Booster voor je applicatie

APACHE KAFKA STREAMS

Nee, we hebben het hier niet over een (COVID) vaccin. We zijn er allemaal wel een beetje klaar mee, juist? Maar dit gaat over een manier van denken, asynchroon (event based) en functioneel code ontwikkelen! En daarnaast is functioneel ook beter te begrijpen (als je het goed aanpakt). Als je kijkt naar wat we als developer coderen, dan verliezen we vaak de performance uit het oog. Wat nou als bepaalde stukken logica asynchroon uitgevoerd kunnen worden en ook in de organisatie gedeeld kunnen worden? Welke platformen en technieken kan ik daarvoor gebruiken? In dit artikel ga ik in op de meest gebruikte platformen en gaan we ons verdiepen in Apache Kafka Streams.

Ko Turk, Developer Blue4IT bij de Rabobank.

{ PLATFORMEN }

Tegenwoordig is er heel veel keus als je een microservice message broker ("man in the middle") zoekt. Super natuurlijk, maar wat moet je precies weten? Wat zijn de belangrijkste verschillen? Hieronder enkele (meest gebruikte) platformen:

- Google Pub Sub.
- RabbitMQ, IBM MQ, Amazon MQ, KubeMQ, in-memory Redis.
- Kafka (Confluent, Axual).

Als we hierboven kijken naar Google's PubSub, RabbitMQ of ander MQ-systeem, dan werken die met subscriptions (publish / subscribe) in plaats van met consumer groups (vergeleken met Kafka). Een consumer group is een set van consumers die luisteren op een topic, zodat berichten parallel verwerkt kunnen worden (en aardig wat sneller zijn). Er kunnen ook meerdere consumer groups zijn (op verschillende servers), wat super handig is voor een mogelijke failover (we noemen dat: fault tolerant). De consumers binnen een group weten van elkaar wanneer er een bericht gelezen is en doen dit niet nog een keer. Ten opzichte van MQ-platformen houdt Kafka een sequence-nummer bij van ieder bericht. Binnen MQ maakt de volgorde niet uit.

Berichten worden binnen Kafka opgeslagen op het filesystem (LOG data structure) en kunnen na een te definiëren aantal dagen verwijderd worden (niet verplicht). Overigens is het opslaan op disk niet per sé trager als je het vergelijkt met in-memory. Kafka voegt alleen berichten toe, en doet verder niks met het opzoeken van berichten (seeking). De verwerkingstijd kan zelfs sneller zijn.

Wat verder voordelig is bij het opslaan van berichten is dat bij een fout, het systeem nog weet waar het gebleven is. Heel erg belangrijk als je bijvoorbeeld transacties verstuurd. Je kan desnoods terug in tijd (en de berichten dus opnieuw verwerken). Binnen MQ is het fire and forget (wanneer het bericht acknowledged is). Dus maakt het niet uit of je berichten worden verwijderd wanneer ze gelezen zijn? Kies dan niet voor Kafka maar voor een MQ platform. Tenzij...

We praten hier over een manier om berichten te versturen en te

verwerken. We produceren data, en consumen het ergens anders. Handig als je informatie wil delen binnen je organisatie, zoals bijvoorbeeld transacties, fraude gevallen, orders en meer. Maar dit kan ieder publish subscribe platform ook. Wat Kafka Streams mooi maakt, is de layer boven de traditionele consumer/producer cliënt, waarbij we real-time streams (een set van events/berichten) kunnen verwerken. Wat is het verschil dan? Enkele voorbeelden:

- Je kan meerdere streams (set van data) met elkaar verbinden, zodat je joins, aggregations en windowing kan uitvoeren. Super handig als je bijvoorbeeld zaken met elkaar wil vergelijken.
- Kafka is meer expressioneel, omdat je een DSL kan gebruiken waarmee je functioneel kan programmeren (denk dan aan map, filter etc).
- Het heeft ook een imperatieve style om complexe events te verwerken.
- Het heeft een eigen testkit, zodat je makkelijk een integratietest kan opzetten in een in-memory cluster.
- Het integreert goed met databases (middels Kafka Connect).
- Het is heel erg snel in het verwerken van berichten (wellicht wel de snelste).

{ KORT SAMENGEVAT }

In het kort, gebruik Kafka vooral als je wil dat berichten niet verloren gaan (over een specifieke aan te geven periode), als je wil dat berichten opnieuw te verwerken zijn (mocht het fout gaan) en wanneer je wellicht streaming wil gebruiken. Daarnaast is Kafka 'highly available' wanneer de inrichting daarvan in orde is (zoals bij een platform provider), schaalt het mee en is de verwerkingssnelheid subliem. Hieronder gaan we nu verder in op Apache Kafka Streams.

{ MAAR EERST, EEN (EIGEN) KAFKA CLUSTER }

Zet je eigen cluster op. Er is een quickstart te vinden op https://kafka.apache.org/quickstart. Ben je meer het type 'luie developer'? Run dan de volgende docker image: https://github.com/KoTurk/Kafka/blob/main/NLJUG/magazine/kafka/PaymentEngine/docker/.

Let er wel op dat je zelf je topics en meer moet aanmaken. Liever je cluster laten beheren? Dan zijn er vendors die deze zorg uit handen nemen, zoals Confluent (marktleider) of Axual (lekker Nederlands).

{ APACHE KAFKA STREAMS! }

Er zijn enkele termen die belangrijk zijn binnen Kafka Streams, zoals:

Stream: kort samengevat een set van records (een stroom van data). In Kafka maken we gebruik van een KStream.

(K)Table: een soort van stateful stream. Deze houdt bij wat er veranderd is doormiddel van changelogs. Als je in een stream zit

en je joint met een table, dan wil je namelijk op de hoogte zijn van de laatste wijzigingen. Denk dan bijvoorbeeld aan de settings van een klant.
Topic: hier worden alle berichten opgeslagen.
Topology: een abstractie van alle stream verwerkende nodes binnen je applicatie.
Node / Processor: een topology bestaat uit allemaal nodes of processors waarover je logica kan uitvoeren, zoals een map, filter of dergelijke.
State Store: een soort van database waarin we data kunnen opslaan en opvragen.

Om uit te leggen waaruit Kafka Streams bestaat, is de volgende terminologie belangrijk. We hebben namelijk binnen Kafka Streams de beschikking over:

- Een DSL, waarin we op hoger niveau streams kunnen abstraheren, voor bijvoorbeeld topologies, tables, state stores en streams. Dit wordt ook wel een 'declaratieve' of 'functionele' DSL genoemd en is goed genoeg voor de meeste gebruikers.
- Een processor API, waarbij je iets meer mogelijkheden hebt dan bij de DSL. Hierin creëer je een processor die tegen state stores kan praten. Deze processor kan je dan ook weer in je topology gebruiken.

{ DEPENDENCIES }

Om van alle event streaming functionaliteiten gebruik te kunnen maken, voegen we de `kafka-streams` dependency toe aan de Maven of Gradle build file. Mocht je Axual of Confluent willen gebruiken? Dan moet je ook een aantal specifieke client configuraties opnemen in je code. Denk dan aan endpoints, SSL-configuratie en meer. Hiervoor zijn ook aparte clients beschikbaar.

In onze voorbeelden maken we gebruik van Kotlin en eigen (Avro) key value objecten, die we 'CustomerID' en 'Message' noemen. Gebruik een schema registry als je zelf je berichten wil definiëren. Voor meer informatie lees je de blog: https://medium.com/slalom-technology/introduction-to-schema-registry-in-kafka-915ccf06b902.

{ FUNCTIONELE STREAMING DSL (MET STREAMSBUILDER) }

Stel je voor, je moet een applicatie maken waarbij je alle transactiegegevens uit een topic ophaalt. Buiten het maken van een Spring Boot applicatie (vergeet hem niet te annoteren met @EnableKafkaStreams) zal je een topology moeten aanmaken, waarin je streams en logica worden gedefinieerd. We doen dit met de klasse `org.apache.kafka.streams.StreamsBuilder`. Het werkt een beetje hetzelfde als in de StreamsAPI van Java.

L1

```
fun create(builder: StreamsBuilder): Topology {
   builder.stream<CustomerID, Message>("transactions")
        .peek {key, value -> log.info("Incoming transaction with '$key' and '$value'")}
   return streamsBuilder.build()
}
```

L2

```
fun create(builder: StreamsBuilder): Topology {
   val transactions = builder.stream<CustomerID, Message>("transactions")
   val balances: KTable<CustomerID, Message> = builder.table("balances")

   transactions
        .peek {key, value -> log.info("Incoming transaction with '$key' and '$value'")}
        .join(balances) { transaction: Message, balance: Message ->
            JoinedMessage(transaction, balance)
        }
        .filter{ _, message -> message.transaction.amount >= message.balance.amount }
        .split()
        .branch(isPossibleFraud(), Branched.withConsumer { ks -> ks.to("fraud") })
        .defaultBranch(Branched.withConsumer { ks -> ks.to("proces") })

   return builder.build()
}
```

```
class TransactionOverviewProcessor : Processor<CustomerID, Message> {

    override fun init(context: ProcessorContext<String, String> ) {
        context.schedule(Duration.ofDays(7), PunctuationType.STREAM_TIME) {
            timestamp ->
            // Meer logica hier
   }

    override process(record: Record<CustomerID, Message>) {
        val transactions = record.value.transactions: Transactions[]
        // Meer logica hier
    }
}
```
L3

In Listing 1 maken we met behulp van de builder een stream, die we uitlezen met peek en loggen. Hierna maken de topology aan met `build()`. Het is relatief simpel, maar wat als we wat meer logica willen toevoegen? Zoals een:
Join: om een balans table te joinen (hierin staat het actuele saldo van een klant).
Filter: om alleen records te verwerken waarbij het saldo toereikend is.
Map: laten we een aparte message maken die we later in het proces kunnen gebruiken.
Split (denk dan aan Black Jack): we delen de stream in sub-streams.
Branch: deze komt na de split. Hierin geven we eerst een conditie aan, zoals 'is het een mogelijke frauduleuze transactie'?
To: dan sturen we het bericht door naar een andere topic (die kan worden afgelezen door een andere applicatie).
DefaultBranch: als bovenstaande branch niet van toepassing is, pak dan de default. Dan sturen we wederom het bericht met een `.to` naar een ander topic.

Zoals je kan zien in Listing 2 staat de `StreamsBuilder` centraal bij het maken van een topology.

Gefeliciteerd! Op dit moment ben je in staat om een werkende Kafka Streams applicatie te maken :-)

{ PROCESSOR API VOOR MEER FLEXIBILITEIT (IN HET KORT) }

Buiten de StreamsDSL is het ook mogelijk om een processor te maken (die je ook in je DSL kan gebruiken). Hierbij heb je de mogelijkheid om per bericht dingen te doen (lower level API), in plaats van op een hoger niveau, zoals de DSL. Je hebt bijvoorbeeld de mogelijkheid om periodiek acties uit te voeren wanneer er een bericht binnenkomt (met een Punctuator). Dit kan op twee manieren, met een:
- Gebeurtenis tijd (dus na x aantal tijdseenheden). Stel dan `PunctuationType.STREAM_TIME` in.
- Wandklok tijd (op een bepaald tijdstip). Ga dan voor `PunctuationType.WALL_CLOCK_TIME`.

Dit stellen we in de `init()` methode in. In de processor bestaat ook een `process()` methode (ja echt). Hierin definieer je de logica per bericht. In de `close()` stop je alle resources die je gebruikt. Een voorbeeld als we bijvoorbeeld per week een transactie overzicht willen, vind je in Listing 3.

{ TOT SLOT }

Wat ik als laatste tip wil meegeven, is dat je bij het gebruik van Streams denkt aan het maken van metrics. In een streaming omgeving is het namelijk moeilijk om berichten te volgen (je hebt vaak geen trace ID of iets dergelijks). Het is mooi om in een dashboard te zien wanneer er zaken mis gaan en dan merk je dat logging toch wat lastiger te volgen is. Lees vooral eens het artikel 'Word een superheld met Micrometer' in het magazine (2022-1) om meer te weten te komen over het maken van juiste monitoring.

Met Apache Kafka Streams zijn de mogelijkheden eindeloos! We hebben bij lange na niet alles kunnen bespreken, maar het belangrijkste is dat je nu aan de slag kan. Heel veel succes! Kijk voor een werkend geheel eens op https://github.com/KoTurk/Kafka