**Caso di studio: Implementazione di un'architettura event-driven con Kafka**

**Obiettivo**: Acquisire competenze nello sviluppo di sistemi asincroni e nella gestione di eventi in tempo reale.

**Tecnologie**: Spring Boot, Apache Kafka, Kafka Streams, Spring Cloud Stream.

**Descrizione dettagliata**:  
Il tirocinante realizzerà una soluzione event-driven basata su Kafka per simulare la gestione di notifiche e alert in tempo reale all’interno di una piattaforma bancaria.   
Nello specifico, dovrà implementare:

* Produttori Kafka per la generazione degli eventi relativi a transazioni e notifiche di sicurezza.
* Consumatori Kafka per la gestione e il trattamento di tali eventi in tempo reale.
* Utilizzo di Kafka Streams per elaborazioni complesse e filtraggio degli eventi, gestendo casi di idempotenza, fault-tolerance e scalabilità.
* Una documentazione tecnica esaustiva delle scelte architetturali, delle configurazioni adottate e delle strategie di resilienza implementate.

1. **Teoria delle tecnologie**

Spring Boot, Apache Kafka, Kafka Streams e Spring Cloud Stream. Queste tecnologie consentono di costruire un sistema basato sulla comunicazione asincrona tramite eventi.   
Di seguito sono elencati gli strumenti e le tecnologie coinvolte nel progetto.

* 1. **Spring Boot**

Spring Boot è un framework open source basato su Spring, progettato per semplificare lo sviluppo di applicazioni Java, offrendo una configurazione minima e componenti pronti all'uso tramite starter.

Elemento centrale di Spring Boot è il concetto di Inversion of Control, che delega al framework la gestione dei componenti dell'applicazione. A questo si lega la Dependency Injection, che consente di fornire automaticamente ai vari componenti le dipendenze di cui necessitano. In Spring, l'iniezione può avvenire via campo, via costruttore o tramite setter.

Spring Boot gestisce automaticamente i componenti dell'applicazione grazie a specifiche annotazioni:

* @Component definisce un componente generico.
* @Service indica una classe di logica applicativa.
* @Bean definisce un metodo che restituisce un oggetto da registrare nel container Spring come bean gestito.
* @Configuration identifica una classe che definisce bean e configurazioni personalizzate per l'applicazione.
* @SpringBootApplication attiva una configurazione automatica completa e la scansione dei componenti per il progetto Spring Boot.
* @EnableKafkaStreams abilita il supporto a Kafka Streams nella configurazione dell'applicazione.
* @Component indica un generico componente da gestire come bean (già descritto sopra).
* @Profile consente di specificare il profilo di esecuzione (ad esempio "dev", "prod") per cui un determinato bean o configurazione deve essere attivato.

Quando Spring Boot avvia l'applicazione, scansiona il progetto alla ricerca di queste annotazioni, registrando le relative classi nel container IoC. Le istanze così registrate vengono chiamate bean.

Una volta gestiti dal container, questi bean sono soggetti alla Dependency Injection: Spring li fornisce automaticamente dove necessario, occupandosi del loro ciclo di vita.

Grazie all'autoconfigurazione, Spring Boot rileva le librerie e dipendenze presenti nel progetto (classpath) e configura automaticamente i componenti, semplificando così l'impostazione iniziale dell'applicazione.

Tra le caratteristiche principali si includono:

* Supporto immediato a microservizi e REST API.
* Riduzione del boilerplate.
* Predisposizione rapida tramite Spring Initializer.
* Integrazione facilitata con Maven.
* Compatibilità con Java moderne.
* Supporto allo sviluppo tramite DevTools per reload automatico.
  1. **Apache Kafka**

Apache Kafka è una piattaforma distribuita per la gestione di flussi di dati in tempo reale. Permette di implementare un modello publish-subscribe efficiente e scalabile, ideale per applicazioni event-driven. Nel progetto è utilizzato come sistema di messaging centrale per la gestione degli eventi relativi a transazioni e notifiche bancarie. Kafka garantisce persistenza, tolleranza ai guasti e alte prestazioni nel trattamento di grandi volumi di dati.

* 1. **Kafka Streams**

Kafka Streams è la libreria client di Kafka dedicata all'elaborazione dei flussi di dati direttamente all'interno dell'applicazione. Consente di costruire pipeline di stream processing sfruttando le API di Kafka senza dover ricorrere a cluster esterni. Nel progetto è utilizzato per implementare funzionalità di filtro e trasformazione dei dati in tempo reale, migliorando la reattività del sistema e la capacità di processare eventi in modo distribuito.

* 1. **Spring Cloud Stream**

Spring Cloud Stream è un framework che semplifica la costruzione di microservizi basati su eventi. Offre un sistema di binding astratto per collegare le applicazioni a piattaforme di messaging come Kafka. Nel progetto, è stato adottato per facilitare la programmazione event-driven e per ridurre il carico di gestione manuale dei topic Kafka, consentendo una maggiore astrazione e una gestione più pulita degli eventi.

* 1. **Docker**

Docker è una piattaforma open-source che consente di automatizzare il deployment, il scaling e la gestione di applicazioni all'interno di container. I container sono ambienti isolati che includono tutto il necessario per eseguire un'applicazione: codice, runtime, strumenti di sistema, librerie e impostazioni. Questo approccio garantisce che le applicazioni funzionino in modo uniforme indipendentemente dall'ambiente in cui vengono eseguite, semplificando il ciclo di vita dello sviluppo e del deployment del software.

1. **Struttura del progetto**

Nel progetto vengono impiegate tre tecnologie, ciascuna con un compito specifico:

* Apache Kafka: funge da bus centrale che riceve gli eventi prodotti, li persiste nei e li rende disponibili ai vari consumer in modo affidabile e asincrono.
* Kafka Streams: elaborazione in tempo reale del topic, filtra le operazioni e le pubblica in un topic.
* Spring Cloud Stream: implementa un consumer dichiarativo per un topic senza codice Kafka esplicito.

Questi tre livelli tecnici permettono di simulare una piattaforma bancaria dove i producer generano eventi di transazione e di sicurezza, Kafka li instrada e li persiste, Kafka Streams crea automaticamente il topic dedicato alle transazioni di alto valore, e i consumer estraggono i messaggi dai rispettivi topic per elaborarli in modo indipendente e asincrono.

* 1. **Scelte architetturali generali**

Di seguito sono elencati i principali componenti e le scelte architetturali adottate:

* TransactionEvent: rappresenta ogni movimento bancario (versamento, prelievo, ecc.).
* SecurityAlertEvent: segnala eventi sospetti legati alla sicurezza (accesso anomalo ecc.).

Apache Kafka gestisce il transito e la persistenza dei messaggi, pubblicandoli sui relativi topic dedicati:

* Topic transactions per gli eventi di transazione.
* Topic security-alerts per gli eventi di sicurezza.
* Topic high-value-transactions per le sole transazioni bancarie con importo pari o superiore a 1000€, generate automaticamente dal sistema di filtraggio.

Dal lato producer:

* Gli eventi di transazione e sicurezza vengono generati e inviati dai rispettivi produttori verso i topic principali di Kafka.

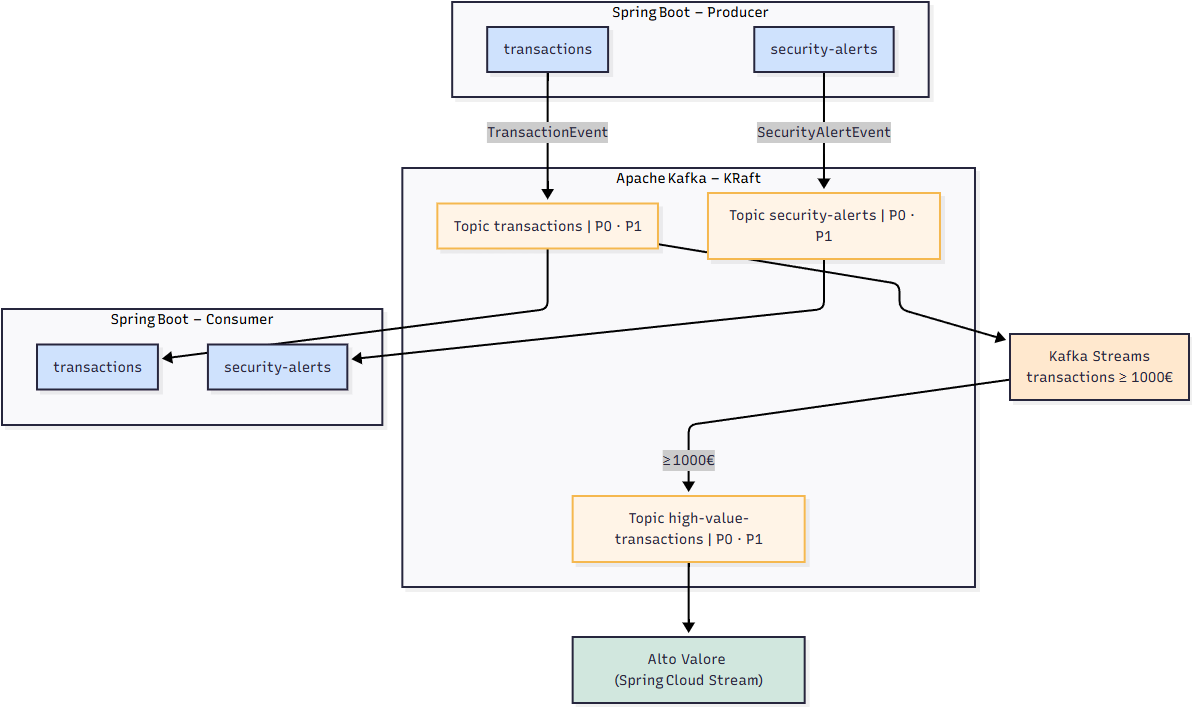
Dal lato Apache Kafka:

* Kafka riceve, memorizza e smista gli eventi verso i consumer, mantenendo la persistenza dei dati tramite log.
* Kafka Streams filtra automaticamente le transazioni ad alto valore e le inoltra nel topic dedicato.

Dal lato consumer:

* Ricevono gli eventi dai topic Kafka dedicati.
* Gli eventi vengono registrati nello storico.
* Gli eventi vengono mostrati nei log dell’applicazione.
* Le transazioni ad alto valore pubblicate nel topic high-value-transactions vengono ricevute da un consumer specifico per consentirne un'eventuale analisi.
  1. **Schema visivo sintetico**

Di seguito è illustrato uno schema visivo dell'applicativo, che mostra le sue componenti e le relazioni tra esse.



1. **Configurazioni adottate**

Per avviare il progetto è stato utilizzato Spring Boot Initializer, uno strumento online che ha consentito di generare la struttura base dell'applicazione.

* 1. **Creazione del pom.xml**

Successivamente, il file pom.xml è stato completato fino ad arrivare alla composizione finale riportata di seguito:

**GAV**

* GroupId: it.alex.kafka
* ArtifactId: event-driven-banking
* Version: 1.0.0

**Parent POM**

* org.springframework.boot:spring-boot-starter-parent:3.5.3

**Properties**

* java.version: 17 scelto per il Long Term Support e la piena compatibilità con Spring Boot 3.x.
* spring-cloud.version: 2025.0.0 per il coordinamento centralizzato delle versioni Spring Cloud.

**Dipendenze principali**

* Kafka Streams: consente l'elaborazione dei flussi tramite le API native di Kafka.
  + La classe KafkaStreamsTopology definisce una pipeline che legge il topic transactions, filtra le transazioni con importo ≥ 1000 €, effettua deduplicazione sulla chiave transactionId e scrive i risultati nel topic high-value-transactions.
* Spring Cloud Stream: facilita la programmazione event-driven.
  + Implementa un consumer dichiarativo tramite la classe HighValueTransactionConsumer, che consuma gli eventi dal topic high-value-transactions. Evit l’uso diretto delle API Kafka e configurazione più modulare tramite il concetto di group-id.
* Spring Cloud Stream Binder Kafka: collega Spring Cloud Stream a Kafka.
  + Rappresenta il connettore tecnico che permette alla funzione dichiarativa del consumer (HighValueTransactionConsumer) di ricevere eventi direttamente dal topic high-value-transactions su Kafka. Gestisce in automatico sottoscrizione al topic, assegnazione del group-id e il flusso dei messaggi verso il metodo consumer configurato.
* Spring Kafka: permette la gestione di producer e consumer Kafka puri.
  + Consente di implementare producer manuali (KafkaProducerService) che inviano eventi ai topic Kafka tramite KafkaTemplate e consumer di basso livello (KafkaConsumerService e KafkaSecurityAlertConsumerService) che ricevono e processano direttamente gli eventi dai topic transactions e security-alerts usando annotazioni @KafkaListener.
* Spring Boot DevTools: semplifica lo sviluppo grazie al supporto per hot reload.
  + Non viene impacchettato nella build finale né nel container di produzione.
* Lombok: riduce il boilerplate generando automaticamente metodi e costruttori.

**Dependency Management**

* org.springframework.cloud:spring-cloud-dependencies:2025.0.0

Questo consente la gestione centralizzata delle versioni per tutte le librerie Spring Cloud utilizzate nel progetto.

**Build**

* maven-compiler-plugin: configurato per supportare Lombok come annotation processor durante la fase di compilazione.  
  1. Docker

Nel progetto docker viene utilizzato esclusivamente per predisporre un'istanza locale di Apache Kafka. Il broker Kafka configurato in modalità KRaft è eseguito all’interno di un container basato sull’immagine bitnami/kafka. Tramite il file docker-compose.yml, Kafka è esposto sulla porta 9092 per consentire la comunicazione con l'applicazione Spring Boot, mentre la porta interna 9093 è riservata al controller KRaft.

L’intero ambiente può essere avviato con il semplice comando docker compose up -d, evitando l’installazione manuale di Kafka o la configurazione di ambienti Java.

In passato, Kafka utilizzava ZooKeeper come sistema esterno per la gestione dei metadata, del quorum e del coordinamento dei broker. In questo progetto, invece, si è scelto di utilizzare KRaft, una modalità più recente in cui Kafka gestisce internamente queste funzioni. L'adozione di KRaft elimina la necessità di un cluster ZooKeeper, riducendo la complessità architetturale e semplificando il setup.

KRaft funziona attraverso un quorum interno di controller che garantisce consistenza dei dati e tolleranza ai guasti, svolgendo il ruolo precedentemente affidato a ZooKeeper. Questo approccio consente a Kafka di operare in modalità standalone.

* 1. Creazione di docker-compose.yml

version: "3.8"  
services:  
 kafka:  
 image: bitnami/kafka:latestcontainer\_name: kafkaports:  
 - "9092:9092"environment:  
 - KAFKA\_CFG\_NODE\_ID=1- KAFKA\_CFG\_PROCESS\_ROLES=broker,controller- KAFKA\_CFG\_CONTROLLER\_LISTENER\_NAMES=CONTROLLER- KAFKA\_CFG\_CONTROLLER\_QUORUM\_VOTERS=1@kafka:9093- KAFKA\_CFG\_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093- KAFKA\_CFG\_ADVERTISED\_LISTENERS=PLAINTEXT://localhost:9092- ALLOW\_PLAINTEXT\_LISTENER=yes

Questo file docker-compose.yml ha lo scopo di predisporre in modo semplice e veloce l'infrastruttura di messaggistica necessaria al progetto. In particolare, definisce un servizio Docker chiamato kafka che avvia un container basato sull'immagine ufficiale di Kafka distribuita da Bitnami. Di seguito, il ruolo delle principali configurazioni:

* image: specifica l'immagine Docker da usare. Nel progetto, si utilizza bitnami/kafka per ottenere una versione pronta all'uso di Apache Kafka.
* container\_name: assegna il nome al container Docker, facilitando la sua gestione (avvio, stop, ecc.).
* ports: espone la porta 9092 del container, necessaria per consentire alle applicazioni esterne (come Spring Boot) di comunicare con Kafka.
* environment: definisce le variabili d'ambiente utilizzate per configurare Kafka:
  + KAFKA\_CFG\_NODE\_ID: identifica univocamente il nodo all'interno del cluster Kafka (anche se si usa un solo nodo).
  + KAFKA\_CFG\_PROCESS\_ROLES: specifica che questo nodo svolge sia il ruolo di broker sia di controller, configurando Kafka in modalità KRaft (Kafka Raft Metadata Mode), evitando l'uso di ZooKeeper.
  + KAFKA\_CFG\_CONTROLLER\_LISTENER\_NAMES e KAFKA\_CFG\_CONTROLLER\_QUORUM\_VOTERS: configurano il quorum del controller KRaft, necessario anche in ambiente a nodo singolo.
  + KAFKA\_CFG\_LISTENERS e KAFKA\_CFG\_ADVERTISED\_LISTENERS: definiscono gli endpoint di ascolto per i client e per il controller interno.
  + ALLOW\_PLAINTEXT\_LISTENER: consente comunicazioni non cifrate, facilitando il testing locale.

Nel complesso, questo file consente l'esecuzione locale di Kafka come sistema di messaggistica stand-alone in modalità KRaft, evitando la complessità di configurare un cluster ZooKeeper e semplificando la fase di sviluppo.

* 1. Creazione del application.yml

spring:  
 application:  
 name: event-driven-bankingprofiles:  
 active: devcloud:  
 stream:  
 bindings:  
 highValueConsumer-in-0:  
 destination: high-value-transactionsgroup: high-value-groupkafka:  
 binder:  
 brokers: localhost:9092kafka:  
 bootstrap-servers: localhost:9092streams:  
 state-dir: /tmp/kafka-streamsproperties:  
 cleanup.on.start: trueproducer:  
 key-serializer: org.apache.kafka.common.serialization.StringSerializervalue-serializer: org.springframework.kafka.support.serializer.JsonSerializerconsumer:  
 key-deserializer: org.apache.kafka.common.serialization.StringDeserializer value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializergroup-id: banking-groupauto-offset-reset: earliestproperties:  
 spring.json.trusted.packages: it.alex.kafka.banking.modelspring.json.value.default.type: it.alex.kafka.banking.model.TransactionEventlogging:  
 level:  
 root: ERRORorg.apache.kafka.streams.processor.internals.StateDirectory: OFFit.alex.kafka.banking: INFO

Questo file application.yml rappresenta la configurazione centralizzata del progetto, consentendo di definire in modo strutturato i parametri operativi di Spring Boot, Apache Kafka e Spring Cloud Stream. Di seguito, la spiegazione delle sezioni principali del file:

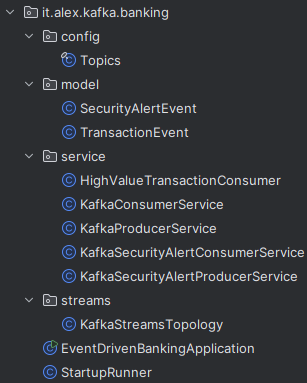
* spring.application.name: imposta il nome logico dell'applicazione Spring Boot.
* spring.profiles.active: specifica il profilo attivo. Nel progetto, viene utilizzato il profilo dev per distinguere l’ambiente di sviluppo.
* spring.cloud.stream: configura i binding di Spring Cloud Stream:
  + bindings.highValueConsumer-in-0.destination: indica che il consumer Cloud Stream ascolta il topic high-value-transactions.
  + bindings.highValueConsumer-in-0.group: definisce il gruppo consumer per il topic di alto valore.
  + kafka.binder.brokers: imposta l’indirizzo del broker Kafka, permettendo la connessione ai topic.
* spring.kafka: configura la parte Kafka nativa usata da Spring Boot:
  + bootstrap-servers: indirizzo del broker Kafka a cui connettersi.
  + streams.state-dir: directory locale usata da Kafka Streams per la memorizzazione dello stato.
  + streams.properties.cleanup.on.start: se impostato a true, forza la pulizia dello stato di Kafka Streams all’avvio.
  + producer.key-serializer e producer.value-serializer: specificano i serializer da usare nella produzione di eventi.
  + consumer.key-deserializer e consumer.value-deserializer: indicano i deserializer usati nella ricezione degli eventi dai topic Kafka.
  + consumer.group-id: identifica il gruppo consumer associato all’ascolto degli eventi di transazione.
  + consumer.auto-offset-reset: definisce la modalità di reset dell’offset: earliest consente di leggere i messaggi non ancora elaborati dal topic.
  + consumer.properties.spring.json.trusted.packages: per la deserializzazione JSON, specifica il pacchetto da considerare affidabile.
  + consumer.properties.spring.json.value.default.type: forza il tipo di evento da deserializzare (nel progetto: TransactionEvent).
* logging.level: regola il livello di dettaglio dei log:
  + root: filtra i log globali di Spring Boot.
  + org.apache.kafka.streams.processor.internals.StateDirectory: disabilita i log interni di Kafka Streams.
  + it.alex.kafka.banking: controlla il livello di log dell'applicazione.

Nel complesso, questo file consente la configurazione coerente di Spring Boot, Kafka e Spring Cloud Stream, garantendo il corretto instradamento e la gestione degli eventi prodotti e consumati all’interno del sistema.

1. **Scelte architetturali**

In questo capitolo viene presentata la struttura architetturale del progetto "event-driven-banking", evidenziando la suddivisione in package, le principali classi implementate e il loro ruolo all'interno dell'architettura event-driven basata su Apache Kafka. Ogni componente è descritto in relazione al suo contributo nella gestione degli eventi di transazione e di sicurezza nella piattaforma bancaria simulata.

* 1. Struttura del Progetto

****

**Descrizione dei Package e delle Classi**

**Package Radice**

* EventDrivenBankingApplication.java: classe principale Spring Boot, avvia l'intera applicazione e inizializza il contesto.
* StartupRunner.java: classe utilizzata per inviare automaticamente eventi di esempio all'avvio del sistema, simulando il comportamento dei producer.

**Package config**

Topics.java: contiene le costanti per i nomi dei topic Kafka utilizzati nel progetto, centralizzandone la gestione.

**Package model**

* SecurityAlertEvent.java: rappresenta il modello dati di un evento di sicurezza (ad esempio un accesso sospetto).
* TransactionEvent.java: rappresenta il modello dati di un evento di transazione bancaria (deposito, prelievo, ecc.).

**Package service**

* HighValueTransactionConsumer.java: consumer implementato tramite Spring Cloud Stream. Consuma transazioni di alto valore pubblicate nel topic high-value-transactions.
* KafkaConsumerService.java: consumer nativo Kafka (Spring Kafka). Riceve eventi di transazione standard dal topic transactions.
* KafkaProducerService.java: producer per gli eventi di transazione. Invia oggetti TransactionEvent al topic Kafka di riferimento.
* KafkaSecurityAlertConsumerService.java: consumer nativo Kafka per la gestione degli eventi di sicurezza ricevuti dal topic security-alerts.
* KafkaSecurityAlertProducerService.java: producer dedicato all'invio di eventi SecurityAlertEvent sul topic Kafka corrispondente.

**Package streams**

KafkaStreamsTopology.java: definisce la topologia Kafka Streams. Filtra le transazioni di importo pari o superiore a 1000€, evitando duplicazioni, e pubblica gli eventi risultanti nel topic high-value-transactions.

* 1. Package radice

**EventDrivenBankingApplication**

@SpringBootApplication  
public class EventDrivenBankingApplication {  
public static void main(String[] args) {  
 SpringApplication.*run*(EventDrivenBankingApplication.class, args);  
 }  
}

Questa classe rappresenta il punto di ingresso principale dell'applicazione Spring Boot. Grazie all'annotazione @SpringBootApplication, viene abilitata la configurazione automatica di Spring e l'inizializzazione del contesto applicativo.

Il metodo main() invoca SpringApplication.run(), che avvia l'intera applicazione, includendo i componenti Kafka, il framework Spring Cloud Stream, i servizi di produzione e consumo eventi e le eventuali configurazioni definite nel progetto. Si tratta di una classe standard nei progetti Spring Boot, responsabile dell'avvio del runtime.

**StartupRunner**

@Component  
@Profile("dev")  
@RequiredArgsConstructor  
@Slf4j  
public class StartupRunner implements CommandLineRunner {

private final KafkaProducerService producerService;

private final KafkaSecurityAlertProducerService securityAlertProducerService;

@Override  
 public void run(String... args) {  
 // Evento di transazione  
 TransactionEvent transaction = new TransactionEvent(  
 UUID.*randomUUID*().toString(),  
 "ACC123456",  
 1050.0,  
 "deposit",  
 LocalDateTime.*now*()  
 );  
 producerService.sendTransaction(transaction);  
 *log*.info("Evento transazione inviato: {}", transaction);  
  
 // Evento di sicurezza  
 SecurityAlertEvent alert = new SecurityAlertEvent(  
 UUID.*randomUUID*().toString(),  
 "ACC123456",  
 "Accesso da IP sospetto",  
 LocalDateTime.*now*()  
 );  
 securityAlertProducerService.sendAlert(alert);  
 *log*.info("Evento sicurezza inviato: {}", alert);  
 }  
}

La classe StartupRunner ha il ruolo di generare automaticamente eventi di test all'avvio dell'applicazione. Questo componente è annotato con @Component e @Profile("dev"), il che significa che viene eseguito solo quando il profilo attivo è impostato su dev (ambiente di sviluppo).

Implementando CommandLineRunner, il metodo run() viene invocato subito dopo il bootstrap di Spring Boot. Al suo interno:

* Viene creato un evento di transazione (TransactionEvent), generando un identificativo univoco e simulando un versamento di 1050 euro sul conto.
* L'evento viene inviato al topic Kafka configurato tramite il servizio KafkaProducerService.
* Successivamente, viene creato un evento di sicurezza (SecurityAlertEvent), simulando un accesso sospetto.
* Anche questo evento viene inviato a Kafka tramite KafkaSecurityAlertProducerService.

Entrambi gli eventi prodotti vengono inoltre registrati nei log, permettendo il monitoraggio delle operazioni svolte. In sintesi, StartupRunner consente di simulare in automatico l'invio di eventi, utile per testare il corretto funzionamento dei producer e del flusso event-driven nel contesto di sviluppo.

* 1. Package config

**Topics**

public final class Topics {  
public static final String *TRANSACTIONS* = "transactions";  
 public static final String *SECURITY\_ALERTS* = "security-alerts";public static final String *HIGH\_VALUE\_TRANSACTIONS* = "high-value-transactions";  
}

Questa classe contiene le costanti relative ai nomi dei topic Kafka utilizzati all'interno dell'applicazione. La sua funzione principale è centralizzare la definizione dei nomi dei topic, evitando errori di battitura e migliorando la manutenibilità del codice.

Essendo una classe di sola utilità, è dichiarata come final e non può essere estesa.

In particolare, definisce:

* TRANSACTIONS: rappresenta il topic principale in cui confluiscono tutti gli eventi relativi alle transazioni bancarie.
* SECURITY\_ALERTS: rappresenta il topic dedicato agli eventi di allerta di sicurezza, ad esempio accessi anomali o sospetti.
* HIGH\_VALUE\_TRANSACTIONS: rappresenta il topic specifico per le transazioni di alto valore (con importo maggiore o uguale a 1000 euro), popolato automaticamente dal filtro implementato tramite Kafka Streams.
  1. Package model

**TransactionEvent**

@Getter  
@NoArgsConstructor  
@AllArgsConstructor  
@ToString  
public class TransactionEvent {  
private String transactionId;private String accountId;private double amount;private String type;private LocalDateTime timestamp;  
}

Questa classe modella un evento di transazione bancaria, inviato o ricevuto attraverso Kafka.

* Contiene tutte le informazioni necessarie per descrivere un'operazione bancaria: ID transazione, ID conto, importo, tipo di transazione e data/ora.
* Viene utilizzata dai producer per generare messaggi Kafka e dai consumer per elaborare le transazioni ricevute.
* Le annotazioni Lombok (@Getter, @NoArgsConstructor, @AllArgsConstructor, @ToString) semplificano il codice, generando automaticamente metodi standard e costruttori.

Questa classe costituisce il formato standard dei messaggi relativi alle transazioni all'interno del sistema event-driven.

**SecurityAlertEvent**

@Getter  
@NoArgsConstructor  
@AllArgsConstructor  
@ToString  
public class SecurityAlertEvent {  
private String alertId;private String accountId;private String message;private LocalDateTime timestamp;  
}

Questa classe rappresenta un evento di allerta di sicurezza, legato a potenziali problemi o accessi sospetti al conto bancario.

* Contiene dati essenziali per descrivere un allarme di sicurezza: ID dell'allerta, ID conto, messaggio descrittivo e timestamp.
* Serve ai producer per inviare notifiche di sicurezza ai topic Kafka e ai consumer per riceverle e gestirle.
* Anche in questo caso, Lombok viene utilizzato per ridurre il boilerplate.

Definisce lo schema dati comune degli eventi di sicurezza trasmessi nella piattaforma bancaria event-driven.

* 1. Package service

**KafkaProducerService**

@Service  
@RequiredArgsConstructor  
public class KafkaProducerService {  
private final KafkaTemplate<String, Object> kafkaTemplate;  
public void sendTransaction(TransactionEvent event) {  
 kafkaTemplate.send(Topics.*TRANSACTIONS*, event.getTransactionId(), event);  
 }  
}

Questa classe fornisce un servizio di invio eventi di transazione verso Kafka. Utilizza un KafkaTemplate per serializzare e inviare oggetti TransactionEvent sul topic transactions. La chiave di ogni messaggio è rappresentata dall'ID della transazione, garantendo così la distribuzione coerente degli eventi sulle partizioni del topic. Il servizio è annotato con @Service per l'iniezione e la gestione da parte del contesto Spring Boot.

**KafkaSecurityAlertProducerService**

@Service  
@RequiredArgsConstructor  
public class KafkaSecurityAlertProducerService {  
private final KafkaTemplate<String, SecurityAlertEvent> securityAlertKafkaTemplate;  
  
 public void sendAlert(SecurityAlertEvent alert) {  
 securityAlertKafkaTemplate.send(Topics.*SECURITY\_ALERTS*, alert.getAlertId(), alert);  
 }  
}

Simile al servizio precedente, questa classe invia eventi di tipo SecurityAlertEvent sul topic security-alerts. Utilizza un KafkaTemplate parametrizzato direttamente per questo tipo di evento. Ogni allerta inviata include una chiave univoca (l'ID dell'allerta). Anche questo servizio è un componente Spring Boot, grazie all'annotazione @Service.

**KafkaConsumerService**

@Service  
@Slf4j  
public class KafkaConsumerService {  
@KafkaListener(topics = Topics.*TRANSACTIONS*)  
 public void consumeTransaction(TransactionEvent event) {  
 *log*.info("Ricevuto evento di transazione: {}", event);  
 }  
}

Questa classe implementa un consumer Kafka classico. Utilizzando l'annotazione @KafkaListener, consuma automaticamente gli eventi dal topic transactions. Gli eventi ricevuti sono di tipo TransactionEvent e vengono semplicemente loggati tramite SLF4J. Il consumer fa parte della gestione standard di Kafka in Spring Boot.

**KafkaSecurityAlertConsumerService**

@Service  
@Slf4j  
public class KafkaSecurityAlertConsumerService {  
@KafkaListener(topics = Topics.*SECURITY\_ALERTS*)  
 public void consumeAlert(SecurityAlertEvent alert) {  
 *log*.info("Ricevuto evento di sicurezza: {}", alert);  
 }  
}

Parallelamente al consumer delle transazioni, questo servizio ascolta il topic security-alerts, ricevendo eventi di tipo SecurityAlertEvent. Anche qui, la ricezione avviene tramite @KafkaListener e gli eventi sono loggati a fini dimostrativi o diagnostici.

**HighValueTransactionConsumer**

@Configuration  
@Slf4j  
public class HighValueTransactionConsumer {  
@Bean  
 public Consumer<TransactionEvent> highValueConsumer() {  
 return event -> {  
 *log*.info("[CloudStream] Transazione ad alto valore: {}", event);  
 };  
 }  
}

Questa classe rappresenta un consumer dichiarativo tramite Spring Cloud Stream. A differenza dei consumer Kafka tradizionali, qui viene definito un Consumer<TransactionEvent> come bean. Tramite configurazione, viene automaticamente connesso al topic high-value-transactions, popolato da Kafka Streams. Ogni transazione ad alto valore ricevuta viene semplicemente stampata a log. La classe usa @Configuration e non @Service, dato il suo ruolo più infrastrutturale.

* 1. Package streams

**KafkaStreamsTopology**

@EnableKafkaStreams  
@Configuration  
public class KafkaStreamsTopology {  
@Bean  
 public KStream<String, TransactionEvent> highValueTransactionsTopology(StreamsBuilder builder) {  
 JsonSerde<TransactionEvent> serde = new JsonSerde<>(TransactionEvent.class);  
  
 KStream<String, TransactionEvent> input = builder.stream(  
 Topics.*TRANSACTIONS*, Consumed.*with*(Serdes.*String*(), serde));  
  
 KTable<String, TransactionEvent> table = input.groupByKey().reduce((  
 agg, val) -> val, Materialized.*with*(Serdes.*String*(), serde));

table.toStream().filter((key, value) -> value.getAmount() >= 1000).to(  
 Topics.*HIGH\_VALUE\_TRANSACTIONS*, Produced.*with*(Serdes.*String*(), serde));  
  
 return input;  
 }  
}

Questa classe definisce la topologia Kafka Streams utilizzata nel progetto per identificare e gestire le transazioni bancarie di alto valore. Fa parte del pacchetto streams e rappresenta un componente fondamentale per l'elaborazione dei dati in streaming.

**Funzione principale**

La classe implementa una pipeline Kafka Streams che consuma eventi dal topic transactions, filtra quelli con importo >= 1000 € e li pubblica nel topic high-value-transactions. L'uso di Kafka Streams consente un'elaborazione in tempo reale degli eventi bancari.

**Dettagli di implementazione**

* È annotata con @EnableKafkaStreams e @Configuration, rendendola una configurazione Spring Boot per l'inizializzazione automatica della topologia.
* Il metodo principale è highValueTransactionsTopology() che costruisce l'intera pipeline:
  + Consuma eventi JSON di tipo TransactionEvent dal topic transactions tramite un KStream.
  + Applica un groupByKey() sulla chiave (ID transazione) e usa una KTable per memorizzare solo l'ultimo evento per ogni ID, garantendo l'idempotenza e riducendo i duplicati.
  + Filtra le transazioni con importo superiore o uguale a 1000 €.
  + Gli eventi filtrati vengono inviati sul topic dedicato high-value-transactions.
* La serializzazione e deserializzazione degli eventi avviene tramite una JsonSerde dedicata a TransactionEvent.

**Ruolo nel progetto**

Questa classe fornisce un **filtro automatico di transazioni ad alto valore**, alleggerendo il carico dei consumer. Gli eventi prodotti nel nuovo topic vengono poi consumati separatamente tramite Spring Cloud Stream. In questo modo, Kafka Streams agisce come un livello intermedio di elaborazione, migliorando la scalabilità e separazione delle responsabilità nel progetto.

1. **Strategie di resilienza**

Per garantire l'affidabilità e la continuità operativa del sistema, sono state adottate diverse strategie di resilienza. Queste soluzioni mirano a prevenire la perdita di eventi, ridurre l’impatto di eventuali crash e consentire un rapido ripristino in caso di guasti.

1. Log Kafka persistenti (KRaft) – gli eventi sono memorizzati su disco; i metadati del cluster vengono replicati tra i controller KRaft, eliminando la dipendenza da ZooKeeper.
2. Idempotenza – la topologia KafkaStreamsTopology materializza una KTable indicizzata per transactionId, scartando i duplicati prima che raggiungano i consumer Spring Cloud Stream.
3. Gestione degli offset – i consumer confermano automaticamente l’avanzamento; in caso di crash riprendono dal last‑committed offset. Nel profilo dev, auto‑offset‑reset: earliest consente di rileggere l’intero backlog.
4. State store di Kafka Streams – lo stato è salvato sul filesystem (percorso indicato in state-dir); al riavvio viene ricaricato, riducendo i tempi di recovery. Nei test locali cleanup.on.start: true assicura un ambiente pulito.

Applicativo in esecuzione:

