Začíname s Apache Kafka

Obsah

1. Ingrediencie 2
2. Spúšťame Kafku 3
   2.1. Docker Compose 3
3. Práca s pluginom Kafka pre IntelliJ IDEA 4
4. Producenti a konzumenti 6
   4.1. Producentský modul 6
   4.2. Konzumentský modul 8
   4.3. Viacerí konzumenti a rebalans 9
5. Architektonické vzorce 12
   5.1. Point to Point 12
   5.2. Publish-subscribe 12
6. Zlyhávajúci konzumenti 14
7. Podpora záznamov vo formáte JSON 16
   7.1. Producent a JSON 17
   7.2. Konzument a JSON 17

Apache Kafka je mnoho vecí:

- distribuovaná platforma pre streaming dát
- message broker ako prostredník pre výmenu správ medzi komponentami distribuovaného systému

Podrobnosti nájdeme v slajdoch.

1. Ingrediencie

- Docker — pre spustenie Kafky
- grafické nástroje
  - Conduktor — grafický nástroj pre Kafku
  - alebo Kafka plugin pre IntelliJ IDEA Ultimate
- Spring Boot — pre rýchle vytvorenie producentov a konzumentov
- IntelliJ IDEA — IDE pre kód v Jave

2. Spúšťame Kafku

Kafku je najjednoduchšie spustiť pomocou nástroja **Docker**, resp. **Docker Compose**, čím si ušetríme množstvo konfigurácie.

Kafka 3.3 (október 2022) si vystačí s jediným kontajnerom.

2.1. Docker Compose

1. Stiahnime si infraštruktúrny súbor `docker-compose.yml`.
2. Spustime dockerovský kontajner:

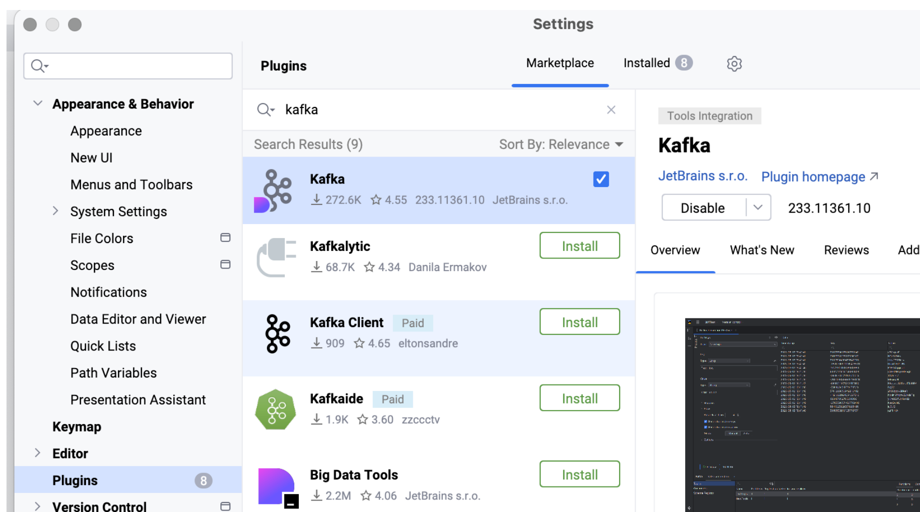
   ```
   docker-compose up
   ```

3. Spustí sa Kafka v Dockeri a v logu uvidíme úspech:

   ```
   kafka-kafka-1: [2023-11-10 15:52:54,884] INFO
   [KafkaRaftServer nodeId=1] Kafka Server started
   (kafka.server.KafkaRaftServer)
   ```

3. Práca s pluginom Kafka pre IntelliJ IDEA

Do IntelliJ IDEA Ultimate Edition stiahneme Kafka plugin



Pozor, používame oficiálny plugin od JetBrains s.r.o.

Z menu *View* > *Tool Windows* > *Kafka* zobrazíme panel pre Kafku.

Pridáme nové pripojenie s použitím dokumentácie.

Configuration Source

Custom

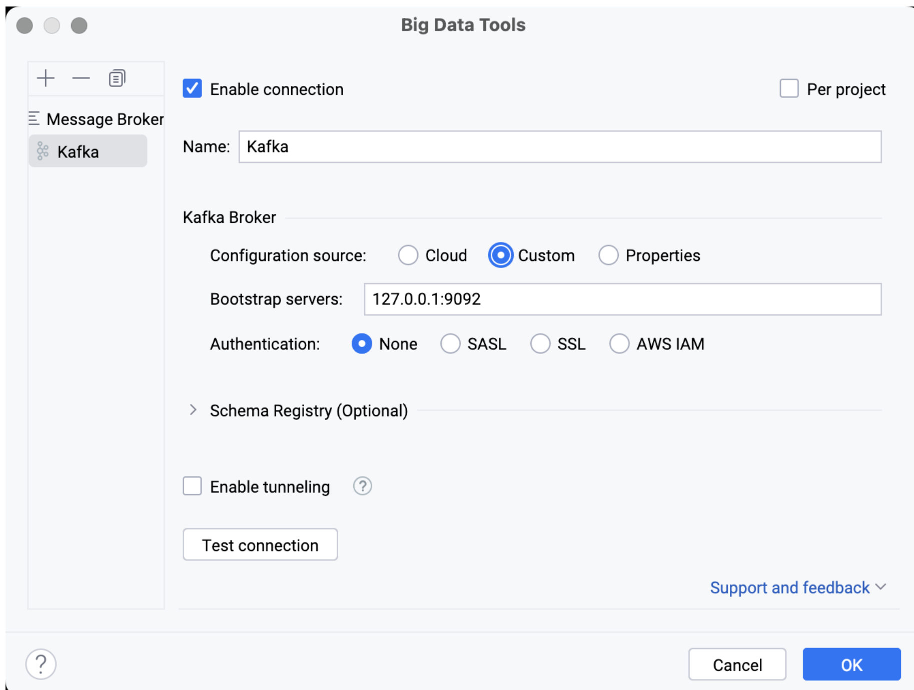
Bootstrap servers

127.0.0.1:9092

Authentication

None

Otestujeme pripojenie a následne ho vytvoríme.


Big Data Tools
Message Broker
Kafka
Enable connection
Per project
Name:
Kafka
Kafka Broker
Configuration source:
Cloud
Custom
Properties
Bootstrap servers:
127.0.0.1:9092
Authentication:
None
SASL
SSL
AWS IAM
Schema Registry (Optional)
Enable tunneling
Test connection
Support and feedback
Cancel
OK

4. Producenti a konzumenti

Producentov a konzumentov vytvoríme pomocou Spring Bootu.

V IntelliJ IDEA vytvorme nový prázdny projekt (**File | New | Empty project**), do ktorého dodáme dva moduly (producentský a konzumentský).

4.1. Producentský modul

4.1.1. Vytvorenie modulu

Vytvorme nový modul (**File | New | Module**) typu **Spring Initializr**. Vyplňme názov skupiny (*Group*) a artefakt nazvime napríklad `producer`. V dialógovom okne následne vyberme z palety Spring Bootu súčasť **Spring for Apache Kafka** (sekcia **Messaging**).

Alternatívne využime projekt https://start.spring.io/, kde si vieme vybrať súčasti, vygenerovať projekt v balíčku ZIP, stiahnuť a následne importovať.

Najdôležitejšou závislosťou je `spring-kafka`:

```
implementation("org.springframework.kafka:spring-kafka")
```

4.1.2. Konfigurácia modulu

Náš broker Kafky beží na lokálnom stroji `localhost` na štandardnom porte 9092, na čo sa spolieha aj knižnica pre Spring Boot. V bežnom prípade nemusíme robiť žiadnu konfiguráciu.

Ak by to tak nebolo — napríklad kvôli inak nastavenému exponovanému portu v Dockeri — musíme to nakonfigurovať.

V súbore `src/main/resources` existuje konfiguračný súbor pre Spring Boot `application.properties`, do ktorého uveďme:

```
spring.kafka.producer.bootstrap-servers=localhost:9092
```

4.1.3. Kód modulu

Na komunikáciu s Kafkou sa využíva `KafkaTemplate`, ktorú si môžeme nechať autowirenúť / injectnúť do triedy aplikácie cez konštruktor

```
class CarProducer(private val kafka: KafkaTemplate<String,
String>) {
    ...
}
```

Trieda má dva generické parametre: pre kľúče (*keys*) a hodnoty (*values*). V tomto prípade budeme posielať v záznamoch reťazce (druhý generický typ).

Vďaka Spring Bootu je mnoho vecí už nakonfigurovaných a sústredíme sa len na to podstatné — odosielanie správ.

Ukážka metódy, ktorá periodicky odosiela správy:

```
public void publish() {
    kafka.send("payment", "Paid some money " + Instant.
now());
}
```

Metóda `send` potrebuje dva parametre: názov *topicu* a samotný záznam (*správu*). Keďže Kafka považuje záznamy za polia bajtov, treba sa postarať o prevod, čo zabezpečí Spring Boot a jeho integrácia s Kafkou.

4.1.4. Spustenie producenta

Keďže aplikácia v Spring Boote má štandardnú metódu `main`, spustíme ju triviálne. Pripojí sa k brokeru a vypublikuje doň správu do topicu `payment`.

Broker je štandardne pripravený automaticky vytvárať topicy pri prvom publikovaní správy.

V ukážkovom kóde na GitHube uvidíme periodické zasielanie

správy.

4.1.5. Kontrola v IntelliJ IDAE

V IntelliJ IDEA môžeme zistiť, že vznikol topic `payment` a produkujú sa doňho správy: počet (*Message Count*) postupne narastá.

4.2. Konzumentský modul

4.2.1. Príprava a konfigurácia

Podobne ako pri konzumentovi vytvorme nový modul (**File | New | Module**) typu **Spring Initializr**. Vyplňme názov skupiny (*Group*) a artefakt nazvime napríklad `producer`. V dialógovom okne následne vyberme z palety Spring Bootu súčasť **Spring for Apache Kafka** (sekcia **Messaging**).

Ak používame Kafku na lokálnom stroji, teda na `localhost`-e a štandardnom porte 9092, nemusíme robiť nič. V opačnom prípade podobne nakonfigurujme cestu k bootstrapovému serveru v `application.properties` identickým spôsobom: do `application.properties` dajme

```
spring.kafka.consumer.bootstrap-servers=localhost:9092
```

4.2.2. Kód konzumenta

Kód konzumenta využíva anotáciu `@KafkaListener`:

```
@KafkaListener(topics = ["payment"]) ❶
fun consumePayment(payment: String) {
    logger.info("Consuming {}", payment) ❷
}
```

❶ Uvedieme *topic*, z ktorého budeme konzumovať.

❷ Spring Boot sa automaticky postará o prevod poľa bajtov z zázname na reťazec.

Skupinu konzumentov (*Consumer Group*) uvedieme v súbore `application.properties`

```
spring.kafka.consumer.group-id=bank-branch ❶
```

❶ Vytvoríme skupinu s názvom `bank-branch`. Názov skupiny si môžeme zvoliť podľa ľubovôle.

4.2.3. Spúšťame konzumenta

Konzumenta spustíme obvyklým spôsobom, cez spustenie hlavnej metódy. Sledujme, ako prijíma údaje z topicu.

4.2.4. Konzumenti v IntelliJ IDEA

V paneli *Kafka* uvidíme, že v sekcii *Consumer Groups* pribudla skupina (*Consumer Group*) s názvom `bank-branch`, ktorá má jediného člena.

4.3. Viacerí konzumenti a rebalans

Vytvorme teraz v IntelliJ IDEA ešte jednu konfiguráciu pre druhého konzumenta. Zvoľme z menu **Run | Edit Configurations**, vyberme z ľavého zoznamu v sekcii **Spring Boot** existujúcu konfiguráciu pre konzumenta a duplikujme ju. Nazvime ju odlišným spôsobom — napr. `Consumer #2`.

Spusťme teraz oboch konzumentov, najprv jedného a potom druhého.

Pri druhom konzumentovi uvidíme v jeho logu hlášku indikujúcu partíciu, na ktorú sa pripojil.

Log druhého konzumenta

```
2023-11-14 17:16:40.515  INFO 11741 --- [ntainer#0-0-C-1]
o.s.k.l.KafkaMessageListenerContainer    : bank-branch:
partitions assigned: [payment-0]
```

Zároveň uvidíme, že prvému konzumentovi bola odobratá partícia:

Log prvého konzumenta

```
2023-11-14 17:16:40.496  INFO 11739 --- [ntainer#0-0-C-1]
o.s.k.l.KafkaMessageListenerContainer    : bank-branch:
partitions assigned: []
```

Inak povedané, nevieme zabezpečiť distribuovanie práce.

4.3.1. Zvýšenie počtu partícií

V Kafke je možné dodatočne navyšovať počet partícií v topicu (nie však znižovať!).

Počet partícií vieme navýšiť cez príkazový riadok, konkrétne cez skript `kafka-topics`, ktorý je k dispozícii v dockerovom kontajneri.

Zistíme si názov kontajnera pre kafku:

```
docker ps
```

Uvidíme:

```
CONTAINER ID   IMAGE                           COMMAND
CREATED              STATUS              PORTS
NAMES
330130b1c4d5   confluentinc/cp-kafka:7.5.1
"/etc/confluent/dock…"   About an hour ago   Up About an
hour   0.0.0.0:9092->9092/tcp   kafka-kafka-1
```

Spustíme skript:

```
docker exec -it 330130b1c4d5 kafka-topics --bootstrap-server
localhost:9092 --alter --topic payment --partitions 3
```

V paneli *Kafka* aktualizujme informácie o topicu, a uvidíme, že pribudli nové partície.

Ak nám beží konzument, uvidíme ako sa záznamy distribuujú medzi partície.

Kafka distribuuje správy primeraným spôsobom. Konkrétny mechanizmus závisí od verzie Kafky, nastavení producenta či od nastavenia kľúča záznamu. V tomto príklade sa môže stať, že záznamy poputujú do jedinej partície.

Konzumentov nemusíme zastavovať, pretože rovno uvidíme, že nastal

rebalance a podľa okolností jeden konzument vyfasoval dve partície a druhý konzument jednu partíciu.

Pohrajte sa so zastavovaním konzumentov a sledujte, ako konzumujú správy. Za štandardných okolností bude pri dvoch konzumentoch rovnomerné rozdelenie správ, približne na striedačku.

5. Architektonické vzorce

Dva najvšeobecnejšie architektonické vzorce sú *point to point* a *publish-subscribe*.

point to point
: vo všeobecnosti sa vytvorí front (*queue*), z ktorého konzumenti vyberajú správy, pričom o ne súperia.

publish-subscribe
: konzumenti sa prihlásia na odber správ z témy (*topic*) a každá správa sa doručí každému prihlásenému konzumentovi.

5.1. Point to Point

Topic potrebujeme rozbiť do toľkých partícií, koľko plánujeme konzumentov v jednej skupine konzumentov *consumer group*.

Pre tri partície v *topicu* podporujeme najviac 3 aktívnych konzumentov v spoločnej skupine *consumer group*.

Pre 2 konzumentoch v spoločnej skupine bude 1 konzument prijímať správy z jednej partície a 1 konzument z dvoch partícií.

5.2. Publish-subscribe

Vysielanie správ pre viacerých konzumentov zabezpečíme cez viacero skupín *consumer group*.

Dve skupiny konzumentov čítajú kópie správ z topicu. Prirodzene, konzumenti v skupinách sa prerozdelia podľa počtu partícií.

V Spring Boote môžeme názov skupiny konzumentov:

- v nastavení `spring.kafka.consumer.group-id` v súbore `application.properties`

- alebo určiť v atribúte `groupId` pri konkrétnom konzumentovi

```
@KafkaListener(topics = ["car"], groupId = "${groupId:car-group1}") (1)
```

(1) V atribúte používame premennú `${…}` s implicitnou hodnotou. Spring zapojí konzumenta do skupiny podľa vlastnosti (*property*) `groupId` a ak taká vlastnosť neexistuje, použije implicitný názov skupiny konzumentov `car-group1`.

Build and run

java 17 SDK of 'consumer' module

com.github.novotnyr.consumer.ConsumerApplication

Press ⌥ for field hints

Active profiles:

Comma separated list of profiles

Environment variables: GROUPID=two

Všimnime si, ako v konfigurácii nastavíme názov druhej skupiny na `two`, pričom použijeme premennú prostredia `GROUPID`, čo sa namapuje na vlastnosť `groupId`. Toto mapovanie zariadi Spring Boot.

6. Zlyhávajúci konzumenti

Konzument môže vyhodiť výnimku a tým reprezentovať zlyhanie.

Štandardné správanie v Spring Kafke 2.8 a 2.9 zoberie typickú výnimku a:

- pokúsi sa o 9 opakovaní
- s odstupom 0 sekúnd
- ak klient zlyhá, pokračuje sa ďalším offsetom

Uvidíme napríklad:

```
2023-11-15 22:36:27.783 ERROR 18687 --- [ntainer#0-0-C-1]
o.s.k.l.KafkaMessageListenerContainer    : Error handler
threw an exception

org.springframework.kafka.KafkaException: Seek to current
after exception; nested exception is
org.springframework.kafka.listener.ListenerExecutionFailedEx
ception: Listener method 'public void
com.github.novotnyr.kafka.KafkaConsumer.consume(org.apache.k
afka.clients.consumer.ConsumerRecord<java.lang.String,
java.lang.String>)' threw exception; nested exception is
java.lang.IllegalArgumentException; nested exception is
java.lang.IllegalArgumentException
```

V aplikačnom kontexte vieme definovať bean s vlastným nastavením pre opakovanie:

```
@Bean
fun kafkaErrorHandler(): DefaultErrorHandler {
    return DefaultErrorHandler(FixedBackOff(5000L, 2L))
```

```
}
```

Voliteľne môžeme nastaviť:

recoverer
: zavolá sa, ak sa potrebujeme zotaviť z chyby, po vyčerpaní možností

not retryable exception
: výnimky, ktoré sa považujú za nezotaviteľné, teda správa sa rovno „zahodí“ a zavolá sa *recoverer*.

```
@Bean
fun kafkaErrorHandler(): DefaultErrorHandler {
    val recoverer = ConsumerRecordRecoverer { record,
exception -> System.err.println(exception) } ❸
    return DefaultErrorHandler(recoverer, FixedBackOff(
5000L, 2L)).apply { ❷
        addNotRetryableExceptions(IllegalArgumentException:
:class.java) ❶
    } ❷
}
```

❶ Výnimku `IllegalArgumentException` považujeme za nezotaviteľnú.

❷ Obsluha chýb sa pokúsi zotaviť z výnimky opakovaním čítania správy z offsetu, každý 5 sekúnd, najviac 2 pokusy.

❸ Ak sa pokusy vyčerpajú alebo ak je výnimka nezotaviteľná, dostaneme správu na štandardný výstup.

7. Podpora záznamov vo formáte JSON

Kafka dokáže posielať záznamy vo formáte JSON. Keďže pre ňu je záznam pole bajtov, o konverziu sa musí postarať producent (odosielateľ) aj konzument (prijímateľ).

Na to potrebujeme:

- Spring Boot
- deklarovať objekt pre správy
- dodať štartér `spring-boot-starter-json`, ktorá zavlečie do projektu knižnicu Jackson.
- špeciálne pre Kotlin dodajme knižnicu `jackson-module-kotlin`, ktorá zavedie podporu pre kotlinovské dátové triedy (*data class*)
- nastaviť serializácie záznamov do JSONov (pre producentov), resp. z JSONov (pre konzumentov)

Deklarujme objekt pre záznam:

```
data class Car(var make: String, var model: String) {
    override fun toString(): String {
        return "$make $model"
    }
}
```

Dôležité je poznačiť si balíček (*package*), v ktorom sa nachádza táto trieda. Tá sa musí nachádzať aj v producentovi, aj v konzumentovi, a to v rovnakom balíčku.

Do producenta i do konzumenta dodajme závislosť na knižniciach pre JSON:

build.gradle.kts

```
implementation("org.springframework.boot:spring-boot-
starter-json")
implementation("com.fasterxml.jackson.module:jackson-module-
kotlin:2.15.2")
```

7.1. Producent a JSON

V producentovi nastavíme serializáciu hodnôt (*values*) v záznamoch.

application.properties

```
spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.Json
Serializer
```

Ak chceme posielať autá, necháme si do producenta nadrôtovať objekt `KafkaTemplate` s príslušným generickým typom `KafkaTemplate<String, Car>`.

```
class CarProducer(private val kafka: KafkaTemplate<String,
Car>) {
    ...
}
```

Zrazu môžeme do topicu posielať autá ako objekty. Serializér `JsonSerializer` sa prostredníctvom knižnice Jackson postará o prevod na JSON.

```
kafka.send("car", car)
```

7.2. Konzument a JSON

Konzument musí obsahovať v zdrojových kódoch takú istú triedu `Car` v takom istom balíčku ako producent.

Konzumentovi nastavíme deserializáciu záznamov z JSONov na objekty `Car`.

application.properties

```
spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.Js
onDeserializer ❶
spring.kafka.consumer.properties.spring.json.trusted.package
s=com.github.novotnyr.car ❷
```

❶ Nastavíme *deserializátor*.

❷ Balíček, v ktorom sa nachádzajú triedy na strane konzumenta, ktoré sa považujú za dôveryhodné pre deserializáciu z JSONu.

Konzument potom dokáže prijímať objekty:

```
@KafkaListener(topics = ["car"])
fun consumeCar(car: Car) {
    //...
}
```