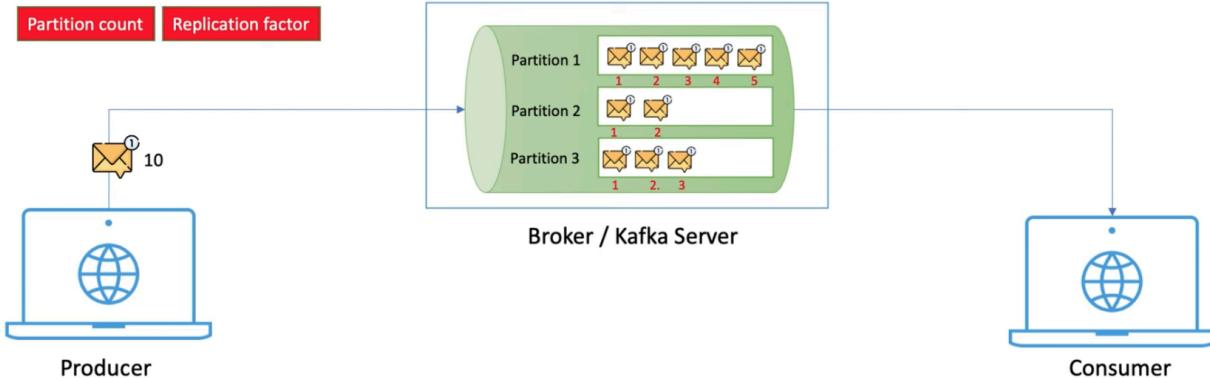
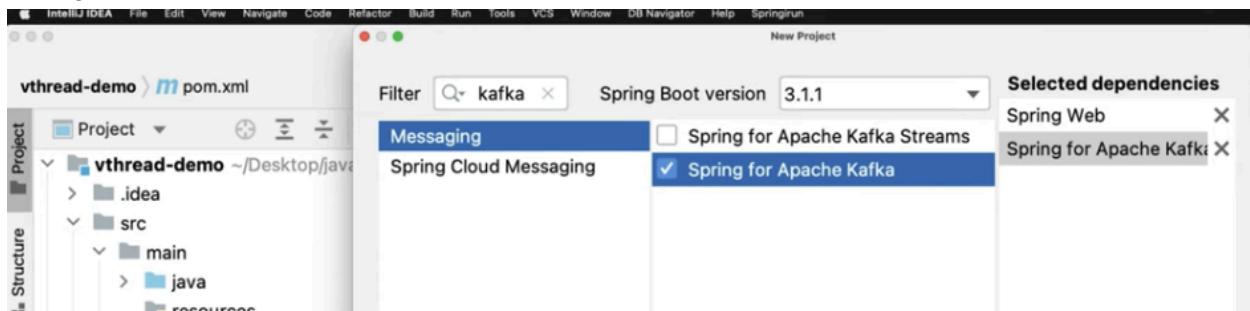


Producer -consumer Flow

1. Start Zookeeper
2. Start Kafka Server
3. Create a Topic

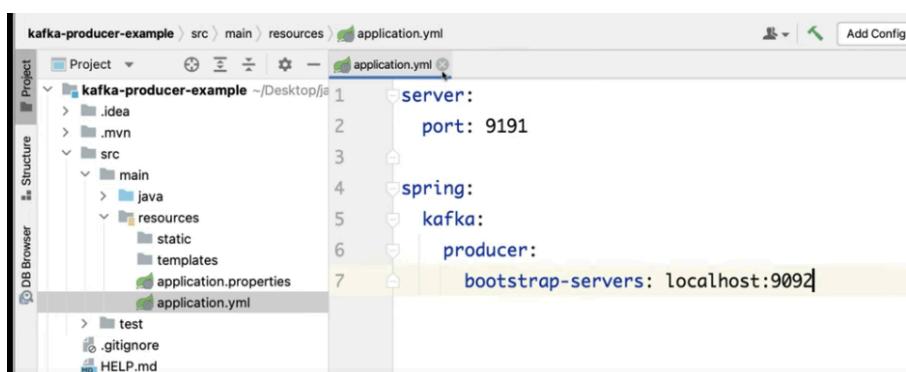


SpringBoot dependencies:



1st start zookeeper and kafka servers

We have to specify these kafka servers in spring boot application



```

11  ic class KafkaMessagePublisher {
12
13     @Autowired
14     private KafkaTemplate<String, Object> template;
15
16     public void sendMessageToTopic(String message){
17         CompletableFuture<SendResult<String, Object>> future = template.send(topic: "jav
18         future.get();
19     }
20
21

```

If `template.send` returns `CompletableFuture`. The `CompletableFuture.get` method blocks the thread => means it will wait for the results to come. It slows down the producer. So we need to handle results asynchronously. So the subsequent message does not wait for the result of the previous message. We can do it using call back implementation as shown below.

Service class

```

15
16     public void sendMessageToTopic(String message){
17         CompletableFuture<SendResult<String, Object>> future = template.send(topic: "jav
18         future.whenComplete((result, ex)->{
19             if (ex == null) {
20                 System.out.println("Sent message=[" + message +
21                     "] with offset=[" + result.getRecordMetadata().offset() + "]");
22             } else {
23                 System.out.println("Unable to send message=[" +
24                     message + "] due to : " + ex.getMessage());
25             }
26         });
27

```

Corresponding Controller class

```

18
19     @GetMapping("/publish/{message}")
20     public ResponseEntity<?> publishMessage(@PathVariable String message){
21         try {
22             publisher.sendMessageToTopic(message);
23             return ResponseEntity.ok("message published successfully ..");
24         } catch (Exception ex){
25             return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
26                 .build();
27         }
28     }
29

```

The screenshot shows the Postman interface. On the left sidebar, there are sections for Collections, APIs, Environments, Mock Servers, Monitors, and History. The History section is currently selected. In the main area, a request is being made to `http://localhost:9191/products/7`. The method is set to `GET`, and the URL is `http://localhost:9191/producer-app/publish/welcome`. The response status is `200 OK`, time is `2.04 s`, and size is `197 B`. The response body contains the text `1 message published successfully ..`.

The screenshot shows a terminal window displaying Kafka logs. The logs show:

- `2023-07-07T13:07:05.352+05:30 WARN 10440 ---`
- `2023-07-07T13:07:05.463+05:30 INFO 10440 ---`
- `sent message=[welcome] with offset=[0]`

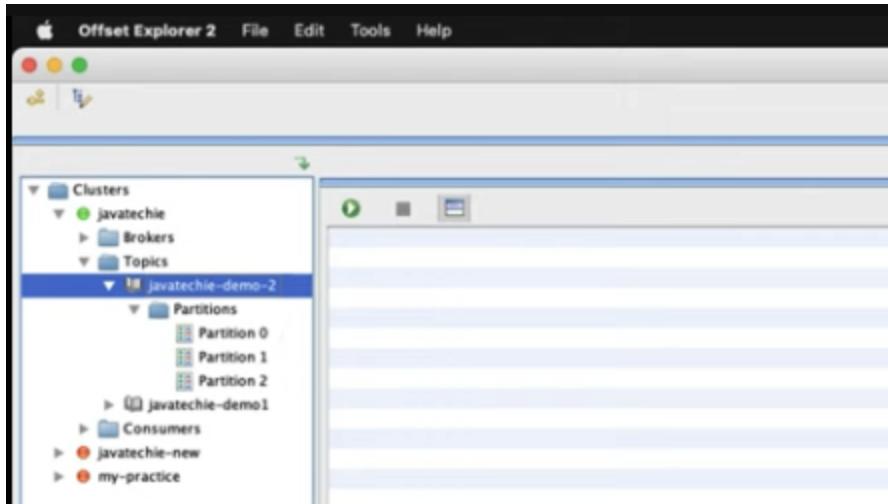
The screenshot shows a terminal window displaying the output of the `kafka-topics.sh --describe` command for the `javatechie-demo1` topic. The output includes:

- `Topic: javatechie-demo1 TopicId: M0dYht_tTGCMcowYl4iyyw PartitionCount: 1 R`
- `eplicationFactor: 1 Configs:`
- `Topic: javatechie-demo1 Partition: 0 Leader: 0 Replicas: 0 I`
- `sr: 0`

By default java will create only one partition dynamically if you have not taken care of creation of topic and partitions. See the above screenshot. See the below screenshot how to create topic with more number of partitions.

The screenshot shows a terminal window displaying the output of the `kafka-topics.sh --create` command for the `javatechie-demo-2` topic. The output includes:

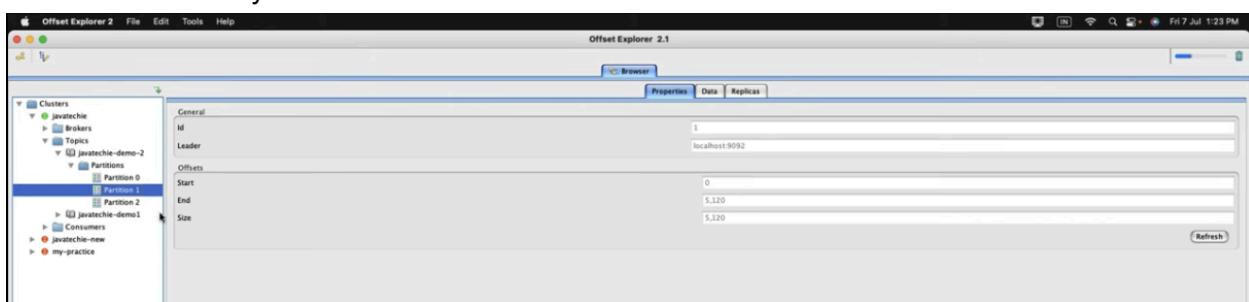
- `sh kafka_2.12-3.4.0 % sh bin/kafka-topics.sh --bootstrap-s`
- `erver localhost:9092 --create --topic javatechie-demo-2 --partitions 3 --replica`
- `tion-factor 1`
- `Created topic javatechie-demo-2.`
- `javatechie@Basantas-iMac kafka_2.12-3.4.0 %`



Let us publish 10000 records with small tweak.

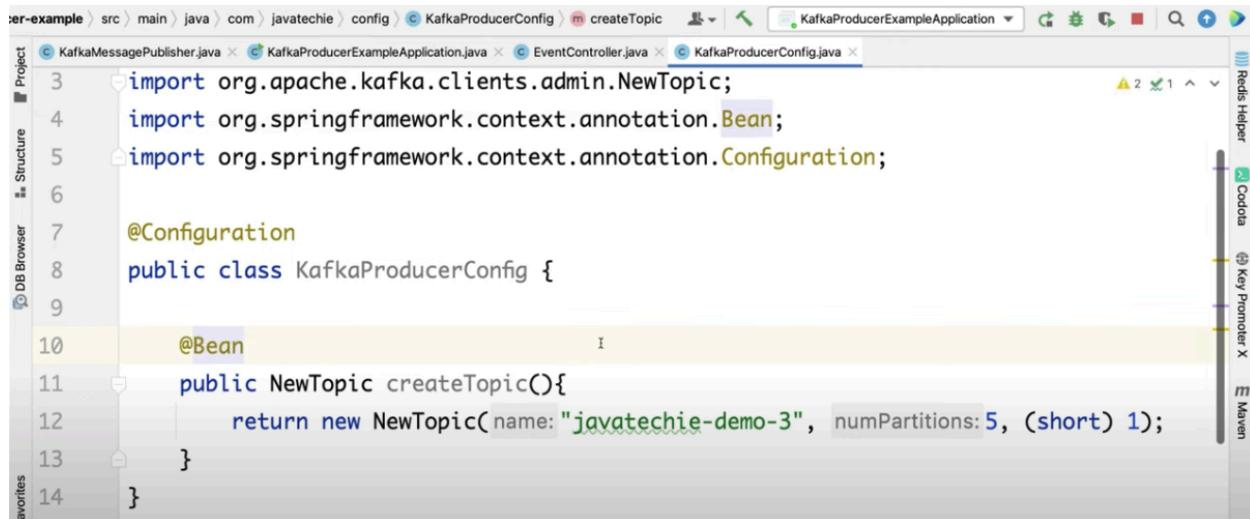
```
@GetMapping("/publish/{message}")
public ResponseEntity<?> publishMessage(@PathVariable String message) {
    try {
        for (int i = 0; i <= 10000; i++) {
            publisher.sendMessageToTopic(message + " : " + i);
        }
        return ResponseEntity.ok("message published successfully ..");
    } catch (Exception ex) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

After hitting the endpoint the load will be distributed between partitions by Kafka scheduler based on availability



In real time we use the kafka portal for creating topics , partitions => this is industry practice and recommended.

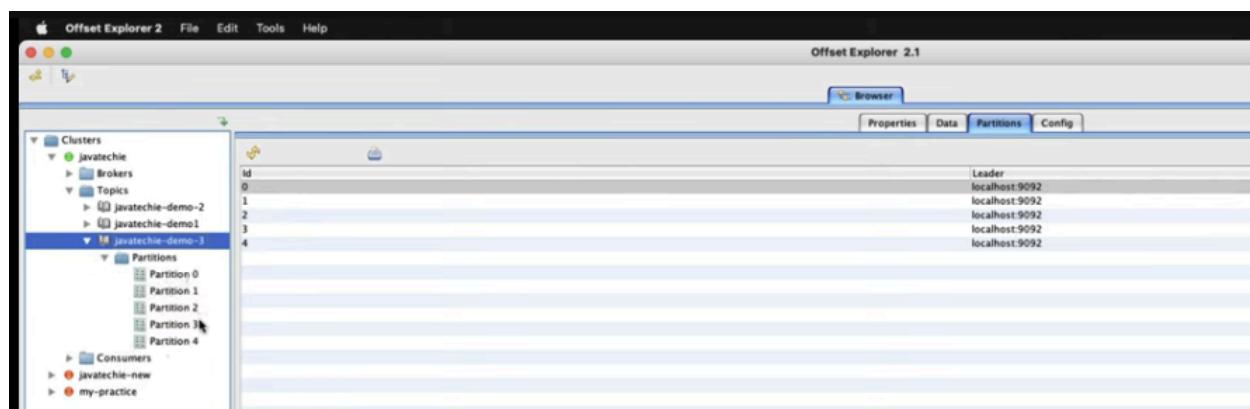
Using springBoot configuration also we can create topic and partition as shown below



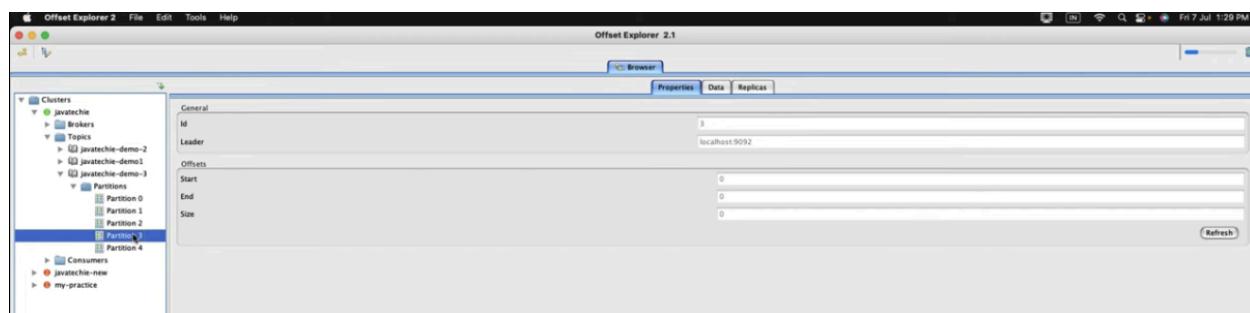
```
import org.apache.kafka.clients.admin.NewTopic;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class KafkaProducerConfig {

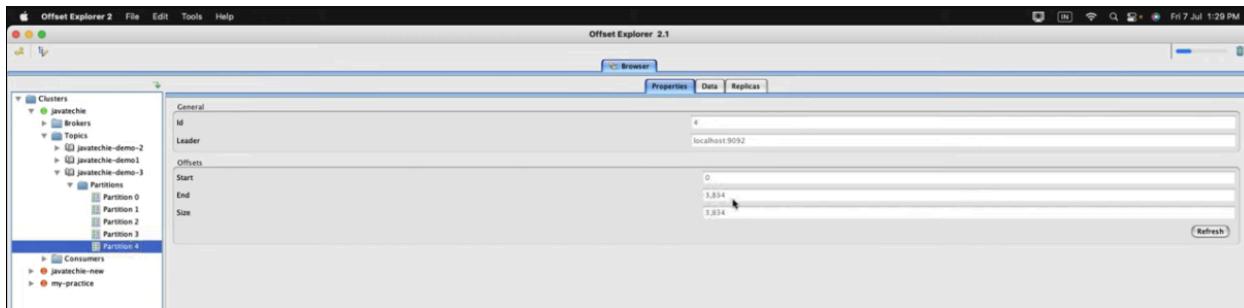
    @Bean
    public NewTopic createTopic(){
        return new NewTopic(name: "javatechie-demo-3", numPartitions: 5, (short) 1);
    }
}
```



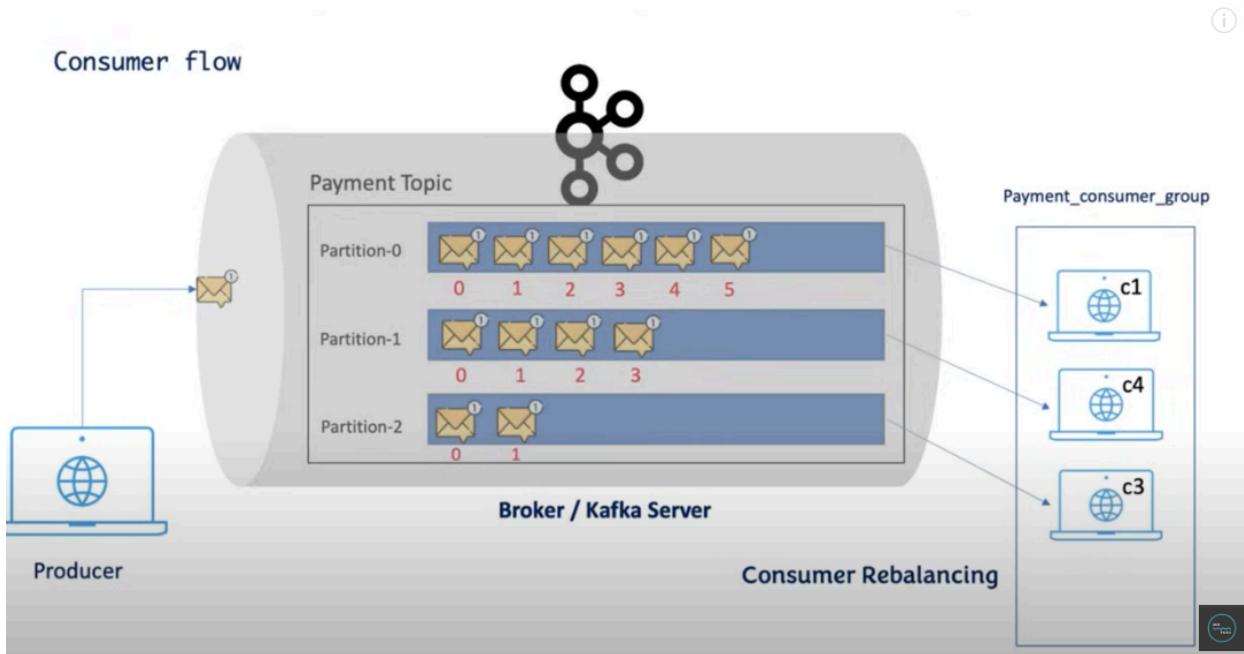
There is a possibility the partition may not receive any messages. That is not in our control. It is in the hands of kafka scheduler



Here partition 3 did not receive any messages but partition 4 received messages.



Now let us see the consumer side



Consumer properties

```

spring:
  kafka:
    consumer:
      bootstrap-servers: localhost:9092
server:
  port: 9292

```

create a listener method

```

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaMessageListener {

    Logger log = LoggerFactory.getLogger(KafkaMessageListener.class);

    @KafkaListener(topics = "javatechie-demo")
    public void consume(String message) {
        log.info("consumer consume the message {} ", message);
    }
}

```

Now start the producer

```

[main] o.apache.kafka.common.metrics.Metrics : Metrics reporters closed
[main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9191 (http) with context
[main] c.j.KafkaProducerExampleApplication : Started KafkaProducerExampleApplication in 4.953 sec

```

Now start the consumer. It gave an error as no group id found in the consumer config.

```

[main] o.s.b.SpringApplication : Metrics reporters closed
[main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9191 (http) with context
[main] c.j.KafkaConsumerExampleApplication : Started KafkaConsumerExampleApplication in 4.953 sec
[main] java.lang.IllegalStateException : Create breakpoint : No group.id found in consumer config, container properties, or @KafkaList

```

For single consumer instance also require group id. Add to listener

```

class KafkaMessageListener {

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

    @KafkaListener(topics = "javatechie-demo", groupId = "jt-group-1")
    public void consume(String message) {
        log.info("consumer consume the message {} ", message);
    }
}

```

Also need to add in the property as well.

```
❶ KafkaMessageListener.java × ❷ application.yml × ❸ KafkaConsumerExampleApplication.java ×
1   spring:
2     kafka:
3       consumer:
4         bootstrap-servers: localhost:9092
5         group-id: jt-group-1
6
7
8   server:
9     port: 9292
```

```
sageListenerContainer : jt-group-1: partitions assigned: [javatechie-de
```

```
    tions assigned: [javatechie-demo-0, javatechie-demo-1, javatechie-demo-2]
```

All the topics are assigned to consumer group

Just hit the producer endpoint

The screenshot shows a POST request in Postman to `http://localhost:9191/producer-app/publish/user`. The response status is `200 OK` with a time of `1507 ms` and a size of `197 B`. The response body is `message published successfully ..`.

Consumer listens all the messages

```

Run: KafkaConsumerExampleApplication
DB Browser
Favorites
Run Problems Terminal Jenkins Linter TODO Services Build Dependencies
Build completed successfully in 5 sec. 206 ms (2 minutes ago)
7001:136 (12 chars) LF UTF-8 4 spaces Event
2:13:29 4:50:15

```

Producer - summary

Partition	Start Offset	End Offset	Size
Partition 0	0	5,150	5,150
Partition 1	0	5,150	5,150
Partition 2	0	5,150	5,150

Consumer summary

Partition	Start	End	Offset	Lag	Last Commit
Partition 0	0	5,150	5,150	0	2023-07-14 16:04:22
Partition 1	0	5,150	5,150	0	2023-07-14 16:04:22
Partition 2	0	5,150	5,150	0	2023-07-14 16:04:22

Notice Log is zero. Which means all the messages were successfully consumed.

Just duplicate consumers. So we have 4 consumers. now

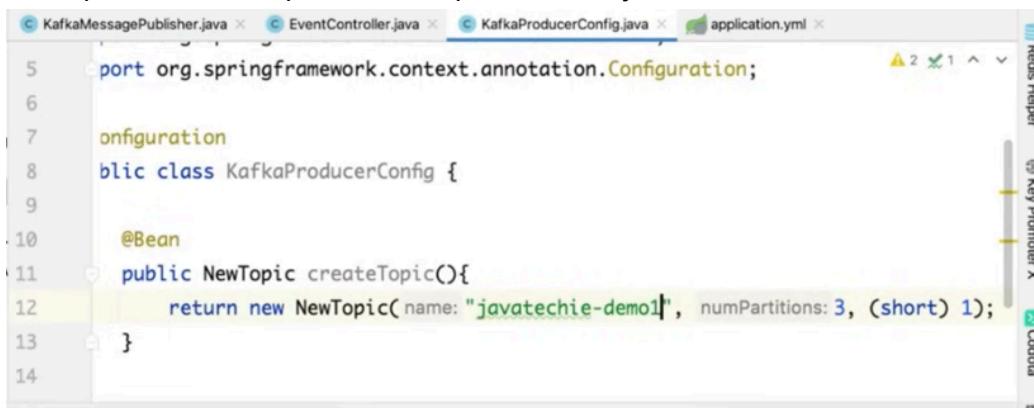
```
11     Logger log = LoggerFactory.getLogger(KafkaMessageListener.class);
12
13     @KafkaListener(topics = "javatechie-demo",groupId = "jt-group-1")
14     public void consume1(String message) {
15         log.info("consumer1 consume the message {} ", message);
16     }
17
18     @KafkaListener(topics = "javatechie-demo",groupId = "jt-group-1")
19     public void consume2(String message) {
20         log.info("consumer2 consume the message {} ", message);
21     }
```

```
21 }
22
23     @KafkaListener(topics = "javatechie-demo",groupId = "jt-group-1")
24     public void consume3(String message) {
25         log.info("consumer3 consume the message {} ", message);
26     }
27
28     @KafkaListener(topics = "javatechie-demo",groupId = "jt-group-1")
29     public void consume4(String message) {
30         log.info("consumer4 consume the message {} ", message);
31     }
32 }
```

Rename the topics and groupId to clear understanding and start the consumer.

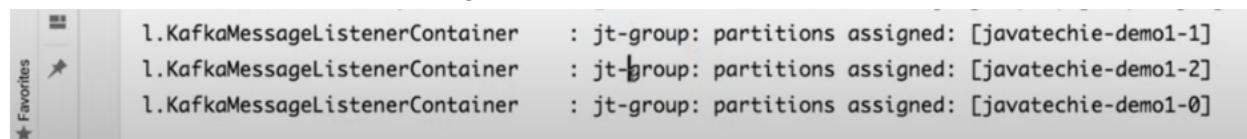
```
12
13     @KafkaListener(topics = "javatechie-demo1",groupId = "jt-group")
14     public void consume1(String message) {
15         log.info("consumer1 consume the message {} ", message);
16     }
17
18     @KafkaListener(topics = "javatechie-demo1",groupId = "jt-group")
19     public void consume2(String message) {
20         log.info("consumer2 consume the message {} ", message);
21     }
22
23     @KafkaListener(topics = "javatechie-demo1",groupId = "jt-group")
24     public void consume3(String message) {
25         log.info("consumer3 consume the message {} ", message);
26     }
```

In the producer also updates the topic name to “javatechie-demo1” and start the producer”.

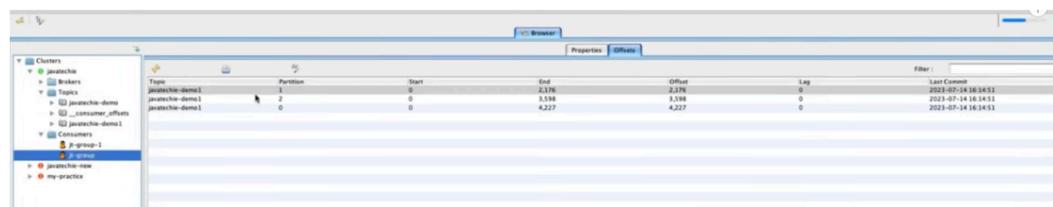
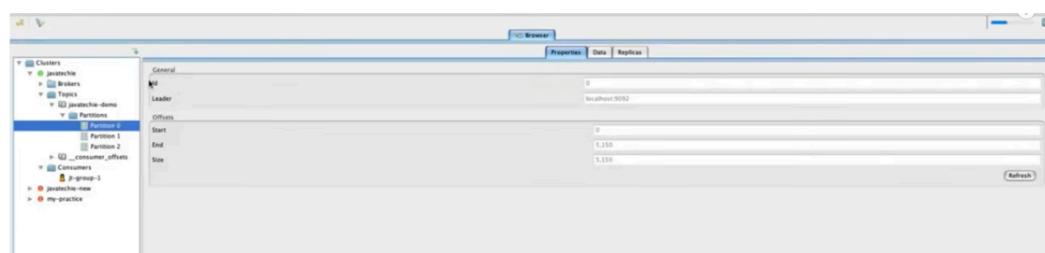
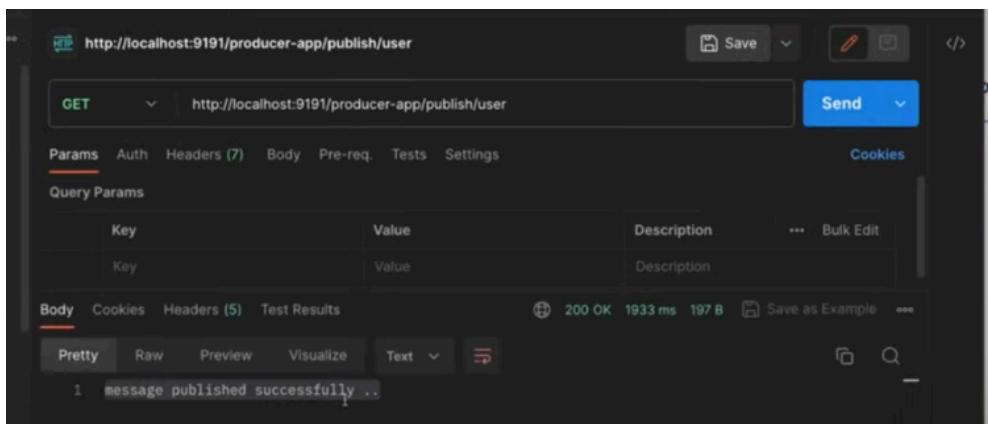


```
5 import org.springframework.context.annotation.Configuration;
6
7 @Configuration
8 public class KafkaProducerConfig {
9
10     @Bean
11     public NewTopic createTopic(){
12         return new NewTopic("javatechie-demo1", 3, 1);
13     }
14 }
```

Now we can see each partition assigned to one consumer. Here consumer names are common.



```
1.KafkaMessageListenerContainer : jt-group: partitions assigned: [javatechie-demo1-1]
1.KafkaMessageListenerContainer : jt-group: partitions assigned: [javatechie-demo1-2]
1.KafkaMessageListenerContainer : jt-group: partitions assigned: [javatechie-demo1-0]
```



DB Browser	Run: KafkaConsumerExampleApplication	consumer1	2023-07-14T16:16:16.976+05:30 INFO 73781 ---
DB Browser	Run: KafkaConsumerExampleApplication	consumer2	2023-07-14T16:16:16.976+05:30 INFO 73781 --- 0 results
DB Browser	Run: KafkaConsumerExampleApplication	consumer3	2023-07-14T16:16:16.681+05:30 INFO 73781 --- [ntai
DB Browser	Run: KafkaConsumerExampleApplication	consumer4	2023-07-14T16:16:16.532+05:30 INFO 73781 --- [ntain

Each partition is assigned to each consumer instance.

In real time we won't create duplicate consumer method instances
We will create duplicate instances of the application. Need to explore.

Lag messages.

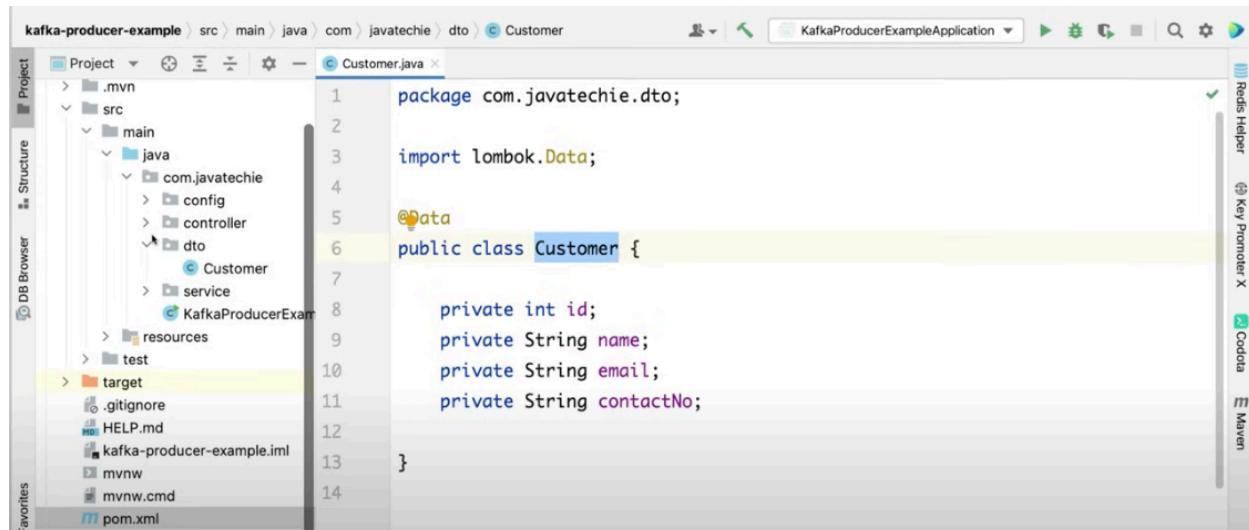
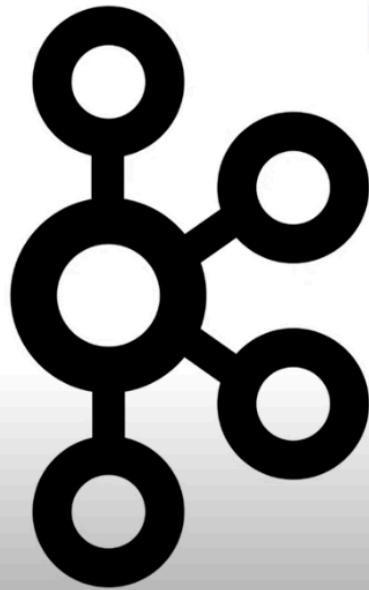
If the consumer is stopped /went down while messages are flowing.

Browser							
Clusters		Topics					
Brokers		Properties Offsets					
Topic	Partition	Start	End	Offset	Lag	Last Commit	
javatechie-demo1	2	0	13,244	13,244	0	2023-07-14 16:34:19	
javatechie-demo1	0	0	9,552	9,552	0	2023-07-14 16:34:19	
javatechie-demo1	1	0	7,207	7,207	0	2023-07-14 16:34:19	
javatechie-demo2	1	0	12,358	976	11,382	2023-07-14 16:34:19	
javatechie-demo2	0	0	32,786	32,737	59,159	2023-07-14 16:34:19	
javatechie-demo2	2	0	34,877	32,557	31,620	2023-07-14 16:34:19	

You may notice lag in consuming the messages as shown above.

KAFKA SERIALIZE & DESERIALIZE

Apache Kafka Object Serialize & Deserialize Example



The screenshot shows the IntelliJ IDEA IDE interface. The left sidebar displays the project structure for 'kafka-producer-example' with packages like .mvn, src, main, and test. The 'src/main/java/com/javatechie/dto' package contains a file named 'Customer.java'. The right pane shows the code for 'Customer.java':

```
package com.javatechie.dto;

import lombok.Data;

@Data
public class Customer {

    private int id;
    private String name;
    private String email;
    private String contactNo;
}
```

Just copy our previous publisher and paste it and replace all String with Customer

The screenshot shows an IDE interface with two tabs open: 'Customer.java' and 'KafkaMessagePublisher.java'. The 'KafkaMessagePublisher.java' tab is active, displaying the following Java code:

```
10  @Service
11  public class KafkaMessagePublisher {
12
13      @Autowired
14      private KafkaTemplate<String, Object> template;
15
16      public void sendMessageToTopic(String message){
17          CompletableFuture<SendResult<String, Object>> future = template.send(topic: "javatechie-demo2", message);
18          future.whenComplete((result, ex)->{
19              if (ex == null) {
20                  System.out.println("Sent message=[" + message +
21                      "] with offset=[" + result.getRecordMetadata().offset() + "]");
22              } else {
23                  System.out.println("Unable to send message=[" +
24                      message + "] due to : " + ex.getMessage());
25              }
26          });
27
28      }
29
30  }
```

The code implements a Kafka producer. It uses a KafkaTemplate to send a message to a topic named 'javatechie-demo2'. The 'whenComplete' method is used to handle the response or any exceptions. The output is printed to the console.

This screenshot shows the same IDE interface, but the cursor is positioned on the 'customer' parameter in the 'sendEventsToTopic' method of 'KafkaMessagePublisher.java'. The code is identical to the one in the first screenshot, except for the highlighted parameter.

```
29
30
31  sendEventsToTopic(Customer customer){
32      CompletableFuture<SendResult<String, Object>> future = template.send(topic: "javatechie-demo2", customer);
33      future.whenComplete((result, ex)->{
34          if (ex == null) {
35              System.out.println("Sent message=[" + customer.toString() +
36                  "] with offset=[" + result.getRecordMetadata().offset() + "]");
37          } else {
38              System.out.println("Unable to send message=[" +
39                  customer.toString() + "] due to : " + ex.getMessage());
40
41
42  }
```

And add it in the try catch block

```
Customer.java KafkaMessagePublisher.java KafkaProducerConfig.java EventController.java

34     future.whenComplete((result, ex) -> {
35         if (ex == null) {
36             System.out.println("Sent message=[" + customer.toString() +
37                 "] with offset=[" + result.getRecordMetadata().offset() + "]");
38         } else {
39             System.out.println("Unable to send message=[" +
40                 customer.toString() + "] due to : " + ex.getMessage());
41         }
42     });
43
44 } catch (Exception ex) {
45     System.out.println("ERROR : " + ex.getMessage());
46 }
47 }
```

Create controller

```
30 @PostMapping("/publish")
31 public void sendEvents(@RequestBody Customer customer) {
32     publisher.sendEventsToTopic(customer);
33 }
```

Create a consumer to listen to customer messages in the consumer app.

Create customer class in the consumer too and start .

```
9 @Service
10 public class KafkaMessageListener {
11
12     Logger log = LoggerFactory.getLogger(KafkaMessageListener.class);
13
14     @KafkaListener(topics = "javatechie-demo",groupId = "jt-group-new")
15     public void consume(Customer customer) {
16         log.info("consumer consume the message {} ", customer.toString());
17     }
18 }
```

Rename group id to jt-group

```

@KafkaListener(topics = "javatechie-demo", groupId = "jt-group")

aConsumerExampleApplication x
[er#0-0-l-1] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-jt-group-1, groupId=jt-group] Found no committed offset for part
er#0-0-C-1] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-jt-group-1, groupId=jt-group] Found no committed offset for part
er#0-0-C-1] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-jt-group-1, groupId=jt-group] Found no committed offset for part
er#0-0-C-1] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-jt-group-1, groupId=jt-group] Found no committed offset for part
er#0-0-C-1] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-jt-group-1, groupId=jt-group] Found no committed offset for part
er#0-0-C-1] o.a.k.c.c.internals.ConsumerCoordinator : [Consumer clientId=consumer-jt-group-1, groupId=jt-group] Resetting offset for partition jav
er#0-0-C-1] o.a.k.c.c.internals.SubscriptionState : [Consumer clientId=consumer-jt-group-1, groupId=jt-group] Resetting offset for partition jav
er#0-0-C-1] o.a.k.c.c.internals.SubscriptionState : [Consumer clientId=consumer-jt-group-1, groupId=jt-group] Resetting offset for partition jav
er#0-0-C-1] o.s.k.l.KafkaMessageListenerContainer : jt-group: partitions assigned: [javatechie-demo-0, javatechie-demo-1, javatechie-demo-2]

```

Up the producer and hit the endpoint. We will get serialization error.

```

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/producer-app")
public class EventController {
}

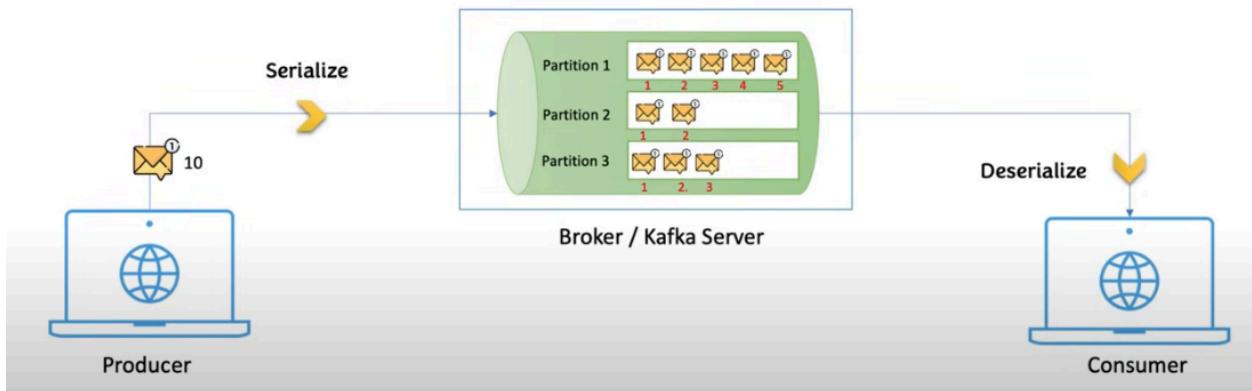
```

Run: KafkaProducerExampleApplication

```

2023-07-28T11:55:52.173+05:30 INFO 6969 --- [nio-9191-exec-2] o.a.kafka.common.utils.AppInfoParser : Kafka commitId: 8a516edc2755df89
2023-07-28T11:55:52.174+05:30 INFO 6969 --- [nio-9191-exec-2] o.a.kafka.common.utils.AppInfoParser : Kafka startTimeMs: 1690525552172
2023-07-28T11:55:52.183+05:30 INFO 6969 --- [ad | producer-1] org.apache.kafka.clients.Metadata : [Producer clientId=producer-1] Cluste
2023-07-28T11:55:52.202+05:30 INFO 6969 --- [ad | producer-1] org.apache.kafka.clients.Metadata : [Producer clientId=producer-1] Resette
2023-07-28T11:55:52.203+05:30 INFO 6969 --- [ad | producer-1] org.apache.kafka.clients.Metadata : [Producer clientId=producer-1] Resette
2023-07-28T11:55:52.203+05:30 INFO 6969 --- [ad | producer-1] org.apache.kafka.clients.Metadata : [Producer clientId=producer-1] Resette
2023-07-28T11:55:52.214+05:30 INFO 6969 --- [ad | producer-1] o.a.k.c.p.internals.TransactionManager : [Producer clientId=producer-1] Prod

```



From the producer we have to serialize and the consumer side we have to deserialize data. It is a common practice we have to do when we send data across network.

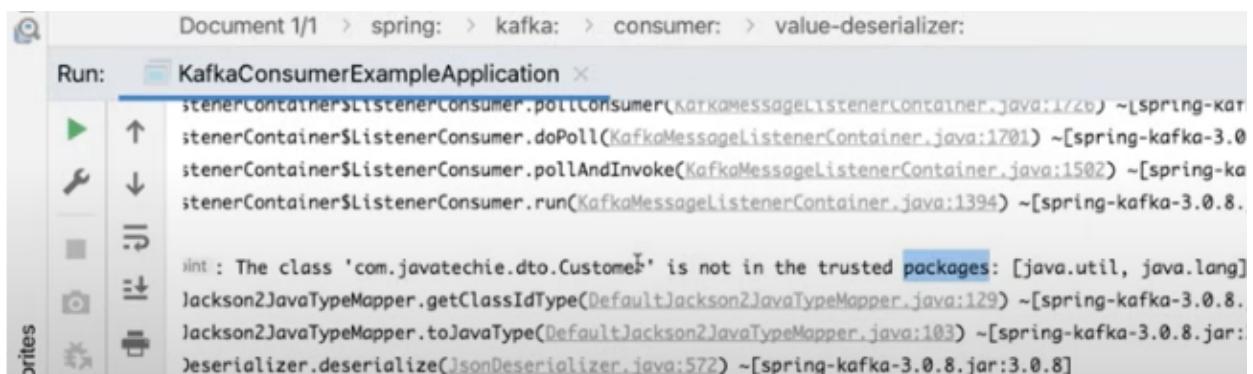
The screenshot shows two instances of IntelliJ IDEA. The top instance is for the 'kafka-producer-example' project, displaying the 'application.yml' configuration file. The bottom instance is for the 'kafka-consumer-example' project, also displaying its 'application.yml' configuration file. Both files contain YAML configurations for Kafka settings, specifically for the producer and consumer respectively, including bootstrap-servers, key-serializer, and value-serializer definitions.

```
kafka-producer-example > src > main > resources > application.yml
server:
  port: 9191

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

kafka-consumer-example > src > main > resources > application.yml
kafka:
  consumer:
    bootstrap-servers: localhost:9092
    group-id: jt-group-1
    key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
```

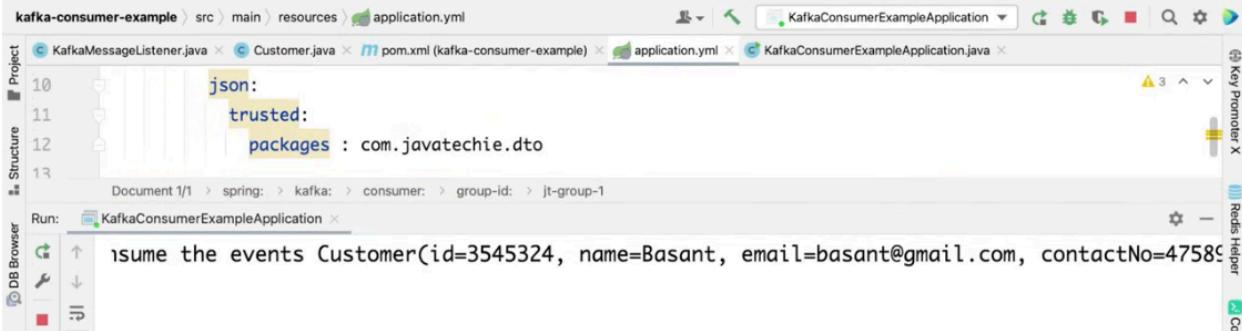
Still we are seeing error as the object is not from the trusted package.





```
group: "jt-group-1"
key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
properties:
  spring:
    json:
      trusted:
        packages : com.javatechie.dto
```

Now consumer



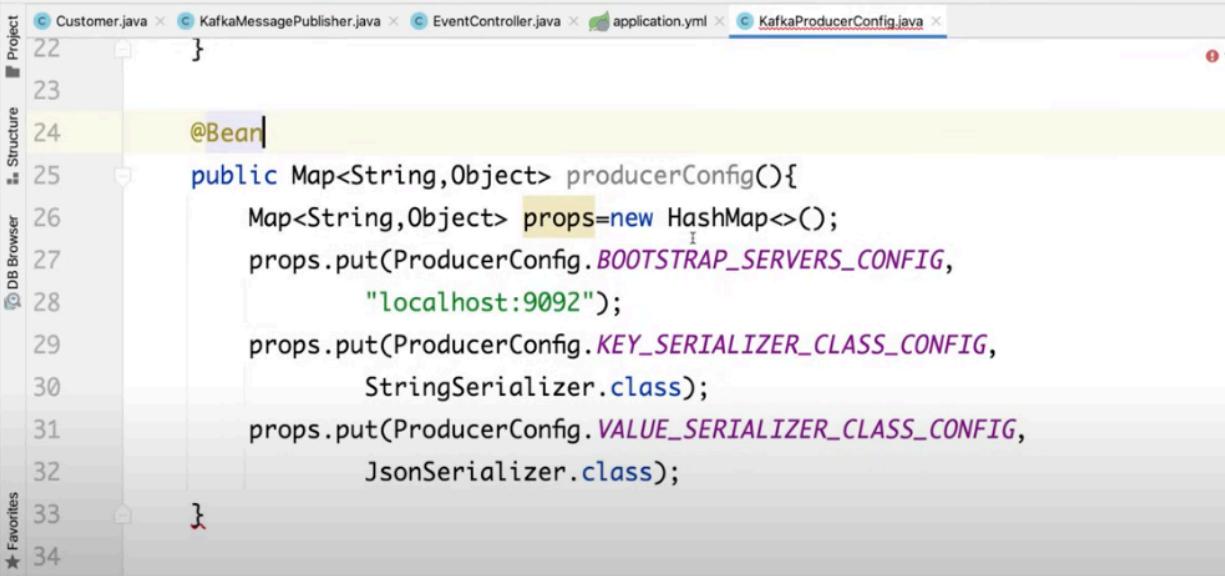
```
Document 1/1 > spring: > kafka: > consumer: > group-id: > jt-group-1
Run: KafkaConsumerExampleApplication
```

Customer	id	name	email	contactNo
Customer	3545324	Basant	basant@gmail.com	47589

But this is not best practice. Comment those properties in both producer and consumer.

Refer to below java configuration for the best practice.

The below one is for producer



```
@Bean
public Map<String, Object> producerConfig(){
    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);
}
```

Same thing for consumer side as well.

```
Customer.java KafkaMessagePublisher.java EventController.java application.yml KafkaProducerConfig.java
33
34
35     @Bean
36     public ProducerFactory<String, Object> producerFactory(){
37         return new DefaultKafkaProducerFactory<>(producerConfig());
38     }
39
40     @Bean
41     public KafkaTemplate<String, Object> kafkaTemplate(){
42         return new KafkaTemplate<>(producerFactory());
43     }
```

Same thing for consumer side

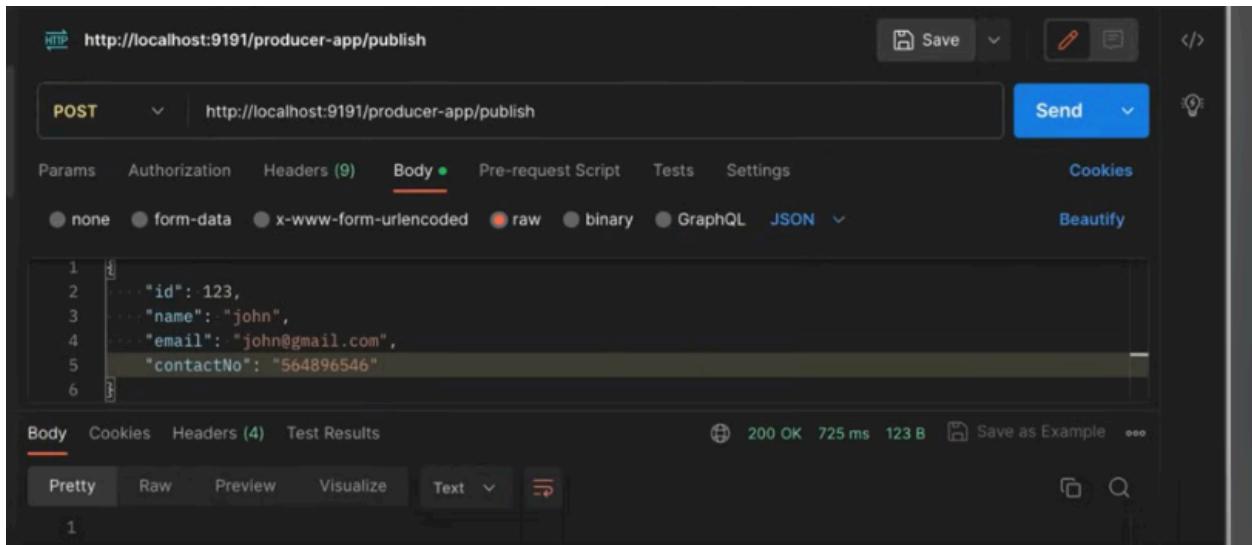
```
KafkaMessageListener.java Customer.java pom.xml (kafka-consumer-example) application.yml KafkaConsumerConfig.java KafkaConsumerExampleApplication.java
15
16     @Bean
17     public Map<String, Object> consumerConfig(){
18         Map<String, Object> props = new HashMap<>();
19         props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
20             "localhost:9092");
21         props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
22             StringDeserializer.class);
23         props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
24             JsonDeserializer.class);
25         props.put(JsonDeserializer.TRUSTED_PACKAGES, "com.javatechie.dto");
26         return props;
27     }
28 }
```

```
34
35     @Bean
36     public ConsumerFactory<String, Object> consumerFactory(){
37         return new DefaultKafkaConsumerFactory<>(consumerConfig());
38     }
39
40     @Bean
41     public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String, Object>> kafkaListenerContainerFactory(){
42         ConcurrentKafkaListenerContainerFactory<String, Object> factory =
43             new ConcurrentKafkaListenerContainerFactory<>();
44         factory.setConsumerFactory(consumerFactory());
45     }
```

Way java config :

If our approach is straight forward - push the event and consume it go for application properties or yaml file.

If you want more customization like to access secure kafka cluster which will be https in that case we have configure certificates using java based config.



The screenshot shows the Postman interface with the following details:

- Method: POST
- URL: <http://localhost:9191/producer-app/publish>
- Body tab selected, showing JSON data:

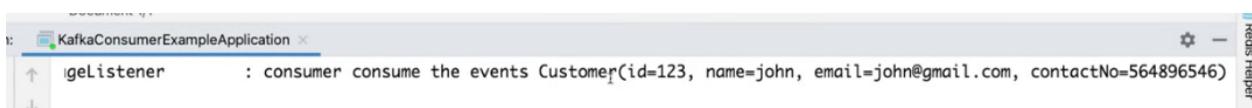
```
1
2   ...
3     "id": 123,
4     "name": "john",
5     "email": "john@gmail.com",
6     "contactNo": "564896546"
```
- Response status: 200 OK (725 ms, 123 B)

producer



```
:39.224+05:30 INFO 13790 --- [ad 1 producer-1] org.apache.kafka.clients.Metadata
stomer(id=123, name=john, email=john@gmail.com, contactNo=564896546) with offset=[0]
```

Consumer



```
KafkaConsumerExampleApplication
geListener : consumer consume the events Customer(id=123, name=john, email=john@gmail.com, contactNo=564896546)
```

Now refer

Control Kafka Partition.pdf file