Apache Kafka для чайников

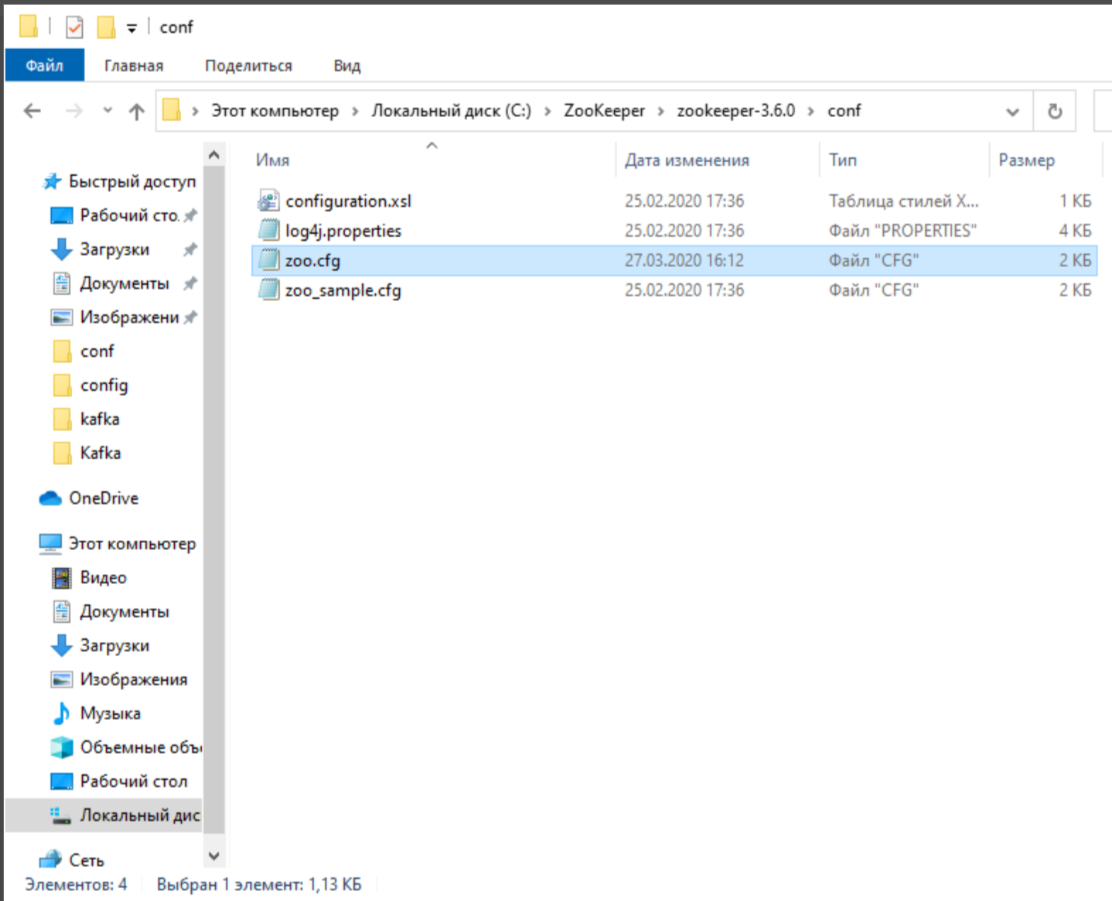
[**https://habr.com/ru/articles/496182/#gallery**](https://habr.com/ru/articles/496182/#gallery)

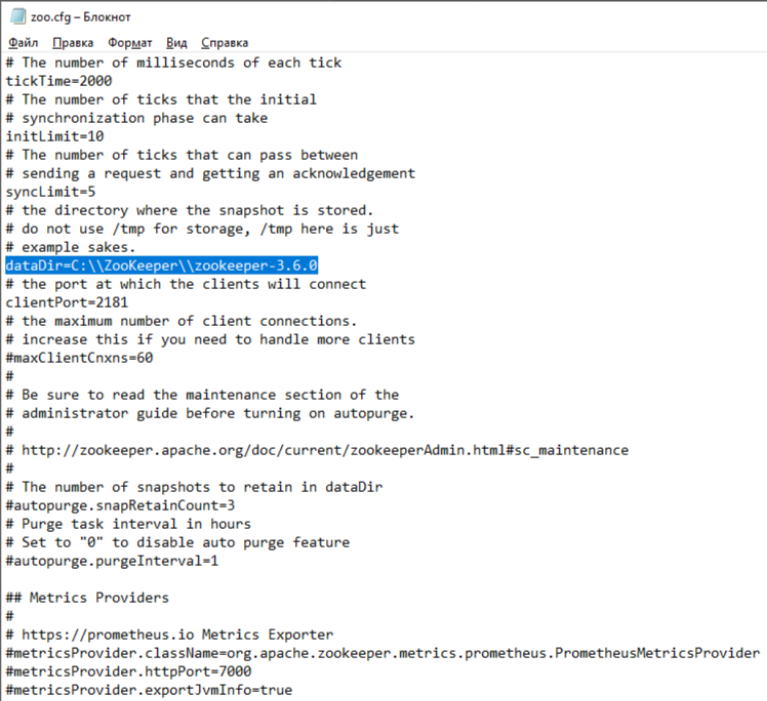
Данная статья будет полезной тем, кто только начал знакомиться с микросервисной архитектурой и с сервисом Apache Kafka. Материал не претендует на подробный туториал, но поможет быстро начать работу с данной технологией. Я расскажу о том, как установить и настроить Kafka на Windows 10. Также мы создадим проект, используя Intellij IDEA и Spring Boot.

**Зачем?**

Трудности в понимании тех или иных инструментов часто связаны с тем, что разработчик никогда не сталкивался с ситуациями, в которых эти инструменты могут понадобиться. С Kafka всё обстоит точно также. Опишем ситуацию, в которой данная технология будет полезной. Если у вас монолитная архитектура приложения, то разумеется, никакая Kafka вам не нужна. Всё меняется с переходом на микросервисы. По сути, каждый микросервис – это отдельная программа, выполняющая ту или иную функцию, и которая может быть запущена независимо от других микросервисов. Микросервисы можно сравнить с сотрудниками в офисе, которые сидят за отдельными столами и независимо от коллег решают свою задачу. Работа такого распределённого коллектива немыслима без централизованной координации. Сотрудники должны иметь возможность обмениваться сообщениями и результатами своей работы между собой. Именно эту проблему и призвана решить Apache Kafka для микросервисов.  
  
Apache Kafka является брокером сообщений. С его помощью микросервисы могут взаимодействовать друг с другом, посылая и получая важную информацию. Возникает вопрос, почему не использовать для этих целей обычный POST – reqest, в теле которого можно передать нужные данные и таким же образом получить ответ? У такого подхода есть ряд очевидных минусов. Например, продюсер (сервис, отправляющий сообщение) может отправить данные только в виде response’а в ответ на запрос консьюмера (сервиса, получающего данные). Допустим, консьюмер отправляет POST – запрос, и продюсер отвечает на него. В это время консьюмер по каким-то причинам не может принять полученный ответ. Что будет с данными? Они будут потеряны. Консьюмеру снова придётся отправлять запрос и надеяться, что данные, которые он хотел получить, за это время не изменились, и продюсер всё ещё готов принять request.  
  
Apache Kafka решает эту и многие другие проблемы, возникающие при обмене сообщениями между микросервисами. Не лишним будет напомнить, что бесперебойный и удобный обмен данными – одна из ключевых проблем, которую необходимо решить для обеспечения устойчивой работы микросервисной архитектуры.

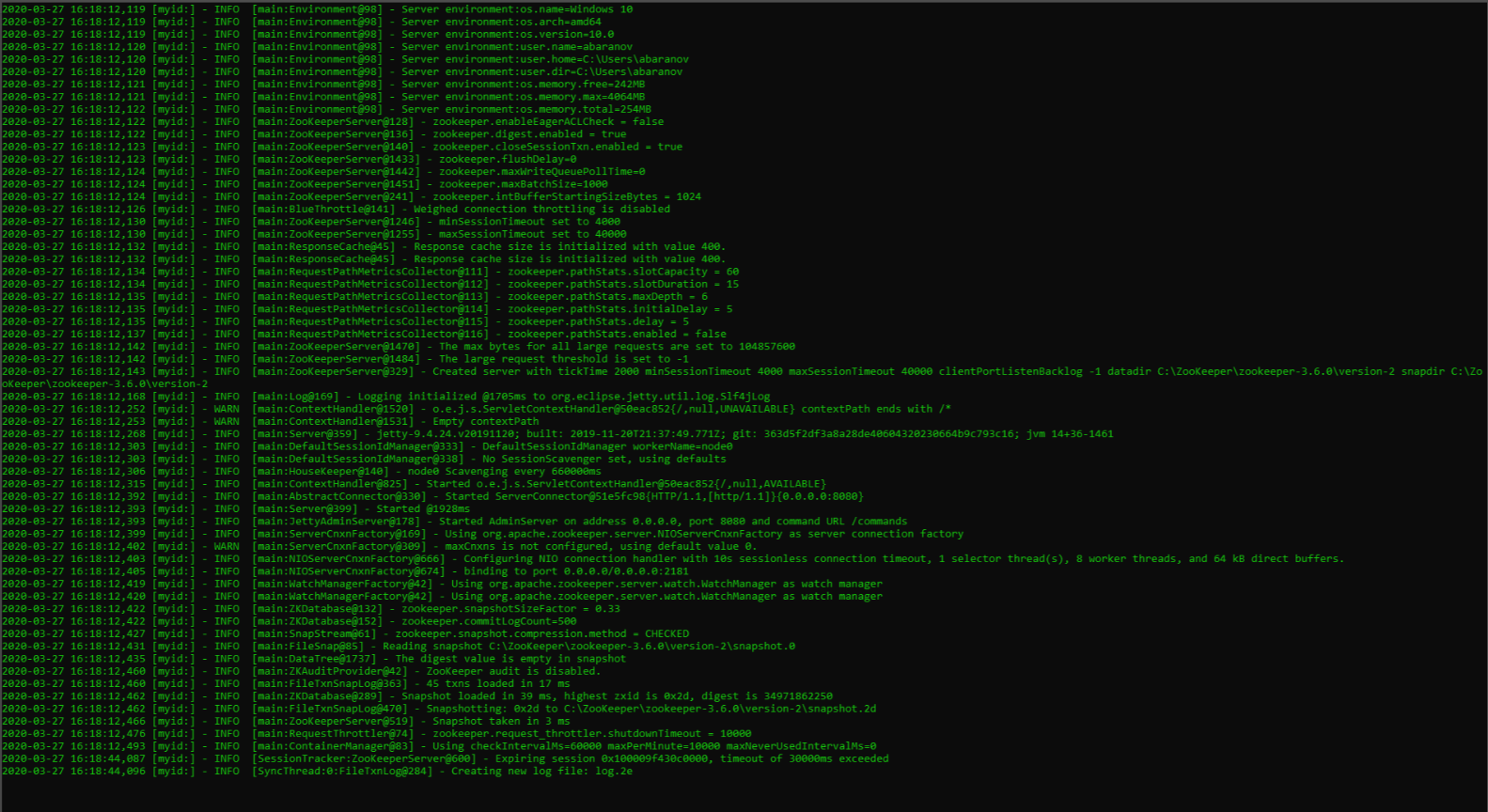
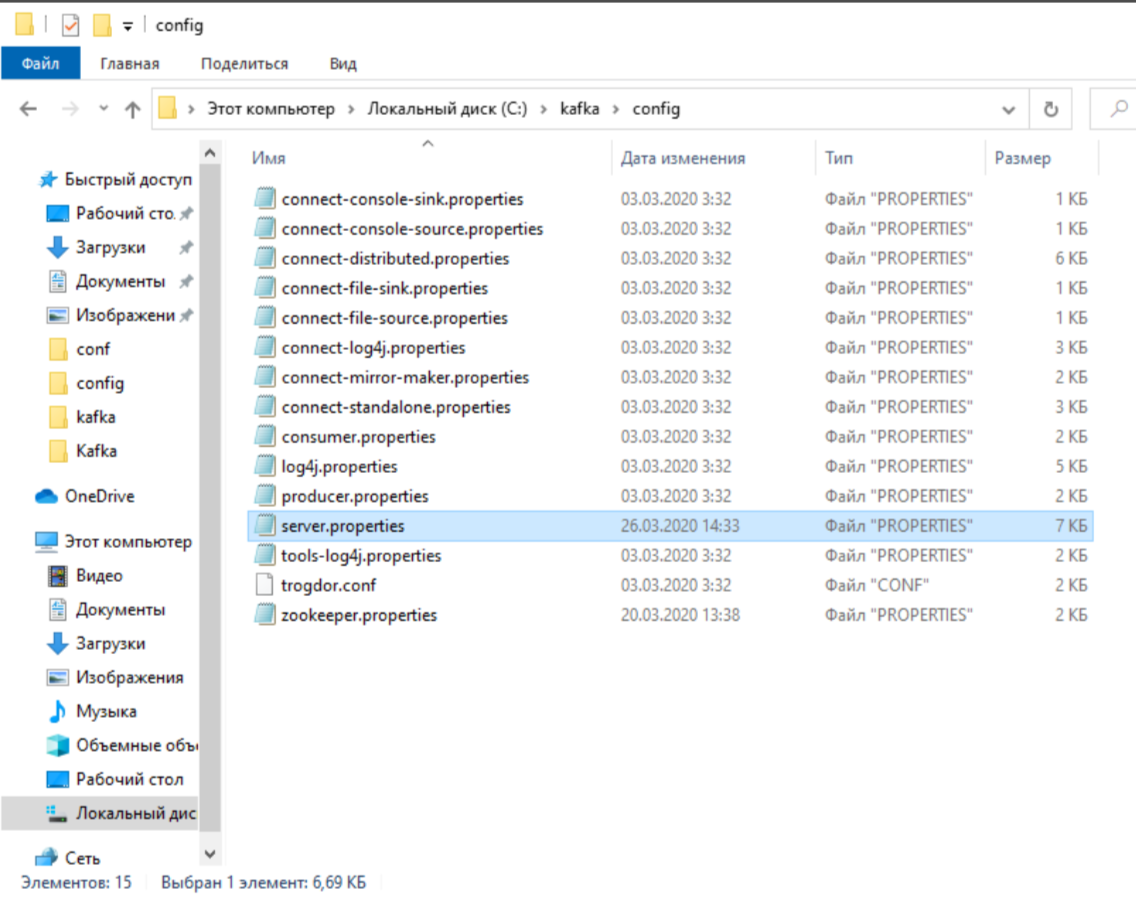
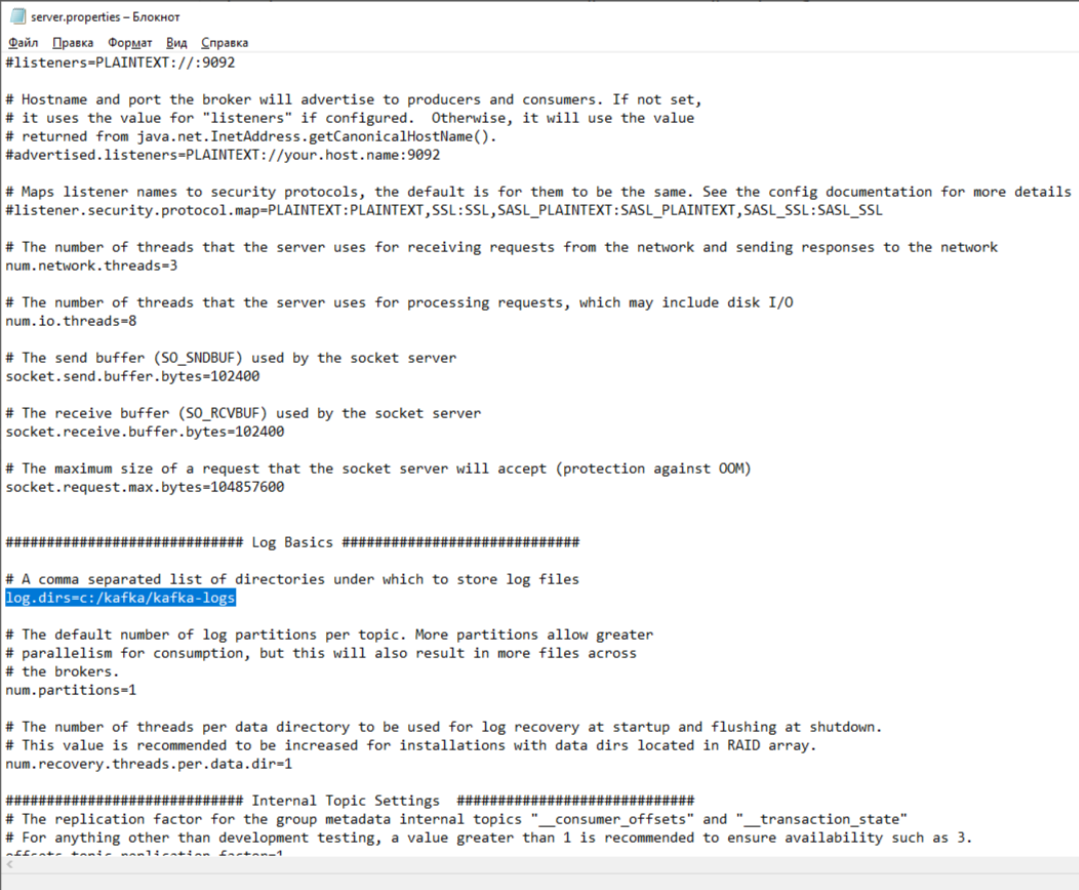
**Установка и настройка ZooKeeper и Apache Kafka на Windows 10**

Первое, что надо знать для начала работы — это то, что Apache Kafka работает поверх сервиса ZooKeeper. ZooKeeper — это распределенный сервис конфигурирования и синхронизации, и это всё, что нам нужно знать о нём в данном контексте. Мы должны скачать, настроить и запустить его перед тем, как начать работу с Kafka. Прежде чем начать работу с ZooKeeper, убедитесь, что у вас установлен и настроен JRE.  
  
[Скачать свежею версию ZooKeeper можно с официального сайта](https://zookeeper.apache.org/releases.html#download).  
  
Извлекаем из скаченного архива ZooKeeper`а файлы в какую-нибудь папку на диске.  
В папке zookeeper с номером версии, находим папку conf и в ней файл “zoo\_sample.cfg”.  
  
  
  
Копируем его и меняем название копии на “zoo.cfg”. Открываем файл-копию и находим в нём строчку dataDir=/tmp/zookeeper. Прописываем в данной строчке полный путь к нашей папке zookeeper-х.х.х. У меня это выглядит так: dataDir=C:\\ZooKeeper\\zookeeper-3.6.0



Теперь добавим системную переменную среды: ZOOKEEPER\_HOME = C:\ ZooKeeper \zookeeper-3.4.9 и в конце системной переменной Path добавим запись: ;%ZOOKEEPER\_HOME%\bin;  
  
Запускаем командную строку и пишем команду:

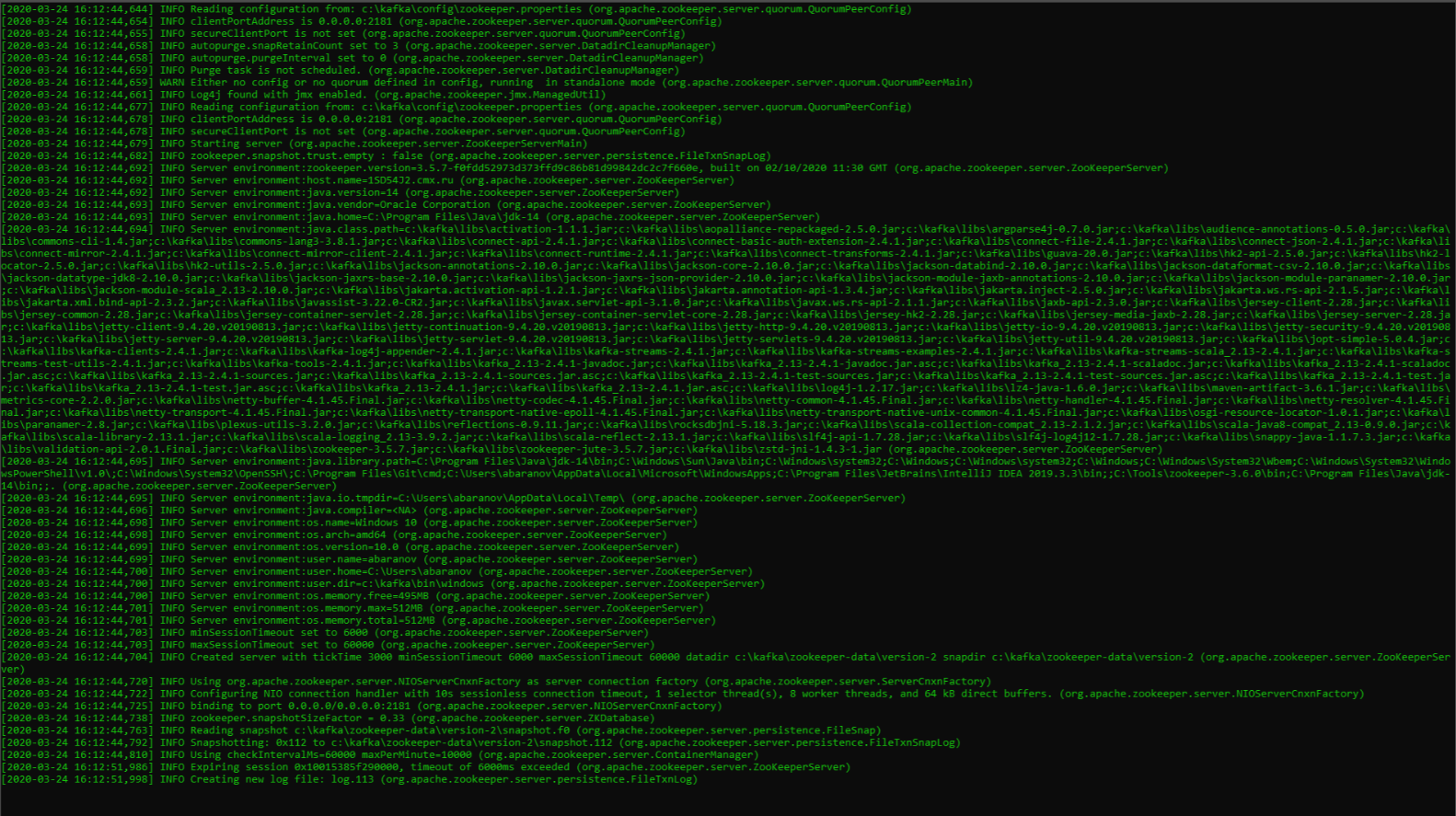
zkserver

Если всё сделано правильно, вы увидите примерно следующее.  
  
  
  
Это означает, что ZooKeeper стартанул нормально. Переходим непосредственно к установке и настройке сервера Apache Kafka. Скачиваем свежую версию с официального сайта и извлекаем содержимое архива: [kafka.apache.org/downloads](https://kafka.apache.org/downloads)  
  
В папке с Kafka находим папку config, в ней находим файл server.properties и открываем его.  
  
  
  
Находим строку log.dirs= /tmp/kafka-logs и указываем в ней путь, куда Kafka будет сохранять логи: log.dirs=c:/kafka/kafka-logs.  
  
  
  
В этой же папке редактируем файл zookeeper.properties. Строчку dataDir=/tmp/zookeeper меняем на dataDir=c:/kafka/zookeeper-data, не забывая при этом, после имени диска указывать путь к своей папке с Kafka. Если вы всё сделали правильно, можно запускать ZooKeeper и Kafka.  
  
Для кого-то может оказаться неприятной неожиданностью, что никакого GUI для управления Kafka нет. Возможно, это потому, что сервис рассчитан на суровых нёрдов, работающих исключительно с консолью. Так или иначе, для запуска кафки нам потребуется командная строка.  
  
Сначала надо запустить ZooKeeper. В папке с кафкой находим папку bin/windows, в ней находим файл для запуска сервиса zookeeper-server-start.bat, кликаем по нему. Ничего не происходит? Так и должно быть. Открываем в этой папке консоль и пишем:

start zookeeper-server-start.bat

Опять не работает? Это норма. Всё потому что zookeeper-server-start.bat для своей работы требует параметры, прописанные в файле zookeeper.properties, который, как мы помним, лежит в папке config. Пишем в консоль:

start zookeeper-server-start.bat c:\kafka\config\zookeeper.properties

Теперь всё должно стартануть нормально.  
  
  
  
Ещё раз открываем консоль в этой папке (ZooKeeper не закрывать!) и запускаем kafka:

start kafka-server-start.bat c:\kafka\config\server.properties

Для того, чтобы не писать каждый раз команды в командной строке, можно воспользоваться старым проверенным способом и создать батник со следующим содержимым:

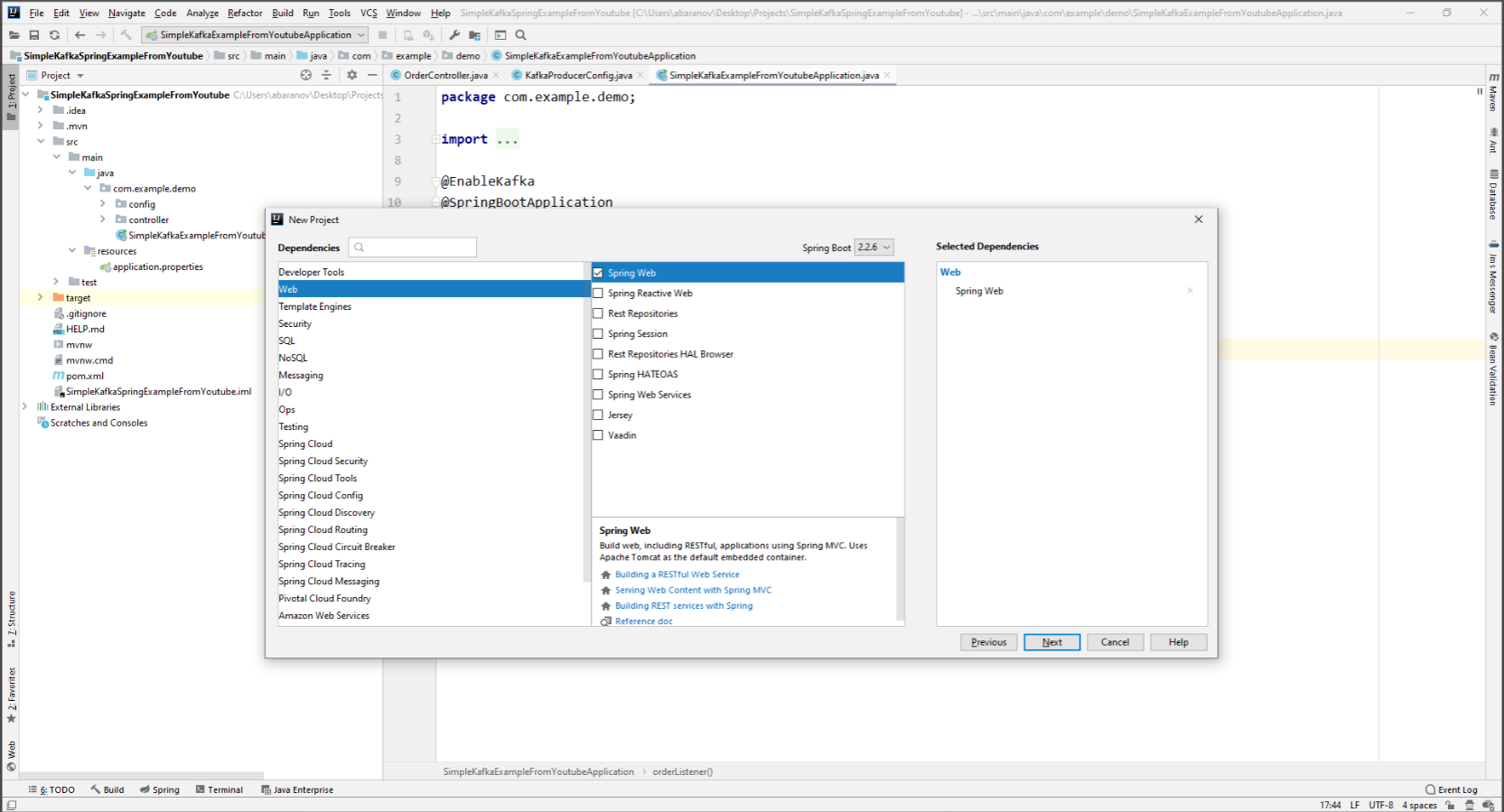
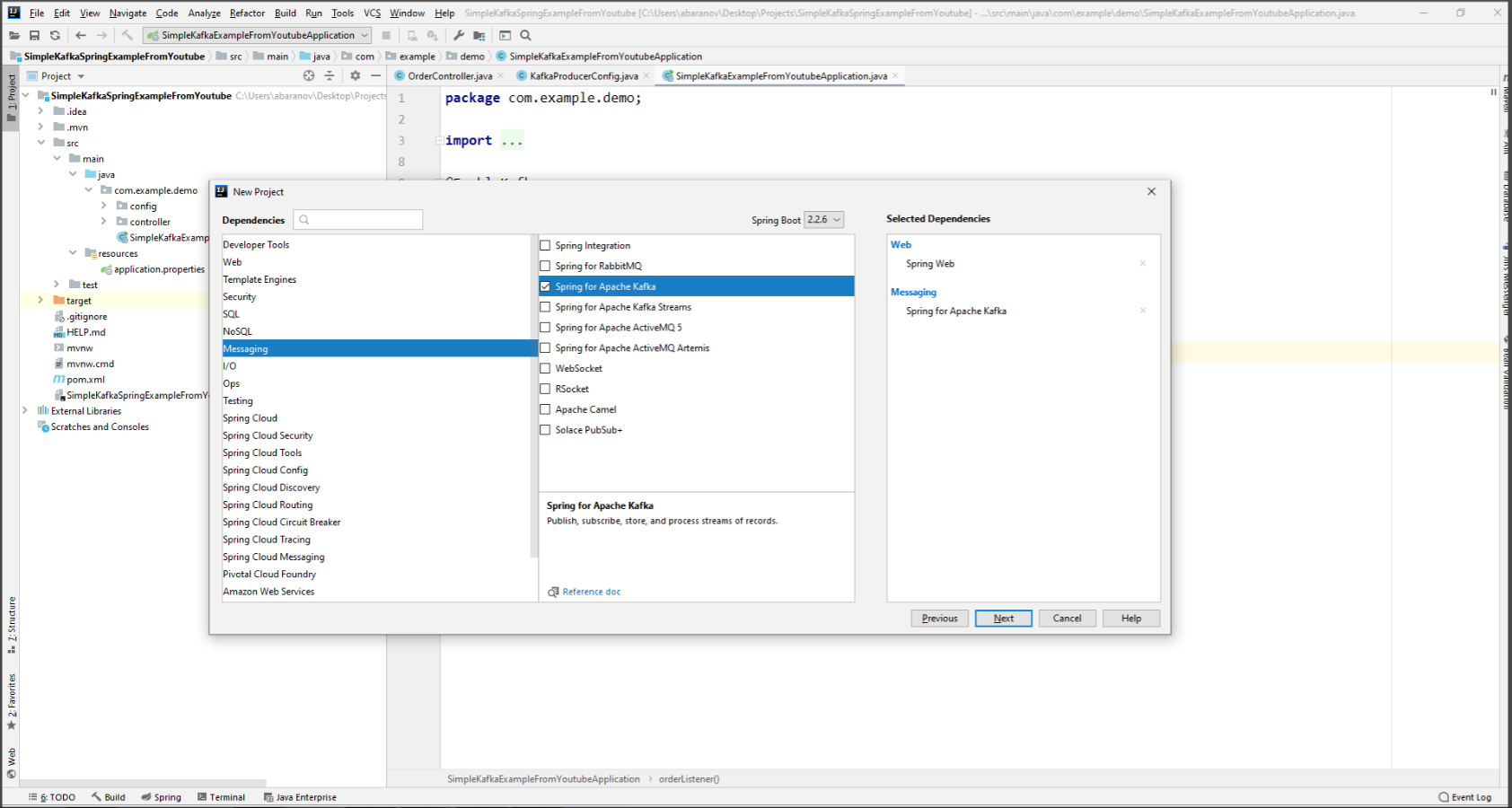
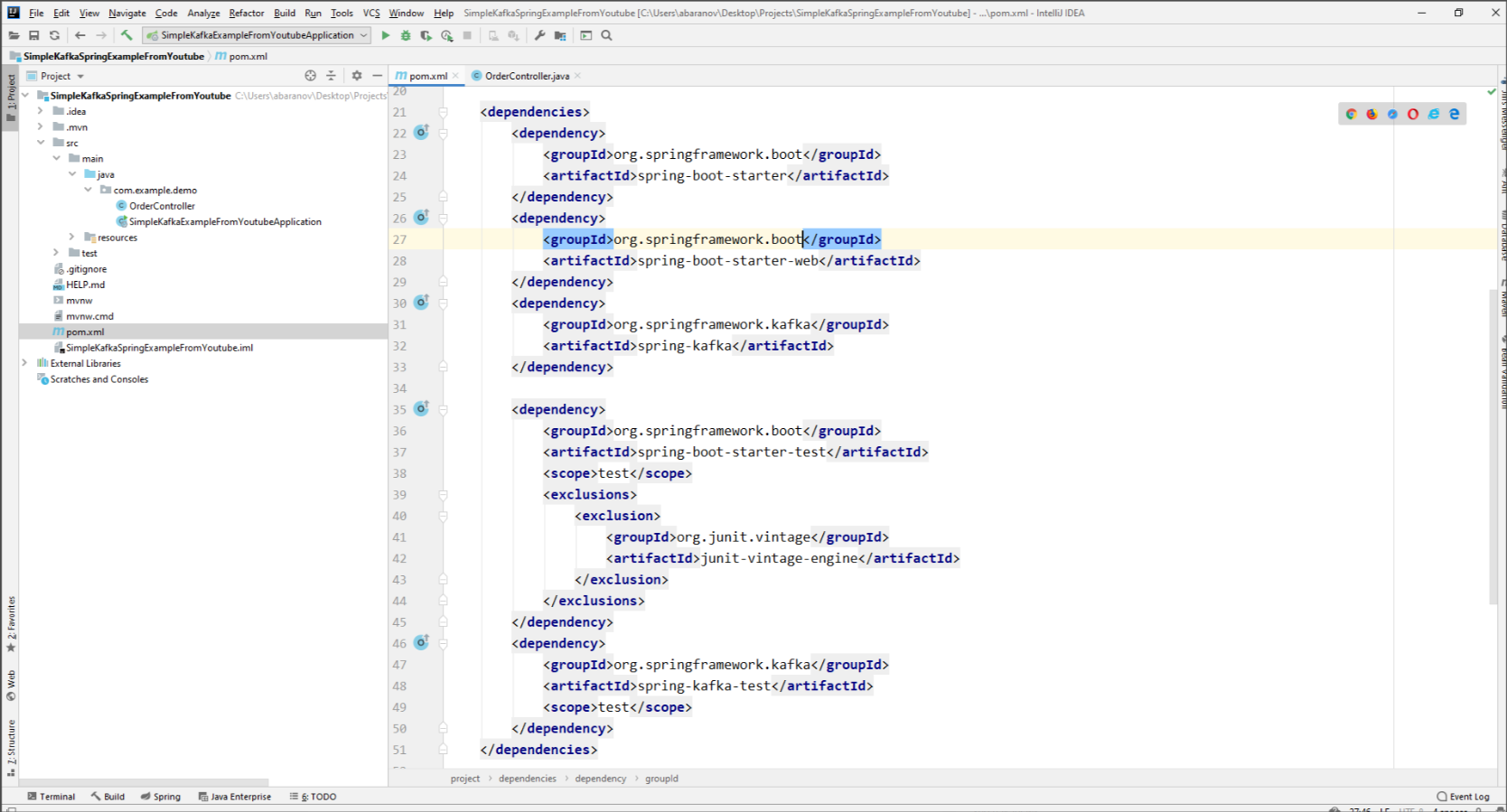
start C:\kafka\bin\windows\zookeeper-server-start.bat C:\kafka\config\zookeeper.properties

timeout 10

start C:\kafka\bin\windows\kafka-server-start.bat C:\kafka\config\server.properties

Строка timeout 10 нужна для того, чтобы задать паузу между запуском zookeeper и kafka. Если вы всё сделали правильно, при клике на батник должны открыться две консоли с запущенным zookeeper и kafka.Теперь мы можем прямо из командной строки создать продюсера сообщений и консьюмера с нужными параметрами. Но, на практике это может понадобиться разве что для тестирования сервиса. Гораздо больше нас будет интересовать, как работать с kafka из IDEA.

**Работа с kafka из IDEA**

Мы напишем максимально простое приложение, которое одновременно будет и продюсером и консьюмером сообщения, а затем добавим в него полезные фичи. Создадим новый спринг-проект. Удобнее всего делать это с помощью спринг-инициалайзера. Добавляем зависимости org.springframework.kafka и spring-boot-starter-web  
  
  
  
  
  
В итоге файл pom.xml должен выглядеть так:  
  
  
  
Для того, чтобы отправлять сообщения, нам потребуется объект KafkaTemplate<K, V>. Как мы видим объект является типизированным. Первый параметр – это тип ключа, второй – самого сообщения. Пока оба параметра мы укажем как String. Объект будем создавать в классе-рестконтроллере. Объявим KafkaTemplate и попросим Spring инициализировать его, поставив аннотацию [Autowired](https://habr.com/ru/users/autowired/).

@Autowired

**private** KafkaTemplate<String, String> kafkaTemplate;

В принципе, наш продюсер готов. Всё что осталось сделать – это вызвать у него метод send(). Имеется несколько перегруженных вариантов данного метода. Мы используем в нашем проекте вариант с 3 параметрами — send(String topic, K key, V data). Так как KafkaTemplate типизирован String-ом, то ключ и данные в методе send будут являться строкой. Первым параметром указывается топик, то есть тема, в которую будут отправляться сообщения, и на которую могут подписываться консьюмеры, чтобы их получать. Если топик, указанный в методе send не существует, он будет создан автоматически. Полный текст класса выглядит так.

@RestController

@RequestMapping("msg")

**public** **class** **MsgController** {

@Autowired

**private** KafkaTemplate<String, String> kafkaTemplate;

@PostMapping

**public** **void** **sendOrder**(String msgId, String msg){

kafkaTemplate.send("msg", msgId, msg);

}

}

Контроллер мапится на [localhost](http://localhost/):8080/msg, в теле запроса передаётся ключ и само сообщений.  
  
Отправитель сообщений готов, теперь создадим слушателя. Spring так же позволяет cделать это без особых усилий. Достаточно создать метод и пометить его аннотацией @KafkaListener, в параметрах которой можно указать только топик, который будет слушаться. В нашем случае это выглядит так.

@KafkaListener(topics="msg")

У самого метода, помеченного аннотацией, можно указать один принимаемый параметр, имеющий тип сообщения, передаваемого продюсером.  
  
Класс, в котором будет создаваться консьюмер необходимо пометить аннотацией @EnableKafka.

@EnableKafka

@SpringBootApplication

**public** **class** **SimpleKafkaExampleApplication** {

@KafkaListener(topics="msg")

**public** **void** **msgListener**(String msg){

System.out.println(msg);

}

**public** **static** **void** **main**(String[] args) {

SpringApplication.run(SimpleKafkaExampleApplication.class, args);

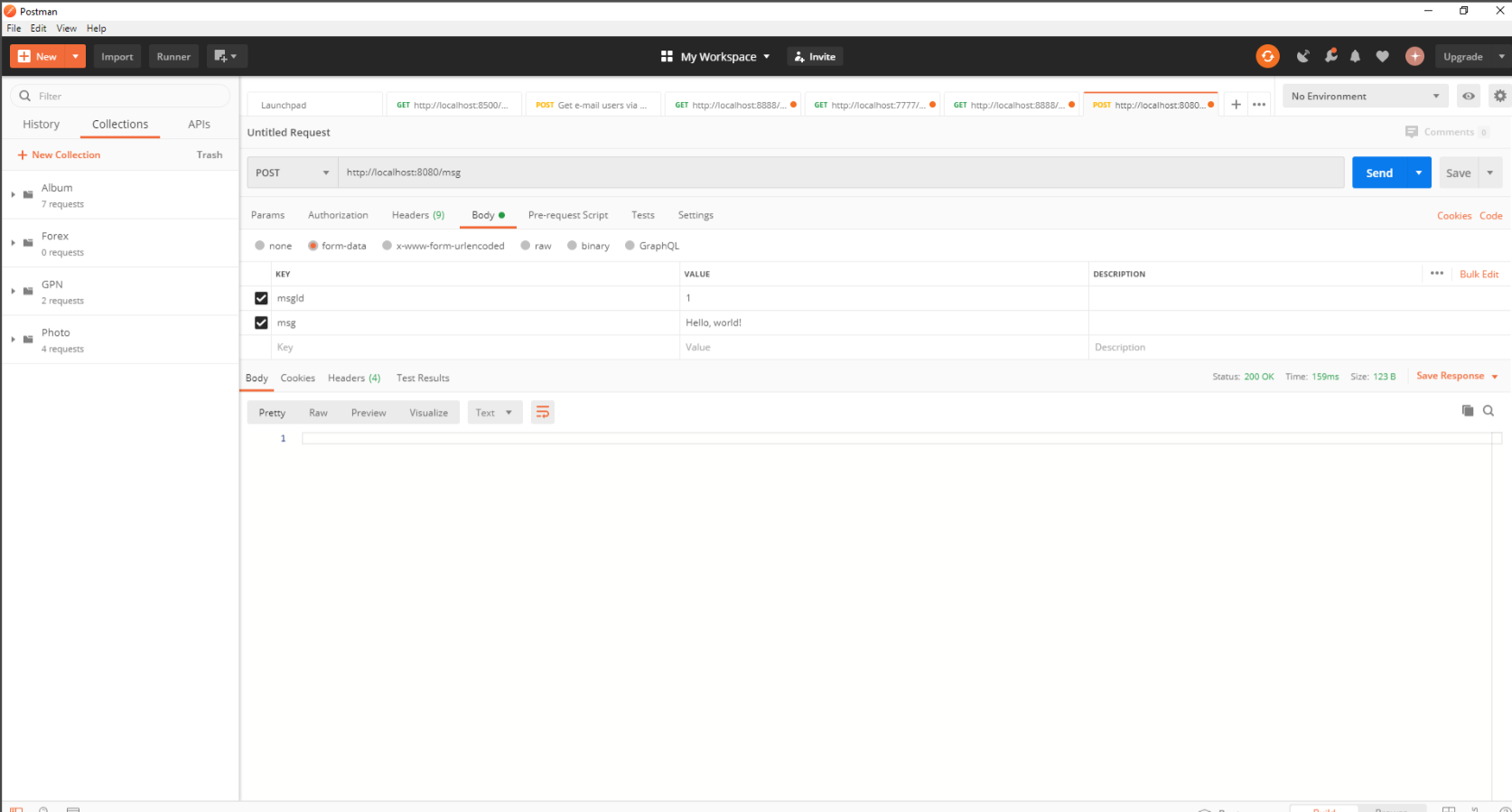
}

}

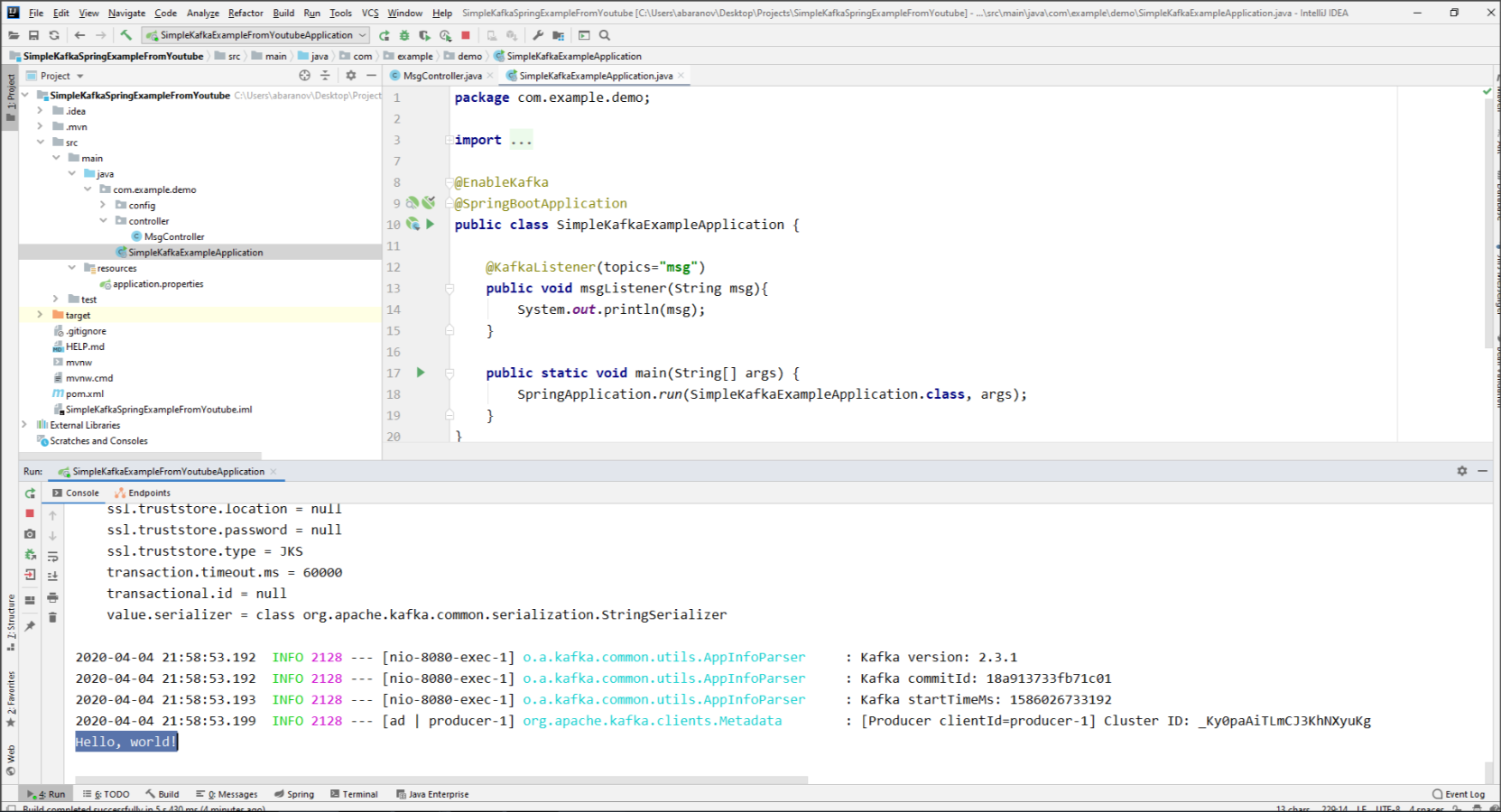
Так же в файле настроек application.property необходимо указать параметр консьюмера groupe-id. Если этого не сделать, приложение не запустится. Параметр имеет тип String и может быть любым.

spring.kafka.consumer.**group**-id=app.1

Наш простейший кафка-проект готов. У нас есть отправитель и получатель сообщений. Осталось только запустить. Для начала запускаем ZooKeeper и Kafka с помощью батника, который мы написали ранее, затем запускаем наше приложение. Отправлять запрос удобнее всего с помощью Postman. В теле запроса не забываем указывать параметры msgId и msg.



Если мы видим в IDEA такую картину, значит всё работает: продюсер отправил сообщение, консьюмер получил его и вывел в консоль.



**Усложняем проект**

Реальные проекты с использованием Kafka конечно же сложнее, чем тот, который мы создали. Теперь, когда мы разобрались с базовыми функциями сервиса, рассмотрим, какие дополнительные возможности он предоставляет. Для начала усовершенствуем продюсера.  
  
Если вы открывали метод send(), то могли заметить, что у всех его вариантов есть возвращаемое значение ListenableFuture<SendResult<K, V>>. Сейчас мы не будем подробно рассматривать возможности данного интерфейса. Здесь будет достаточно сказать, что он нужен для просмотра результата отправки сообщения.

@PostMapping

**public** **void** **sendMsg**(String msgId, String msg){

ListenableFuture<SendResult<String, String>> future = kafkaTemplate.send("msg", msgId, msg);

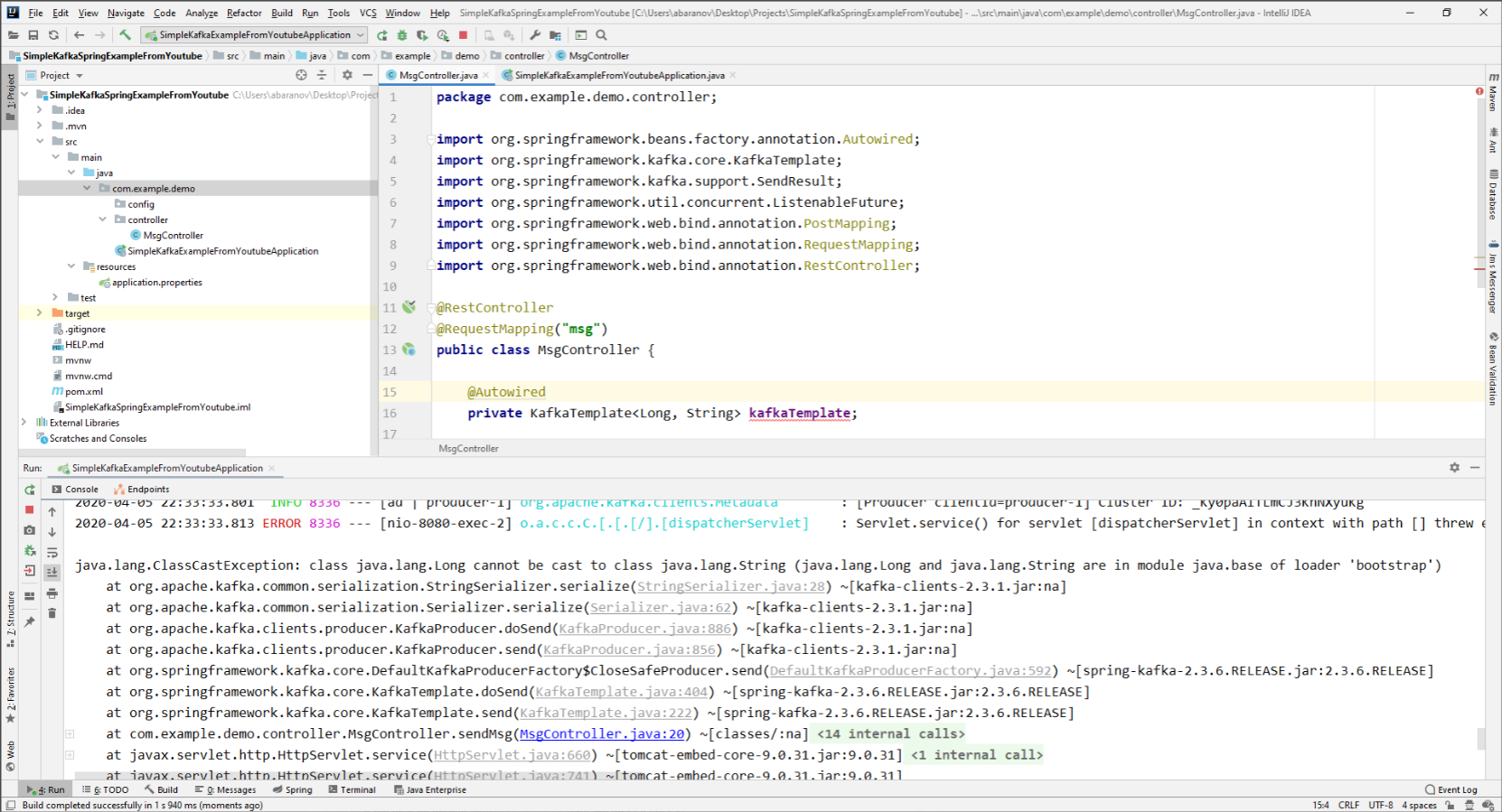
future.addCallback(System.out::println, System.err::println);

kafkaTemplate.flush();

}

Метод addCallback() принимает два параметра – SuccessCallback и FailureCallback. Оба они являются функциональными интерфейсами. Из названия можно понять, что метод первого будет вызван в результате успешной отправки сообщения, второго – в результате ошибки.Теперь, если мы запустим проект, то увидим на консоли примерно следующее:

SendResult [producerRecord=ProducerRecord(topic=msg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=1, **value**=Hello, world!, timestamp=null), recordMetadata=msg-0@6]

Посмотрим ещё раз внимательно на нашего продюсера. Интересно, что будет если в качестве ключа будет не String, а, допустим, Long, а в качестве передаваемого сообщения и того хуже – какая-нибудь сложная DTO? Попробуем для начала изменить ключ на числовое значение…  
Если мы укажем в продюсере в качестве ключа Long, то приложение нормально запуститься, но при попытке отправить сообщение будет выброшен ClassCastException и будет сообщено, что класс Long не может быть приведён к классу String.  
  
  
  
Если мы попробуем вручную создать объект KafkaTemplate, то увидим, что в конструктор в качестве параметра передаётся объект интерфейса ProducerFactory<K, V>, например DefaultKafkaProducerFactory<>. Для того, чтобы создать DefaultKafkaProducerFactory, нам нужно в его конструктор передать Map, содержащий настройки продюсера. Весь код по конфигурации и созданию продюсера вынесем в отдельный класс. Для этого создадим пакет config и в нём класс KafkaProducerConfig.

@Configuration

**public** **class** **KafkaProducerConfig** {

**private** String kafkaServer="localhost:9092";

@Bean

**public** Map<String, Object> **producerConfigs**() {

Map<String, Object> props = **new** HashMap<>();

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG,

kafkaServer);

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG,

LongSerializer.class);

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG,

StringSerializer.class);

**return** props;

}

@Bean

**public** ProducerFactory<Long, String> **producerFactory**() {

**return** **new** DefaultKafkaProducerFactory<>(producerConfigs());

}

@Bean

**public** KafkaTemplate<Long, String> **kafkaTemplate**() {

**return** **new** KafkaTemplate<>(producerFactory());

}

}

В методе producerConfigs() создаём мапу с конфигурациями и в качестве сериализатора для ключа указываем LongSerializer.class. Запускаем, отправляем запрос из Postman и видим, что теперь всё работает, как надо: продюсер отправляет сообщение, а консьюмер принимает его.  
  
Теперь изменим тип передаваемого значения. Что если у нас не стандартный класс из библиотеки Java, а какой-нибудь кастомный DTO. Допустим такой.

@Data

**public** **class** **UserDto** {

**private** Long age;

**private** String name;

**private** Address address;

}

@Data

@AllArgsConstructor

**public** **class** **Address** {

**private** String country;

**private** String city;

**private** String street;

**private** Long homeNumber;

**private** Long flatNumber;

}

Для отправки DTO в качестве сообщения, нужно внести некоторые изменения в конфигурацию продюсера. В качестве сериализатора значения сообщения укажем JsonSerializer.class и не забудем везде изменить тип String на UserDto.

@Configuration

**public** **class** **KafkaProducerConfig** {

**private** String kafkaServer="localhost:9092";

@Bean

**public** Map<String, Object> **producerConfigs**() {

Map<String, Object> props = **new** HashMap<>();

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG,

kafkaServer);

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG,

LongSerializer.class);

props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG,

JsonSerializer.class);

**return** props;

}

@Bean

**public** ProducerFactory<Long, UserDto> **producerFactory**() {

**return** **new** DefaultKafkaProducerFactory<>(producerConfigs());

}

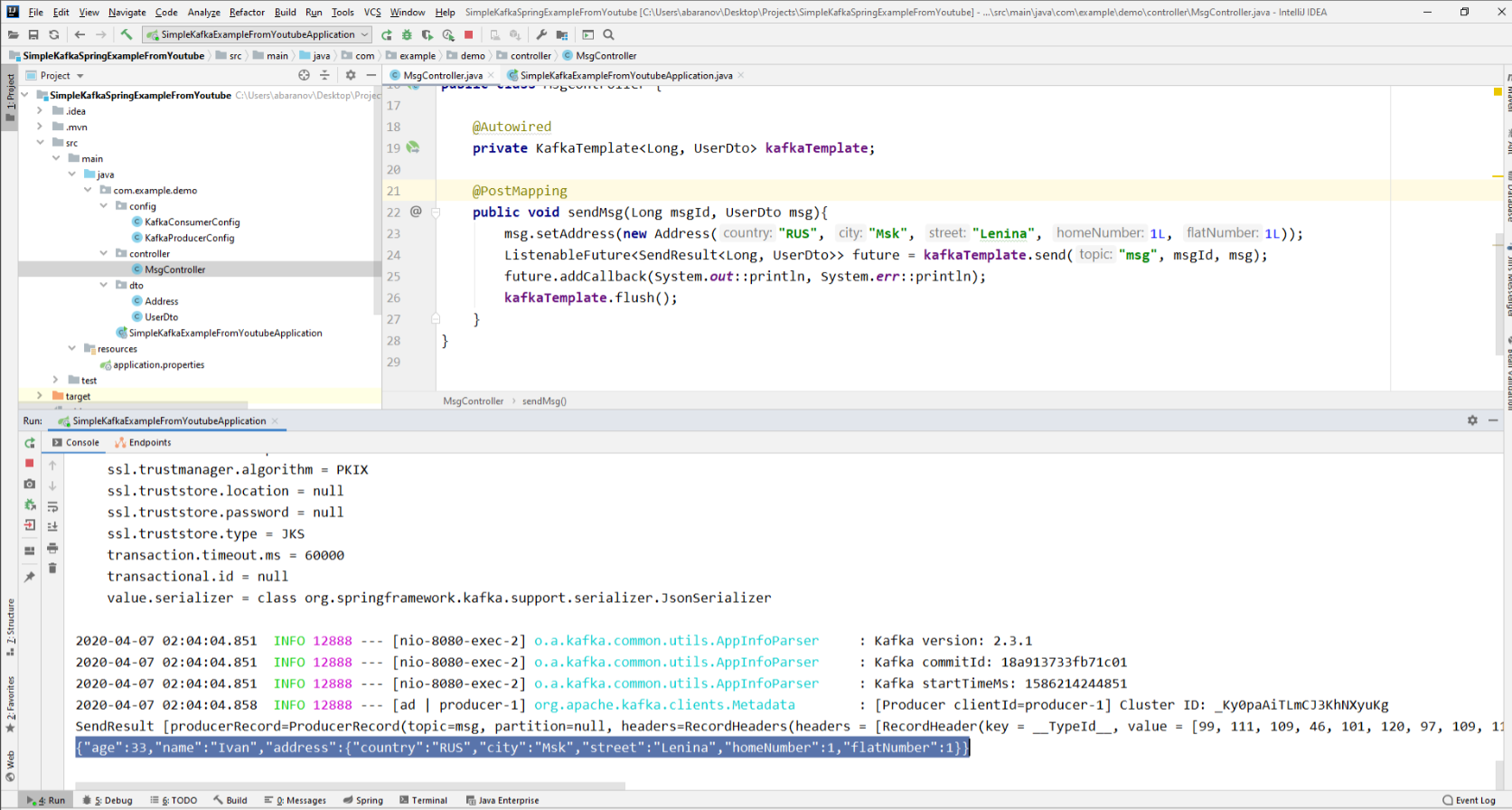
@Bean

**public** KafkaTemplate<Long, UserDto> **kafkaTemplate**() {

**return** **new** KafkaTemplate<>(producerFactory());

}

}

Отправим сообщение. В консоль будет выведена следующая строка:  
  
  
  
Теперь займёмся усложнением консьюмера. До этого наш метод public void msgListener(String msg), помеченный аннотацией @KafkaListener(topics=«msg») в качестве параметра принимал String и выводил его на консоль. Как быть, если мы хотим получить другие параметры передаваемого сообщения, например, ключ или партицию? В этом случае тип передаваемого значения необходимо изменить.

@KafkaListener(topics="msg")

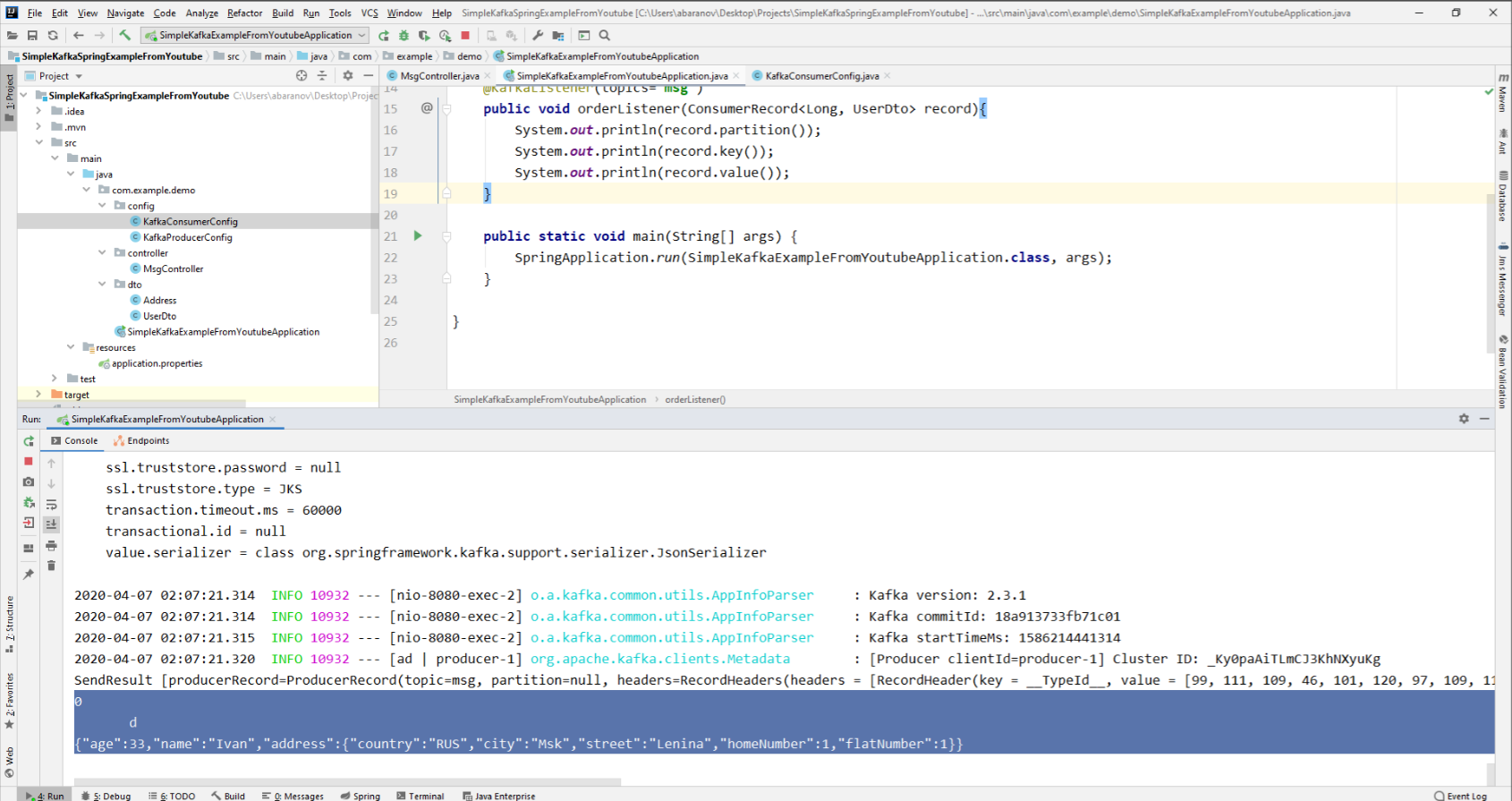
**public** **void** **orderListener**(ConsumerRecord<Long, UserDto> record){

System.out.println(record.partition());

System.out.println(record.key());

System.out.println(record.value());

}

Из объекта ConsumerRecord мы можем получить все интересующие нас параметры.  
  
  
  
Мы видим, что вместо ключа на консоль выводятся какие-то кракозябры. Это потому, что для десериализации ключа по умолчанию используется StringDeserializer, и если мы хотим, чтобы ключ в целочисленном формате корректно отображался, мы должны изменить его на LongDeserializer. Для настройки консьюмера в пакете config создадим класс KafkaConsumerConfig.

@Configuration

**public** **class** **KafkaConsumerConfig** {

@Value("${spring.kafka.bootstrap-servers}")

**private** String kafkaServer;

@Value("${spring.kafka.consumer.group-id}")

**private** String kafkaGroupId;

@Bean

**public** Map<String, Object> **consumerConfigs**() {

Map<String, Object> props = **new** HashMap<>();

props.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, kafkaServer);

props.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, LongDeserializer.class);

props.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

props.put(ConsumerConfig.GROUP\_ID\_CONFIG, kafkaGroupId);

**return** props;

}

@Bean

**public** KafkaListenerContainerFactory<?> kafkaListenerContainerFactory() {

ConcurrentKafkaListenerContainerFactory<Long, UserDto> factory =

**new** ConcurrentKafkaListenerContainerFactory<>();

factory.setConsumerFactory(consumerFactory());

**return** factory;

}

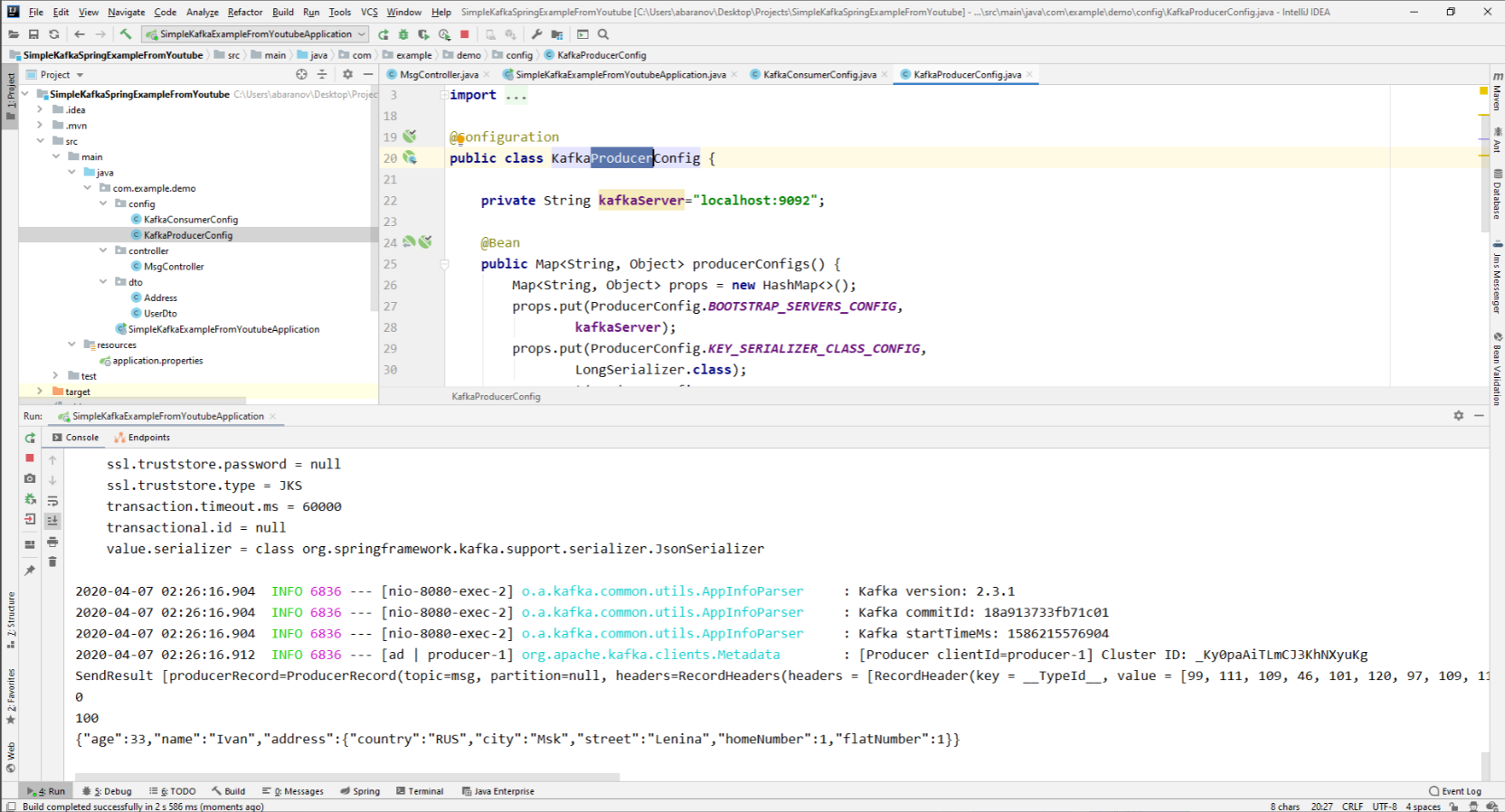
@Bean

**public** ConsumerFactory<Long, UserDto> **consumerFactory**() {

**return** **new** DefaultKafkaConsumerFactory<>(consumerConfigs());

}

}

Класс KafkaConsumerConfig очень похож на KafkaProducerConfig, который мы создавали ранее. Здесь так же присутствует Map, содержащий необходимые конфигурации, например, такие как десериализатор для ключа и значения. Созданная мапа используется при создании ConsumerFactory<>, которая в свою очередь, нужна для создания KafkaListenerContainerFactory<?>. Важная деталь: метод возвращающий KafkaListenerContainerFactory<?> должен называться kafkaListenerContainerFactory(), иначе Spring не сможет найти нужного бина и проект не скомпилируется. Запускаем.  
  
  
Видим, что теперь ключ отображается как надо, а это значит, что всё работает. Конечно, возможности Apache Kafka далеко выходят за пределы тех, что описаны в данной статье, однако, надеюсь, прочитав её, вы составите представление о данном сервисе и, самое главное, сможете начать работу с ним.