrapServers;

    @Value("group-consumer")
    private String consumerGroup;

    // Bean di creazione di una mappa con i parametri di configurazione per il client kafka
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                bootstrapServers); // Host e porta sulla quale kafka è in ascolto
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
                consumerGroup);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class); // Classe di serializzazione da utilizzare per serializzare la chiave
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class); // Classe di serializzazione da utilizzare per serializzare i valori
        return props;
    }
    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }
    @Bean
    public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String, String>> kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }
```

Il KafkaConsumer fa parte del groupID "group-consumer" e permette di consumare i messaggi in arrivo sul topic *ordersupdates.*

La funzione **listenOrderValidation** prende come paramentri in ingresso un *ConsumerRecord* che rappresenta il messaggio in arrivo.
Per prima cosa viene controllato se il messaggio è diverso da null.
Successivamente verrà eseguito un controllo che permetterà di applicare azioni diverse in base al valore della chiave, che può assumere valore pari a *order_paid* o *order_validation.*

```
@Component
public class KafkaConsumer {

    @Autowired
    TotalOrderRepository repository;

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @KafkaListener(topics = "orderupdates", groupId = "group-consumer")
    public void listenOrderValidation(ConsumerRecord<String, String> record) {
        if(record != null){
            if(record.key().equals("order_validation")) {
                System.out.println(record.key());
                System.out.println(record.value());
                OrderValidation orderValidation = new Gson().fromJson(record.value(), OrderValidation.class);
                System.out.println(orderValidation);
                if (orderValidation.getStatus() != 0) {
                    TotalOrder order = repository.findBy_id(orderValidation.getOrderId());
                    order.setStatus("Abort");
                    repository.save(order);
                }
        }else if(record.key().equals("order_paid")){
            System.out.println(record.key());
            System.out.println(record.value());
            OrderPaid orderPaid = new Gson().fromJson(record.value(), OrderPaid.class);
            TotalOrder order = repository.findTotalOrderBy_idAndUserIdAndAmount(orderPaid.getOrderId(),
                    orderPaid.getUserId(), orderPaid.getAmount());
            if(order != null){
                order.setStatus("Paid");
                repository.save(order);
                kafkaTemplate.send( topic: "pushnotifications",  key: "order_paid", new Gson().toJson(order));
                kafkaTemplate.send( topic: "invoicing",  key: "order_paid", new Gson().toJson(order));
            }else{
                TotalOrder order_error = repository.findTotalOrderBy_idAndUserId(orderPaid.getOrderId(),
                        orderPaid.getUserId());
                if(order_error != null){
                    //esiste un ordine ma c'è l'amount sbagliato.
                    order_error.setStatus("Abort");
                    repository.save(order_error);
                    orderPaid.getExtraArgs().put("error", "WRONG_AMOUNT_PAID");

                }else{
                    orderPaid.getExtraArgs().put("error", "ORDER_NOT_FOUND");
                }
                kafkaTemplate.send( topic: "logging",  key: "order_paid_validation_failure", new Gson().toJson(orderPaid));
            }
        }
```

## 3.3 Docker

Per la gestione dei microservizi è stato utilizzato Docker, il quale permette il loro sviluppo all'interno dei container.

### 3.3.1 DockerFile

Il DockerFile per il microservizio *order_management* è così formato:

```
#builder
FROM maven:3-jdk-8 AS builder
WORKDIR /project
COPY orders_management .
RUN mvn package


FROM openjdk:8-jdk-alpine

#Informa la rete su quale porta si collegherà il container
EXPOSE 8088

WORKDIR /app
#viene copiato l'artefatto costruito dal builder nell'immagine docker
COPY --from=builder /project/target/orders_management-0.0.1-SNAPSHOT.jar ./orders_management.jar
CMD java -jar orders_management.jar
```

In questo caso viene utilizzata un'implementazione multi-stage.
Il file contiene una serie di comandi per la creazione dell'immagine a partire da un contesto di applicazione, ovvero una directory locale.
Questo permetterà di costrituire un'immagine di base, utilizzando il comando FROM.
Il comando RUN serve ad eseguire in fase di building dell'immagine un comando, in questo caso il mvn package.
Viene usato l'EXPOSE per informare la rete sulla porta dove il container si porrà in ascolto e si prosegue con l'esecuzione del container mediante il CMD.

### 3.3.2 Docker-compose

Il docker-compose viene inserito all'interno della cartella complessiva del progetto quindi risulterà essere comune ad entrambi i microservizi.
Vengono creati i seguenti containers:

1. Mongo;

```yaml
mongo:
  image: mongo
  restart: always
  #environment:
   # MONGO_INITDB_ROOT_USERNAME: *mongo-user
   # MONGO_INITDB_ROOT_PASSWORD: *mongo-pass
  ports:
    - 27017:27017
  volumes:
    - mongo-db-data:/data/db
```

2. Orders_management;

```yaml
orders_management:
  build:
    context: .
    dockerfile: orders_management/Dockerfile
  ports:
    - 8088:8080
  restart: always
  environment:
    *mongo-credentials
```

3. Fake_producer;

```yaml
fake_producer:
  build:
    context: .
    dockerfile: fake_producer/Dockerfile
  ports:
    - 8089:8080
  restart: always
```

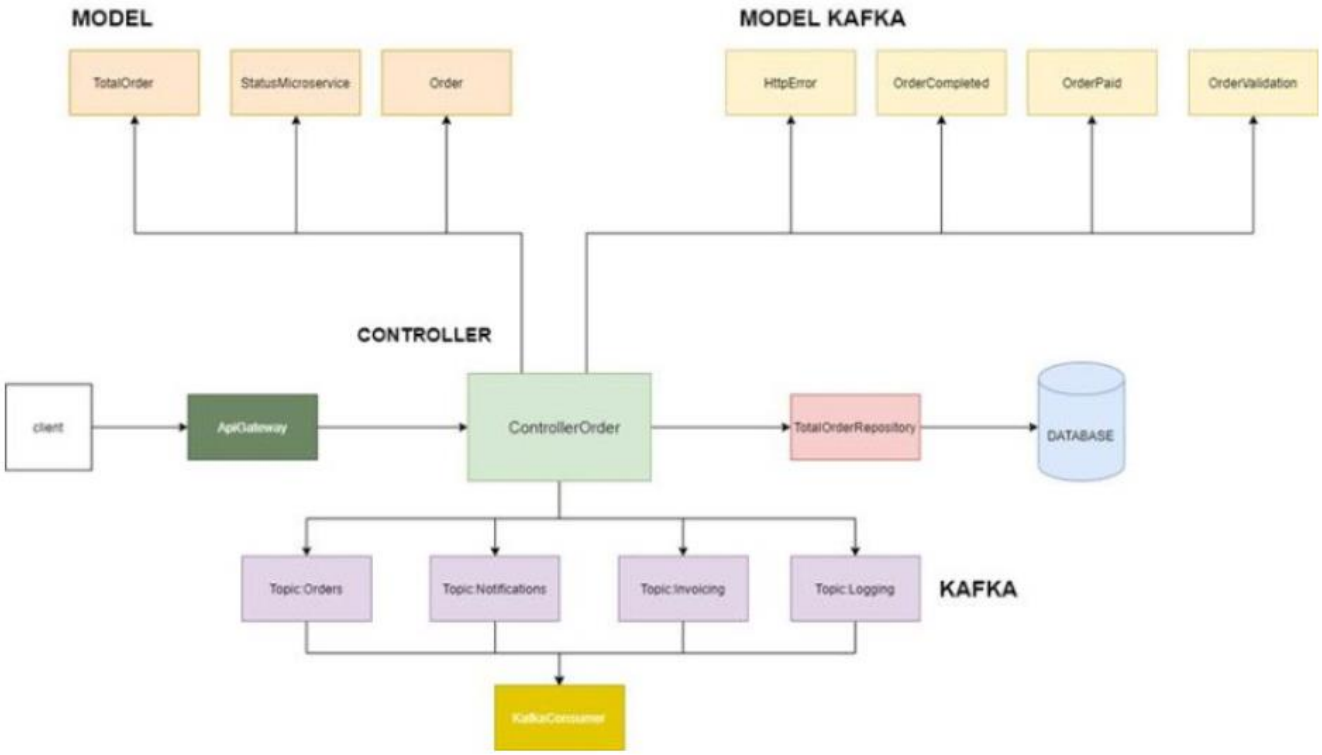
4. Zoo;

```yaml
zoo:
  image: library/zookeeper:3.4.13
  environment:
   ZOO_MY_ID: 1
  restart: always
```

5. Kafka.

```
kafka:
  environment: *kafka-env
  image: wurstmeister/kafka:2.11-2.0.0
  restart: always
  ports:
    - 9092:9092
```

## 3.4 Struttura microservizio order_management



# 4. Fake Producer

## 4.1 Spring Boot

Il Fake Producer è stato strutturato seguendo il pattern Model-View-Controller.

Il FakeProducerController offre le seguenti API:

1. POST "/producer/prova1": è raggiungibile mediante la porta 8089 e permette di inviare un messaggio di tipo *OrderValidation* al topic *orderupdates* con chiave *order_validation.* I parametri attesi dalla funzione sono: *orderId e status.* Il primo è l'identificativo dell'ordine mentre il secondo lo status dell'ordine;

```
@PostMapping(path="/prova_1")
public @ResponseBody
String prova_1(@RequestParam("id") ObjectId orderId, @RequestParam Integer status){
    OrderValidation orderValidation = new OrderValidation(LocalDateTime.now(), status, orderId);
    kafkaTemplate.send(topicNameOrder, key: "order_validation", new Gson().toJson(orderValidation));
    return "Inviato con successo";
}
```

2. POST "/producer/prova2": è raggiungibile mediante la porta 8089 e permette di inviare un messaggio di tipo *OrderPaid* al topic *ordersupdates* con chiave *order_paid.* I parametri attesi dalla funzione sono: *orderId, userId e amount.* Il primo è analogo a quello espresso nel punto 1, il secondo è l'indentificatiovo dell'utente mentre il terzo rappresenta il prezzo totale dell'ordine.

```
@PostMapping(path="/prova_2")
public @ResponseBody
String prova_2(@RequestParam("id") ObjectId orderId, @RequestParam String userId, @RequestParam double amount){
    OrderPaid orderPaid = new OrderPaid(orderId, userId, amount);
    kafkaTemplate.send(topicNameOrder, key: "order_paid", new Gson().toJson(orderPaid));
    return "Inviato con successo";
}
```

## 4.2 DockerFile

Il DockerFile è analogo a quello spiegato nel punto 3.3.1.

```
#builder
FROM maven:3-jdk-8 AS builder
WORKDIR /project
COPY fake_producer .
RUN mvn package

FROM openjdk:8-jdk-alpine
#Informa la rete su quale porta si collegherà il container
EXPOSE 8089
WORKDIR /app
#viene copiato l'artefatto costruito dal builder nell'immagine docker
COPY --from=builder /project/target/fake_producer-0.0.1-SNAPSHOT.jar ./fake_producer.jar
CMD java -jar fake_producer.jar
```

## 4.3 Kafka Producer

Il Kafka producer configuration è analogo a quello spiegato nel punto 3.2.1.

```java
@Configuration
public class KafkaFakeProducerConfiguration {

        @Value("kafka:9092")
        private String bootstrapServers;

        // Bean di creazione di una mappa con i parametri di configurazione per il client kafka
        @Bean
        public Map<String, Object> producerConfigs() {
            Map<String, Object> props = new HashMap<>();
            props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                    bootstrapServers); // Host e porta sulla quale kafka è in ascolto
            props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                    StringSerializer.class); // Classe di serializzazione da utilizzare per serializzare la chiave
            props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                    StringSerializer.class); // Classe di serializzazione da utilizzare per serializzare i valori
            return props;
        }

        @Bean
        public ProducerFactory<String, String> producerFactory() {
            // implementation for a singleton shared Producer instance.
            return new DefaultKafkaProducerFactory<>(producerConfigs());
        }

        // KafkaTemplate fornisce le utility per inviare messaggi a kafka
        @Bean
        public KafkaTemplate<String, String> kafkaTemplate() { return new KafkaTemplate<>(producerFactory()); }
}
```

## 4.4 Struttura microservizio fake_producer