Spring Boot + Apache Kafka и SSL в Docker контейнере (<https://habr.com/ru/articles/505720/>)

Проект лежит - *https://github.com/layonez/kafka-example*

### Выбор docker образа Apache Kafka + Zookeeper

### !!!!!Странный пример, где клиенты (Consumer u Producer) зачем-то запущенны будут не в Docker-е и его локальной сети, а снаружи, просто отдельно на нашем компьютере с использованием хоста “localhost”. И это ломает основную логику микросервисов, где каждый из которых должен быть запущен на разных компьютерах *(и точно не не нашем компьютере т.к. мы разработчик ПО, а не сервер фирмы заказавшей это ПО)*, для иммитации чего и создан Docker.!!!

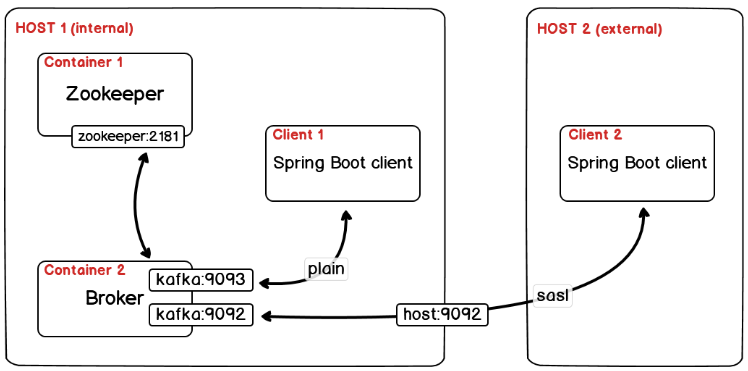
Для начала придется определиться с вендором образа Apache Kafka для нашего контейнера. Требования каждого конкретного проекта разнятся по необходимому уровню безопасности и надежности решения, в некоторых случаях конечно придется собирать свой собственный образ, но для большинства проектов будет разумным выбрать один из доступных на docker hub. На 2020й год есть три основных популярных вендора образов kafka+zookeper:

* [oбразы от confluent](https://docs.confluent.io/current/quickstart/ce-docker-quickstart.html) [ субъективно, большой плюс — поставляются разработчиками самой Apache Kafka и наиболее надежны в плане информационной безопасности ]
* [образы от bitnami](https://github.com/bitnami/bitnami-docker-kafka) [плюс, в сравнении с confluent, в том что образы нуждаются в минимальной конфигурации и “стартуют из коробки”]
* [образы от wurstmeister](https://github.com/wurstmeister/kafka-docker) [в этом примере я буду использовать их, потому что у проекта наибольшее активное комьюнити на github и [hub.docker](https://hub.docker.com/r/wurstmeister/kafka/), а так же отличная документация ]

### Установка Docker-compose

Для запуска и конфигурирования образов используется [docker-compose](https://docs.docker.com/compose/), который позволяет легко запускать мультиконтейнерные приложения. Без него не обойтись, потому что минимальная конфигурация kafka предусматривает связку из zookeeper и хотя бы одного брокера.  
Сompose идет в комплекте с mac/windows — дистрибутивом Docker, так что установка необходимого окружения сводится к загрузке и установке дистрибутива с [docs.docker.com/engine/install](https://docs.docker.com/engine/install/).  
А вот linux пользователям его необходимо устанавливать отдельно. За гайдом по установке docker + compose лучше всего обратиться к официальной документации [docs.docker.com/compose/install](https://docs.docker.com/compose/install/).

### Пишем yaml файл для compose

Общая схема нашей конфигурации для всех клиентов и контейнеров будет следующей:  
  
  
  
К одному координатору будет подключаться один брокер с открытыми портами 9093 — для localhost подключений и 9092 для подключений с других машин.  
  
Для доступа из внешних сетей включим авторизацию через логин пароль без ssl шифрования. Для работы через ssl необходимо выпускать собственные ключи и подписывать сертификаты, весь процесс хорошо расписан в [официальной документации](https://docs.confluent.io/current/security/security_tutorial.html).  
  
Следующий файл docker-compose.yml, который создает одноузловой сервер Kafka с 1 Zookeeper и 1 экземпляром брокера. Конфигурация также гарантирует, что сервер Zookeeper всегда запускается раньше сервера Kafka (брокера) и останавливается после него. В итоге файл будет иметь следующий вид(адрес нужно заменить на ваш публичный IP для возможности доступа из других сетей):  
**docker-compose.yml**

version: '2'

services:

zookeeper:

image: wurstmeister/zookeeper

expose:

- "2181"

kafka:

image: wurstmeister/kafka

ports:

- "9092:9092"

expose:

- "9093"

environment:

KAFKA\_AUTO\_CREATE\_TOPICS\_ENABLE: 'true'

# отправляум информацию за пределы Docker-а и его внутренней сети на Клиентов, зачем-то запущенных не в Docker-е, а просто отдельно на нашем компьютере с использованием хоста “localhost”. И это ломает основную логику микросервисов, каждый из которых должен быть запущен на разных компьютерах, для иммитации чего и создан Docker.

KAFKA\_ADVERTISED\_LISTENERS: INSIDE://kafka:9093,OUTSIDE://1.2.3.4:9092

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: INSIDE:PLAINTEXT,OUTSIDE:SASL\_PLAINTEXT

KAFKA\_LISTENERS: INSIDE://0.0.0.0:9093,OUTSIDE://0.0.0.0:9092

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: INSIDE

KAFKA\_OPTS: "-Djava.security.auth.login.config=/etc/kafka/kafka\_server\_jaas.conf"

KAFKA\_SASL\_ENABLED\_MECHANISMS: PLAIN

KAFKA\_SASL\_MECHANISM\_INTER\_BROKER\_PROTOCOL: PLAIN

depends\_on:

- zookeeper

volumes:

- ./:/etc/kafka

KAFKA\_BROKER\_ID — свойство broker.id — это уникальное и постоянное имя каждого узла в кластере.

KAFKA\_AUTO\_CREATE\_TOPICS\_ENABLE – если значение истинно, то оно позволяет брокерам создавать темы, когда на них впервые ссылается производитель или потребитель. Если для значения установлено значение false, тему следует сначала создать с помощью команды Kafka, а затем использовать.

KAFKA\_ZOOKEEPER\_CONNECT — указывает Кафке, как связаться с Zookeeper.

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR –показывает, на сколько брокеров-последователей будут скопированы данные с ведущего-лидера. Требуется, когда мы работаем с кластером с одним узлом. Если у вас три или более узлов, мы можем использовать значение по умолчанию.

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP –определяет пары ключ/значение для протокола безопасности, которые будут использоваться для каждого имени прослушивателя.

KAFKA\_ADVERTISED\_LISTENERS — это то, что будет возвращать брокер кафки клиентам, говоря как с ним (или, в зависимости от темы, с необходимым клиенту брокером) связаться и делает Kafka доступным снаружи контейнера, объявляя его местоположение на хосте Docker.

### Авторизация клиентов

Для того что бы включить авторизацию необходимо передать kafka\_server\_jaas.conf файл в kafka. В папке с compose yaml файлом создадим kafka\_server\_jaas.conf и в нем добавим одного пользователя admin с паролем admin-secret:

**kafka\_server\_jaas.conf**

KafkaServer {

org.apache.kafka.common.security.plain.PlainLoginModule required

username="admin"

password="admin-secret"

user\_admin="admin-secret";

};

Client {};

Этот файл будет передаваться в образ с kafka брокером через:

1. указание пути к файлу в непосредственно конфиге kafka  
   KAFKA\_OPTS: "-Djava.security.auth.login.config=/etc/kafka/kafka\_server\_jaas.conf"
2. связь локальной папки*(текущей папки на нашем компе)* с папкой образа “/etc/kafka” :  
   volumes:  
   - ./ : /etc/kafka

### Запуск контейнеров

В итоге мы должны получить 2 файла, которые находятся в одной директории:

1. docker-compose.yml
2. kafka\_server\_jaas.conf

В этой директории нужно вызвать команду:

$ docker-compose up -d

флаг *-d* позволяет запуститься в Detached mode и закрыть консоль при необходимости без выключения образов.

### Java клиент на Spring Boot

Напишем минималистичный клиент с возможностью публикации и чтения сообщений из kafka. Для этого создадим новый проект через [Spring Initializr](https://start.spring.io/) или любой другой привычный вам инструмент. Будет достаточно подключить одну зависимость:  
  
Настройку Spring Boot приложения я предпочитаю по возможности реализовывать через [конфигурационные yaml файлы](https://docs.spring.io/autorepo/docs/spring-boot/1.2.0.M2/reference/html/boot-features-external-config.html#boot-features-external-config-yaml) с минимальной конфигурацией в Java коде. За счет встроенного структурирования в Yaml-формат, эти файлы гораздо удобнее для человеческого восприятия, чем .properties файлы и позволяют группировать настройки по модулям. Единственный недостаток .yml конфигурации в том, что она не поддерживается в тестах, но он легко обходится встроенными инструментами boot’а.

*Очень рекомендую попробовать посмотреть как изменится ваше восприятие старого .properties файла (особенно для больших проектов), если его переписать в yaml. Это можно сделать в пару кликов на*[*www.toyaml.com/index.html*](https://www.toyaml.com/index.html)*.*

В resources проекта создадим файл application.yml cо следующим содержанием, где сетап будет работать для последовательной вычитки consumer-ом по одному сообщению (**max-poll-records: 1**) одним слушателем (**concurrency: 1**) и требовать ручного подтверждения обработки сообщения в listener (**ack-mode: manual\_immediate**):

**application.yaml**

example:

**kafka:**

consumer-enabled: ${consumer-enabled:true}

**spring:**

**kafka:**

bootstrap-servers: ${kafka\_bootstrap\_servers:localhost:9092}

**properties:**

sasl:

jaas:

config: org.apache.kafka.common.security.plain.PlainLoginModule required username=${kafka\_username:'admin'} password=${kafka\_password:'admin-secret'};

mechanism: PLAIN

security:

protocol: SASL\_PLAINTEXT

**consumer:**

client-id: example

key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

auto-offset-reset: earliest

group-id: example

max-poll-records: 1

fetch-max-wait: 36000

enable-auto-commit: false

**producer:**

client-id: example

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: org.apache.kafka.common.serialization.StringSerializer

retries: 2

**jaas:**

enabled: true

listener:

poll-timeout: 1800000

concurrency: 1

ack-mode: manual\_immediate

**Добавим классы producer и consumer**

Слушатель получает сообщение и логирует все доступные сведения о нем в консоль и комитит факт считывания.

**Запустим добавление сообщений в очередь каждые 3 секунды**

для включения шедулера надо повесить аннотацию @EnableScheduling на наше spring приложение

Voilà! и приложение готово к запуску. Можно запускать dev вариант с локальным клиентом и локальной kafka, а так же к kafka можно получить доступ из других сетей если открыть 9092 порт и указать в application.yml адрес хоста.

**KafkaExampleApplication.java**

package com.layo.kafkaexample;

@EnableScheduling

@SpringBootApplication

public class KafkaExampleApplication {

public static void main(String[] args) {

SpringApplication.run(KafkaExampleApplication.class, args);

}

}

**Producer.java**

package com.layo.kafkaexample.engine;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.kafka.support.SendResult;

import org.springframework.stereotype.Service;

import org.springframework.util.concurrent.ListenableFuture;

@Service

@SuppressWarnings("SpringJavaInjectionPointsAutowiringInspection")

public class **Producer** {

private final KafkaTemplate<String, String> kafkaTemplate;

public Producer(KafkaTemplate<String, String> kafkaTemplate) {

this.kafkaTemplate = kafkaTemplate;

}

public ListenableFuture<SendResult<String, String>> sendMessage(String topic, String key, String message) {

return this.kafkaTemplate.send(topic, key, message);

}

}

**SendMessageTask.java**

package com.layo.kafkaexample.tasks;

import com.layo.kafkaexample.engine.Producer;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.kafka.support.SendResult;

import org.springframework.scheduling.annotation.Scheduled;

import org.springframework.stereotype.Component;

import org.springframework.util.concurrent.ListenableFuture;

import java.time.LocalDate;

import java.util.concurrent.ExecutionException;

@Component

public class SendMessageTask {

private final Logger logger = LoggerFactory.getLogger(SendMessageTask.class);

private final Producer producer;

public SendMessageTask(Producer producer) {

this.producer = producer;

}

// run every 3 sec

@Scheduled(fixedRateString = "3000")

public void send() throws ExecutionException, InterruptedException {

ListenableFuture<SendResult<String, String>> listenableFuture =

this.producer.sendMessage("INPUT\_DATA", "IN\_KEY", LocalDate.now().toString());

SendResult<String, String> result = listenableFuture.get();

logger.info(String.format("Produced:\ntopic: %s\noffset: %d\npartition: %d\nvalue size: %d",result.getRecordMetadata().topic(),result.getRecordMetadata().offset(),result.getRecordMetadata().partition(),result.getRecordMetadata().serializedValueSize()));

}

}

**Consumer.java**

package com.layo.kafkaexample.engine;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.kafka.support.Acknowledgment;

import org.springframework.kafka.support.KafkaHeaders;

import org.springframework.messaging.handler.annotation.Header;

import org.springframework.messaging.handler.annotation.Payload;

import org.springframework.stereotype.Service;

@Service

@ConditionalOnProperty(value = "example.kafka.consumer-enabled", havingValue = "true")

public class **Consumer** {

private final Logger logger = LoggerFactory.getLogger(Producer.class);

@KafkaListener(topics = {"INPUT\_DATA"})

public void consume(final @Payload String message,

final @Header(KafkaHeaders.OFFSET) Integer offset,

final @Header(KafkaHeaders.RECEIVED\_MESSAGE\_KEY) String key,

final @Header(KafkaHeaders.RECEIVED\_PARTITION\_ID) int partition,

final @Header(KafkaHeaders.RECEIVED\_TOPIC) String topic,

final @Header(KafkaHeaders.RECEIVED\_TIMESTAMP) long ts,

final Acknowledgment acknowledgment

) {

logger.info(String.format("#### -> Consumed message -> TIMESTAMP: %d\n%s\noffset: %d\nkey: %s\npartition: %d\ntopic: %s", ts, message, offset, key, partition, topic));

acknowledgment.acknowledge();

}

}

**application.yaml**

example:

kafka:

consumer-enabled: ${consumer-enabled:true}

spring:

kafka:

bootstrap-servers: ${kafka\_bootstrap\_servers:localhost:9092}

properties:

sasl:

jaas:

config: org.apache.kafka.common.security.plain.PlainLoginModule required username=${kafka\_username:'admin'} password=${kafka\_password:'admin-secret'};

mechanism: PLAIN

security:

protocol: SASL\_PLAINTEXT

consumer:

auto-offset-reset: earliest

group-id: example

key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

max-poll-records: 1

fetch-max-wait: 36000

enable-auto-commit: false

client-id: example

producer:

client-id: example

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: org.apache.kafka.common.serialization.StringSerializer

retries: 2

jaas:

enabled: true

listener:

poll-timeout: 1800000

concurrency: 1

ack-mode: manual\_immediate

**docker-compose.yml**

version: '2'

services:

zookeeper:

image: wurstmeister/zookeeper

expose:

- "2181"

kafka:

image: wurstmeister/kafka

ports:

- "9092:9092"

expose:

- "9093"

environment:

KAFKA\_AUTO\_CREATE\_TOPICS\_ENABLE: 'true'

KAFKA\_ADVERTISED\_LISTENERS: INSIDE://kafka:9093,OUTSIDE://1.2.3.4:9092

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: INSIDE:PLAINTEXT,OUTSIDE:SASL\_PLAINTEXT

KAFKA\_LISTENERS: INSIDE://0.0.0.0:9093,OUTSIDE://0.0.0.0:9092

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: INSIDE

KAFKA\_OPTS: "-Djava.security.auth.login.config=/etc/kafka/kafka\_server\_jaas.conf"

KAFKA\_SASL\_ENABLED\_MECHANISMS: PLAIN

KAFKA\_SASL\_MECHANISM\_INTER\_BROKER\_PROTOCOL: PLAIN

depends\_on:

- zookeeper

volumes:

- ./:/etc/kafka

**kafka\_server\_jaas.conf**

KafkaServer {

org.apache.kafka.common.security.plain.PlainLoginModule required

username="admin"

password="admin-secret"

user\_admin="admin-secret";

};

Client {};

**how\_to.md**

[To start messaging with docker-compose installed go to terminal and run](https://github.com/layonez/kafka-example/blob/master/kafka_conf/how_to.md#to-start-messaging-with-docker-compose-installed-go-to-terminal-and-run)

$ docker-compose -f ./docker-compose.yml up -d