УДК 004.42
DOI: 10.46548/21vek-2021-1053-0005

О ПРАКТИЧЕСКИХ АСПЕКТАХ СОЗДАНИЯ ПРИЛОЖЕНИЙ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ СОВМЕСТНО С РАСПРЕДЕЛЁННЫМ ПРОГРАММНЫМ БРОКЕРОМ СООБЩЕНИЙ *APACHE KAFKA*

Тамбовцев Антон Юрьевич, магистрант
Смольянов Андрей Григорьевич, кандидат физико-математических наук, доцент,
заведующий кафедрой фундаментальной информатики
Национальный исследовательский Мордовский государственный университет имени Н. П. Огарёва
(430005, Россия, г. Саранск, улица Большевистская, д. 68,
e-mails: grafvector725@gmail.com, mgutech@mail.ru)

Аннотация. В настоящее время большое развитие получила идея микросервисной архитектуры. В статье обсуждаются вопросы организации обмена сообщениями между сервисами в микросервисной экосистеме при помощи брокера сообщений Apache Kafka и распределенного хранилища Apache Zookeeper. В контексте микросервисов, сервис - это не какой-то класс или функция, а изолированное приложение, которое отвечает на запросы. Оно способно также отправлять запросы, иметь свою собственную базу данных и необходимый набор библиотек и фреймворков. Сервис не должен быть зависим от конкретного языка программирования, он должен предоставлять интерфейс для обращения к нему, это и есть изолированность, то есть для каждого сервиса можно подобрать тот стек технологий, который подходит для решения задач данного сервиса. Принято создавать сервисы максимально компактными (отсюда и название – «микросервис»), которые решают конкретные бизнес-задачи. В статье рассматривается реализация простого приложения, написанного на языках Java с использованием фреймворка Spring Boot и Golang вместе с пакетом kafka-go. Данные стек технологий сегодня самый популярный в контексте высоконагруженных и распределенных систем, как вместе, так и по отдельности. Так же показывается независимость приложения от конкретного языка программирования. Демонстрируются идеи построения масштабируемых приложений на абстракциях и подходах. Исследование показало полезность и популярность брокеров сообщений и самой идеи обмена сообщениями между сервисами посредством рассмотренного подхода.

Ключевые слова: микросервис, брокер сообщений, распределенная система, веб-сервис, подписчик, порт, канал связи.

ON THE PRACTICAL ASPECTS OF CREATING MICROSERVICE ARCHITECTURE APPLICATIONS IN CONJUNCTION WITH THE *APACHE KAFKA* DISTRIBUTED SOFTWARE MESSAGE BROKER

Tambovtsev Anton Yurievich, master's student
Smolyanov Andrey Grigorievich, candidate of Physical and Mathematical Sciences,
head of the Department of Fundamental Informatics
National Research Mordovia State University named after N.P. Ogarev
(430005, Russia, Saransk, street Bolshevik, 67, e-mails: grafvector725@gmail.com, mgutech@mail.ru)

Abstract. Currently, the idea of microservice architecture has received a lot of development. This article discusses how to organize messaging between services in a microservice ecosystem using Apache Kafka message broker and Apache Zookeeper distributed storage. In the context of microservices, a service is not a class or function, but a sandboxed application that responds to requests. It can also send requests, have its own database and the necessary set of libraries and frameworks. The service should not be dependent on a specific programming language, it should provide an interface for accessing it, this is isolation, that is, for each service, you can choose the technology stack that is suitable for solving the tasks of this service. It is customary to create services as compact as possible (hence the name - "microservice") that solve specific business problems. The article discusses the implementation of a simple application written in Java using the Spring Boot framework and Golang along with the kafka-go package. These technology stacks are the most popular today in the context of high-load and distributed systems, both together and separately. The independence of the application from a specific programming language is also shown. The ideas of building scalable applications on abstractions and approaches are demonstrated. The study showed the usefulness and popularity of message brokers and the very idea of messaging between services through the considered approach.

Keywords: microservice, message broker, distributed system, web service, subscriber, port, communication channel.

Введение. Современные технологии разработки распределенных систем предложили программистам специальные инструменты – брокеры сообщений, предназначенные для обмена данными микросервисов между собой [1]. Обсудим идеи их применения. Как известно, микросервисы должны быть изолированы друг от друга [2]. По этой причине возникает необходимость в каком-то протоколе, к примеру, *http*,

который позволил бы сервисам обмениваться между собой сообщениями [3]. Возникает вопрос: как быть, если сервис, на который послано сообщение, в данный момент занят и не может обработать посланное сообщение? Одно из решений – где-то временно его разместить. Здесь и оказывается полезен брокер сообщений. Таким образом, в качестве цели работы можно обозначить исследование методов построения микросервисных приложений с помощью брокера сообщений. Задачей исследования была разработка простого микросервисного приложения с использованием *Apache Kafka* и языков программирования *Java* и *Golang*.

Материалы и результаты исследования. Напомним, что брокер сообщений – архитектурный паттерн в структуре распределенной системы. Это приложение, которое преобразует в соответствие с некоторым протоколом сообщение от приложения-источника в сообщение протокола приложения-приемника, выступая между ними посредником.

Кроме преобразования сообщений из одного формата в другой, в задачи брокера сообщений также входят следующие функции:

1) проверка сообщений на ошибки;

2) маршрутизация сообщений конкретным приемникам;

3) разбиение сообщения на несколько частей, агрегирование ответов приемников и отправка результата источнику;

4) сохранение сообщений в базе данных;

5) вызов веб-сервисов;

6) рассылка сообщений подписчикам, если используются шаблоны типа «издатель-подписчик» [4].

Кратко обсудим идеи построения микросервисного приложения с использованием `Spring Boot`, `Golang` и `Apache Kafka` [5, 6]. Упрощённая структура общения сервисов через `Apache Kafka` показана на рисунке 1.

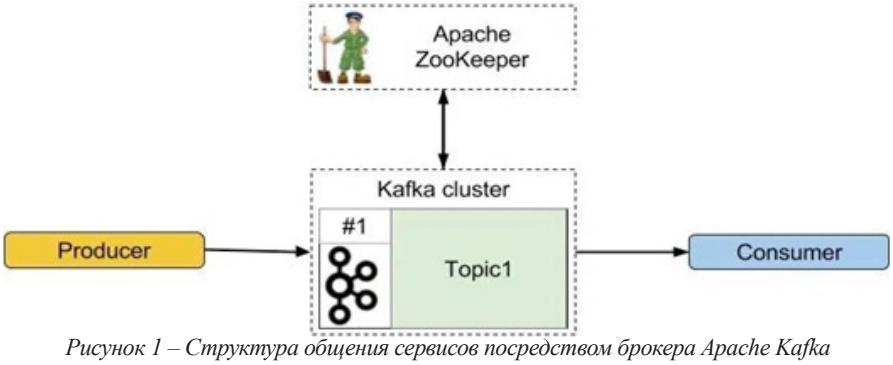

Рисунок 1 – Структура общения сервисов посредством брокера Apache Kafka

Как видно из рисунка 1, на схеме присутствует еще один сервис – `Apache Zookeeper` [7].

Это некое хранилище объектов типа `{key: value}`[8]. Он будет использоваться для хранения наших тем.

Здесь уместно напомнить ряд важных для исследования терминов:

– Тема (*topic*) – некий канал связи, к которому подключатся сервисы и через него передают сообщения;

– Подписчики (*consumer*) – это те, кто будет принимать сообщения из темы;

– Издатель (*producer*) – сервис, который посылает сообщения;

– Группа (*group*) - группа подписчиков;

– Раздел (*partition*) - группа тем;

– Потоки (*streams*) - инструменты для тем.

Реализация поставленной задачи потребует развернуть среду `Apache Kafka`. В первую очередь нам потребуется «поднять» хранилище объектов. В файле конфигураций `Zookeeper` минимальное, что требуется сделать, указать порт, на котором будет запущен экземпляр хранилища и путь к папке, в которую будут направляться данные [9]. Время, через которое хранилище будет искать *JVM* машины в нашей системе, указывается в миллисекундах:

```
tickTime=2000
clientPort=2181
dataDir=/var/zookeeper
```

Старт сервиса `Zookeeper`:

`./zookeeper/bin/zkServer.sh start.`

Если все прошло успешно, то пользователь увидит сообщение типа:

```
/usr/bin/java
ZooKeeper JMX enabled by default
Using config: /home/ramfan/study/kafka_broker/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

Проверить работоспособность сервиса можно следующей командой [10]:

```
echo ruok | nc localhost 2181
```

Если в ответ пришло `imok`, значит, все прошло хорошо.

Следующий этап – поднятие самого брокера `Apache Kafka`.

```
./kafka_2.12-2.6.0/bin/kafka-server-start.sh
./kafka_2.12-2.6.0/config/ server.properties
```

Теперь создадим `topic` с названием `message`:

```
/kafka_2.12-2.6.0/bin/kafka-topics.sh
                --create
        --zookeeper localhost:2181
        --replication-factor 1
```

```
        --partitions 1
        --topic message
```

Теперь можно приступать к написанию микросервисного приложения.

Создадим *Java*-проект и подключим в нем фреймворки *Spring Boot* и *Spring Kafka* [11, 12].

Наши сервисы могут быть либо подписчиками, либо издателями. По этой причине сервис нужно сконфигурировать либо под подписчика, либо под издателя [13].

```
@Configuration
public class KafkaConsumerConfiguration {
   @Bean
    public KafkaListenerContainerFactory<?>
kafkaListenerSingleFactory() {
     ConcurrentKafkaListenerContainerFactor
y<Long, AbstractDto> factory =
         new ConcurrentKafkaListenerContain
erFactory<>();
     factory.setConsumerFactory(kafkaConsum
erProprieties());
     factory.setBatchListener(false);
     factory.setMessageConverter(stringJson
MessageConverter());
     return factory;
    }
   @Bean
    public StringJsonMessageConverter
stringJsonMessageConverter() {
     return new StringJsonMessageConverter();
    }
   @Bean
    public DefaultKafkaConsumerFactory<Long
, AbstractDto>
        kafkaConsumerProprieties() {
     Map<String, Object> proprieties = new
HashMap<>();
     proprieties.put(ConsumerConfig.GROUP_ID
_CONFIG, «server.broadcast»);
     proprieties.put(ConsumerConfig.ENABLE_AU
TO_COMMIT_CONFIG, true);
     proprieties.put(ConsumerConfig.BOOTSTRA
P_SERVERS_CONFIG, «localhost:9092»);
     proprieties.put(ConsumerConfig.KEY_DESE
RIALIZER_CLASS_CONFIG,
                StringDeserializer.class);
     proprieties.put(ConsumerConfig.VALUE_DE
SERIALIZER_CLASS_CONFIG,
                StringDeserializer.class);
     return  new DefaultKafkaConsumerFactor
y<Long, AbstractDto>(proprieties);
    }
}
```

В данной конфигурации нам нужно указать сериализаторы и десериализаторы для перевода сообщения из строки в объект и наоборот [14]. Так же нужно указать адрес, на котором работает сам брокер.

Создадим фабрику подписчиков – *kafkaListener SingleFactory* [15]. Проинициализированная нужными нам свойствами, она в дальнейшем будет служить стандартной фабрикой подписчиков. Здесь устанавливается *factory.setBatchListener(false)*, то есть данная фабрика будет использоваться исключительно для приема одиночных сообщений. Если же нам потребуется получать списки таких сообщений, как массив *JSON*- объектов, то нам будет нужна другая фабрика, но уже со свойством *factory. setBatchListener(true)*. Это один из ключевых моментов в конфигурации подписчиков. Далее в классе-сервисе, который будет обрабатывать полученные сообщения, укажем название данного метода – *kafkaListenerSingleFactory*.

kafkaConsumerProprieties – это просто список конфигураций. Здесь указываем адрес сервера, группу и десериализаторы.

Опишем класс-сервис, который будет прослушивать нашу тему *gateway_topic*.

```
@Service
@Slf4j
public    class    MessageManagerServiceImpl
implements MessageManagerService {
   private  final  KafkaTemplate<Long,  Messa
geDto>
              starshipDtoKafkaTemplate;
    private final ObjectMapper objectMapper;
    private final Logger log;
    @Autowired
    public  MessageManagerServiceImpl(Kafka
Template
             <Long, MessageDto>
             kafkaStarshipTemplate,
             ObjectMapper objectMapper,
             Logger log)  {
      this.starshipDtoKafkaTemplate = kafka
Star shipTemplate;
      this.objectMapper = objectMapper;
      this.log = log;
    }
    @Override
    @KafkaListener(id = "MessageManager",to
pics = {"message"},
                containerFactory = "kafkaLi
stenerSingleFactory")
    public void consume(MessageDto dto) {
      log.info("=> consumed {}", writeValue
AsString(dto));
    }
   private String writeValueAsString(Message
Dto dto) {
      try {
            return objectMapperwriteValueAs
String(dto);
      } catch (JsonProcessingException e) {
            e.printStackTrace();
            throw new RuntimeException
                ("Writing value to JSON fai
led: " + dto.toString());
      }
   }
}
```

Заметим, что код

```
@KafkaListener(id = "MessageManager",
           topics = {"message"},
           containerFactory = "kafkaListe
ner SingleFactory")
```

говорит о том, какую тему прослушивает метод, и какая фабрика при этом будет использоваться [16].

Перейдем к рассмотрению второго сервиса.

Для примера был использован другой язык программирования – *Go* [17]. Здесь использовалась библиотека для конфигурации *Apache Kafka, kafka-go* [18, 19]. Данный сервис будет издателем. Функция createProducer фактически является конфигурацией

издателя. Она возвращает указатель на функцию `Writer` библиотеки [20]. Можно заметить значительное сокращение кода.

```
func createProducer() *kafka.Writer  {
       producer    :=    &kafka.Writer{Addr:
kafka.TCP("localhost:9092"),
         Topic:  "gateway_topic",  Balancer:
&kafka.LeastBytes{}}
        return producer
}
type MessageDTO struct {
        ID      int   `json:"id"`
        Action  string  `json:"actionType"`
       Payload interface{} `json: "payload"`
}
func main() {
       kafkaProducer := createProducer()
       for i:=0; ;i++ {
          messageBody := &MessageDTO{ID: i,
Action: "SEND_MESSAGE_ACTION",
                          Payload: nil}
         messageBodyAsJson, err := json.
Marshal(messageBody)
          if err != nil {
              fmt.Println(err)
         }
          msg := kafka.Message{
               Key: []byte(fmt.Sprintf("Key-
%d", i)),
               Value: []byte(fmt.Sprint(stri
ng(messageBodyAsJson))),
          }
          err = kafkaProducer.WriteMes sages
(context.Background(), msg)
               if err != nil{
               fmt.Println(err)
          }
          time.Sleep(322000)
     }
     defer kafkaProducer.Close()
}
```

В главной функции `main` в цикле идет создание структуры сообщения и происходит передача самого сообщения для передачи библиотеке `kafka-go`. Структура `Message` библиотеки подготавливает наше сообщение для метода `WriteMessages`, который делает отправку нашего сообщения.

Результат работы программного обеспечения показан на рисунках 2 и 3.

```
.ServiceApplication     : <= sending {"id":41249944,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
.ServiceApplication     : <= sending {"id":41252946,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
.ServiceApplication     : <= sending {"id":41255947,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
.ServiceApplication     : <= sending {"id":41258949,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
.ServiceApplication     : <= sending {"id":41261950,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
```

Рисунок 2 – Результат отправки сообщений

```
.ServerApplication   : => consumed {"id":41249944,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
.ServerApplication   : => consumed {"id":41252946,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
.ServerApplication   : => consumed {"id":41255947,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
.ServerApplication   : => consumed {"id":41258949,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
.ServerApplication   : => consumed {"id":41261950,"payload":null,"actionType":"SEND_MESSAGE_ACTION"}
```

Рисунок 3 – Результат приема сообщений из темы

Заключение. Рассмотренные в настоящем материале программные инструменты продемонстрировали технологию взаимодействия микросервисных приложений. Показаны способы организации взаимодействия микросервисных приложений посредством брокера сообщений. Разработанный в рамках настоящего исследования программный код показал свою корректность и работоспособность.

СПИСОК ЛИТЕРАТУРЫ:

1. Распределённая система [Электронный ресурс] //URL: https://ru.wikipedia.org/wiki/Распределённая_система (дата обращения: 12.01.2021).
2. Микросервисная архитектура [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/Микросервисная_архитектура (дата обращения: 12.01.2021).
3. Microservices [Электронный ресурс] // URL: https://martinfowler.com/articles/microservices.html (дата обращения: 12.01.2021).
4. Брокер сообщений [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/Брокер сообщений (дата обращения: 12.01.2021)
5. Documentation. Kafka 2.6 Documentation [Электронный ресурс] // URL: https://kafka.apache.org/documentation (дата обращения: 12.01.2021)
6. Spring Boot Reference Documentation [Электронный ресурс] // URL: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle (дата обращения: 12.01.2021)
7. ZooKeeper 3.6 Documentation [Электронный ресурс] // URL: https://zookeeper.apache.org/doc/r3.6.2/index.html (дата обращения: 12.01.2021)
8. Распределенное хранилище [Электронный ресурс] // URL: https://ru.qaz.wiki/wiki/Distributed_data_store (дата обращения: 12.01.2021).
9. ZooKeeper Getting Started Guide [Электронный ресурс] // URL: https://zookeeper.apache.org/doc/r3.1.2/zookeeperStarted.html (дата обращения: 12.01.2021).
10. Netcat [Электронный ресурс] // URL: https://docs.oracle.com/cd/E36784_01/html/E36870/netcat-1.html (дата обращения: 12.01.2021).
11. Spring for Apache Kafka [Электронный ресурс] // URL: https://docs.spring.io/spring-kafka/reference/html (дата обращения: 12.01.2021).
12. Building an Application with Spring Boot [Электронный ресурс] // URL: https://spring.io/guides/gs/spring-boot/ (дата обращения: 12.01.2021).
13. Spring Bean Annotations [Электронный ресурс] // URL: https://www.baeldung.com/spring-bean-annotations (дата обращения: 12.01.2021).
14. Сериализация [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/Сериализация (дата обращения: 12.01.2021).
15. Фабричный метод (шаблон проектирования) [Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/ Фабричный_метод_(шаблон_проектирования) (дата обращения: 12.01.2021).
16. Using Kafka with Spring Boot [Электронный ресурс] // URL: https://reflectoring.io/spring-boot-kafka/ (дата обращения: 12.01.2021).
17. Using Kafka with Spring Boot [Электронный ресурс] // URL: https://reflectoring.io/spring-boot-kafka/ (дата обращения: 12.01.2021).
18. Golang documentation [Электронный ресурс] // URL: https://golang.org/doc/ (дата обращения: 12.01.2021).
19. Package kafka // URL: https://godoc.org/github.com/segmentio/kafka-go (дата обращения: 12.01.2021).
20. Getting Started with Kafka in Golang // URL:https://godoc.org/github.com/segmentio/kafka-gohttps://yusufs.medium.com/getting-started-with-kafka-in-golang-14ccab5fa26(дата обращения: 12.01.2021).

Статья поступила в редакцию 25.01.2021
Статья принята к публикации 12.03.2021