Introduzione

In questo progetto è stata implementata un’architettura event‑driven per la simulazione di un sistema bancario con transazioni e notifiche in tempo reale.

Il core dell’applicazione è un microservizio basato su Spring Boot, mentre Apache Kafka funge da broker di messaggi su cui viaggiano due tipi di eventi: transactions e security‑alerts.

Per l’elaborazione dei flussi in tempo reale si utilizza Kafka Streams, e Spring Cloud Stream fornisce l’astrazione di alto livello per collegare in modo reattivo e scalabile il microservizio a Kafka.

Per l’esecuzione locale del progetto si impiega Docker, che crea un ambiente di test isolato e replicabile; Kafka gira in modalità standalone con controller KRaft abilitato, eliminando la dipendenza da ZooKeeper.

Di seguito vengono presentate le tecnologie impiegate: per ciascuna descriveremo cos’è, il suo ruolo e come si integra nel sistema.

Spring Boot

Spring Boot è un framework open source sviluppato dal team Spring che estende il core di Spring Framework per semplificare la creazione e il deploy di applicazioni Java standalone e microservizi. Grazie all’autoconfigurazione riduce drasticamente il codice boilerplate necessario per avviare un’applicazione.

Al runtime il framework rileva le dipendenze presenti nel POM e configura automaticamente le componenti come Kafka. Spring Boot gestisce l’inizializzazione del contesto applicativo, registrando i bean definiti nel codice e caricando le proprietà da application.yml.

All’avvio del microservizio, il metodo SpringApplication.run(...) innesca il processo di creazione del container Spring, in cui vengono configurati sia i producer e consumer Kafka sia le topologie Kafka Streams. In questo modo, Spring Boot fornisce la base infrastrutturale su cui si appoggiano tutti gli altri componenti event-driven, consentendo di concentrare l’attenzione sulla logica di business senza doversi occupare di configurazioni ripetitive e dettagli implementativi delle infrastrutture sottostanti.

Spring Boot è un framework di base caratterizzato dalla sua rapidità di sviluppo e l’ottima integrazione con componenti esterni come Kafka. Velocizza infatti il processo di creazione di servizi back-end grazie all’autoconfigurazione e alla vasta collezione di “starter” (pacchetti di dipendenze preconfigurate, ad esempio per Kafka), garantendo applicazioni performanti e facilmente scalabili.

L’applicazione bancaria event-driven è implementata come un servizio Spring Boot standalone, che si avvia con un semplice comando ed espone la funzionalità di produzione e consumo di eventi Kafka attraverso le librerie Spring. Spring Boot gestisce la configurazione di connessione a Kafka tramite le proprietà di configurazione standard come quelle scritte nel file YAML. Inoltre, grazie al supporto nativo di Spring Boot per Spring Kafka e Spring Cloud Stream, il microservizio può produrre e consumare messaggi da Kafka senza dover scrivere boilerplate aggiuntivo: è sufficiente definire metodi annotati o bean funzionali per i consumer, e il framework si occupa di collegarli ai topic Kafka appropriati in base alla configurazione.

Apache Kafka

Apache Kafka è una piattaforma di stream processing concepita per pubblicare, memorizzare ed elaborare flussi di dati in tempo reale. In pratica Kafka funge da broker di messaggi altamente scalabile e resistente ai guasti, strutturato con dei log le applicazioni possono inviare eventi (messaggi) su topic Kafka, da cui altre applicazioni possono leggere in modo asincrono.

Kafka è progettato per gestire elevati volumi di dati con bassa latenza e alta velocità, permettendo di spostare grandi quantità di informazioni.

Kafka sostituisce i tradizionali message broker aziendali offrendo migliori prestazioni e throughput su scala orizzontale. Grazie alla persistenza su disco dei messaggi e alla replica dei dati nel cluster, garantisce durabilità e tolleranza ai guasti: i messaggi pubblicati vengono conservati e replicati su più nodi, assicurando che nessun dato vada perso anche in caso di failure di un nodo. Inoltre separa i produttori dai consumatori tramite il modello publish/subscribe e supporta consumer group concorrenti, consentendo a più istanze di consumare in parallelo messaggi da uno stesso topic. Queste caratteristiche lo rendono ideale in architetture event-driven moderne che richiedono elevata affidabilità e scalabilità.

Nel progetto Kafka svolge il ruolo centrale di bus di eventi: tutte le transazioni bancarie e notifiche di sicurezza vengono pubblicate come messaggi su topic Kafka dedicati. La generazione e l’elaborazione degli eventi avviene in maniera asincrona.

Quando si verifica una nuova transazione, il servizio Spring Boot produce un evento TransactionEvent sul topic Kafka corrispondente. Uno o più consumer, iscritti a quel topic, riceveranno l’evento e lo elaboreranno quasi in tempo reale.

Nel sistema sono presenti sia consumer standard come ad esempio un listener Kafka che sottoscrive direttamente il topic degli alert di sicurezza sia pipeline di stream processing più complesse che coinvolgono Kafka Streams e ulteriori topic intermedi.

I messaggi restano persistenti nel log Kafka finché non vengono elaborati, e tramite la replicazione e le conferme di ricezione Kafka garantisce che anche in caso di errori o ritardi lato consumer le informazioni non vadano perdute.

Kafka Streams

Kafka Streams è una libreria Java, parte del progetto Apache Kafka e offre un framework per costruire applicazioni e microservizi dedicati all’elaborazione di flussi di dati in tempo reale.

A differenza di Kafka stesso che si occupa del trasporto e storage dei messaggi, Kafka Streams opera lato client: permette a un’applicazione Spring Boot di definire trasformazioni, filtri, aggregazioni e altre operazioni sui dati in transito nei topic Kafka, il tutto utilizzando API a flusso di tipo DSL funzionale.

Consente di elaborare streaming di eventi direttamente all’interno dell’applicazione senza bisogno di cluster esterni di processing. Si integra nativamente con Kafka: un’applicazione con Kafka Streams può consumare dati da uno o più topic in input, applicare logica di processamento in-memory, e scrivere i risultati su topic di output, il tutto beneficiando delle garanzie di Kafka come ad esempio elaborazione stateful con store locali e reprocessing in caso di errori.

Il modello di programmazione di Kafka Streams offre due livelli di astrazione: una DSL ad alto livello basata su concetti di flusso KStream, KTable, ecc. per le operazioni comuni, e un’API Processor più low-level per casi avanzati. Questa flessibilità, unita alla natura leggera, rende Kafka Streams adatto a costruire microservizi streaming senza introdurre ulteriori componenti infrastrutturali.

Nel nostro sistema bancario event-driven, Kafka Streams viene utilizzato per implementare logiche di stream processing sulle transazioni in arrivo. In particolare è stata definita una *topologia* Kafka Streams che sottoscrive il topic delle transazioni e filtra gli eventi in base a determinati criteri di business, ovvero, la topologia individua le transazioni di importo ≥ 1000€ e le inoltra in tempo reale verso un topic dedicato per ulteriori notifiche.

Durante questo processo, Kafka Streams mantiene l’idempotenza usando una KTable: gli eventi di transazione sono aggregati per chiave, ovvero l’ID della transazione così da eliminare eventuali duplicati, garantendo che la stessa transazione non venga conteggiata due volte. Questa combinazione di filtro avviene in streaming man mano che i messaggi fluiscono nel topic.

Kafka Streams consente di esprimere tali trasformazioni direttamente nel codice Java, con poche configurazioni aggiuntive. La topologia Kafka Streams nel progetto è configurata e avviata automaticamente da Spring; questo significa che definendo un bean Spring @Configuration con annotazione @EnableKafkaStreams, la libreria costruisce il flusso di elaborazione e si sottoscrive ai topic Kafka appropriati all’avvio dell’applicazione.

Spring Cloud Stream

Spring Cloud Stream è un modulo del framework Spring che facilita lo sviluppo di applicazioni event-driven e reactive astratte dai dettagli del broker sottostante. Ha come obiettivo di permettere agli sviluppatori di concentrarsi unicamente sulla logica di business, eliminando la complessità del codice specifico per i diversi sistemi di messaggistica. Fornisce un modello di programmazione dichiarativo in cui si definiscono producer e consumer come funzioni o metodi Spring Bean, mentre la connessione a Kafka o ad altri broker viene gestita automaticamente attraverso componenti chiamati *Binder*.

Grazie ai binder, lo stesso codice applicativo può inviare/ricevere messaggi su diversi middleware (Kafka, RabbitMQ, ecc.) senza modifiche, semplicemente cambiando la dipendenza del binder e la configurazione. Questo rende l’applicazione sostanzialmente *broker-agnostic*: ad esempio, un consumer implementato con Spring Cloud Stream non contiene chiamate alle API Kafka, ma solo una funzione che accetta un messaggio, il resto (sottoscrizione al topic, deserializzazione, backpressure, etc.) è delegato al framework.

Inoltre supporta nativamente la programmazione reattiva basata su *Spring Cloud Function*: sviluppatori possono definire supplier, function o consumer come lambda/funzioni e, opzionalmente, utilizzare tipi reattivi per gestire stream asincroni di messaggi. Il framework si occupa di collegare queste funzioni agli endpoint di messaggistica. Un altro vantaggio chiave di Spring Cloud Stream è il supporto incorporato per partitioning e consumer grouping indipendentemente dal broker utilizzato. Ciò significa che è possibile creare più istanze parallele di una stessa applicazione consumer che si suddividono le partizioni di un topic, aumentando la scalabilità del sistema senza dover gestire manualmente la coordinazione tra processi.

Nel progetto Spring Cloud Stream è stato impiegato per implementare in modo semplice e scalabile uno dei componenti consumer. In particolare il flusso di transazioni filtrate da Kafka Streams pubblicato sul topic high-value-transactions  viene consumato da un microservizio reattivo definito con Spring Cloud Stream.

L’applicazione dichiara un consumer Spring Cloud Stream ad esempio, una funzione Consumer<Transactions> registrata come bean che il framework lega automaticamente al topic Kafka appropriato tramite il binder Kafka. Grazie a questa soluzione, il codice del consumer risulta estremamente pulito, non contiene chiamate dirette a Kafka e possiamo eseguirlo in più istanze concorrenti per gestire un alto volume di eventi.

Infatti, utilizzando i gruppi consumer di Kafka attraverso Spring Cloud Stream, il sistema può scalare orizzontalmente: più istanze del servizio Cloud Stream in esecuzione si divideranno il carico dei messaggi sul topic high-value-transactions, garantendo throughput maggiore e fault tolerance (se un’istanza si guasta, le altre continueranno a consumare le partizioni rimanenti).

Quindi Spring Cloud Stream fornisce il layer di astrazione che semplifica la connessione del codice applicativo a Kafka in modalità *reactive* e robusta. Può essere una soluzione reattiva e cloud-ready per consumare eventi. Spring Cloud Stream soddisfa i requisiti permettendo di scrivere il consumer come semplice funzione Java e di delegare al framework tutti gli aspetti di gestione del messaging (binding al topic, serializzazione del messaggio, gestione degli errori e retry, ecc.).

**Docker & Kafka**

Docker è una piattaforma software che consente di creare, eseguire e gestire applicazioni all’interno di container, ossia unità standard che includono tutto il necessario (codice, librerie, runtime) per l’esecuzione di un software in qualsiasi ambiente.

I container condividono le risorse del sistema operativo risultando più leggeri rispetto alle macchine virtuali tradizionali. Utilizzato per costruire ambienti di sviluppo e test riproducibili e per facilitare il deployment continuo dei servizi.

Nel progetto **Docker** viene utilizzato per **orchestrare in locale l’infrastruttura di Apache Kafka in modalità KRaft**. Grazie a Docker è stato possibile definire nel file docker‑compose.yml un container che avvia un singolo broker Kafka già configurato con il comando docker‑compose up -d.