Luca Cabibbo

Architettura dei Sistemi Software

Comunicazione asincrona: Kafka

dispensa asw840

marzo 2021

When you come out of the storm,
you won't be the same person
who walked in.
That's what this storm's all about.

Haruki Murakami (Kafka on the Shore)

- Riferimenti

- Luca Cabibbo. **Architettura del Software**: Strutture e Qualità. Edizioni Efesto, 2021.
  - Capitolo 25, Comunicazione asincrona
- Richardson, C. **Microservices Patterns**: With examples in Java. Manning, 2019.
  - Chapter 3, Interprocess communication in a microservice architecture
- Scott, D.D. **Kafka in Action**, MEAP. Manning, 2020.
- Bejeck Jr., W.P. **Kafka Streams in Action**: Real-time apps and microservices with the Kafka Streams API. Manning, 2018.
  - Chapter 2, Kafka Quickly
- Apache Kafka: A distributed streaming platform
  - https://kafka.apache.org/
- Spring for Apache Kafka
  - https://spring.io/projects/spring-kafka

- Obiettivi e argomenti

- Obiettivi
  - presentare Kafka come esempio di message broker per la comunicazione asincrona
- Argomenti
  - introduzione a Kafka
  - esempi
  - discussione

* Introduzione a Kafka

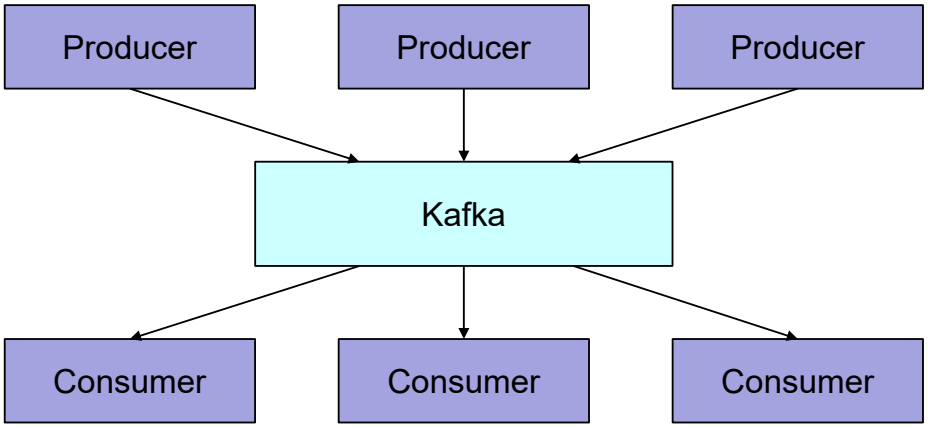
- **Apache Kafka** è una piattaforma distribuita per lo streaming
  - Kafka fornisce tre capacità fondamentali
    - publish-subscribe su stream (flussi) di record – ovvero, è in grado di agire da message broker
    - memorizzazione di stream di record in modo duraturo e tollerante ai guasti
    - elaborazione di stream di record, mentre occorrono (mentre vengono prodotti)
  - qui ci interessa la capacità di Kafka di agire da message broker

Concetti e API fondamentali

- Alcuni concetti di Kafka
  - un *broker* è un server (un nodo) utilizzato per eseguire Kafka
  - un *cluster* è un insieme di broker Kafka – nel seguito, un cluster Kafka sarà chiamato semplicemente *Kafka*
  - un cluster memorizza e distribuisce flussi di *record* (messaggi), organizzati in categorie chiamate *topic* (canali)
  - ogni *record* consiste di una chiave, un valore e un timestamp
- Tra le API fondamentali di Kafka, ce ne interessano due
  - *Producer API* – consente a un servizio o applicazione ("produttore") di pubblicare un flusso di record su uno o più topic
  - *Consumer API* – consente a un servizio o applicazione ("consumatore") di abbonarsi a uno o più topic e di ricevere i corrispondenti flussi di record

Produttori e consumatori

- Produttori e consumatori (e flussi di record)



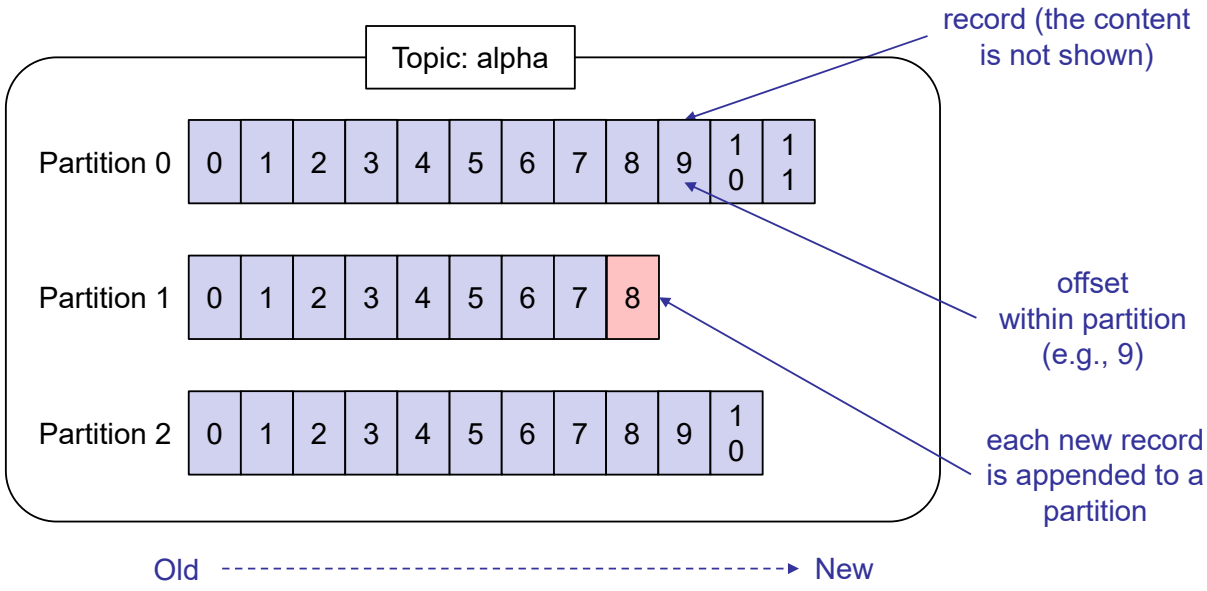
  - i produttori e i consumatori agiscono come client di Kafka
  - la comunicazione tra Kafka e i suoi client avviene mediante un protocollo richiesta/risposta basato su TCP – con implementazioni per molti linguaggi di programmazione
  - in effetti, ogni client Kafka può agire sia da produttore che da consumatore

Topic e partizioni

- Un *topic* è una categoria, identificata da un nome, utilizzata per pubblicare e ricevere record
  - su un topic possono pubblicare messaggi molti produttori (zero, uno o più)
  - a un topic possono abbonarsi, per riceverne i messaggi, molti consumatori (zero, uno o più)
- Per ciascun topic, Kafka mantiene i record pubblicati sul topic in modo partizionato
  - una *partizione* di un topic è una sequenza ordinata e immutabile di messaggi (che memorizza un sottoinsieme disgiunto dei messaggi del topic) – a cui vengono dinamicamente appesi nuovi messaggi
  - ogni record di una partizione ha un id sequenziale, chiamato *offset*, che identifica il record nella partizione

Topic e partizioni

- Un topic, con le sue partizioni e i suoi record



  - nota: i record mostrati in questa figura sono tutti distinti tra loro – all'interno è mostrato l'offset del record, non il contenuto

Produttori

- Ogni produttore, durante la sua esistenza, può pubblicare molti record sui topic che vuole
  - ciascun record pubblicato su un topic viene appeso a una sola delle partizioni del topic
    - la scelta della partizione può avvenire in modalità round-robin oppure utilizzando una qualche funzione di partizionamento semantico

Consumatori e gruppi

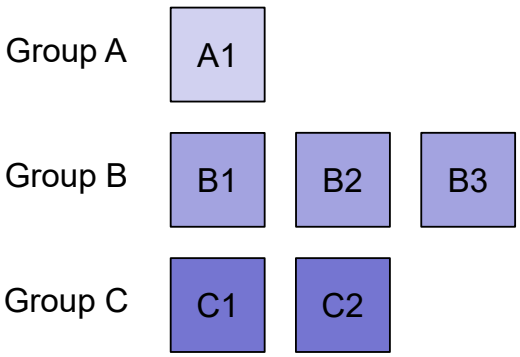
- Ogni consumatore (istanza di consumatore), per ricevere i record di un topic, deve abbonarsi al topic
  - quando un consumatore si abbona a un topic, lo fa specificando il nome del suo *consumer group* (gruppo) – che viene utilizzato da Kafka per la distribuzione dei record del topic ai consumatori abbonati al topic
  - Kafka distribuisce i record del topic consegnando ciascun record pubblicato sul topic a un consumatore (istanza di consumatore) per ciascuno dei gruppi
    - ogni record di un topic viene dunque consegnato a molti consumatori – viene consegnato a tutti i gruppi, e precisamente a un solo consumatore per ciascun gruppo
    - nell'ambito di un gruppo, i diversi record di un topic vengono in genere consegnati a consumatori differenti di quel gruppo (e non tutti a uno stesso consumatore)

Consumatori, gruppi e partizioni

- In pratica
  - Kafka assegna (dinamicamente) zero, una o più partizioni del topic a ciascun consumatore (istanza di consumatore) attivo di un gruppo, e consegna tutti i record di quelle partizioni a quel consumatore
  - se (in un certo periodo di tempo), in un gruppo, il numero dei consumatori attivi nel gruppo è maggiore del numero delle partizioni del topic, allora (in quel periodo di tempo) alcuni consumatori di quel gruppo non riceveranno nessun messaggio dal topic

Consumatori e gruppi

- Kafka distribuisce i record di un topic consegnando ciascun record pubblicato sul topic a un solo consumatore (istanza di consumatore) per gruppo
  - consideriamo alcuni consumatori per un topic suddivisi su più gruppi



  - il consumatore A1 riceverà tutti i messaggi del topic
  - i consumatori B1, B2 e B3 riceveranno, ciascuno, parte dei messaggi del topic – in modo simile, anche i consumatori C1 e C2

Consumatori e gruppi

- Kafka distribuisce i record di un topic consegnando ciascun record pubblicato sul topic a un solo consumatore (istanza di consumatore) per gruppo
  - se tutti i consumatori appartengono a un solo gruppo, il topic si comporta come un canale point-to-point
  - se tutti i consumatori appartengono a gruppi differenti, il topic si comporta come un canale publish-subscribe
  - sono possibili anche modalità di distribuzione diversificate dei record di un topic
  - dunque, il modello di Kafka generalizza i modelli per la distribuzione dei messaggi che vengono in genere utilizzati da altri message broker
    - ad es., i message broker basati su JMS offrono solo canali point-to-point e canali publish-subscribe

Ordine dei record

- Kafka offre le seguenti garanzie sull'ordinamento dei record pubblicati su un topic
  - i record pubblicati da un produttore su un topic verranno appesi alle rispettive partizioni nell'ordine in cui sono stati pubblicati
  - un consumatore (istanza di consumatore) riceverà i record da una partizione di un topic nell'ordine in cui sono memorizzati nella partizione
  - pertanto, se c'è un solo topic, con una sola partizione, un solo produttore e un solo consumatore, i record verranno ricevuti dal consumatore nell'ordine in cui sono stati pubblicati dal produttore
    - tuttavia, questo non è garantito se il topic è composto da più partizioni

* Esempi

- Vengono ora mostrati alcuni esempi di utilizzo di Kafka
  - installazione e configurazione di Kafka
  - un semplice esempio basato su un produttore e un consumatore – vengono anche discussi alcuni esperimenti relativi a questa configurazione
  - una semplice pipeline basata su un produttore, un filtro e un consumatore
  - l'utilizzo di Kafka con riferimento al servizio **`restaurant-service`** per la gestione di un insieme di ristoranti – nell'ambito di un'applicazione **`efood`** per la gestione di un servizio di ordinazione e spedizione a domicilio di pasti da ristoranti, su scala nazionale

- Installazione e configurazione di Kafka

- L'utilizzo di Kafka richiede, in genere, la definizione di un cluster, con molti nodi (almeno uno) – nel cluster deve essere installato sia Kafka che ZooKeeper (usato per il coordinamento dei nodi Kafka)
  - un modo semplice di utilizzare Kafka, soprattutto durante lo sviluppo, è di mandarlo in esecuzione con Docker (Docker Compose)

Installazione e configurazione di Kafka

- Il file docker-compose.yml per Kafka

```
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper
    ports:
      - "2181:2181"
  kafka:
    image: wurstmeister/kafka:latest
    depends_on:
      - "zookeeper"
    ports:
      - "9092:9092"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: 10.11.1.121
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_CREATE_TOPICS: "alpha:4:1,beta:4:1,gamma:4:1"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
```

il nome o l'indirizzo IP dell'host di Docker

crea tre topic (alpha, beta e gamma), ciascuno con 4 partizioni e con replicazione 1

- Un produttore e un consumatore

- Consideriamo ora un semplice esempio, con un produttore e un consumatore, che si scambiano messaggi testuali su un topic (alpha)
  - realizziamo il produttore come un'applicazione Spring Boot, il cui package di base è asw.kafka.simpleproducer
  - realizziamo anche il consumatore come un'altra applicazione Spring Boot, il cui package di base è asw.kafka.simpleconsumer
  - utilizziamo il progetto *Spring for Apache Kafka*, che semplifica l'accesso a Kafka, mediante l'uso di template
    - va utilizzata la dipendenza starter org.springframework.kafka:spring-kafka

Produttore

- Il produttore definisce un semplice servizio per la pubblicazione di messaggi testuali

```
package asw.kafka.simpleproducer.domain;

import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Autowired

@Service
public class SimpleProducerService {

   @Autowired
   private SimpleMessagePublisher simpleMessagePublisher;

   public void publish(String message) {
      simpleMessagePublisher.publish(message);
   }

}
```


Produttore

- Ecco una porzione di esempio del produttore “finale”
  - è qui che va definita la logica applicativa del produttore

```
package asw.kafka.simpleproducer.domain;

import ...;

@Component
public class SimpleProducerRunner implements CommandLineRunner {

   @Autowired
   private SimpleProducerService simpleProducerService;

   public void run(String[] args) {

      String message = ... produce un messaggio message ...
      simpleProducerService.publish(message);

   }

}
```

Qui va definita la logica applicativa del produttore.

Produttore

- Per consentire l'invio di messaggi su Kafka è necessario un outbound adapter (messagepublisher) per Kafka
  - ecco la sua interfaccia

```
package asw.kafka.simpleproducer.domain;

public interface SimpleMessagePublisher {

    public void publish(String message);

}
```


Produttore

- Per consentire l'invio di messaggi su Kafka è necessario un outbound adapter (messagepublisher) per Kafka
  - ecco la sua implementazione

```
package asw.kafka.simpleproducer.messagepublisher;

import asw.kafka.simpleproducer.domain.SimpleMessagePublisher;

import org.springframework.kafka.core.KafkaTemplate;

import ...

@Component
public class SimpleMessagePublisherImpl
                                implements SimpleMessagePublisher {

   ... vedi dopo ...

}
```

in **rosso** indichiamo il codice relativo a Kafka

Produttore

- Per consentire l'invio di messaggi su Kafka è necessario un outbound adapter (messagepublisher) per Kafka
  - ecco la sua implementazione

```
@Value("${asw.kafka.channel.out}")
private String channel;

@Autowired
private KafkaTemplate<String, String> template;

public void publish(String message) {
   template.send(channel, message);
}
```

```
# application.properties
asw.kafka.channel.out=alpha
```

  - **KafkaTemplate** è il template Spring per la pubblicazione di messaggi con Kafka
  - **channel** è il nome del canale (topic) su cui inviare messaggi

Produttore

- Un'occhiata al file application.properties

```
# NON ESEGUIRE COME APPLICAZIONE WEB
spring.main.web-application-type=NONE

# MESSAGING
asw.kafka.channel.out=alpha
asw.kafka.groupid=simple-producer

# KAFKA
spring.kafka.bootstrap-servers=10.11.1.121:9092
spring.kafka.producer.key-serializer=
              org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=
              org.springframework.kafka.support.serializer.JsonSerializer
```

nome del canale su cui inviare messaggi

nome del gruppo del componente (irrilevante in questo caso)

indirizzo IP e porta su cui accedere a Kafka (in questo caso, l'host di Docker)

Consumatore

- Il consumatore definisce un semplice servizio per la ricezione e l'elaborazione di messaggi testuali
  - è qui che va definita la logica applicativa del consumatore – il metodo onMessage deve specificare che cosa fare quando viene ricevuto un messaggio

```
package asw.kafka.simpleconsumer.domain;

import org.springframework.stereotype.Service;

@Service
public class SimpleConsumerService {

    public void onMessage(String message) {

        ... fa qualcosa con message ...

    }

}
```

Qui va definita la logica applicativa del consumatore.

Consumatore

- Per consentire la ricezione di messaggi da Kafka è necessario un inbound adapter (messagelistener) per Kafka
  - ecco la sua implementazione

```
package asw.kafka.simpleconsumer.messagelistener;

import asw.kafka.simpleconsumer.domain.SimpleConsumerService;

import org.springframework.kafka.annotation.KafkaListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;

import ...

@Component
public class SimpleMessageListener {

    ... vedi dopo ...

}
```

Consumatore

- Per consentire la ricezione di messaggi da Kafka è necessario un inbound adapter (messagelistener) per Kafka

```
@Value("${asw.kafka.channel.in}")
private String channel;
@Value("${asw.kafka.groupid}")
private String groupId;

@Autowired
private SimpleConsumerService simpleConsumerService;

@KafkaListener(topics="${asw.kafka.channel.in}",
               groupId="${asw.kafka.groupid}")
public void listen(ConsumerRecord<String, String> record)
                                                  throws Exception {
   String message = record.value();
   simpleConsumerService.onMessage(message);
}
```

```
# application.properties
asw.kafka.channel.in=alpha
asw.kafka.groupid=simple-consumer
```

  - si noti l'invocazione del metodo che definisce la logica applicativa del consumatore

Consumatore

- Per consentire la ricezione di messaggi da Kafka è necessario un inbound adapter (messagelistener) per Kafka
  - l'annotazione **@KafkaListener**, gestita dal framework Spring, svolge quasi tutto il lavoro
    - all'avvio dell'applicazione, Spring richiede a Kafka di abbonare questo consumatore (istanza di consumatore) ai topic elencati (in questo caso, al solo topic alpha) usando il gruppo specificato (in questo caso, simple-consumer) – in corrispondenza, Kafka gli assegna (dinamicamente) zero, una o più partizioni del topic alpha
    - per ogni messaggio pubblicato su una di queste partizioni del topic alpha, Kafka (tramite Spring) consegna il messaggio a questo consumatore (istanza di consumatore), invocando proprio il metodo listen annotato con **@KafkaListener** (consumo in modalità "subscription")

Il metodo listen() e @KafkaListener

- Un aspetto cruciale del consumo dei messaggi è l'invocazione del metodo listen()? Chi lo invoca? Quando?
  - si consideri
    - un topic T (con più partizioni)
    - un produttore P per T
    - due consumatori C1 e C2 (potrebbero essere due istanze di una stessa classe) per T che
      - tramite **@KafkaListener** hanno entrambi dichiarato di essere consumatori per T e di appartenere a uno stesso gruppo G
  - che succede quando P invia un messaggio M su T?

Il metodo listen() e @KafkaListener

- Un aspetto cruciale del consumo dei messaggi è l'invocazione del metodo listen()? Chi lo invoca? Quando?
  - che succede quando P invia un messaggio M su T?
    - in prima battuta, il messaggio M non viene ricevuto né da C1 né da C2
    - piuttosto, la pubblicazione del messaggio M su T viene preso in carico da Kafka
    - è Kafka che sa quali consumatori sono abbonati presso un certo topic – inoltre è sempre Kafka che decide a quale dei consumatori abbonati, per ciascun gruppo, (in questo caso, nel gruppo G, C1 oppure C2) consegnare il messaggio M
    - infine, è Kafka (tramite il framework Spring) che consegna il messaggio invocando il metodo annotato **@KafkaListener** – in questo caso, listen() – delle istanze di consumatori (una per gruppo) che sono state selezionate per il messaggio

Consumatore

- Un'occhiata al file application.properties

```
# NON ESEGUIRE COME APPLICAZIONE WEB
spring.main.web-application-type=NONE

# MESSAGING
asw.kafka.channel.in=alpha
asw.kafka.groupid=simple-consumer

# KAFKA
spring.kafka.bootstrap-servers=10.11.1.121:9092
spring.kafka.consumer.group-id=${asw.kafka.groupid}
# spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.auto-offset-reset=latest
spring.kafka.consumer.key-deserializer=
              org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=
              org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.trusted.packages=*
```

nome del canale da cui ricevere messaggi

nome del gruppo del componente

indirizzo IP e porta su cui accedere a Kafka
(in questo caso, l'host di Docker)

Consumatore

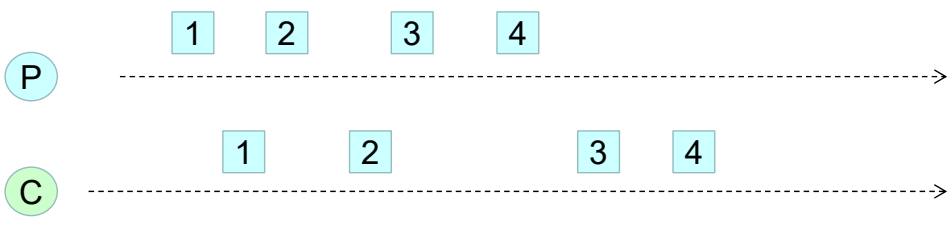
- Nel file application.properties si noti anche la proprietà spring.kafka.consumer.auto-offset-reset
  - questa proprietà consente di regolare gli aspetti temporali della consegna di messaggi a un gruppo di consumatori su un topic
    - il valore latest specifica che i consumatori di quel gruppo debbano ricevere solo i messaggi pubblicati sul topic dal momento del loro abbonamento – escludendo quelli pubblicati prima dell'inizio dell'abbonamento
    - il valore earliest specifica invece che i consumatori di quel gruppo debbano ricevere tutti i messaggi pubblicati sul topic – compresi quelli pubblicati in passato, anche prima del loro abbonamento

Discussione

- Abbiamo mostrato come realizzare la comunicazione asincrona tra una coppia di servizi/applicazioni
  - nel produttore di messaggi va utilizzato un adattatore outbound (messagepublisher) per l'invio di messaggi
    - l'interfaccia di questo adattatore è definita nel dominio del servizio – l'invio di messaggi sarà richiesto probabilmente dai servizi o da altri oggetti del dominio
  - nel consumatore di messaggi va utilizzato un adattatore inbound (messagelistener) per la ricezione di messaggi
    - questo adattatore, alla ricezione di un messaggio, invocherà probabilmente qualche servizio del dominio
  - nell'esempio, non sono state esemplificate le logiche di produzione e di consumo dei messaggi – che costituiscono la "ragion d'essere" per la comunicazione asincrona – ma tuttavia ne sono stati mostrati i "segnaposti"

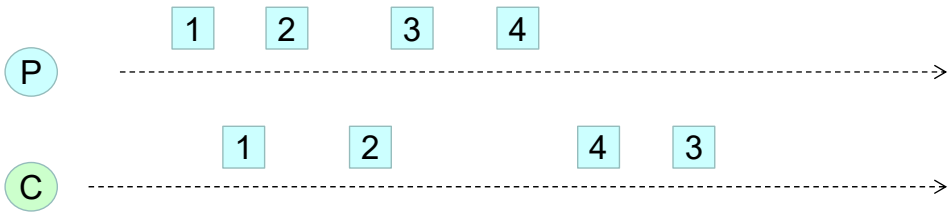
- Alcuni esperimenti con Kafka

- Un topic T (1 partizione), un produttore P per T, un gruppo G di consumatori per T, un consumatore C per T nel gruppo G
  - avvio C, poi avvio P che invia N messaggi
  - conseguenze
    - C riceve N messaggi (nell'ordine in cui sono stati inviati)



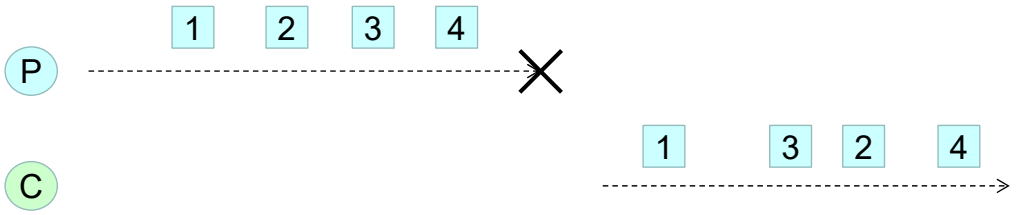
Alcuni esperimenti con Kafka

- Un topic T (più partizioni), un produttore P per T, un gruppo G di consumatori per T, un consumatore C per T nel gruppo G
  - avvio C, poi avvio P che invia N messaggi
  - conseguenze
    - C riceve N messaggi (non necessariamente nell'ordine in cui sono stati inviati)



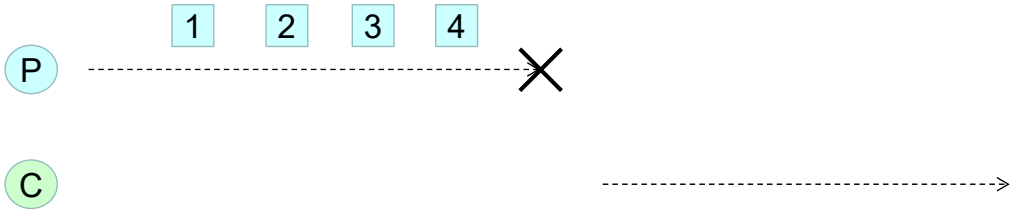
Alcuni esperimenti con Kafka

- Un topic T (più partizioni), un produttore P per T, un gruppo G di consumatori per T, un consumatore C per T nel gruppo G (consegna earliest)
  - C non è inizialmente attivo, avvio P che invia N messaggi e termina, e poi avvio C
  - conseguenze
    - C riceve N messaggi (non necessariamente nell'ordine in cui sono stati inviati)



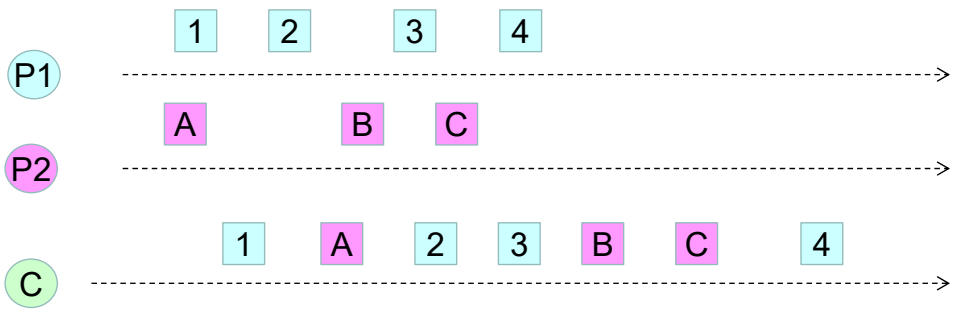
Alcuni esperimenti con Kafka

- Un topic T (più partizioni), un produttore P per T, un gruppo G di consumatori per T, un consumatore C per T nel gruppo G (consegna latest)
  - C non è inizialmente attivo, avvio P che invia N messaggi e poi avvio C
  - conseguenze
    - C non riceve alcun messaggio



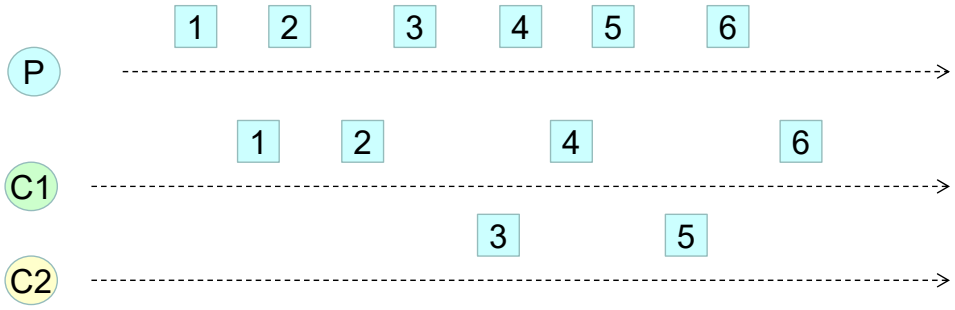
Alcuni esperimenti con Kafka

- Un topic T (più partizioni), più produttori P1 e P2 per T, un gruppo G di consumatori per T, un consumatore C per T nel gruppo G
  - avvio C, poi avvio P1 e P2 che inviano N1 e N2 messaggi ciascuno
  - conseguenze
    - C riceve N1+N2 messaggi



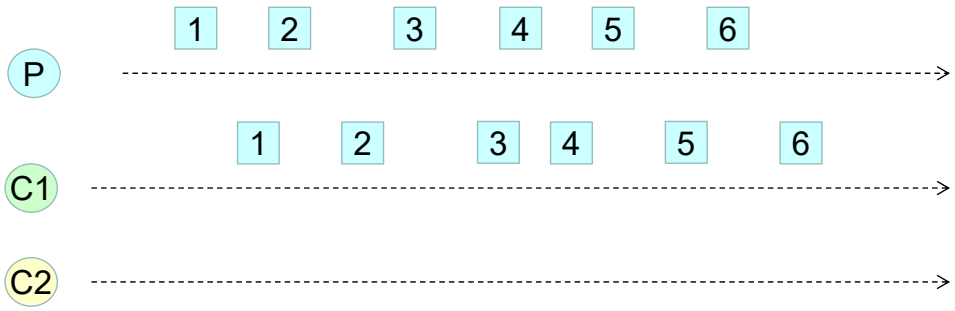
Alcuni esperimenti con Kafka

- Un topic T (più partizioni), un produttore P per T, un gruppo G di consumatori per T, più consumatori C1 e C2 per T nel gruppo G
  - avvio C1 e C2, poi avvio P che invia N messaggi
  - conseguenze
    - il consumatore C1 riceve X messaggi
    - l'altro consumatore C2 riceve gli altri N-X messaggi



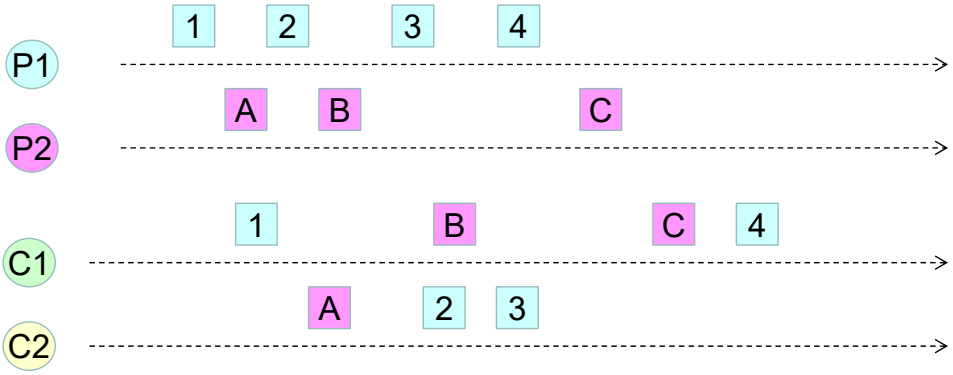
Alcuni esperimenti con Kafka

- Un topic T (1 partizione), un produttore P per T, un gruppo G di consumatori per T, più consumatori C1 e C2 per T nel gruppo G
  - avvio C1 e C2, poi avvio P che invia N messaggi
  - conseguenze
    - il consumatore C1 riceve tutti i messaggi
    - l'altro consumatore C2 non riceve alcun messaggio



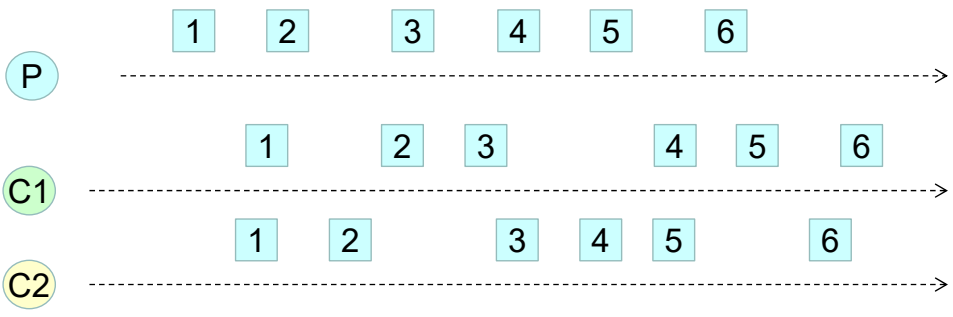
Alcuni esperimenti con Kafka

- Un topic T (più partizioni), più produttori P1 e P2 per T, un gruppo G di consumatori per T, più consumatori C1 e C2 per T in G
  - avvio C1 e C2, poi avvio P1 e P2 che inviano N1 e N2 messaggi ciascuno
  - conseguenze
    - il consumatore C1 riceve X messaggi
    - l'altro consumatore C2 riceve gli altri N1+N2-X messaggi



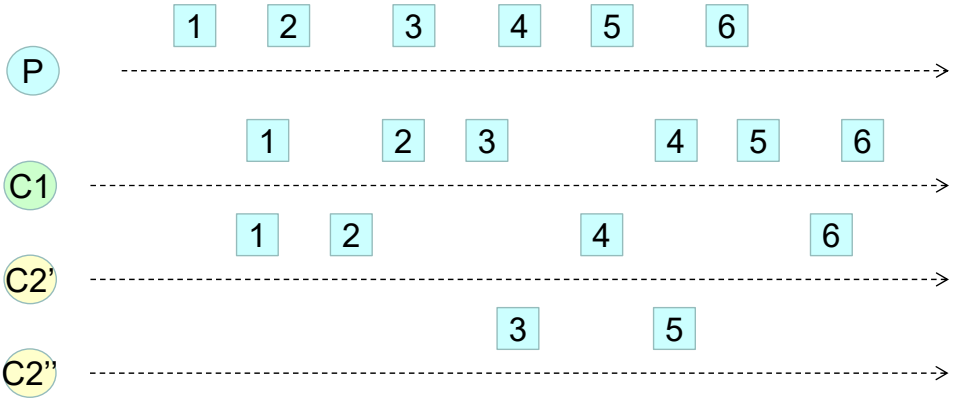
Alcuni esperimenti con Kafka

- Un topic T (più partizioni), un produttore P per T, più gruppi G1 e G2 di consumatori per T, un consumatore C1 per T in G1 e un consumatore C2 per T in G2
  - avvio C1 e C2, poi avvio P che invia N messaggi
  - conseguenze
    - ciascuno dei consumatori C1 e C2 riceve N messaggi



Alcuni esperimenti con Kafka

- Un topic T (più partizioni), un produttore P per T, più gruppi G1 e G2 di consumatori per T, un consumatore C1 per T in G1 e più consumatori C2' e C2'' per T in G2
  - avvio C1, C2' e C2'', poi avvio P che invia N messaggi
  - conseguenze
    - il consumatore C1 riceve N messaggi
    - C2' riceve X messaggi, C2'' riceve gli altri N-X messaggi



- Produttore, filtro e consumatore

- Consideriamo ora una semplice pipeline, con un produttore, un filtro e un consumatore, si scambiano messaggi testuali
  - il produttore invia messaggi sul topic alpha
  - il filtro riceve messaggi dal topic alpha, li elabora, e poi invia messaggi sul topic beta
  - il consumatore riceve messaggi dal topic beta
  - il produttore è come nell'esempio precedente
  - il consumatore è come nell'esempio precedente – ma riceve messaggi da beta anziché da alpha
  - anche il filtro può essere realizzato come un'ulteriore applicazione Spring Boot

Filtro

- Realizziamo il filtro come un'applicazione Spring Boot, il cui package di base è asw.kafka.simplefilter
  - il filtro ha bisogno di un inbound adapter (messagelistener) per Kafka – come il consumatore – per consentire la ricezione di messaggi da Kafka
  - il filtro ha anche bisogno di un outbound adapter (messagepublisher) per Kafka, con la rispettiva interfaccia – come il produttore – per consentire l'invio di messaggi su Kafka
  - inoltre, il dominio deve definire, in un servizio, la logica di elaborazione (filtraggio) dei messaggi
    - per semplicità, supponiamo che il filtro debba inviare un messaggio in corrispondenza a ciascun messaggio ricevuto

Filtro

- Il filtro definisce un servizio per l'elaborazione di messaggi testuali
  - è qui che va definita la logica applicativa del filtro

```
package asw.kafka.simplefilter.domain;

import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Autowired

@Service
public class SimpleFilterService {

   @Autowired
   private SimpleMessagePublisher simpleMessagePublisher;

   public void filter(String inMessage) {
      String outMessage = ... elabora il messaggio message ricevuto ...
      simpleMessagePublisher.publish(outMessage);
   }

}
```

Qui va definita la logica applicativa del filtro.

Filtro

- L'interfaccia dell'outbound adapter (messagepublisher)
  - è come per il produttore – in questo caso cambia il package

```
package asw.kafka.simplefilter.domain;

public interface SimpleMessagePublisher {

    public void publish(String message);

}
```


Filtro

- L'implementazione dell'outbound adapter (messagepublisher)
  - è come per il produttore – in questo caso cambia il package e il file di configurazione application.properties

```
package asw.kafka.simplefilter.messagepublisher;

import asw.kafka.simplefilter.domain.SimpleMessagePublisher;

import org.springframework.kafka.core.KafkaTemplate;

import ...

@Component
public class SimpleMessagePublisherImpl
                                implements SimpleMessagePublisher {

   ... vedi dopo ...

}
```

Filtro

- L'implementazione dell'outbound adapter (messagepublisher)
  - è come per il produttore – in questo caso cambia il package e il file di configurazione application.properties

```
@Value("${asw.kafka.channel.out}")
private String channel;


@Autowired
private KafkaTemplate<String, String> template;

public void publish(String message) {

   template.send(channel, message);

}
```

```
# application.properties
asw.kafka.channel.out=beta
```


Filtro

- L'implementazione dell'inbound adapter (messagelistener)
  - è simile a quella del consumatore – cambia il package, il file di configurazione application.properties, e soprattutto il servizio invocato quando viene ricevuto un messaggio

```
package asw.kafka.simplefilter.messagelistener;

import asw.kafka.simplefilter.domain.SimpleFilterService;

import org.springframework.kafka.annotation.KafkaListener;
import org.apache.kafka.clients.consumer.ConsumerRecord;

import ...

@Component
public class SimpleMessageListener {

   ... vedi dopo ...

}
```

Filtro

- L'implementazione dell'inbound adapter (messagelistener)
  - è simile a quella del consumatore – cambia il package, il file di configurazione application.properties, e soprattutto il servizio invocato quando viene ricevuto un messaggio

```
@Value("${asw.kafka.channel.in}")
private String channel;
@Value("${asw.kafka.groupid}")
private String groupId;

@Autowired
private SimpleFilterService simpleFilterService;

@KafkaListener(topics="${asw.kafka.channel.in}",
               groupId="${asw.kafka.groupid}")
public void listen(ConsumerRecord<String, String> record)
                                                throws Exception {
   String message = record.value();
   simpleFilter.filter(message);
}
```

```
# application.properties
asw.kafka.channel.in=alpha
asw.kafka.groupid=simple-filter
```


Filtro

- Un'occhiata al file application.properties

```
# NON ESEGUIRE COME APPLICAZIONE WEB
spring.main.web-application-type=NONE

# MESSAGING
asw.kafka.channel.in=alpha
asw.kafka.channel.out=beta
asw.kafka.groupid=simple-filter

# KAFKA
spring.kafka.bootstrap-servers=10.11.1.121:9092
spring.kafka.consumer.group-id=${asw.kafka.groupid}
# spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.auto-offset-reset=latest
spring.kafka.consumer.key-deserializer=
             org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=
             org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.trusted.packages=*
spring.kafka.producer.key-serializer=
             org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=
             org.springframework.kafka.support.serializer.JsonSerializer
```

- Il servizio restaurant-service

- Consideriamo ora il servizio **restaurant-service** per la gestione di un insieme di ristoranti – nell'ambito di un'applicazione **efood** per la gestione di un servizio di ordinazione e spedizione a domicilio di pasti da ristoranti, su scala nazionale – già introdotto in una dispensa precedente
  - i ristoranti sono definiti come un'entità JPA **Restaurant** – con attributi **id**, **name** e **location**
  - la gestione dei ristoranti avviene tramite il servizio **RestaurantService**

Servizio restaurant-service e comunicazione asincrona

- Ecco alcune possibili applicazioni della comunicazione asincrona per il servizio **restaurant-service** – nel contesto dell'applicazione **efood**, in cui ci sono diversi servizi applicativi
  - pubblicazione di *eventi* relativi a cambiamenti di stato avvenuti in questo servizio
    - altri servizi potrebbero essere interessati a questi eventi, per poter eseguire delle azioni in corrispondenza al loro verificarsi
  - ascolto di *eventi* pubblicati da altri servizi applicativi
    - questo servizio potrebbe essere interessato a tali eventi, per poter eseguire delle azioni in corrispondenza al loro verificarsi

Servizio restaurant-service e comunicazione asincrona

- Ecco alcune possibili applicazioni della comunicazione asincrona per il servizio **restaurant-service** – nel contesto dell'applicazione **efood**, in cui ci sono diversi servizi applicativi
  - ricezione di *comandi* provenienti da altri servizi applicativi
    - questo servizio potrebbe fornire un'interfaccia asincrona per l'invocazione delle proprie operazioni
  - invio di *comandi* ad altri servizi applicativi
    - per invocare in modo asincrono le operazioni di altri servizi

Pubblicazione di eventi

- Il servizio per la gestione dei ristoranti può pubblicare eventi di dominio mediante un outbound adapter (eventpublisher)
  - questo richiede
    - la definizione degli eventi del dominio dei ristoranti
    - la specifica del canale su cui scambiare gli eventi del dominio dei ristoranti
    - la definizione di un'interfaccia per l'adapter eventpublisher e la sua implementazione
    - l'utilizzo dell'adapter eventpublisher – ad es., da parte del servizio **RestaurantService**

Pubblicazione di eventi

- La definizione degli eventi del dominio dei ristoranti
  - l'interfaccia "radice" degli eventi di dominio per l'applicazione **efood**

```
package asw.efood.common.api.event;

public interface DomainEvent {

}
```


Pubblicazione di eventi

- La definizione degli eventi del dominio dei ristoranti
  - l'evento di dominio RestaurantCreatedEvent

```
package asw.efood.restaurantservice.api.event;

import asw.efood.common.api.event.DomainEvent;

public class RestaurantCreatedEvent implements DomainEvent {

   private Long id;
   private String name;
   private String location;

    … costruttori e metodi get, set e toString …

}
```

Pubblicazione di eventi

- La specifica del canale su cui scambiare gli eventi del dominio dei ristoranti

```
package asw.efood.restaurantservice.api.event;

public class RestaurantServiceEventChannel {

   public static final String channel =
                "restaurant-service-event-channel";

}
```


Pubblicazione di eventi

- La definizione di un'interfaccia per l'adapter eventpublisher e la sua implementazione

```
package asw.efood.restaurantservice.domain;

import asw.efood.common.api.event.DomainEvent;

public interface RestaurantDomainEventPublisher {

   public void publish(DomainEvent event);

}
```

Pubblicazione di eventi

- La definizione di un'interfaccia per l'adapter eventpublisher e la sua implementazione

```
package asw.efood.restaurantservice.eventpublisher;

import …

@Component
public class RestaurantDomainEventPublisherImpl
             implements RestaurantDomainEventPublisher {

   @Autowired
   private KafkaTemplate<String, DomainEvent> template;

   private String channel = RestaurantServiceEventChannel.channel;

   public void publish(DomainEvent event) {

      template.send(channel, event);

   }

}
```


Pubblicazione di eventi

- L'utilizzo dell'adapter eventpublisher – ad es., da parte del servizio **RestaurantService**
  - sono evidenziate in rosso le differenze rispetto alla versione precedente del servizio

```
package asw.efood.restaurantservice.domain;

import …

@Service
@Transactional
public class RestaurantService {

   @Autowired
   private RestaurantRepository restaurantRepository;

   @Autowired
   private RestaurantDomainEventPublisher domainEventPublisher;

   ... vedi dopo ...

}
```

Pubblicazione di eventi

- L'utilizzo dell'adapter eventpublisher – ad es., da parte del servizio **RestaurantService**
  - sono evidenziate in rosso le differenze rispetto alla versione precedente del servizio

```
public Restaurant createRestaurant(String name, String location) {

   Restaurant restaurant = new Restaurant(name, location);
   restaurant = restaurantRepository.save(restaurant);
   DomainEvent event = new RestaurantCreatedEvent(
                                      restaurant.getId(),
                                      restaurant.getName(),
                                      restaurant.getLocation()
                                   );
   domainEventPublisher.publish(event);
   return restaurant;

}
```


Ricezione di comandi

- Il servizio per la gestione dei ristoranti può ricevere comandi per le proprie operazioni mediante un inbound adapter (commandlistener)
  - questo richiede
    - la definizione dei comandi del servizio dei ristoranti
    - la specifica del canale su cui scambiare i comandi del servizio dei ristoranti
    - l'implementazione di un command handler (gestore dei comandi) per il servizio dei ristoranti
    - l'implementazione dell'adapter commandlistener

Ricezione di comandi

- La definizione dei comandi del servizio dei ristoranti
  - l'interfaccia "radice" dei comandi per l'applicazione **efood**

```
package asw.efood.common.api.command;

public interface Command {

}
```


Ricezione di comandi

- La definizione dei comandi del servizio dei ristoranti
  - il comando CreateRestaurantCommand

```
package asw.efood.restaurantservice.api.command;

import asw.efood.common.api.command.Command;

public class CreateRestaurantCommand implements Command {

   private String name;
   private String location;

    … costruttori e metodi get, set e toString …

}
```

Ricezione di comandi

- La specifica del canale su cui scambiare i comandi del servizio dei ristoranti

```
package asw.efood.restaurantservice.api.command;

public class RestaurantServiceCommandChannel {

    public static final String channel =
                "restaurant-service-command-channel";

}
```


Ricezione di comandi

- L'implementazione di un command handler per il servizio dei ristoranti
  - definisce il metodo pubblico onCommand per la gestione dei comandi

```
package asw.efood.restaurantservice.domain;

import asw.efood.common.api.command.Command;
import asw.efood.restaurantservice.api.command.*;

import …

@Service
public class RestaurantCommandHandler {

    @Autowired
    private RestaurantService restaurantService;

    public void onCommand(Command command) {
        ... vedi dopo ...
    }

}
```

Ricezione di comandi

- L'implementazione di un command handler per il servizio dei ristoranti
  - definisce il metodo pubblico onCommand per la gestione dei comandi

```
public void onCommand(Command command) {
   if (command.getClass().equals(CreateRestaurantCommand.class)) {
      CreateRestaurantCommand c = (CreateRestaurantCommand) command;
      this.createRestaurant(c);
   } else if (command.getClass().equals(AnotherOpCommand.class)) {
      AnotherOpCommand c = (AnotherOpCommand) command;
      this.anotherOp(c);
   } else {
      ... unknown command ...
   }
}
```


Ricezione di comandi

- L'implementazione di un command handler per il servizio dei ristoranti
  - inoltre, definisce un metodo di supporto per ciascuno dei comandi

```
private void createRestaurant(CreateRestaurantCommand command) {
   restaurantService.createRestaurant(
                           command.getName(),
                           command.getLocation()
                         );
}
```

Ricezione di comandi

- L'implementazione dell'adapter commandlistener
  - alla ricezione di un messaggio per un comando, invoca il command handler

```
package asw.efood.restaurantservice.commandlistener;

import …;

@Component
public class RestaurantCommandListener {

   @Autowired
   private RestaurantCommandHandler restaurantCommandHandler;

   @KafkaListener(topics = RestaurantServiceCommandChannel.channel)
   public void listen(ConsumerRecord<String, Command> record)
                                                  throws Exception {

      Command command = record.value();

      restaurantCommandHandler.onCommand(command);
   }

}
```


Configurazione

- Un'occhiata al file application.properties del servizio dei ristoranti – limitatamente alla configurazione di Kafka

```
# KAFKA
spring.kafka.bootstrap-servers=10.11.1.121:9092
spring.kafka.consumer.group-id=${spring.application.name}
spring.kafka.consumer.auto-offset-reset=earliest

spring.kafka.producer.key-serializer=
              org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=
              org.springframework.kafka.support.serializer.JsonSerializer

spring.kafka.consumer.key-deserializer=
              org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=
              org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.trusted.packages=*
```

  - si noti l'uso di JSON come formato per l'interscambio dei messaggi – il framework Spring effettua automaticamente le conversioni da e verso le classi Java per i comandi e gli eventi

Esercizi

- Con riferimento al servizio per la gestione dei ristoranti
  - realizzare un consumatore per gli eventi di dominio dei ristoranti
  - realizzare un produttore di comandi
- In un precedente esercizio è stato richiesto di estendere il servizio per la gestione dei ristoranti con la gestione dei menu dei ristoranti
  - qui si chiede di definire e realizzare nuovi comandi (ad es., CreateRestaurantMenuCommand e ReviseRestaurantMenuCommand) e nuovi eventi di dominio (ad es., RestaurantMenuCreatedOrRevisedEvent) relativi alla gestione dei menu dei ristoranti

- Discussione

- Ecco alcune considerazioni sull'utilizzo di Kafka
  - consente di inviare e ricevere messaggi (record) tramite canali (topic) – con un modello basato su gruppi che generalizza quello dei canali point-to-point e publish-subscribe
  - con riferimento all'architettura esagonale
    - l'invio di messaggi, da parte di un produttore, richiede la definizione di un outbound adapter
    - la ricezione di messaggi, da parte di un consumatore, richiede la definizione di un inbound adapter
  - un componente può agire sia da produttore che da consumatore di messaggi
  - i messaggi scambiati possono essere documenti, eventi di dominio e comandi – ciascuna tipologia di essi richiederà un canale specifico

Discussione

- Ecco alcune considerazioni sull'utilizzo di Kafka
  - i componenti produttori e consumatori agiscono da client nei confronti di Kafka
    - i produttori e consumatori comunicano con Kafka, come client, in modo sincrono
    - tuttavia, i produttori e consumatori comunicano tra loro in modo asincrono
  - i messaggi vengono scambiati con Kafka tramite un protocollo specifico per Kafka
  - i messaggi sono "opachi" per Kafka (che ne ignora il contenuto) – e possono essere scambiati tra i client Kafka nel formato di interscambio preferito – negli esempi precedenti, in JSON, ma sono possibili anche XML e Protocol Buffers

* Discussione

- In questa dispensa abbiamo presentato Apache Kafka come piattaforma per la comunicazione asincrona
  - Kafka consente di agire da message broker – ovvero supporta il pattern publish-subscribe per la trasmissione di stream di record (flussi di messaggi)
  - i canali (chiamati topic e organizzati in partizioni) consentono di pubblicare e di ricevere record
  - i produttori possono pubblicare flussi di messaggi (record) su uno o più topic
  - i consumatori possono ricevere flussi di messaggi (record) da uno o più topic
  - i consumatori di messaggi sono organizzati in gruppi – utilizzati per la distribuzione dei messaggi ai consumatori – secondo un modello che generalizza quello dei canali point-to-point e publish-subscribe