Урок 17. Основы Kafka

Содержание

Чему вы научитесь на этом уроке?

- Основам архитектуры распределенного брокера сообщений Kafka
- Взаимодействию с Kafka из Spring Boot приложения

Шаг 0. Структура проекта

```
payment-service/
├── pom.xml
├── payment-service-app/
│   ├── pom.xml
│   ├── Dockerfile
│   ├── docker-compose.yml
│   └── src/
│       ├── main/
```

```
│   │   ├── java/com/iprody/payment/service/app/
│   │   │   ├── PaymentServiceApplication.java
│   │   │   ├── security/
│   │   │   │   ├── SecurityConfig.java
│   │   │   │   └── KeycloakRealmRoleConverter.java
│   │   │   ├── controller/
│   │   │   │   └── PaymentController.java
│   │   │   ├── persistence/
│   │   │   │   ├── entity/
│   │   │   │   │   ├── Payment.java
│   │   │   │   │   └── PaymentStatus.java
│   │   │   │   ├── PaymentRepository.java
│   │   │   │   ├── PaymentFilter.java
│   │   │   │   ├── PaymentSpecifications.java
│   │   │   │   └── PaymentFilterFactory.java
│   │   │   ├── dto/
│   │   │   │   └── PaymentDto.java
│   │   │   ├── mapper/
│   │   │   │   ├── PaymentMapper.java
│   │   │   │   └── XPaymentAdapterMapper.java
│   │   │   ├── services/
│   │   │   │   └── PaymentService.java
│   │   │   └── async/
│   │   │       ├── Message.java
│   │   │       ├── AsyncListener.java
│   │   │       ├── AsyncSender.java
│   │   │       ├── XPaymentAdapterRequestMessage.java
│   │   │       ├── XPaymentAdapterResponseMessage.java
│   │   │       └── kafka/
│   │   │           ├── KafkaXPaymentAdapterRequestSender.java
│   │   │           └──
KafkaXPaymentAdapterResultListenerAdapter.java
│   │   └── resources/
│   │       ├── application.yml
│   │       ├── application-kafka.yml
│   │       └── logback-spring.xml
│   └── test/
│       ├── java/com/iprody/payment/service/app/
│       │   ├── AbstractPostgresIntegrationTest.java
│       │   ├── controller/
│       │   │   └── PaymentControllerIntegrationTest.java
│       │   ├── mapper/
│       │   │   └── PaymentMapperTest.java
│       │   └── services/
│       │       └── PaymentServiceTest.java
│       └── resources/
```

```
|                └── db
|                    ├──master-test-changelog.yml
|                    └──db.changelog-test-data.yml
```

Шаг 1. Зависимости приложения. Файл pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.5.3</version>
    <relativePath/>
  </parent>

  <groupId>com.iprody</groupId>
  <artifactId>payment-service-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Payment Service App</name>
  <description>Payment service web application</description>
  <packaging>jar</packaging>

  <properties>
    <java.version>21</java.version>
    <map.struct.version>1.6.3</map.struct.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
```

```
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

    <dependency>
      <groupId>org.mapstruct</groupId>
       <artifactId>mapstruct</artifactId>
       <version>${map.struct.version}</version>
    </dependency>

    <dependency>
       <groupId>org.mapstruct</groupId>
       <artifactId>mapstruct-processor</artifactId>
       <version>${map.struct.version}</version>
       <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.7.6</version>
    </dependency>

    <dependency>
      <groupId>org.testcontainers</groupId>
      <artifactId>postgresql</artifactId>
      <version>1.19.3</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.kafka</groupId>
      <artifactId>spring-kafka</artifactId>
    </dependency>

    <dependency>
      <groupId>org.testcontainers</groupId>
```

```
      <artifactId>junit-jupiter</artifactId>
      <version>1.19.3</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-test</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
             <version>3.14.0</version>
             <configuration>
               <source>${java.version}</source>
               <target>${java.version}</target>
               <annotationProcessorPaths>
                  <path>
                     <groupId>org.mapstruct</groupId>
                     <artifactId>mapstruct-processor</artifactId>
                     <version>${map.struct.version}</version>
                  </path>
               </annotationProcessorPaths>
            </configuration>
       </plugin>
    </plugins>
  </build>

</project>
```

Шаг 2. Что такое Kafka

Apache Kafka — это распределенная платформа потоковой передачи данных, основанная на принципе **журнала событий** (*event log*).
Ее можно представить как **бесконечный массив**, в который последовательно добавляются новые элементы. Каждый элемент этого массива — это событие (сообщение), а его позиция определяется **смещением** (*offset*). Порядок элементов фиксирован: новые сообщения добавляются в конец, а уже записанные никогда не изменяются. Потребители могут начинать чтение с любого «индекса» — с самого начала или с определённого места — и двигаться вперед по мере появления новых данных.

Удивительным образом настолько простая по своей сути архитектура позволяет строить **высокопроизводительные и надежные системы обмена информацией**, способные обрабатывать сотни тысяч событий в секунду, масштабироваться горизонтально и интегрировать множество разнородных сервисов в единую потоковую экосистему. Стоит отметить что именно простота базовой архитектуры в виде журнала событий и позволяет добиться высокой горизонтальной масштабируемости и производительности.

Если сравнивать Kafka и RabbitMQ, то у RabbitMQ сообщения живут только до тех пор, пока не будут доставлены и подтверждены получателем, после чего удаляются из очереди. Это упрощает модель и хорошо подходит для классических сценариев «точка-точка» или сложной маршрутизации сообщений, но не позволяет легко повторно прочитать старые данные. У Kafka же сообщения хранятся в журнале событий заданное время, и любой потребитель может начать чтение с нужного момента, независимо от того, когда сообщение было отправлено. Такой подход дает возможность строить системы, где важно как текущее, так и историческое состояние потока данных, обеспечивая высокую производительность и легкое масштабирование за счет параллельной обработки. При этом RabbitMQ обладает интуитивно понятной архитектурой и простотой конфигурирования, тогда как Kafka — это более низкоуровневый инструмент, требующий глубокого понимания принципов построения потоковых систем.

Шаг 3. Основные компоненты Кафка

Основные компоненты Apache Kafka

1. Брокер (Broker)

Запущенный экземпляр (инстанс) Kafka, отвечающий за прием, хранение и выдачу сообщений. Кластер Kafka состоит из нескольких брокеров, которые совместно хранят данные и обеспечивают отказоустойчивость. Один брокер может обслуживать множество топиков (topic) разделенных на партиции (partition).

2. Топик (Topic)

Это упорядоченная, неизменяемая последовательность сообщений, в которую постоянно добавляются новые записи. Сообщения внутри темы могут храниться ограниченное время, определяемое настройками.

3. Партиция (Partition)

Физическая часть топика, которая хранит упорядоченную последовательность сообщений. Каждая партиция размещается на одном из брокеров кластера и может иметь копии (реплики) на других брокерах для обеспечения отказоустойчивости. Нагрузка распределяется между брокерами: разные разделы одного топика могут находиться на разных брокерах, что позволяет обрабатывать данные параллельно и масштабировать производительность.

4. Смещение (Offset)

Порядковый номер сообщения внутри раздела. Offset используется для отслеживания того, какие сообщения уже были прочитаны или обработаны. Порядок сообщений гарантируется только внутри одной партиции.

5. Продюсер (Producer)

Приложение или сервис, отправляющий сообщения в топики Kafka. Продюсер может указывать конкретную партиция для записи или позволять Kafka автоматически распределять сообщения между партициями.

6. Консьюмер (Consumer)

Приложение или сервис, читающий сообщения из топиков Kafka. консьюмер может начинать чтение с самого последнего сообщения, с начала истории или с заданного offset.

7. Группа консьюмеров (Consumer Group)

Набор консьюмеров, совместно читающих данные из топика. Каждая партиция топика внутри одной группы потребителей обрабатывается только одним консьюмером, что обеспечивает балансировку нагрузки и параллельную обработку.

8. ZooKeeper *(в старых версиях)* / KRaft *(в новых версиях)*

Механизм координации кластера Kafka.

- В старых версиях Kafka использует Apache ZooKeeper для хранения метаданных и управления брокерами.
- Начиная с Kafka 2.8, доступен режим **KRaft** (Kafka Raft Metadata Mode), позволяющий обойтись без ZooKeeper, храня метаданные внутри самой Kafka.

Шаг 4. Сообщение в Kafka

В Kafka сообщение - это **запись** (*record*), передаваемая в брокер для сохранения в партиции топика. Каждое сообщение состоит из нескольких логических частей:

1. Ключ (*key*)

- Последовательность байт, которую задаёт продюсер.
- Может быть использован для:
  - логической группировки сообщений,
  - выбора раздела топика для записи.
- Может быть `null`

2. Значение (*value*)

- Последовательность байт с данными сообщения.
- Интерпретация зависит от используемой сериализации: JSON, Avro, Protobuf, String и т.д.
- Может быть `null` (в некоторых сценариях используется для «логического удаления»).

3. Заголовки (*headers*)

- Набор пар «имя → значение» (имя — строка, значение — байтовый массив).
- Используются для передачи дополнительной метаинформации без изменения `value`.
- Примеры: тип события, версия схемы, идентификатор корреляции.

4. Метаданные, назначаемые системой (устанавливаются при записи в Kafka)

- **timestamp** — момент времени, связанный с сообщением:
  - `CreateTime` — время, когда сообщение создано продюсером.
  - `LogAppendTime` — время, когда сообщение было записано брокером.
- **checksum / CRC** — контрольная сумма для проверки целостности.
- **размер сообщения** — длина ключа, значения, заголовков.
- Внутренние поля для формата хранения (Record Batch, версия формата и т.д.).

Шаг 5. Топик (topic) и партиции (partition)

Журнал сообщений в Kafka представлен в виде топика, который разделен на партиции, хранящиеся на различных брокерах в кластере. То как происходит это распределение является ключевым в понимании принципов работы и масштабирования Kafka.

При создании топика необходимо указать два параметра

- num.partitions - количество партиций в топике
- replication.factor - количество реплик (копий) для каждой партиции

Важно понимать что каждая из партиций топика всегда будет целиком храниться на одном из брокеров. Таким образом данные топика оказываются распределены между брокерами. Если количество реплик равно 1, то это распределение будет насколько возможно равномерным - на каждом брокере примерно равное количество партиций каждого топика. Именно это и делает Kafka почти неограниченно горизонтально масштабируемой. Запись и чтение в разделы топиков может происходить параллельно, а при необходимости можно легко увеличивать количество партиций и подключать к кластеру новые брокеры, увеличивая пропускную способность.

Если количество реплик больше единицы, а так бывает практически всегда, то каждая партиция будет храниться в виде нескольких копий распределенных между брокерами: причем одна из копий будет главной (master или lead), а остальные ведомыми (slave или follower). Чтение и запись всегда происходят в главную копию партиции, а потом происходит репликация сообщений между ее репликами так чтобы их содержимое максимально совпадало с главной партицией. В случае если брокер на котором находится главная партиция топика вдруг станет недоступен, то его роль на себя быстро возьмёт наиболее полная реплика, которая станет новой главной партицией. Это дает кластеру не только высокую пропускную способность, но и высокую доступность. Неполадки на части брокеров не приведут к остановке обработки данных.

Шаг 6. Продюсер и запись сообщений

При создании продюсера в его параметрах указывается топик для отправки. Запись происходит в следующем порядке

1. Подготовка сообщения. Продюсер формирует запись (record), состоящую из:
    - ключа (key)
    - данных (value)
    - заголовков (headers)
2. Выбор партиции для записи производится
    - по хэшу ключа (hash(key) mod N), если ключ задан,
    - round-robin или sticky-partition алгоритмом, если ключа нет,
    - или кастомным партиционером.
3. Отправка лидеру партиции (брокеру на котором находится партиция-лидер)
4. Запись и репликация
    - Лидер партиции сохраняет запись в свой commit log (на диск).
    - Если replication.factor > 1, лидер отправляет запись своим ведомым партициям.
5. Подтверждение (acknowledgement) продюсеру, которое определяется количеством реплик запись сообщения в которые должна быть подтверждена
        - acks=0 — продюсер не ждёт подтверждения.
        - acks=1 — ждёт только подтверждение от лидера.
        - acks=all — ждёт подтверждения от всех синхронных реплик (min.insync.replicas).
6. Возврат метаданных. После выполнения условий acks лидер возвращает продюсеру RecordMetadata (топик, партиция, offset).

Шаг 7. Консьюмеры и чтение сообщений

Консьюмер (consumer) – это клиентское приложение, которое читает сообщения из Kafka-топиков.

Каждое сообщение внутри партиции топика имеет уникальный **офсет** – порядковый номер или индекс,под которым оно было записано в партицию.

- консьюмер всегда читает сообщения из конкретной партиции строго в порядке их записи.
- Чтобы «помнить», где он остановился, консьюмер хранит последний прочитанный оффсет.
- По умолчанию информация об офсете сохраняется в специальном служебном топике `__consumer_offsets`. Это позволяет консьюмеру после перезапуска продолжить чтение «с того же места».

Группа консьюмеров – это набор процессов, которые совместно читают один и тот же топик.

- Каждой партиции топика назначается **ровно один консьюмер в группе**. Это значит, что если в топике 6 партиций и в группе 3 консьюмера, то каждый получит по 2 партиции.
- Если консьюмеров больше, чем партиций, «лишние» будут простаивать.
- Если консьюмеров меньше, чем партиций, один консьюмер может обслуживать несколько партиций.

Следует отметить что Kafka гарантирует последовательное чтение только для сообщений в рамках одной партиции. Если консьюмер читает несколько партиций, то он будет циклически чередовать чтение из них, что нужно учитывать в рамках конкретной бизнес-логики.

Шаг 8. Запуск Kafka при помощи docker-compose

Мы будем запускать Kafka при помощи docker-compose в режиме KRaft. Для простоты ограничимся одним брокером. Также запустим приложение с графическим интерфейсом для удобного мониторинга состояния брокера

```
version: '3.8'

services:
  payment-service-app:
    image: payment-service-app:0.0.1
    ports:
      - "8080:8080"

  postgres:
    image: postgres:16
```

```yaml
    container_name: postgres-db
    environment:
      POSTGRES_USER: admin          # имя пользователя
      POSTGRES_PASSWORD: secret     # пароль
      POSTGRES_DB: payment-db       # имя БД по умолчанию
      LC_MESSAGES: en_US.UTF-8
    ports:
      - "5432:5432"                 # проброс порта PostgreSQL
    volumes:
      - pgdata:/var/lib/postgresql/data
    restart: unless-stopped

  pgadmin:
    image: dpage/pgadmin4:latest
    container_name: pgadmin
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@mail.com
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "8081:80"
    depends_on:
      - postgres
    restart: unless-stopped

  keycloak:
    image: quay.io/keycloak/keycloak:24.0.3
    container_name: keycloak
    command:
      - start-dev
      - --http-port=8080
      - --import-realm
      - --log-level=DEBUG
    environment:
      KEYCLOAK_ADMIN: admin
      KEYCLOAK_ADMIN_PASSWORD: admin
    ports:
      - "8085:8080"
    volumes:
      - ./realm-export.json:/opt/keycloak/data/import/realm-export.json

  # ДОБАВИТЬ
  kafka:
    image: bitnami/kafka:4.0.0
    container_name: kafka
    ports:
      - "9092:9092"   # для клиентов из docker-сети (Kafka UI, другие
```

```
сервисы)
      - "9093:9093"   # для внешних клиентов (ваш хост)
    environment:
      - KAFKA_ENABLE_KRAFT=yes
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_NODE_ID=1
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka:9094
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER

      # слушатели
      -
KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,EXTERNAL://:9093,CONTROLLER://:909
4
      -
KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,EXTERNAL:PL
AINTEXT,CONTROLLER:PLAINTEXT

      # рекламируем: внутри докера - kafka:9092, снаружи -
localhost:9093
      -
KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092,EXTERNAL://localho
st:9093

      # брокеры общаются через PLAINTEXT (у вас кластер из одного узла)
      - KAFKA_CFG_INTER_BROKER_LISTENER_NAME=PLAINTEXT

      - KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE=true
    volumes:
      - kafka_data:/bitnami/kafka

  # ДОБАВИТЬ
  kafka-ui:
    image: provectuslabs/kafka-ui:latest
    container_name: kafka-ui
    ports:
      - "8090:8080"
    environment:
      - KAFKA_CLUSTERS_0_NAME=local
      - KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS=kafka:9092
    depends_on:
      - kafka

volumes:
  pgdata:
  # ДОБАВИТЬ
  kafka_data:
```

Шаг 9. Проверка работоспособности Kafka

Перезапустите конфигурацию docker-compose и попробуйте перейти по ссылке http://localhost:8090. Если всё правильно то по этому адресу будет доступен UI для мониторинга Kafka.

Для проверки давайте создадим тему и отправим в неё сообщение из командной строки. Для этого введите команду

```
docker exec -it kafka /opt/bitnami/kafka/bin/kafka-console-producer.sh \
  --bootstrap-server localhost:9092 \
  --topic demo
```

В интерактивной консоли напечатай строку, например: Hello Kafka! и нажми Enter. Сообщение отправится в топик demo.

Отметим что запуск kafka-ui может занять некоторое время. Чтобы убедиться что он запустился лучше всего посмотреть логи

```
docker-compose logs kafka-ui -f
```

Теперь найдем сообщения через Kafka UI

1. Открой в браузере http://localhost:8090.
2. Перейти в раздел Clusters → local → Topics.
3. Найди топик demo и кликни на него.
4. Там будет несколько вкладок: Messages, Partitions и др.
    - В Messages ты увидишь отправленные данные.
    - Можно выбрать конкретную партицию и просматривать сообщения по их офсетом.
5. Если сообщение одно и партиций несколько, оно всё равно попадёт ровно в одну из них (алгоритм распределения зависит от ключа). В UI будет видно:
    - номер партиции (например, Partition 0),
    - оффсет (например, offset = 0),
    - само сообщение (Hello Kafka!).

Шаг 10. Настройка соединения с брокером Kafka в Spring Boot

Для начала добавим зависимость в pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         https://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <parent>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.5.3</version>
    <relativePath/>
  </parent>

  <groupId>com.iprody</groupId>
  <artifactId>payment-service-app</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Payment Service App</name>
  <description>Payment service web application</description>
  <packaging>jar</packaging>

  <properties>
    <java.version>21</java.version>
    <map.struct.version>1.6.3</map.struct.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

    <dependency>
      <groupId>org.mapstruct</groupId>
```

```
    <artifactId>mapstruct</artifactId>
    <version>${map.struct.version}</version>
</dependency>

<dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct-processor</artifactId>
    <version>${map.struct.version}</version>
    <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.6</version>
</dependency>

<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>postgresql</artifactId>
  <version>1.19.3</version>
  <scope>test</scope>
</dependency>

<!-- ДОБАВИТЬ -->
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>

<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>1.19.3</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-test</artifactId>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
```

```
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
             <version>3.14.0</version>
             <configuration>
               <source>${java.version}</source>
               <target>${java.version}</target>
               <annotationProcessorPaths>
                  <path>
                     <groupId>org.mapstruct</groupId>
                     <artifactId>mapstruct-processor</artifactId>
                     <version>${map.struct.version}</version>
                  </path>
               </annotationProcessorPaths>
            </configuration>
       </plugin>
    </plugins>
  </build>

</project>
```

Теперь обновим файл конфигурации application.yaml с настройками подключения к Kafka. Также в разделе app.kafka.topics мы указали имена очередей для входящих и исходящих сообщений

Мы создаём продюсер для отправки платежей на оплату и одного консьюмера с group-id: xpayment-adapter-result-consumers. Количество консьюмеров указано в параметром concurrency в разделе listener.

```
server:
  port: 8088

spring:
  application:
```

```
    name: payment-service
  datasource:
    url: jdbc:postgresql://localhost:5432/payment-db
    username: admin
    password: secret
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: validate
    show-sql: true
    properties:
      hibernate:
        format_sql: true
    database-platform: org.hibernate.dialect.PostgreSQLDialect
  liquibase:
    enabled: true
    change-log: classpath:db/changelog/db.changelog-master.yaml
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8085/realms/iprody-lms

  kafka:
    bootstrap-servers: localhost:9093

    producer:
      key-serializer:
org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer

    consumer:
      group-id: xpayment-adapter-result-consumers
      key-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer
      auto-offset-reset: earliest
      enable-auto-commit: false
      properties:
        spring.json.trusted.packages:
"com.iprody.payment.service.app.async"
        spring.json.value.default.type:
"com.iprody.payment.service.app.async.XPaymentAdapterResponseMessage"
```

```
    listener:
      ack-mode: manual
      concurrency: 1

app:
  kafka:
    topics:
      xpayment-adapter:
        request: xpayment-adapter.requests
        response: xpayment-adapter.responses
```

Шаг 11. Реализация внутреннего API для взаимодействия с Kafka

Теперь интегрируем Kafka продюсер с нашим API из методички 16. Обратите внимание, что интеграция происходит практически бесшовно и не затрагивает бизнес логику.

```
package com.iprody.payment.service.app.async.kafka;

import com.iprody.payment.service.app.async.AsyncSender;
import
com.iprody.payment.service.app.async.XPaymentAdapterRequestMessage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class KafkaXPaymentAdapterRequestSender
        implements AsyncSender<XPaymentAdapterRequestMessage> {

    private static final Logger log =
LoggerFactory.getLogger(KafkaXPaymentAdapterRequestSender.class);

    private final KafkaTemplate<String, XPaymentAdapterRequestMessage>
template;

    private final String topic;

    public KafkaXPaymentAdapterRequestSender(
            KafkaTemplate<String, XPaymentAdapterRequestMessage>
template,
```

```
@Value("${app.kafka.topics.xpayment-adapter.request:xpayment-adapter.req
uests}") String topic
    ) {
        this.template = template;
        this.topic = topic;
    }

    @Override
    public void send(XPaymentAdapterRequestMessage msg) {
        String key = msg.getPaymentGuid().toString(); // фиксируем
партиционирование по платежу
        log.info("Sending XPayment Adapter request: guid={}, amount={},
currency={} -> topic={}",
                msg.getPaymentGuid(), msg.getAmount(),
msg.getCurrency(), topic);
        template.send(topic, key, msg);
    }
}
```

Аналогично для консьюмера

```
package com.iprody.payment.service.app.async.kafka;

import com.iprody.payment.service.app.async.MessageHandler;
import
com.iprody.payment.service.app.async.XPaymentAdapterResponseMessage;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.support.Acknowledgment;
import org.springframework.stereotype.Component;

@Component
public class KafkaXPaymentAdapterResultListenerAdapter
        implements AsyncListener<XPaymentAdapterResponseMessage> {

    private static final Logger log =
LoggerFactory.getLogger(KafkaXPaymentAdapterResultListenerAdapter.class)
;

    private final MessageHandler<XPaymentAdapterResponseMessage>
handler;

    public
KafkaXPaymentAdapterResultListenerAdapter(MessageHandler<XPaymentAdapter
```

```
ResponseMessage> handler) {
        this.handler = handler;
    }

    @Override
    public void onMessage(XPaymentAdapterResponseMessage message) {
        handler.handle(message);
    }

    @KafkaListener(topics =
"${app.kafka.topics.xpayment-adapter.response}",
                   groupId = "${spring.kafka.consumer.group-id}")
    public void consume(
            XPaymentAdapterResponseMessage message,
            ConsumerRecord<String, XPaymentAdapterResponseMessage>
record,
            Acknowledgment ack
    ) {
        try {
            log.info("Received XPayment Adapter response:
paymentGuid={}, status={}, partition={}, offset={}",
                     message.getPaymentGuid(), message.getStatus(),
record.partition(), record.offset());
            onMessage(message);
            ack.acknowledge();
        } catch (Exception e) {
            log.error("Error handling XPayment Adapter response for
paymentGuid={}", message.getPaymentGuid(), e);
            throw e; // отдаём в error handler Spring Kafka
        }
    }
}
```

Реализации интерфейсов AsyncSender и AsyncListener из урока 16 необходимо удалить чтобы они не создавали конфликтов с новыми бинами Kafka.

Домашнее задание

1. Откройте ваш рабочий проект, переключитесь на ветку main и синхронизируйтесь с удаленным репозиторием выполнив команду git pull origin main
2. Создайте новую ветку с именем homework-17 и переключитесь на нее выполнив команду git checkout -b <your_branch>
3. Необходимо расширить существующий docker-compose.yml новыми сервисами: kafka и kafka-ui. После обновления файла конфигурации, необходимо удостовериться в работоспособности новых сервисов перейдя на Kafka UI по

адресу http://localhost:8090 и проанализировать данные. В случае трудностей воспользуйтесь инструкцией из шага 8 и шага 9
4. Необходимо добавить org.springframework.kafka:spring-kafka в качестве зависимости для использования возможностей Spring Framework по интеграции с Apache Kafka. В случае трудностей воспользуйтесь инструкцией из шага 10
5. Необходимо расширить существующий application.yml конфигурацией Apache Kafka (server, producer, consumer, listener), а также кастомной конфигурацией именования топиков для коммуникации с X Payment Adapter: запрос о выполнение платежа, ответ о выполнении платежа. В случае трудностей воспользуйтесь инструкцией из шага 10
6. Необходимо реализовать интерфейс AsyncSender<XPaymentAdapterRequestMessage>. Данный класс будет играть роль отправителя сообщений в Apache Kafka по соответствующему топику для обработки этого сообщения на стороне X Payment Adapter. Реализация заключается в принятии подготовленного (на основе объекта типа Payment) сообщения. После отправки сообщения реализация сразу же должна вернуть ответ. В случае трудностей воспользуйтесь инструкцией из шага 11
7. Необходимо реализовать интерфейс AsyncListener<XPaymentAdapterResponseMessage>. Данный класс будет играть роль слушателя сообщений соответствующего топика, через который будут передаваться сообщения после обработки платежной транзакции на стороне X Payment Adapter. Реализация заключается в принятии сообщения и его передаче классу реализующему интерфейс MessageHandler<XPaymentAdapterResponseMessage>.handle(...) для последующей обработки. В случае успешной обработки, необходимо подтвердить Apache Kafka успешность принятия сообщения. В случае трудностей воспользуйтесь инструкцией из шага 11
8. Для избежания конфликта бинов в Spring IoC-контейнере, необходимо удалить старые реализации интерфейсов AsyncSender и AsyncListener из предыдущего урока.
9. Пересоберите приложение и сторонние компоненты и перезапустите их. Удостоверьтесь в работоспособности всей системы целиком и особенно в отправке запроса о платежной транзакции в Apache Kafka и получении ответа об обработке платежной транзакции из Apache Kafka.
10. Убедитесь в том, что вы находитесь в рабочей ветке homework-17 выполнив команду git branch. Создайте коммит с сообщением, которое соответствует реализованным изменениям. Выполните команду git push для отправки изменений в удаленный репозиторий.
11. Перейдите на GitHub, убедитесь в том, что новая ветка и изменения достигли GitHub. Создайте PR на основании реализованных изменений.