Luca Cabibbo

Architettura dei Sistemi Software

Comunicazione asincrona: Kafka

dispensa asw840

ottobre 2023

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

- 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., Gamov, V., Klein, D. **Kafka in Action**, Manning, 2022.
- 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 eventi – ovvero, è in grado di agire da message broker
    - memorizzazione di stream di eventi in modo duraturo e tollerante ai guasti
    - elaborazione di stream di eventi, mentre questi stream vengono prodotti
  - qui siamo interessati alla capacità di Kafka di agire da message broker

Concetti fondamentali

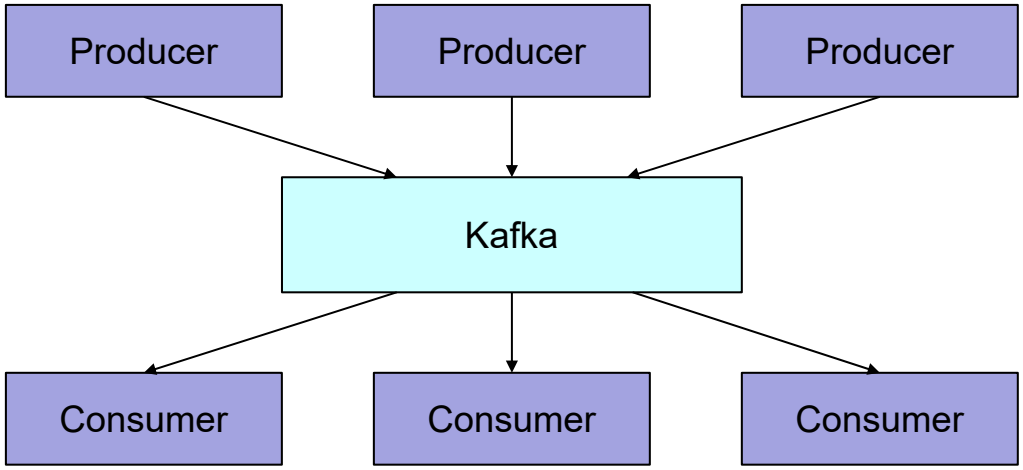
- Alcuni concetti di Kafka
  - un *broker* è un server (un nodo) utilizzato per eseguire Kafka
  - un *cluster* è un insieme di broker Kafka
    - nel seguito, un cluster Kafka sarà chiamato semplicemente *Kafka*
  - un cluster consente di memorizzare e distribuire flussi di *eventi* (messaggi), organizzati in categorie chiamate *topic* (canali)
    - si noti la terminologia specifica di Kafka
      - messaggio → evento canale → topic
  - ogni *evento* consiste di una chiave, un valore e un timestamp
    - la chiave è opzionale, il timestamp è assegnato da Kafka
  - nota: gli eventi di Kafka (in precedenza chiamati record) sono messaggi, e il loro uso non va limitato alla notifica di "eventi di dominio"

API fondamentali

- Tra le API fondamentali di Kafka, ce ne interessano due
  - *Producer API* – consente a un componente, servizio o applicazione ("produttore") di pubblicare un flusso di eventi su uno o più topic
  - *Consumer API* – consente a un componente, servizio o applicazione ("consumatore") di abbonarsi a uno o più topic e di ricevere i corrispondenti flussi di eventi

Produttori e consumatori

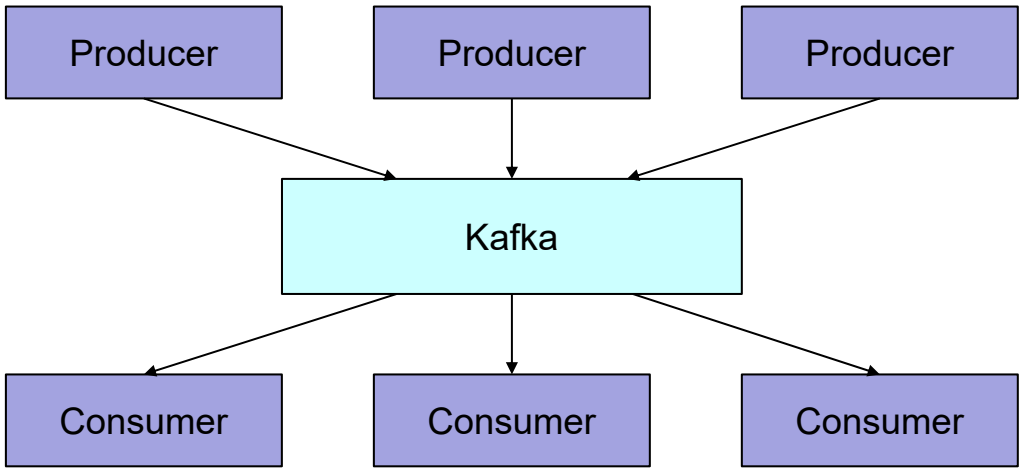
- Kafka, produttori e consumatori (e flussi di eventi)



  - i produttori inviano flussi di eventi ai consumatori mediante l'indirezione di Kafka, che agisce da message broker
  - in effetti, ogni componente può agire sia da produttore che da consumatore di eventi

Produttori e consumatori

- Kafka, produttori e consumatori (e flussi di eventi)



  - i produttori e i consumatori agiscono come client nei confronti 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

Topic

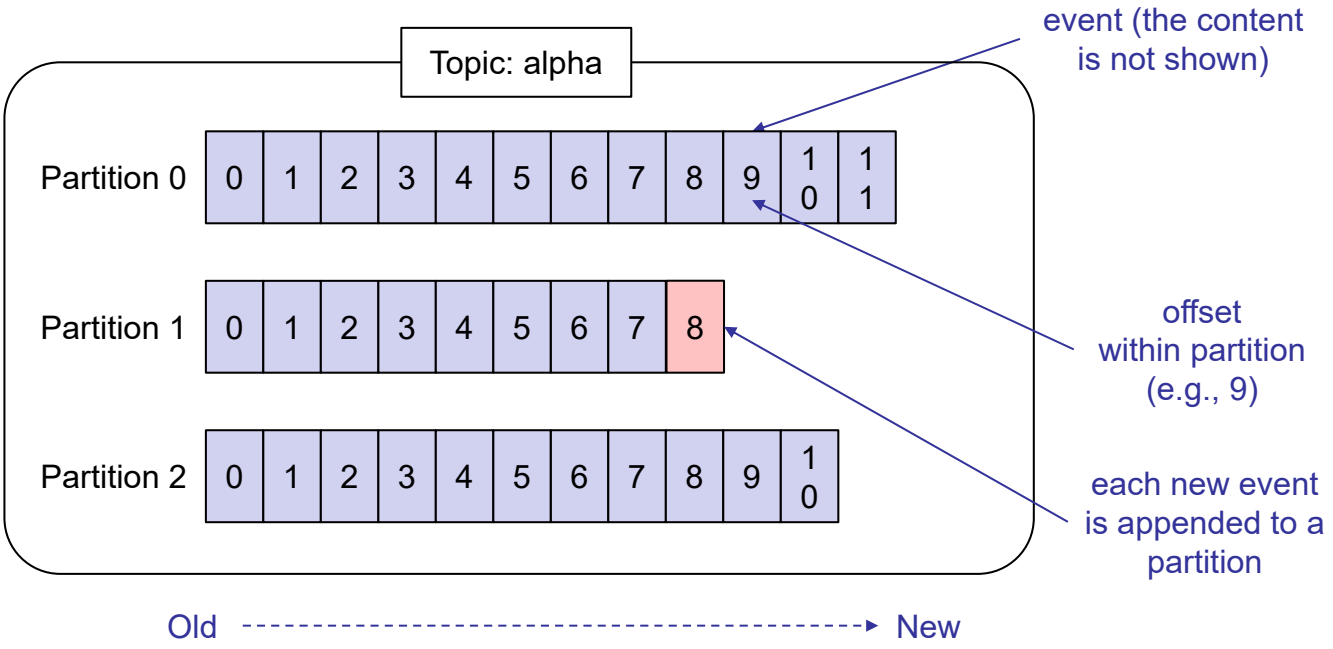
- Un *topic* è una categoria, identificata da un nome, utilizzata per pubblicare e ricevere eventi
  - su un topic possono pubblicare eventi molti produttori (zero, uno o più)
  - a un topic possono abbonarsi, per riceverne gli eventi, molti consumatori (zero, uno o più)

Partizioni

- Per ciascun topic, Kafka mantiene gli eventi pubblicati sul topic in modo partizionato
  - una *partizione* di un topic è una sequenza ordinata e immutabile di eventi
    - che memorizza un sottoinsieme disgiunto degli eventi del topic
    - a cui vengono dinamicamente appesi nuovi eventi
  - ogni evento di una partizione ha un id sequenziale, chiamato *offset*, che identifica l'evento nella partizione

Topic e partizioni

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



  - nota: gli eventi mostrati in questa figura sono tutti distinti tra loro – all'interno è mostrato l'offset dell'evento, non il suo contenuto

Creazione di topic e partizioni

- I topic devono essere in genere creati esplicitamente
  - con un nome, il numero di partizioni e il livello di replicazione (per replicare il topic e le sue partizioni sui diversi nodi del cluster)
  - è anche possibile la creazione automatica di topic (ma è poco sconsigliata)
    - un topic viene creato quando si accede a un topic, con un certo nome, che non esiste ancora
    - per default il topic viene creato con 1 partizione e livello di replicazione 1

Produttori

- Ogni produttore, durante la sua esistenza, può pubblicare molti eventi sui topic che vuole
  - ciascun evento pubblicato su un topic viene appeso a una sola delle partizioni del topic
  - la scelta della partizione a cui appendere il nuovo evento può essere configurata – ad esempio
    - per appenderli in modalità round-robin
    - utilizzando una qualche funzione di partizionamento semantico sulla chiave dell'evento
    - per appendere gli eventi in lotti (batch), per migliorare le prestazioni

Consumatori e gruppi

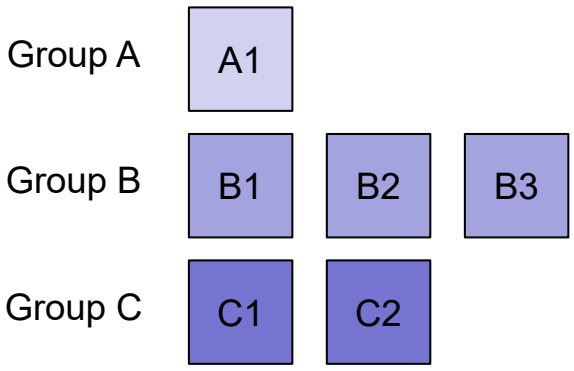
- Ogni consumatore (istanza di consumatore), per ricevere gli eventi di un topic, deve abbonarsi al topic
  - quando un consumatore si abbona a un topic, lo fa specificando il nome del suo *consumer group* (gruppo) – i gruppi vengono utilizzati da Kafka per la distribuzione degli eventi del topic ai consumatori abbonati al topic
  - Kafka distribuisce gli eventi del topic consegnando ciascun evento pubblicato sul topic a un consumatore (istanza di consumatore) per ciascuno dei gruppi
    - ogni evento 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 eventi 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 gli eventi 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 gli eventi di un topic consegnando ciascun evento 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 gli eventi del topic
  - i consumatori B1, B2 e B3 riceveranno, ciascuno, una parte degli eventi del topic
  - in modo simile, anche i consumatori C1 e C2

Consumatori e gruppi

- Kafka distribuisce gli eventi di un topic consegnando ciascun evento 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 degli eventi 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 degli eventi

- Kafka offre le seguenti garanzie sull'ordinamento degli eventi pubblicati su un topic
  - gli eventi 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 gli eventi da una partizione di un topic nell'ordine in cui sono memorizzati nella partizione

Ordine degli eventi

- Kafka offre le seguenti garanzie sull'ordinamento degli eventi pubblicati su un topic
  - pertanto, se c'è un solo topic, con una sola partizione, un solo produttore e un solo consumatore, gli eventi 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
    - d'altra parte, l'uso di una sola partizione per topic non consente, in pratica, di distribuire gli eventi tra i diversi consumatori abbonati al topic

* 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 uno o più nodi – nel cluster deve essere installato Kafka e, di solito, anche 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) – di cui parleremo più avanti nel corso

Installazione e configurazione di Kafka

- Il file docker-compose.yml (semplificato) per Kafka

```
version: '3'

services:
  kafka:
    image: docker.io/bitnami/kafka:3.6
    ports:
      - "9092:9092"
    volumes:
      - "kafka_data:/bitnami"
    environment:
      - KAFKA_CFG_NODE_ID=0
      - KAFKA_CFG_PROCESS_ROLES=controller,broker
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=0@kafka:9093
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093
      - ...

volumes:
  kafka_data:
    driver: local
```

la porta su cui ascolta Kafka

Installazione e configurazione di Kafka

- Script per creare i topic asw.alpha, asw.beta e asw.gamma
  - ciascuno con 4 partizioni e con replicazione 1

```
#!/bin/bash

KAFKA_DOCKER=...trova il container in cui è in esecuzione Kafka...

docker exec -it $KAFKA_DOCKER \
   kafka-topics.sh --bootstrap-server localhost:9092 \
   --create --topic asw.alpha --replication-factor 1 --partitions 4

docker exec -it $KAFKA_DOCKER \
   kafka-topics.sh --bootstrap-server localhost:9092 \
   --create --topic asw.beta --replication-factor 1 --partitions 4

docker exec -it $KAFKA_DOCKER \
   kafka-topics.sh --bootstrap-server localhost:9092 \
   --create --topic asw.gamma --replication-factor 1 --partitions 4
```


Installazione e configurazione di Kafka

- Script per elencare i topic esistenti

```
#!/bin/bash

KAFKA_DOCKER=...trova il container in cui è in esecuzione Kafka...

docker exec -it $KAFKA_DOCKER \
   kafka-topics.sh --bootstrap-server localhost:9092 \
   --list
```

- Un produttore e un consumatore

- Consideriamo ora un semplice esempio, con un produttore e un consumatore, che si scambiano messaggi testuali su un topic (asw.alpha)
  - realizziamo il produttore come un'applicazione Spring Boot, il cui package di base è asw.simpleproducer
  - realizziamo anche il consumatore come un'altra applicazione Spring Boot, il cui package di base è asw.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.simpleproducer.domain;

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

@Service
public class SimpleProducerService {

   @Autowired
   private SimpleMessagePublisherPort 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 di business del produttore

```
package asw.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 di business del produttore.

Produttore

- Per consentire l’invio di messaggi è necessaria una outbound port
  - ecco l’interfaccia per la porta

```
package asw.simpleproducer.domain;

public interface SimpleMessagePublisherPort {

    public void publish(String message);

}
```

Produttore

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

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

import asw.simpleproducer.domain.SimpleMessagePublisherPort;

import org.springframework.kafka.core.KafkaTemplate;

import ...

@Component
public class SimpleMessageKafkaPublisher
                        implements SimpleMessagePublisherPort {

   ... vedi dopo ...

}
```

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

Produttore

- Per consentire l'invio di messaggi tramite Kafka è necessario un outbound adapter (messagepublisher.kafka) 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=asw.alpha
```

  - **KafkaTemplate** è il template Spring per la pubblicazione di messaggi con Kafka
    - i due parametri indicano il tipo della chiave e del valore degli eventi (messaggi)
  - **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=asw.alpha
asw.kafka.groupid=simple-producer

# KAFKA
spring.kafka.bootstrap-servers=10.11.1.121:9092

# KAFKA PRODUCER
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)

Produttore

- Architettura esagonale del produttore

Consumatore

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

```
package asw.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 di business del consumatore.

Consumatore

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

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

import asw.simpleconsumer.domain.SimpleConsumerService;

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

import ...

@Component
public class SimpleMessageKafkaListener {

    ... vedi dopo ...

}
```

Consumatore

- Per consentire la ricezione di messaggi tramite Kafka è necessario un inbound adapter (messagelistener.kafka) 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=asw.alpha
asw.kafka.groupid=simple-consumer
```

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

Consumatore

- Per consentire la ricezione di messaggi tramite Kafka è necessario un inbound adapter (messagelistener.kafka) 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")
    - notare il tipo di chiave e valore degli eventi (messaggi)

Consumatore

- Per consentire la ricezione di messaggi tramite Kafka è necessario un inbound adapter (messagelistener.kafka) per Kafka
  - l'annotazione **@KafkaListener**, gestita dal framework Spring, svolge quasi tutto il lavoro
    - l'assegnazione delle partizioni di un topic ai consumatori abbonati al topic avviene in modo dinamico – ma che vuol dire "dinamicamente"? ecco un esempio
      - avvio un primo consumatore di un gruppo – gli vengono assegnate tutte le partizioni del topic
      - avvio un secondo consumatore dello stesso gruppo – gli vengono assegnate (circa) metà delle partizioni, che vengono sottratte al primo
      - un consumatore del gruppo viene arrestato – le sue partizioni vengono riassegnate agli altri consumatori del gruppo

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=asw.alpha
asw.kafka.groupid=simple-consumer

# KAFKA
spring.kafka.bootstrap-servers=10.11.1.121:9092

# KAFKA CONSUMER
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 devono 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 devono ricevere tutti i messaggi pubblicati sul topic – compresi quelli pubblicati in passato, anche prima del loro abbonamento

Consumatore

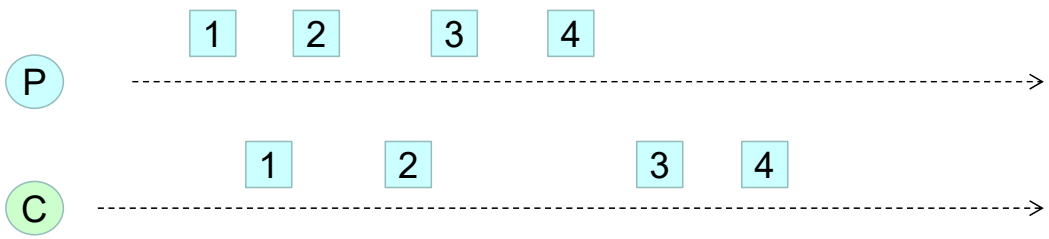
- Architettura esagonale del consumatore

Discussione

- Abbiamo mostrato come realizzare la comunicazione asincrona tra una coppia di servizi/applicazioni
  - nel produttore di messaggi va utilizzato una porta e un adattatore outbound (messagepublisher.kafka) per l'invio di messaggi
    - l'interfaccia per questa porta 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.kafka) 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 business 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"

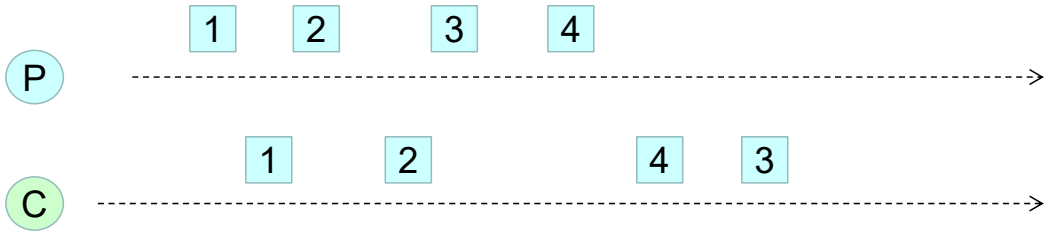
- Alcuni esperimenti con Kafka

- 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)



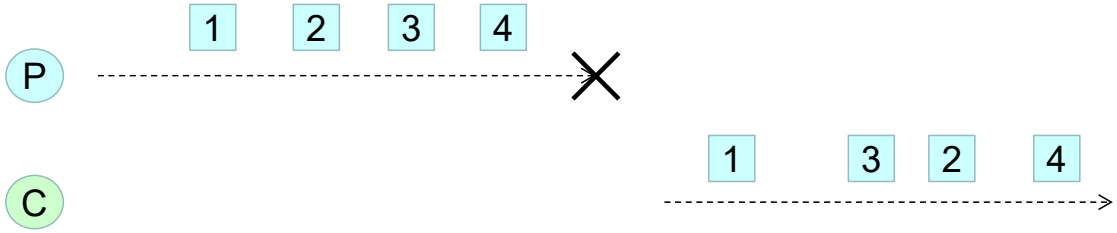
Alcuni esperimenti con Kafka

- 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 (ma non necessariamente nell'ordine in cui sono stati inviati)



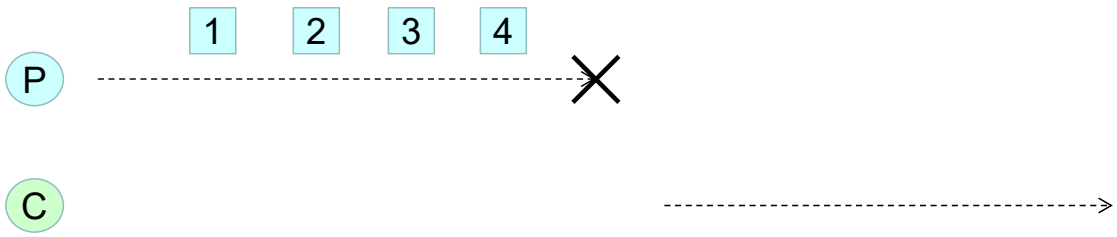
Alcuni esperimenti con Kafka

- 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)



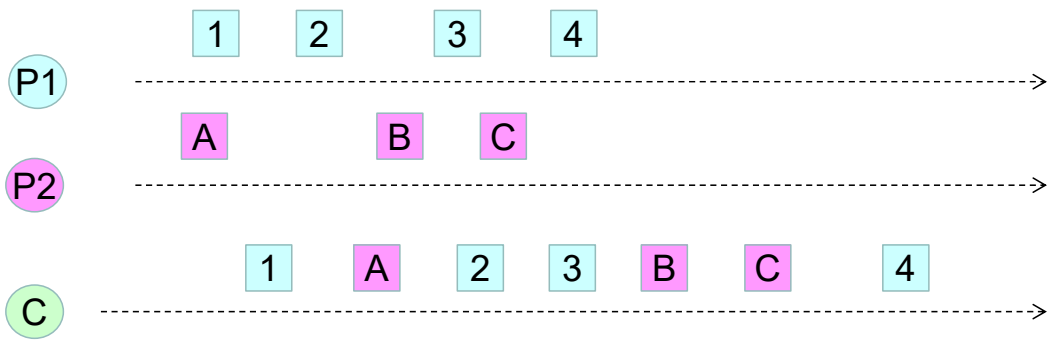
Alcuni esperimenti con Kafka

- 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



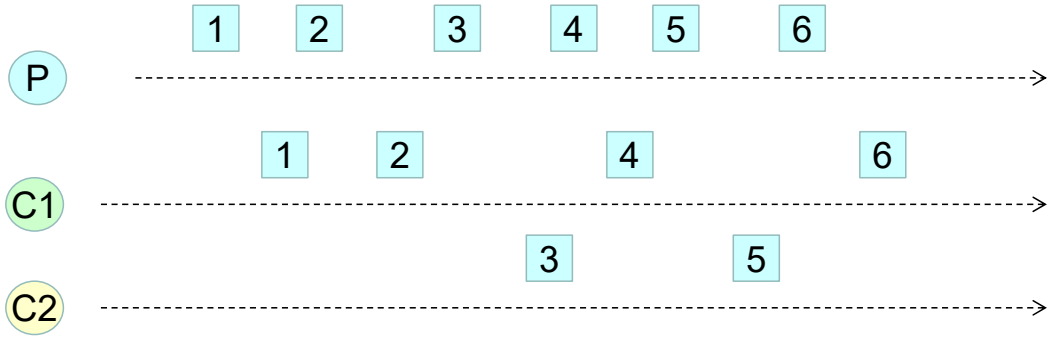
Alcuni esperimenti con Kafka

- 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



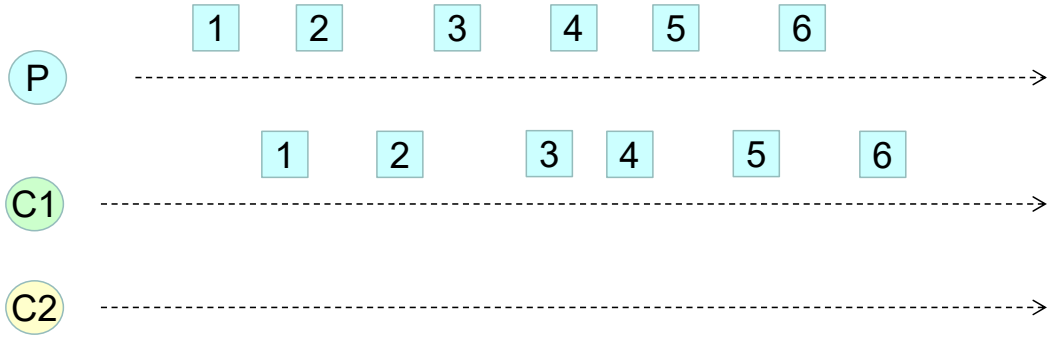
Alcuni esperimenti con Kafka

- 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



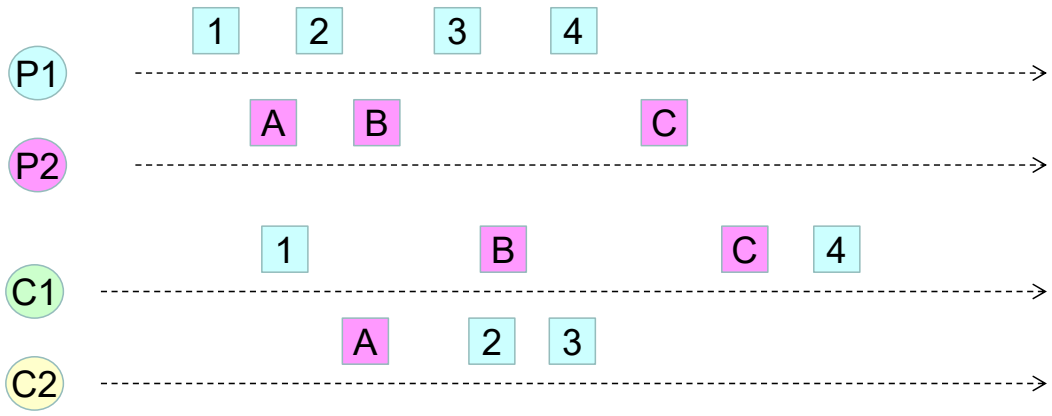
Alcuni esperimenti con Kafka

- 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



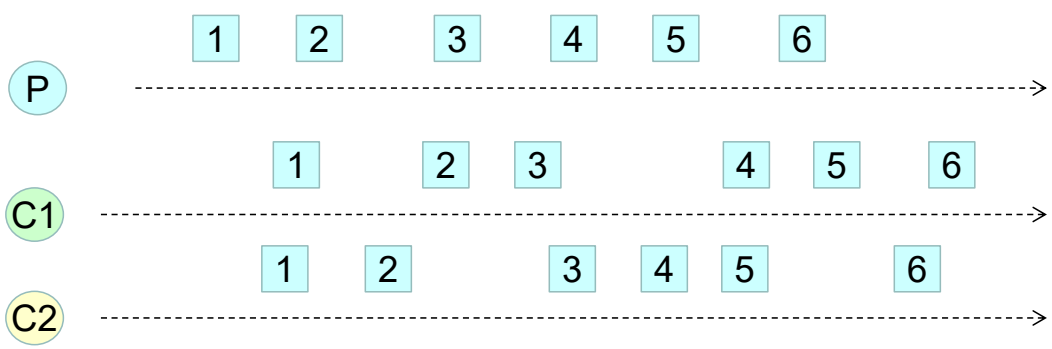
Alcuni esperimenti con Kafka

- 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



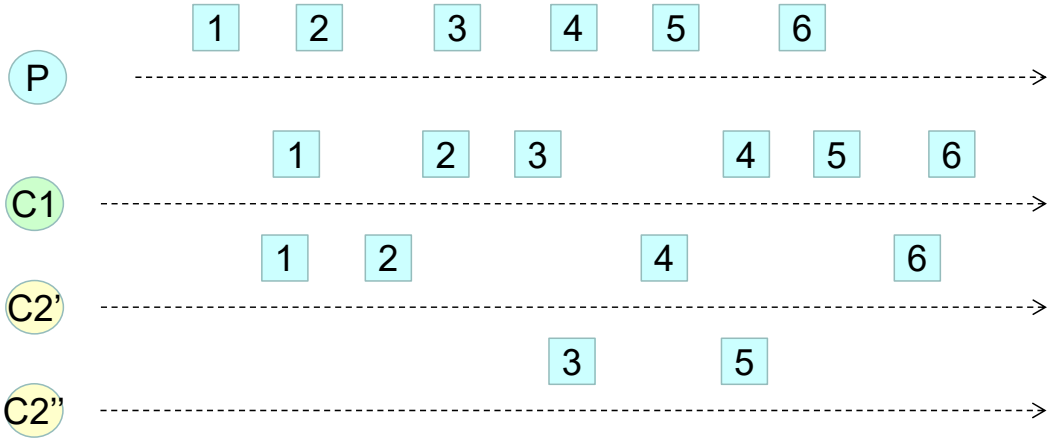
Alcuni esperimenti con Kafka

- 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 due 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 asw.alpha
  - il filtro riceve messaggi dal topic asw.alpha, li elabora, e poi invia messaggi sul topic asw.beta
  - il consumatore riceve messaggi dal topic asw.beta
  - il produttore è come nell'esempio precedente
  - il consumatore è come nell'esempio precedente – ma riceve messaggi da asw.beta anziché da asw.alpha (cambia solo il file di configurazione)
  - 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.simplefilter
  - il filtro ha bisogno di un inbound adapter (messagelistener.kafka) per Kafka – come il consumatore – per consentire la ricezione di messaggi da Kafka
  - il filtro ha anche bisogno di un outbound adapter (messagepublisher.kafka) per Kafka, con la rispettiva interfaccia/porta – 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 per ciascuno dei messaggi ricevuti

Produttore

- Architettura esagonale del filtro

Filtro

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

```
package asw.simplefilter.domain;

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

@Service
public class SimpleFilterService {

    @Autowired
    private SimpleMessagePublisherPort simpleMessagePublisher;

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

}
```

Qui va definita la logica di business del filtro.

Filtro

- L'interfaccia per l'outbound port (messagepublisher)
  - è come per il produttore – in questo caso cambia solo il package

```
package asw.simplefilter.domain;

public interface SimpleMessagePublisherPort {

    public void publish(String message);

}
```

Filtro

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

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

import asw.simplefilter.domain.SimpleMessagePublisherPort;

import org.springframework.kafka.core.KafkaTemplate;

import ...

@Component
public class SimpleMessageKafkaPublisher
                        implements SimpleMessagePublisherPort {

   ... vedi dopo ...

}
```


Filtro

- L'implementazione dell'outbound adapter (messagepublisher.kafka)
  - 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=asw.beta
```

Filtro

- L'implementazione dell'inbound adapter (messagelistener.kafka)
  - 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.simplefilter.messagelistener.kafka;

import asw.simplefilter.domain.SimpleFilterService;

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

import ...

@Component
public class SimpleMessageKafkaListener {

   ... vedi dopo ...

}
```


Filtro

- L'implementazione dell'inbound adapter (messagelistener.kafka)
  - 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=asw.alpha
asw.kafka.groupid=simple-filter
```

Filtro

- Un'occhiata al file application.properties – contiene sia le proprietà dei consumatori che quelle dei produttori

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

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

# KAFKA
spring.kafka.bootstrap-servers=10.11.1.121:9092

# KAFKA CONSUMER
spring.kafka.consumer.group-id=${asw.kafka.groupid}
spring.kafka.consumer.auto-offset-reset=latest
spring.kafka.consumer.key-deserializer=...
spring.kafka.consumer.value-deserializer=...
spring.kafka.consumer.properties.spring.json.trusted.packages=*

# KAFKA PRODUCER
spring.kafka.producer.key-serializer=...
spring.kafka.producer.value-serializer=...
```


- 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
  - la gestione dei ristoranti avviene tramite il servizio **RestaurantService**
  - i ristoranti sono definiti come un'entità JPA **Restaurant** – con attributi **id**, **name** e **location**
  - internamente al servizio, i ristoranti vengono acceduti da una base di dati mediante un repository **RestaurantRepository**

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 di dominio* 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 di dominio* 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 una outbound port e 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 la porta per l'invio di eventi e l'implementazione di un adattatore eventpublisher (per Kafka)
    - l'utilizzo della porta/adattatore 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";

}
```

  - il topic restaurant-service-event-channel va creato in sede di configurazione dell'applicazione

Pubblicazione di eventi

- La definizione di un'interfaccia per la porta per l'invio di eventi e l'implementazione di un adattatore eventpublisher (per Kafka)

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

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

public interface RestaurantEventPublisher {

    public void publish(DomainEvent event);

}
```


Pubblicazione di eventi

- La definizione di un'interfaccia per la porta per l'invio di eventi e l'implementazione di un adattatore eventpublisher (per Kafka)

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

import …

@Component
public class RestaurantEventKafkaPublisher
             implements RestaurantEventPublisher {

    @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 della porta/adattatore 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 asw.efood.common.api.event.DomainEvent;
import asw.efood.restaurantservice.api.event.*;

import …

@Service @Transactional
public class RestaurantService {

   @Autowired
   private RestaurantRepository restaurantRepository;

   @Autowired
   private RestaurantEventPublisher restaurantEventPublisher;

   ... vedi dopo ...

}
```


Pubblicazione di eventi

- L'utilizzo della porta/adattatore 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()
                                );
   restaurantEventPublisher.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) e una relativa inbound port
  - 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 – è un servizio che fa parte della logica di business e definisce implicitamente la porta per l'adapter commandlistener
    - l'implementazione dell'inbound adapter commandlistener (per Kafka)

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";

}
```

  - il topic restaurant-service-command-channel va creato in sede di configurazione dell'applicazione

Ricezione di comandi

- L'implementazione di un command handler per il servizio dei ristoranti – è un servizio che fa parte della logica di business
  - 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 – è un servizio che fa parte della logica di business
  - il metodo onCommand per la gestione dei comandi può essere basato su un'istruzione condizionale per capire quale comando è stato richiesto

```
public void onCommand(Command command) {

   if (command instanceof CreateRestaurantCommand cmd) {

      this.createRestaurant(cmd);

   } else if (command instanceof AnotherOpCommand cmd) {

      this.anotherOp(cmd);

   } else {
      ... unknown command ...
   }

}
```

Ricezione di comandi

- L'implementazione di un command handler per il servizio dei ristoranti – è un servizio che fa parte della logica di business
  - inoltre, il command handler definisce un metodo di supporto per ciascuno dei comandi – a cui delegare il controllo quando viene ricevuto uno specifico comando

```
private void createRestaurant(CreateRestaurantCommand command) {

   restaurantService.createRestaurant(
                              command.getName(),
                              command.getLocation()
                           );

}
```


Ricezione di comandi

- L'implementazione dell'adapter commandlistener (per Kafka)
  - alla ricezione di un messaggio per un comando (qui viene usato il group-id dell'applicazione), invoca il command handler

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

import …;

@Component
public class RestaurantCommandKafkaListener {

   @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 producer e consumer

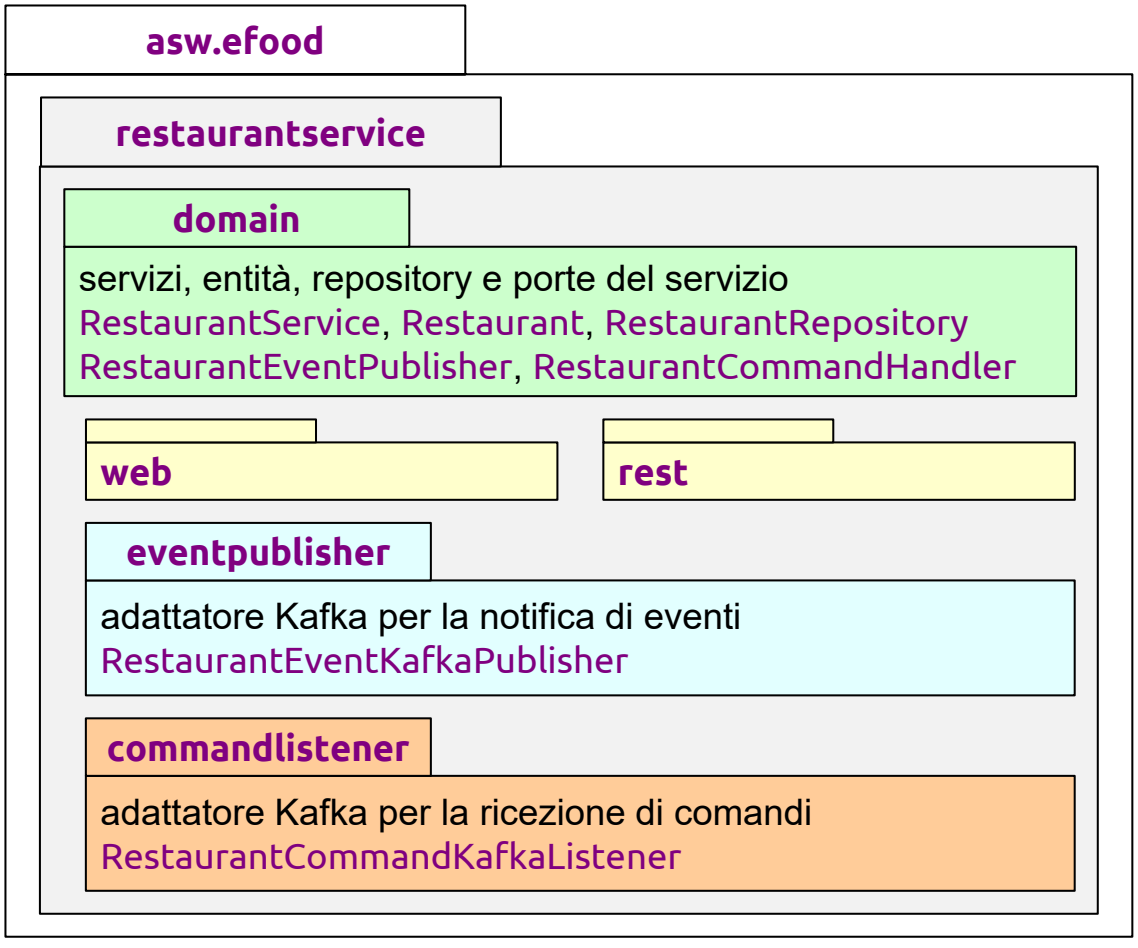
```
# KAFKA CONSUMER
spring.kafka.consumer.group-id=${spring.application.name}
spring.kafka.consumer.auto-offset-reset=earliest

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=*

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

  - 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

Architettura esagonale



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 (eventi) tramite canali (topic) – con un modello basato su gruppi che generalizza quello dei canali point-to-point e publish-subscribe
  - ogni 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
  - con riferimento all'architettura esagonale
    - l'invio di messaggi, da parte di un produttore, richiede la definizione di un outbound adapter (e della relativa porta)
    - la ricezione di messaggi, da parte di un consumatore, richiede la definizione di un inbound adapter (e della relativa porta)
  - in termini di pattern per il Messaging, questi adapter sono dei *message endpoint* [POSA4] – più precisamente, sono dei *messaging gateway* [EIP]
    - un message endpoint è un connettore che consente lo scambio di messaggi
    - un messaging gateway è un message endpoint che ha lo scopo di disaccoppiare i componenti dalla tecnologia di messaging utilizzata

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 è anche possibile utilizzare XML o Protocol Buffers

Discussione

- Ecco alcune considerazioni sull'utilizzo di Kafka
  - come realizzare un'applicazione Spring Boot che è interessata a ricevere due tipi di messaggi da due diversi canali? – ad, un applicazione che è sia un ascoltare di comandi (da un canale comandi) che un ascoltatore di eventi (da un canale eventi)
  - ci sono almeno due possibilità
    - usare un unico **@KafkaListener** per entrambi i canali e i tipi di messaggi
    - usare due diversi **@KafkaListener**, uno per ciascun canale e tipo di messaggi (di solito preferibile)

Discussione

- Ecco alcune considerazioni sull'utilizzo di Kafka
  - come realizzare un'applicazione Spring Boot che è interessata a ricevere due tipi di messaggi da due diversi canali?
    - usare un unico **@KafkaListener** per entrambi i canali e i tipi di messaggi
    - bisogna anche definire un supertipo dei messaggi

```
@Component
public class RestaurantMessageKafkaListener {

   @Autowired
   private RestaurantMessageListener restaurantMessageListener;

   @KafkaListener(topics = { "comandi", "eventi" })
   public void listen(ConsumerRecord<String, Message> record)
                                                 throws Exception {
      Message message = record.value();
      restaurantMessageListener.onMessage(message);
   }

}
```

Discussione

- Ecco alcune considerazioni sull'utilizzo di Kafka
  - come realizzare un'applicazione Spring Boot che è interessata a ricevere due tipi di messaggi da due diversi canali?
    - usare due diversi **@KafkaListener**, uno per ciascun canale e tipo di messaggi (di solito preferibile)
    - ogni listener però deve appartenere a un gruppo differente

```
@Component
public class RestaurantCommandKafkaListener {

   @Autowired
   private RestaurantCommandHandler restaurantCommandHandler;

   @KafkaListener(topics = "comandi",
                  groupId = "restaurant-command-listener")
   public void listen(ConsumerRecord<String, Command> record)
                                                  throws Exception {
      Command command = record.value();
      restaurantCommandHandler.onCommand(command);
   }
```


* 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 eventi (flussi di messaggi)
  - i canali (chiamati topic e organizzati in partizioni) consentono di pubblicare e di ricevere eventi
  - i produttori possono pubblicare flussi di messaggi (eventi) su uno o più topic
  - i consumatori possono ricevere flussi di messaggi (eventi) 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