ChatGPT

Интеграция Apache Kafka в Spring Boot для отправки сообщений при сохранении Payment

Развертывание Kafka через Docker (локальная разработка)

Для локального запуска Kafka кластера удобнее всего воспользоваться Docker. Kafka зависит от сервиса Zookeeper, поэтому мы поднимем два контейнера: один с Zookeeper и один с Kafka-брокером. Ниже приведён пример файла `docker-compose.yml` для развёртывания Kafka локально:

```
version: '3.8'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.0.1
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181

  kafka:
    image: confluentinc/cp-kafka:7.0.1
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

Что делает эта конфигурация:

- `zookeeper`: Запускает контейнер Zookeeper на порту 2181, необходимый для координации Kafka.
- `kafka`: Запускает контейнер Kafka, который зависит от Zookeeper (параметр `depends_on` гарантирует, что Zookeeper стартует первым).
- Переменная среды `KAFKA_ZOOKEEPER_CONNECT` указывает Kafka, где найти Zookeeper (в данном случае – по адресу `zookeeper:2181`, т.е. имя сервиса Zookeeper и его порт) [1].
- `KAFKA_ADVERTISED_LISTENERS` задаёт адрес и порт, по которому Kafka-брокер будет доступен клиентам. Здесь мы настраиваем обычное (PLAINTEXT) соединение на

`localhost:9092` [1]. Это важно, чтобы приложения, работающие вне Docker-контейнера, могли подключиться к Kafka через `localhost:9092`.

- `KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1` – устанавливает фактор репликации для служебных топиков Kafka равным 1, так как у нас всего один брокер (иначе Kafka может не запуститься, ожидая большего числа брокеров) [1].
- `KAFKA_BROKER_ID: 1` – задаёт идентификатор брокера (для одного брокера можно оставить `1`).

Создав этот `docker-compose.yml` в корне проекта, запустите Docker Compose командой:

```
docker-compose up -d
```

Это скачает необходимые образы и запустит контейнеры в фоновом режиме. После запуска убедитесь, что Kafka слушает порт 9092 (например, командой `docker-compose ps` или просмотром логов с `docker-compose logs -f kafka`). Теперь у вас запущен Kafka-брокер, доступный по адресу `localhost:9092`, готовый принимать и отдавать сообщения.

Добавление зависимостей Spring Boot для Kafka

Прежде всего, в проект Spring Boot необходимо добавить зависимость **Spring for Apache Kafka**. Если вы используете Maven, откройте `pom.xml` и добавьте зависимость:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>3.0.0</version> <!-- используйте актуальную версию -->
</dependency>
```

Если вы пользуетесь Spring Boot BOM, то можно указать только `spring-kafka` без версии – возьмётся совместимая версия. Для Gradle аналогично добавляется зависимость `implementation 'org.springframework.kafka:spring-kafka:3.0.0'`. Подключение этой зависимости позволит воспользоваться автоматической конфигурацией Kafka в Spring Boot [2].

Кроме того, убедитесь, что у вас присутствует Jackson (библиотека для работы с JSON). Обычно `spring-boot-starter-web` уже включает Jackson. Если же веб-зависимость не подключена, добавьте зависимость `com.fasterxml.jackson.core:jackson-databind` той же версии, что используется в Spring Boot, поскольку JSON-сериализация сообщений Kafka будет выполняться через Jackson [3].

Настройка Kafka Producer в Spring Boot

После подключения зависимости, необходимо сконфигурировать продюсер Kafka в вашем приложении. Это включает две части: настройки в файле свойств (application.yml) и конфигурационный класс Java, определяющий бины продюсера (ProducerFactory, KafkaTemplate и т.д.).

Настройки в application.yml

Добавьте в `src/main/resources/application.yml` следующие настройки:

```
spring:
  kafka:
    bootstrap-servers: "localhost:9092"
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
```

Что означают эти настройки:

- `spring.kafka.bootstrap-servers` – адрес Kafka-брокера. Поскольку мы запускаем локально, указываем `localhost:9092` (порт 9092, как сконфигурировано в Docker). Это позволяет приложению подключиться к брокеру Kafka [4].
- `spring.kafka.producer.key-serializer` – класс-сериализатор для ключа сообщений. Мы используем `StringSerializer`, чтобы, если понадобится, ключи сообщений были в текстовом формате (String) [5]. В простейшем случае можно и вовсе не задавать ключ (он может быть `null`), но сериализатор всё равно должен быть указан.
- `spring.kafka.producer.value-serializer` – класс-сериализатор для значения сообщения. Указываем `JsonSerializer`, чтобы объекты автоматически конвертировались в JSON перед отправкой в Kafka [5]. Этот сериализатор использует библиотеку Jackson внутри себя для преобразования объекта в JSON-байты [3].

Spring Boot автоматически прочитает эти настройки и подготовит необходимые свойства для Kafka-производителя. Однако, нам всё равно нужно объявить бины продюсера в конфигурации, либо воспользоваться автонастройкой. Мы рассмотрим явную конфигурацию для наглядности.

Java-конфигурация Kafka Producer (ProducerFactory и KafkaTemplate)

Создадим класс конфигурации, например `KafkaProducerConfig`, пометив его аннотацией `@Configuration`. В нём определим бины для ProducerFactory, KafkaTemplate и, при необходимости, для Kafka-топика:

```
@Configuration
public class KafkaProducerConfig {

    @Bean
    public ProducerFactory<String, Payment> producerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        return new DefaultKafkaProducerFactory<>(props);
    }
```

```
    @Bean
    public KafkaTemplate<String, Payment> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }

    // Опционально: создаём топик при запуске приложения
    @Bean
    public NewTopic paymentsTopic() {
        return
TopicBuilder.name("payments").partitions(1).replicas(1).build();
    }
}
```

Пояснение конфигурации:

- В `producerFactory()` мы создаём фабрику продюсеров Kafka с нужными настройками. Здесь мы программно задаём те же свойства, что и в YAML: адрес брокера, сериализатор ключей (StringSerializer) и сериализатор значений (JsonSerializer) [5]. Благодаря JsonSerializer значения типа `Payment` будут автоматически преобразовываться в JSON при отправке.
- Бин `KafkaTemplate<String, Payment>` создаётся на основе ProducerFactory. **KafkaTemplate** — это шаблон (обёртка), предоставляемый Spring, упрощающий отправку сообщений в Kafka; мы будем использовать его для отправки сообщений в нужный топик [6].
- Метод `paymentsTopic()` показывает, как создавать топик через Spring. Бин типа `NewTopic` с именем **payments** заставит Spring Boot при запуске проверить наличие топика и создать его, если его ещё нет [7]. Мы задаём 1 раздел (partition) и фактор репликации 1 (так как у нас единственный брокер). Этот шаг необязателен, если у вас включено автоматическое создание топиков на стороне Kafka, но является хорошей практикой явно создавать нужные топики из приложения.

Обратите внимание: Если вы указали настройки продюсера в `application.yml` (как выше), можно не дублировать `BOOTSTRAP_SERVERS_CONFIG` в коде, а вместо этого воспользоваться объектом `KafkaProperties`, предоставляемым Spring Boot. В нашем примере для простоты значения указаны явно, но вы можете инъецировать `KafkaProperties` и вызвать `kafkaProperties.buildProducerProperties()` для получения настроек из application.yml [8].

Теперь приложение сконфигурировано для отправки сообщений в Kafka. При запуске Spring Boot создаст `KafkaTemplate` и свяжется с брокером Kafka на `localhost:9092`.

JSON-сериализация сущности `Payment`

Наша задача – отправлять в Kafka сообщения с данными платежа (`Payment`) в формате JSON. Благодаря указанной конфигурации `JsonSerializer` Spring Kafka сам будет выполнять сериализацию объекта в JSON. **JsonSerializer** использует Jackson для преобразования Java-объекта в последовательность байт JSON [3]. Это означает, что объект `Payment` будет автоматически конвертирован в JSON при вызове `kafkaTemplate.send(...)`.

Чтобы сериализация прошла успешно, убедитесь, что класс `Payment` соответствует требованиям Jackson: у него есть публичные геттеры/сеттеры для полей или аннотации Jackson, и он не содержит рекурсивных ссылок (например, в случае связей @OneToMany/@ManyToOne — такие поля можно пометить как `@JsonIgnore` или использовать DTO для отправки). Пример упрощённой сущности `Payment`:

```
@Entity
public class Payment {
    @Id
    private UUID id;
    private BigDecimal amount;
    private String method;
    // ... конструкторы, геттеры и сеттеры ...
}
```

Допустим, `Payment` содержит информацию о сумме платежа, способе оплаты и пр. При сохранении такого объекта в БД мы хотим сразу отправить его же в Kafka-топик **payments** в виде JSON-сообщения.

Отправка сообщения в Kafka после сохранения сущности

Осталось реализовать логику, которая будет отправлять сообщение в Kafka сразу после того, как мы сохранили `Payment` в базе данных. Обычно это делают на уровне сервисного слоя. Например, у нас есть `PaymentService` с методом создания/сохранения платежа:

```
@Service
public class PaymentService {

    @Autowired
    private PaymentRepository paymentRepository;

    @Autowired
    private KafkaTemplate<String, Payment> kafkaTemplate;

    private static final String TOPIC = "payments";

    public Payment createPayment(Payment payment) {
        // Сохранение платежа в базе данных
        Payment savedPayment = paymentRepository.save(payment);
        // Отправка сообщения в Kafka-топик
        kafkaTemplate.send(TOPIC, savedPayment);
        return savedPayment;
    }
}
```

Здесь после сохранения через `JpaRepository` мы вызываем `kafkaTemplate.send("payments", savedPayment)`. Это поместит событие в Kafka-топик

payments, используя настроенный JSON-сериализатор – в топик уйдёт JSON, содержащий поля сохранённого платежа.

Spring Boot автоматически сконфигурированный KafkaTemplate выполняет эту отправку асинхронно. Вы можете также получить `ListenableFuture` от метода `send` для обработки результата отправки или ошибок, но в простом случае это не обязательно. Главное – теперь каждое сохранение Payment будет сопровождаться отправкой сообщения в Kafka.

Использование событий (опционально)

В приведённом выше подходе есть нюанс: операция сохранения в базу данных и отправка в Kafka не связаны общей транзакцией. Если сохранение в БД по каким-то причинам откатится (например, выбросится исключение при валидации или проблемах с соединением), то сообщение в Kafka уже могло уйти, что приведёт к рассинхрону данных.

Чтобы избежать такой ситуации, можно использовать событийный подход. Spring позволяет публиковать и слушать события после завершения транзакции. Например, вы можете опубликовать событие после сохранения Payment и обработать его в слушателе с аннотацией `@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)`. Такой слушатель будет вызываться только если транзакция по сохранению успешно зафиксирована, и уже внутри него вы вызовете отправку сообщения в Kafka [9]. Это гарантирует, что в Kafka попадут только успешно сохранённые в БД платежи (хотя всё равно следует обрабатывать потенциальные ошибки отправки в Kafka).

Для простоты, если транзакционная целостность не критична, можно оставить отправку сразу после `save()`, как показано в сервисе выше. Но в более надёжных сценариях стоит рассмотреть шаблон **Outbox** или упомянутый событийный механизм для гарантированной доставки сообщений только по завершению транзакций.

Пример итоговой реализации

Подводя итог, интеграция Kafka в Spring Boot для задачи отправки сообщений о платежах включает следующие шаги:

1. **Запуск Kafka**: Развернули Kafka локально через Docker (Kafka + Zookeeper) [1].
2. **Конфигурация приложения**: Добавили зависимость Spring Kafka [2] и прописали настройки Kafka (адрес брокера, сериализаторы) в `application.yml`.
3. **Kafka Producer**: Создали конфигурационный класс `KafkaProducerConfig` для продюсера, определив `ProducerFactory`, `KafkaTemplate` и топик **payments** [5] [7].
4. **JSON-сериализация**: Настроили `JsonSerializer` для значений, благодаря чему объекты `Payment` автоматически превращаются в JSON при отправке [3].
5. **Отправка сообщений**: Реализовали сервис, который при сохранении `Payment` в базу сразу отправляет сообщение в Kafka-топик с тем же объектом [9] (в JSON формате).

Следуя этому гайду, вы получите полноценную интеграцию: при вызове метода сохранения платежа (например, через REST-контроллер или другой сервис) запись попадёт в базу данных **и** событие с данными платежа отправится в Kafka. Впоследствии другой микросервис или компонент, подписанный на Kafka-топик **payments**, сможет получить эти данные и обработать их в режиме реального времени.

---

[1] Spring Boot with Apache Kafka Using Docker Compose: A Step-by-Step Tutorial
https://www.javaguides.net/2024/05/spring-boot-with-apache-kafka-using-docker-compose.html

[2] [3] [5] Apache Kafka и Spring Boot: лёгкая интеграция / Хабр
https://habr.com/ru/companies/slurm/articles/772818/

[4] [7] Apache Kafka Support :: Spring Boot
https://docs.spring.io/spring-boot/reference/messaging/kafka.html

[6] [8] Spring Boot app with Apache Kafka in Docker container
https://devapo.io/blog/technology/kafka-in-spring-boot-on-docker/

[9] java - Good practice when using kafka with jpa - Stack Overflow
https://stackoverflow.com/questions/51052406/good-practice-when-using-kafka-with-jpa/51055336