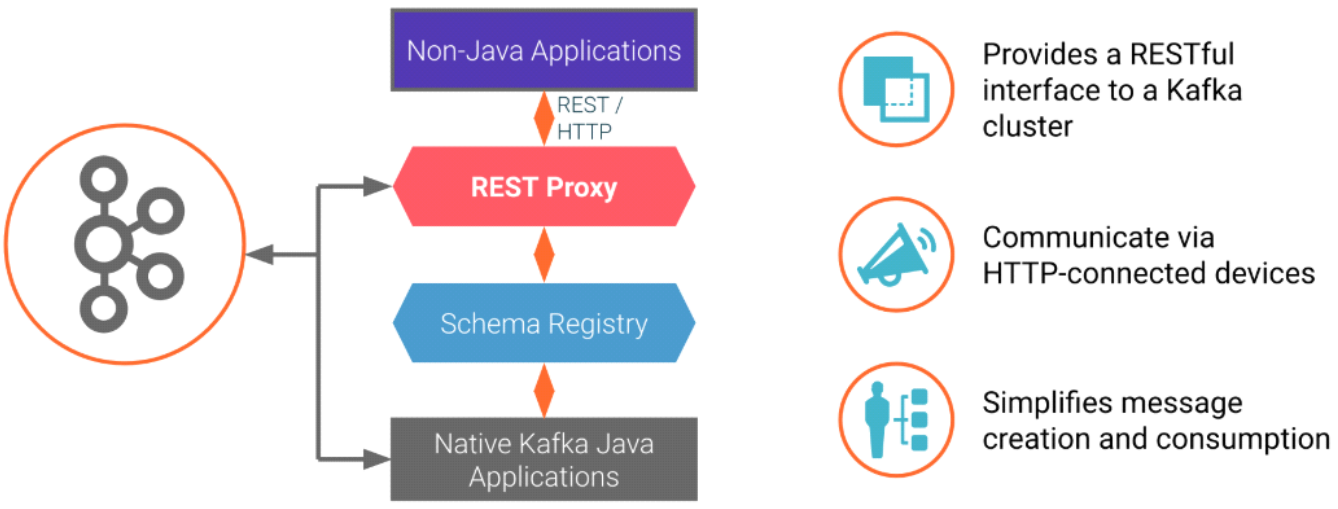
_общее

14 декабря 2020 г. 13:49

REST Proxy



Provides a RESTful interface to a Kafka cluster

Communicate via HTTP-connected devices

Simplifies message creation and consumption

- Confluent Open Source also includes a REST Proxy for Kafka. This allows **any language to access Kafka** via REST. We will discuss REST Proxy in much detail in a later module
- Confluent Enterprise also includes an **MQTT proxy** to allow direct ingestion of IoT data. This will also be discussed in a later module

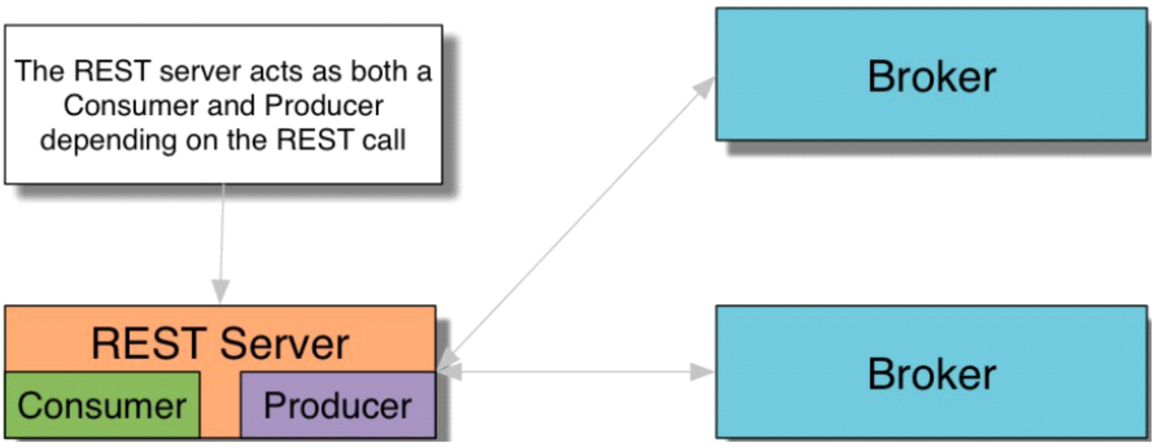
Three uses for the REST Proxy:

1. For remote clients (such as outside the datacenter including over the public internet)
2. For internal client apps that are written in a language not supported by native Kafka client libraries (i.e. other than Java, C/C++, Python, and Go)
3. For developers or existing apps for which REST is more productive and familiar than Kafka Producer/Consumer API.

в

About the Confluent REST Proxy

- The Confluent REST Proxy allows you to use HTTP to perform actions on the Kafka cluster
- The REST calls are translated into native Kafka calls
- This allows virtually any language to access Kafka
- Uses POST to send data to Kafka
  - Embedded formats: JSON, base64-encoded JSON, or Avro-encoded JSON
- Uses GET to retrieve data from Kafka


The REST server acts as both a Consumer and Producer depending on the REST call
Broker
REST Server
Consumer
Producer
Broker

The Confluent REST Proxy is part of Confluent Open Source and Confluent Enterprise distributions. The proxy provides a RESTful interface to a Kafka cluster, making it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients.

Some example use cases are:

- Reporting data to Kafka from any frontend app built in any language not supported by official Kafka clients
- Ingesting messages into a stream processing framework that doesn't yet support Kafka
- Scripting administrative actions

The configuration files for the REST proxy are preconfigured in our lab environments. Configuring the REST proxy is not part of this course. Refer students to the documentation for more details if they ask: https://docs.confluent.io/current/kafka-rest/docs/deployment.html

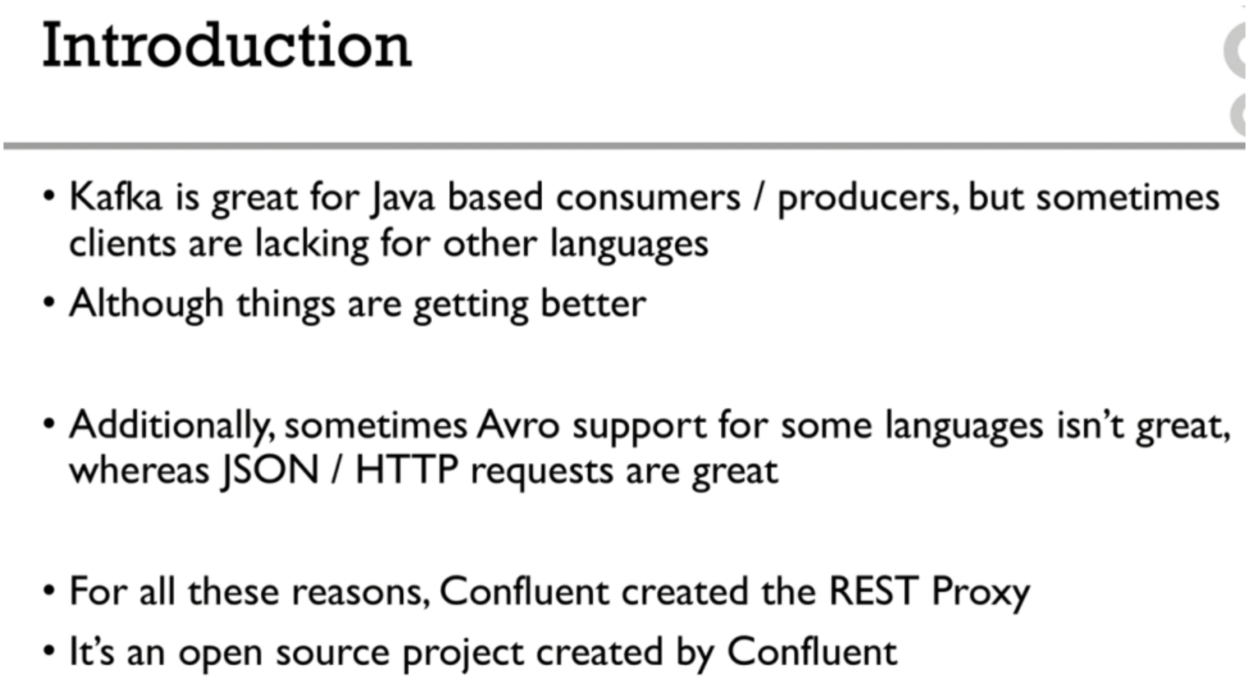
работа с кафкой через HTTP (вполовину медленее для продьюсера и в пять раз медленнее для консьюмера) чем через TCP

- REST это 66% от TCP для продьюсеров
- REST это 20% от TCP для консьюмеров

HTTP REST protocol

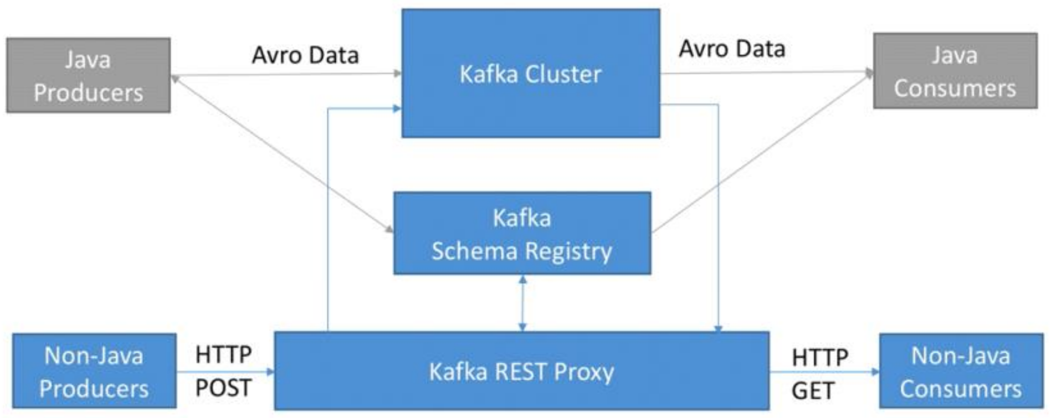
5 декабря 2020 г. 20:26

https://github.com/confluentinc/kafka-rest

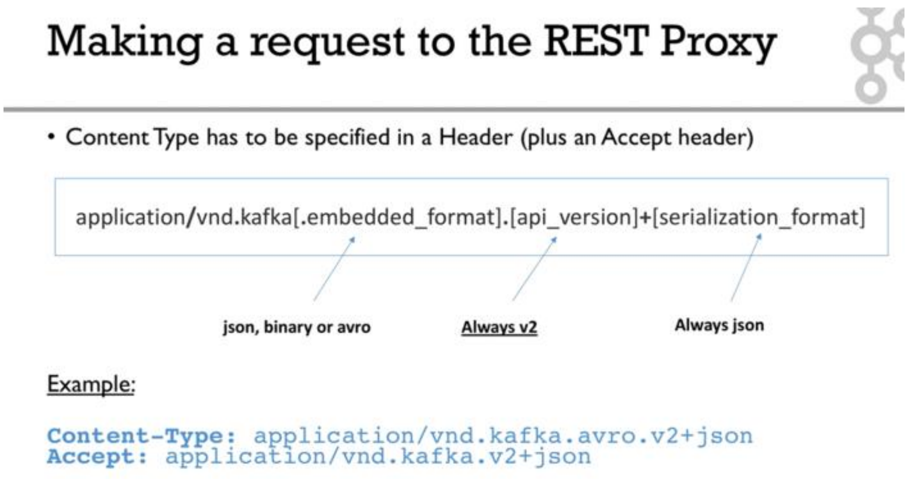


Introduction

- Kafka is great for Java based consumers / producers, but sometimes clients are lacking for other languages
- Although things are getting better
- Additionally, sometimes Avro support for some languages isn't great, whereas JSON / HTTP requests are great
- For all these reasons, Confluent created the REST Proxy
- It's an open source project created by Confluent



медленнее работает потому что приходится работать через HTTP а не kafka протоколом



Making a request to the REST Proxy

- Content Type has to be specified in a Header (plus an Accept header)

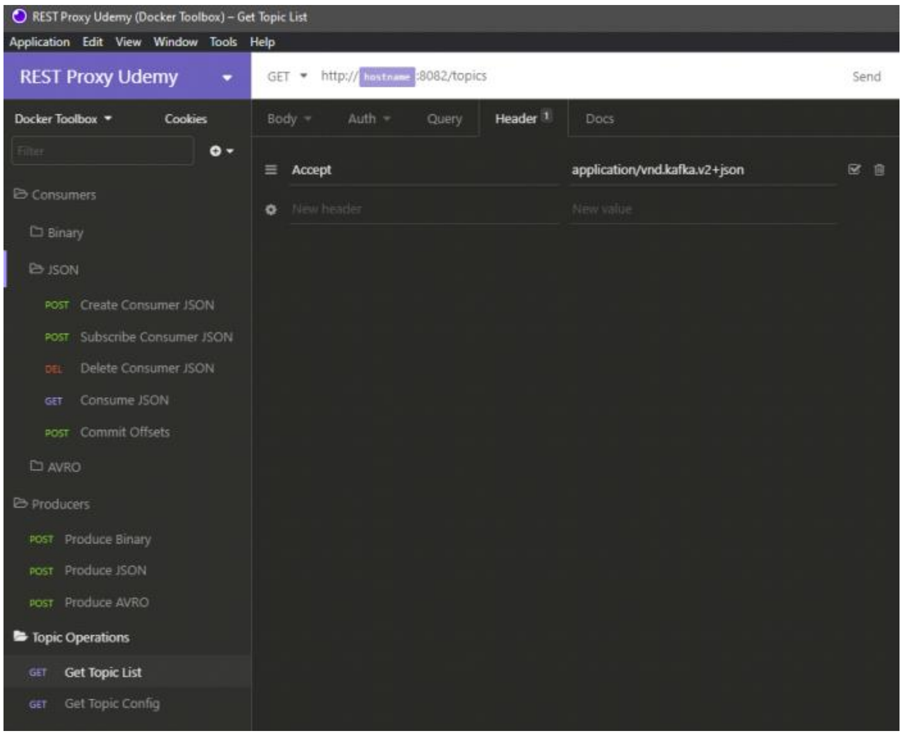
```
application/vnd.kafka[.embedded_format].[api_version]+[serialization_format]
```

json, binary or avro
Always v2
Always json

Example:

```
Content-Type: application/vnd.kafka.avro.v2+json
Accept: application/vnd.kafka.v2+json
```

теперь операции consume/produce можно выполнять через proxy по HTTP-REST



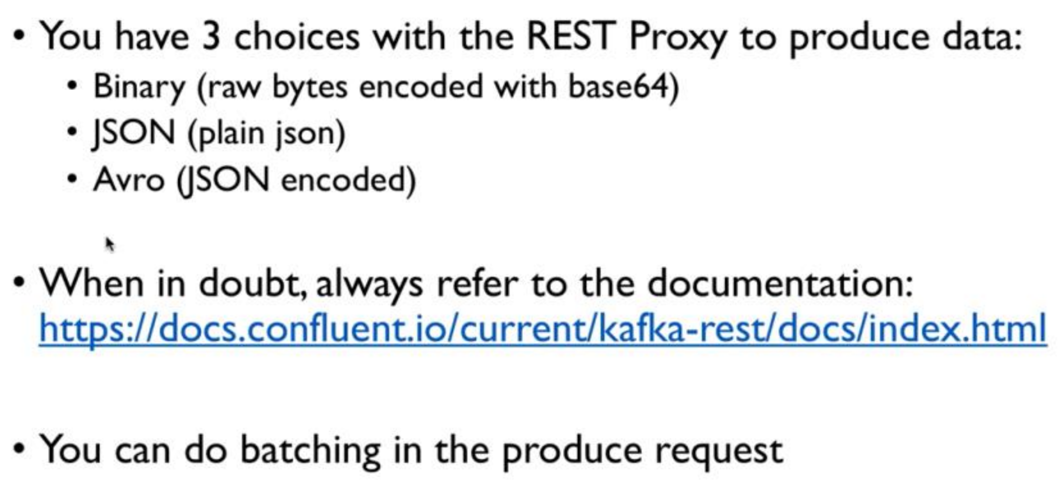
REST Proxy Udemy (Docker Toolbox) – Get Topic List

GET http://hostname:8082/topics

Accept application/vnd.kafka.v2+json

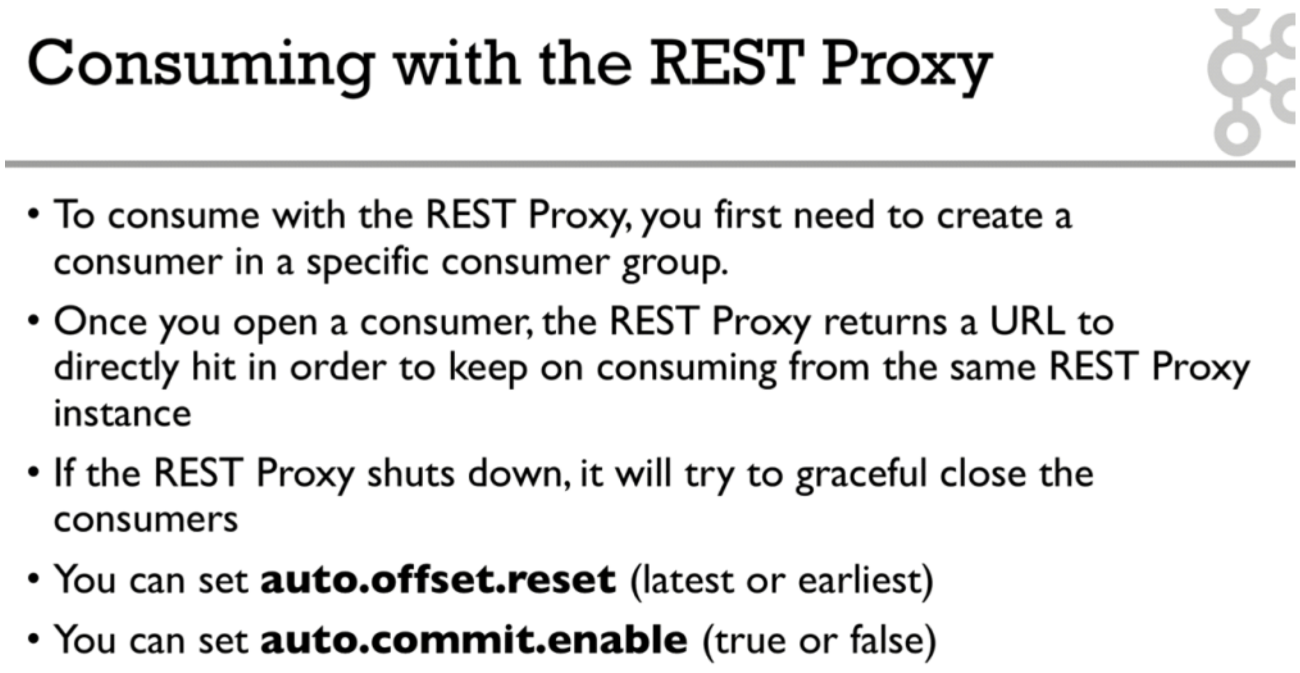
прокси - позволяет с кафктой работать по ресту например чтоб кафку дергать из SAP PI или Informatica

- удобно также чтобы FRONTEND напрямую(через HTTP) мог работать с кафкой



- You have 3 choices with the REST Proxy to produce data:
  - Binary (raw bytes encoded with base64)
  - JSON (plain json)
  - Avro (JSON encoded)
- When in doubt, always refer to the documentation: https://docs.confluent.io/current/kafka-rest/docs/index.html
- You can do batching in the produce request

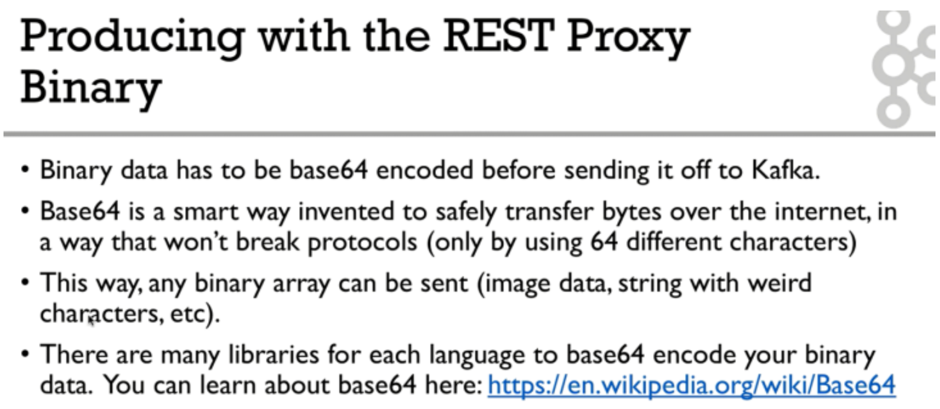
особенности консьюмеров



Consuming with the REST Proxy

- To consume with the REST Proxy, you first need to create a consumer in a specific consumer group.
- Once you open a consumer, the REST Proxy returns a URL to directly hit in order to keep on consuming from the same REST Proxy instance
- If the REST Proxy shuts down, it will try to graceful close the consumers
- You can set **auto.offset.reset** (latest or earliest)
- You can set **auto.commit.enable** (true or false)

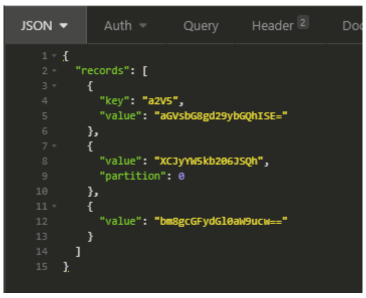
binary формат закодированный BASE64



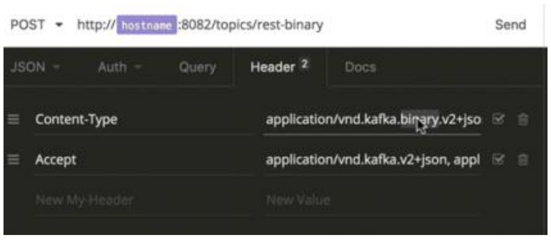
Producing with the REST Proxy Binary

- Binary data has to be base64 encoded before sending it off to Kafka.
- Base64 is a smart way invented to safely transfer bytes over the internet, in a way that won't break protocols (only by using 64 different characters)
- This way, any binary array can be sent (image data, string with weird characters, etc).
- There are many libraries for each language to base64 encode your binary data. You can learn about base64 here: https://en.wikipedia.org/wiki/Base64

1 продьюсером запостим сообщение



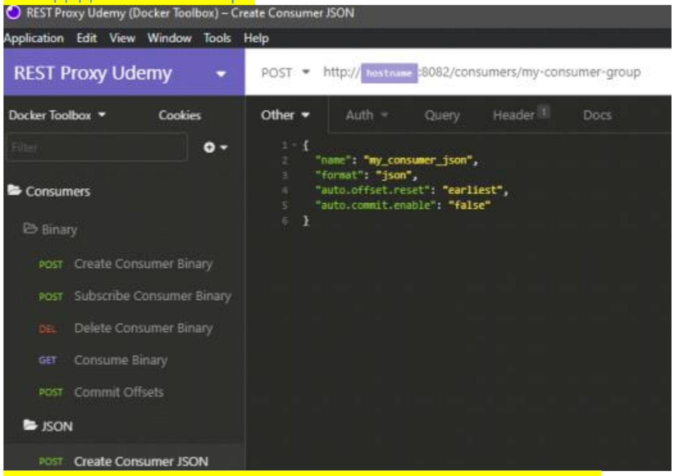
```
{
  "records": [
    {
      "key": "a2V5",
      "value": "aGVsbG8gd29ybGQhISE="
    },
    {
      "value": "XCJyYW5kb206JSQh",
      "partition": 0
    },
    {
      "value": "bm8gcGFydGl0aW9ucw=="
    }
  ]
}
```



POST http://hostname:8082/topics/rest-binary

Content-Type	application/vnd.kafka.binary.v2+jso
Accept	application/vnd.kafka.v2+json, appl

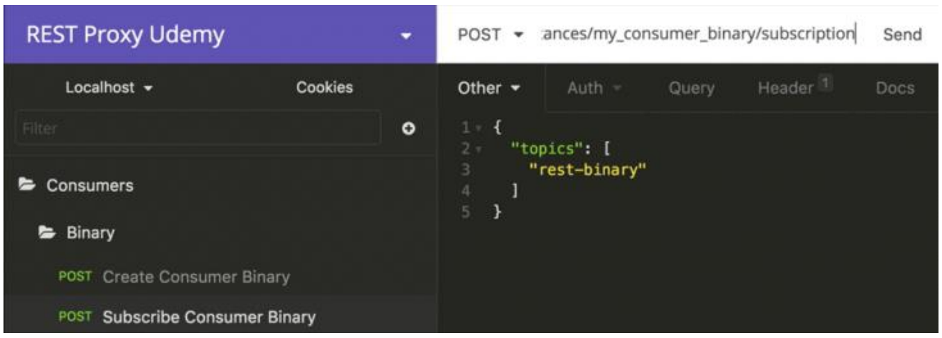
1 создадим консьюмера


REST Proxy Udemy (Docker Toolbox) – Create Consumer JSON
Application
Edit
View
Window
Tools
Help
REST Proxy Udemy
Docker Toolbox
Cookies
Filter
Consumers
Binary
POST Create Consumer Binary
POST Subscribe Consumer Binary
DEL Delete Consumer Binary
GET Consume Binary
POST Commit Offsets
JSON
POST Create Consumer JSON
Other
Auth
Query
Header 1
Docs

POST http://hostname:8082/consumers/my-consumer-group

```
{
  "name": "my_consumer_json",
  "format": "json",
  "auto.offset.reset": "earliest",
  "auto.commit.enable": "false"
}
```

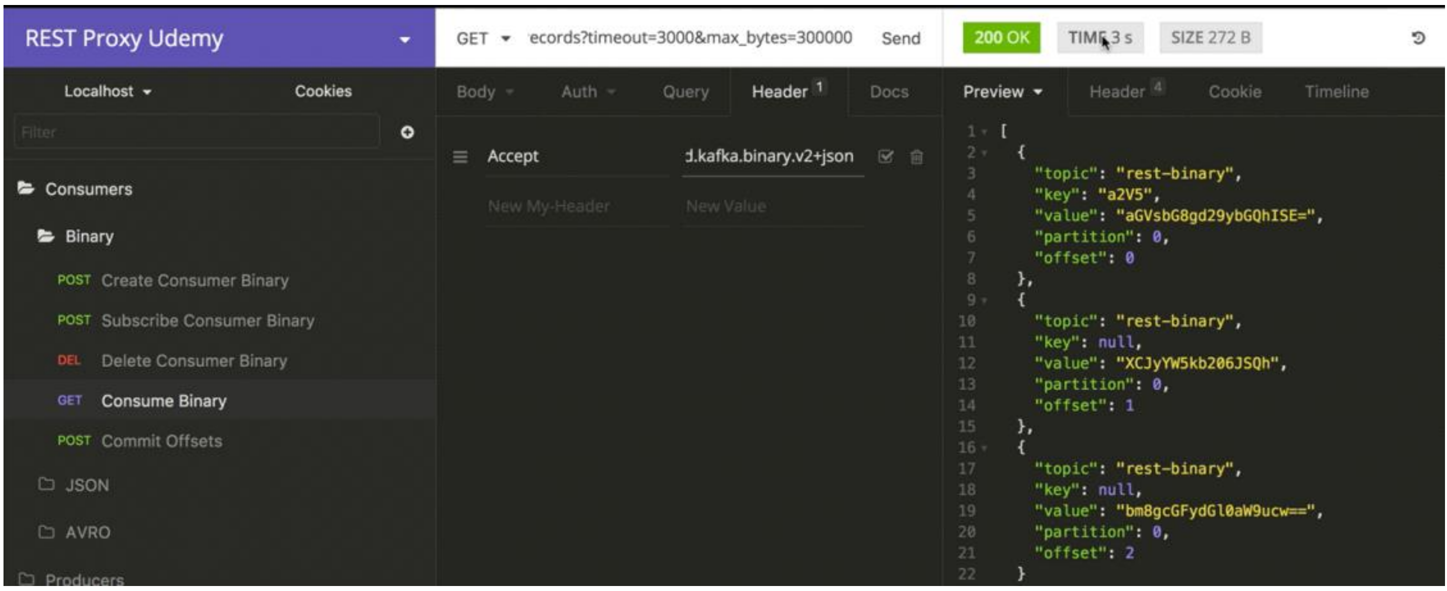
2 чтобы читать консьюмером сначала нужно подписаться



POST ances/my_consumer_binary/subscription

```
{
  "topics": [
    "rest-binary"
  ]
}
```

3 законсьюмим сообщения те прочитаем

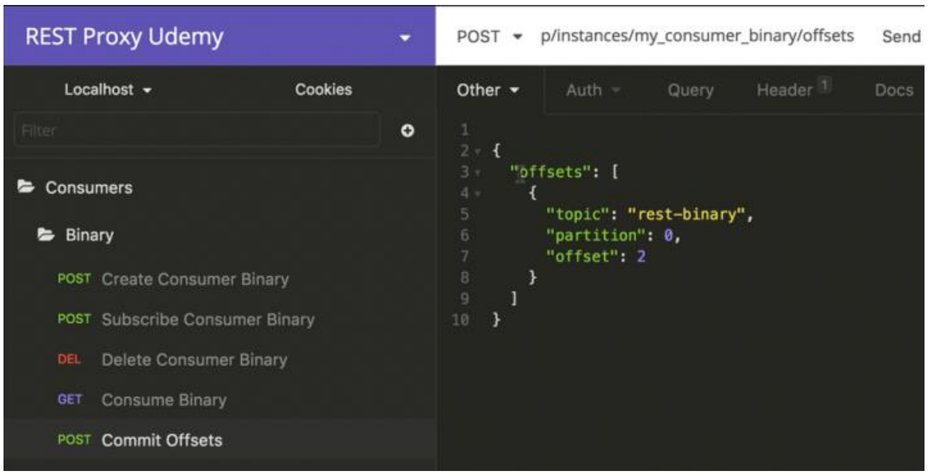


GET ecords?timeout=3000&max_bytes=300000

Accept: d.kafka.binary.v2+json

```
[
  {
    "topic": "rest-binary",
    "key": "a2V5",
    "value": "aGVsbG8gd29ybGQhISE=",
    "partition": 0,
    "offset": 0
  },
  {
    "topic": "rest-binary",
    "key": null,
    "value": "XCJyYW5kb206JSQh",
    "partition": 0,
    "offset": 1
  },
  {
    "topic": "rest-binary",
    "key": null,
    "value": "bm8gcGFydGl0aW9ucw==",
    "partition": 0,
    "offset": 2
  }
```

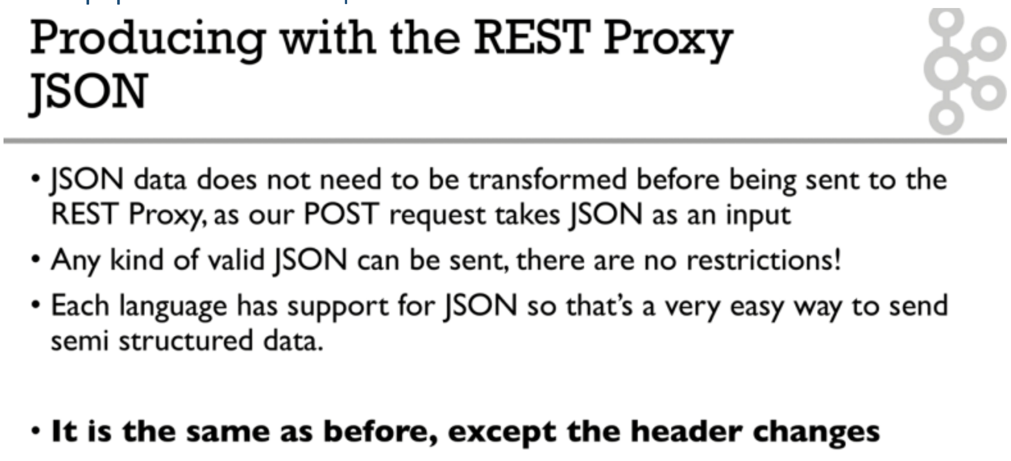
4 закомитим офсет



POST p/instances/my_consumer_binary/offsets

```
{
  "offsets": [
    {
      "topic": "rest-binary",
      "partition": 0,
      "offset": 2
    }
  ]
}
```

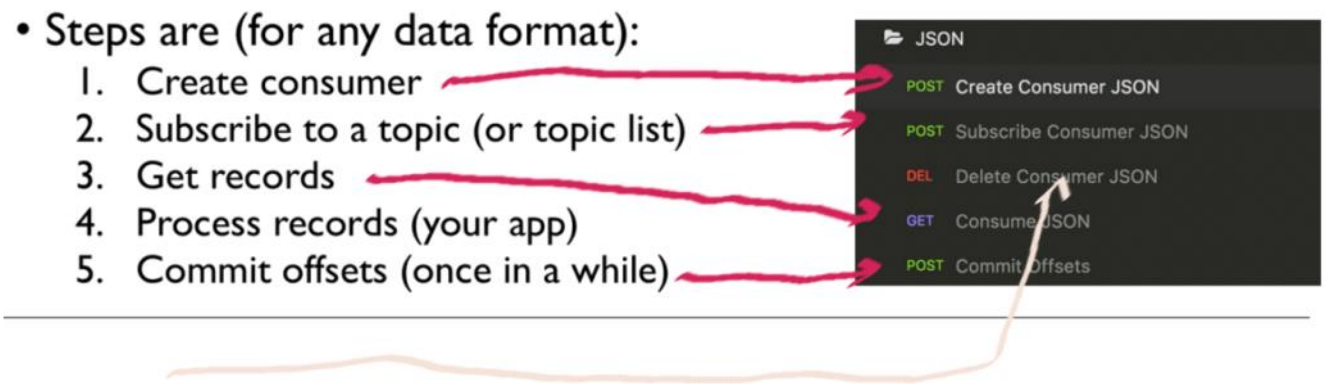
JSON формат REST сообщения



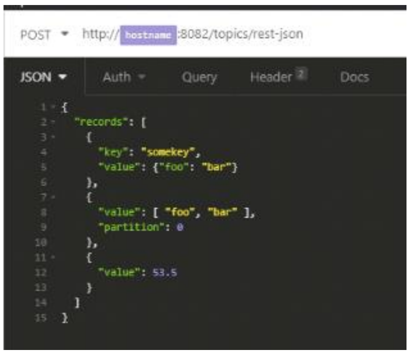
Producing with the REST Proxy JSON

- JSON data does not need to be transformed before being sent to the REST Proxy, as our POST request takes JSON as an input
- Any kind of valid JSON can be sent, there are no restrictions!
- Each language has support for JSON so that's a very easy way to send semi structured data.

- **It is the same as before, except the header changes**



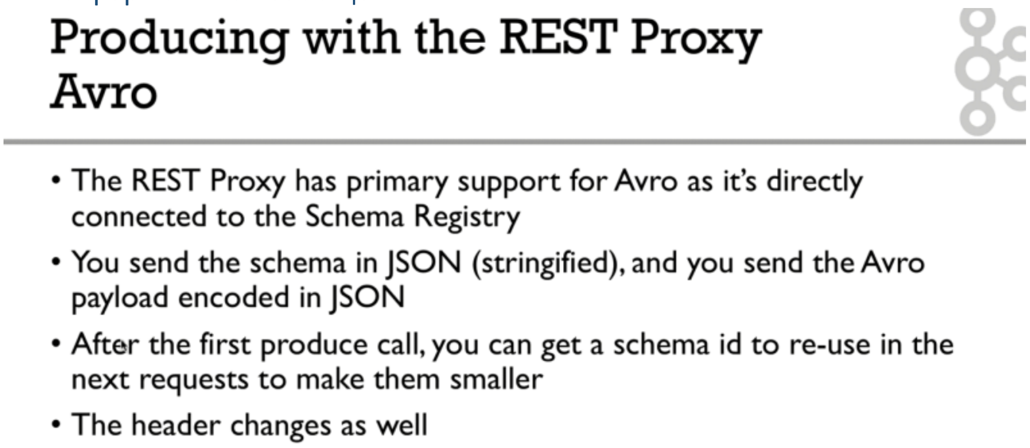
- Steps are (for any data format):
  1. Create consumer
  2. Subscribe to a topic (or topic list)
  3. Get records
  4. Process records (your app)
  5. Commit offsets (once in a while)



```
POST http://hostname:8082/topics/rest-json
```

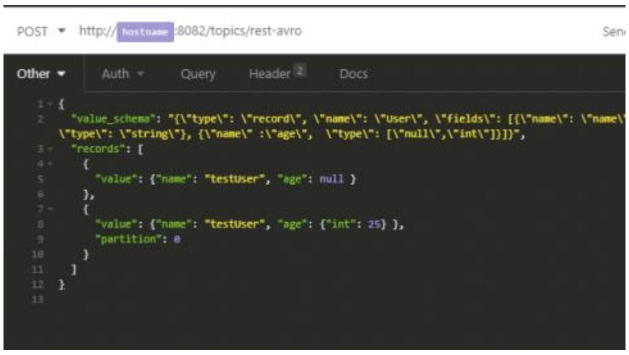
```
{
  "records": [
    {
      "key": "somekey",
      "value": {"foo": "bar"}
    },
    {
      "value": [ "foo", "bar" ],
      "partition": 0
    },
    {
      "value": 53.5
    }
  ]
}
```

AVRO формат REST сообщения



Producing with the REST Proxy Avro

- The REST Proxy has primary support for Avro as it's directly connected to the Schema Registry
- You send the schema in JSON (stringified), and you send the Avro payload encoded in JSON
- After the first produce call, you can get a schema id to re-use in the next requests to make them smaller
- The header changes as well

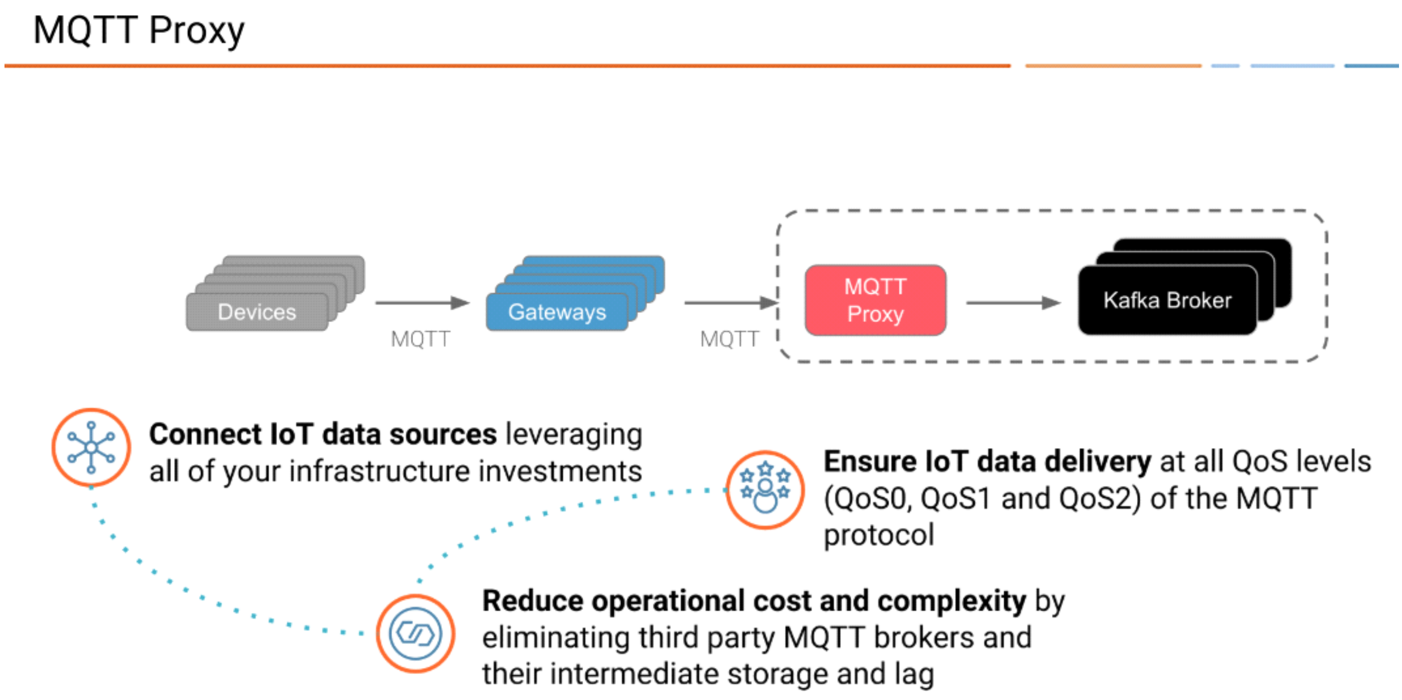


```
POST http://hostname:8082/topics/rest-avro
```

```
{
  "value_schema": "{\"type\": \"record\", \"name\": \"User\", \"fields\": [{\"name\": \"name\"
\"type\": \"string\"}, {\"name\" :\"age\",  \"type\": [\"null\",\"int\"]}]}",
  "records": [
    {
      "value": {"name": "testUser", "age": null }
    },
    {
      "value": {"name": "testUser", "age": {"int": 25} },
      "partition": 0
    }
  ]
}
```

MQTT protocol (для интернета вещей)

14 декабря 2020 г.   13:51



MQTT Proxy

Connect IoT data sources leveraging all of your infrastructure investments

Ensure IoT data delivery at all QoS levels (QoS0, QoS1 and QoS2) of the MQTT protocol

Reduce operational cost and complexity by eliminating third party MQTT brokers and their intermediate storage and lag

- Confluent Enterprise includes an MQTT (Message Queuing Telemetry Transport) proxy to allow direct ingestion of IoT data
- The MQTT proxy is intended to allow customers to add Kafka to their IoT architecture to enable stream processing. MQTT is a pub/sub messaging transport protocol (the de facto standard for IoT devices) that is designed to support thin lightweight clients that run in a very diverse set of devices and communicate asynchronously with scalable, more heavy weight, protocol aware MQTT brokers. Without this proxy, customers would have to maintain separate MQTT clients and brokers, adding layers and complexity to their architectures. The current implementation of the proxy is read-only (i.e., producer only).

пример продьюсера

```
private final int qos = 1;
private String host = "ssl://mqtt.hsl.fi:8883";
private String clientId = "MQTT-Java-Example";
private String topic = "/hfp/v2/journey/ongoing/vp/#";
private String kafka_topic = "vehicle-positions";
private MqttClient client;
```

We have the following:

- `qos` defines the **quality of service** of the MQTT service
- `host` is the URI of the host providing the data, which we're going to connect to
- `clientId` is just how we identify ourselves to the host
- `topic` is the MQTT topic (**NOT** the Kafka topic!) we're going to subscribe to
- `kafka_topic` is the name of the Kafka topic we're going to write the MQTT data to
- Finally we are defining a variable `client` for the `MqttClient` instance

If you have problems with the amount of data that the MQTT source produces, you can adjust the value of `topic` to say only return vehicle positions for **trams**. The corresponding value would then be: `/hfp/v2/journey/ongoing/vp/tram/#`
For more details please consult https://digitransit.fi/en/developers/apis/4-realtime-api/vehicle-positions/

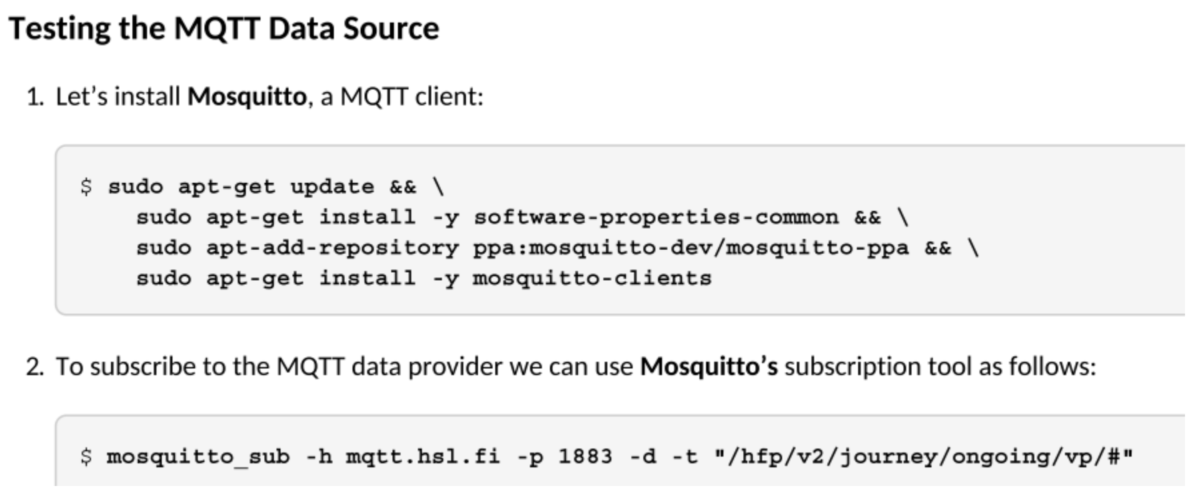
9. Let's now define the `start` method:

```
package clients;
import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.eclipse.paho.client.mqttv3.*;
public class VehiclePositionProducer {
    public static void main(String[] args) throws MqttException {
        System.out.println("*** Starting VP Producer ***");
        Properties settings = new Properties();
        settings.put(ProducerConfig.CLIENT_ID_CONFIG, "vp-producer");
        settings.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka:9092");
        settings.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        settings.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        final KafkaProducer<String, String> producer = new KafkaProducer<>(settings);

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("### Stopping VP Producer ###");
            producer.close();
        }));

        Subscriber subscriber = new Subscriber(producer);
        subscriber.start();
    }
}
```

```
package clients;
import java.util.UUID;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.eclipse.paho.client.mqttv3.*;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
public class Subscriber implements MqttCallback {
    private final int qos = 1;
    private String host = "ssl://mqtt.hsl.fi:8883";
    private String clientId = "MQTT-Java-Example";
    private String topic = "/hfp/v2/journey/ongoing/vp/#";
    private String kafka_topic = "vehicle-positions";
    private MqttClient client;
    private KafkaProducer<String, String> producer;
    public Subscriber(KafkaProducer<String, String> producer) {
        this.producer = producer;
    }
    public void start() throws MqttException {
        MqttConnectOptions conOpt = new MqttConnectOptions();
        conOpt.setCleanSession(true);
        final String uuid = UUID.randomUUID().toString().replace("-", "");

        String clientId = this.clientId + "-" + uuid;
        this.client = new MqttClient(this.host, clientId, new MemoryPersistence());
        this.client.setCallback(this);
        this.client.connect(conOpt);

        this.client.subscribe(this.topic, this.qos);
    }
    public void connectionLost(Throwable cause) {
        System.out.println("Connection lost because: " + cause);
        System.exit(1);
    }
    public void deliveryComplete(IMqttDeliveryToken token) {
    }
    public void messageArrived(String topic, MqttMessage message) throws MqttException {
        System.out.println(String.format("[%s] %
s", topic, new String(message.getPayload())));
        final String key = topic;
        final String value = new String(message.getPayload());
        final ProducerRecord<String, String> record = new ProducerRecord<>
(this.kafka_topic, key, value);
        producer.send(record);
    }
}
```

(var1) MQTT connector

18 декабря 2020 г.   14:41



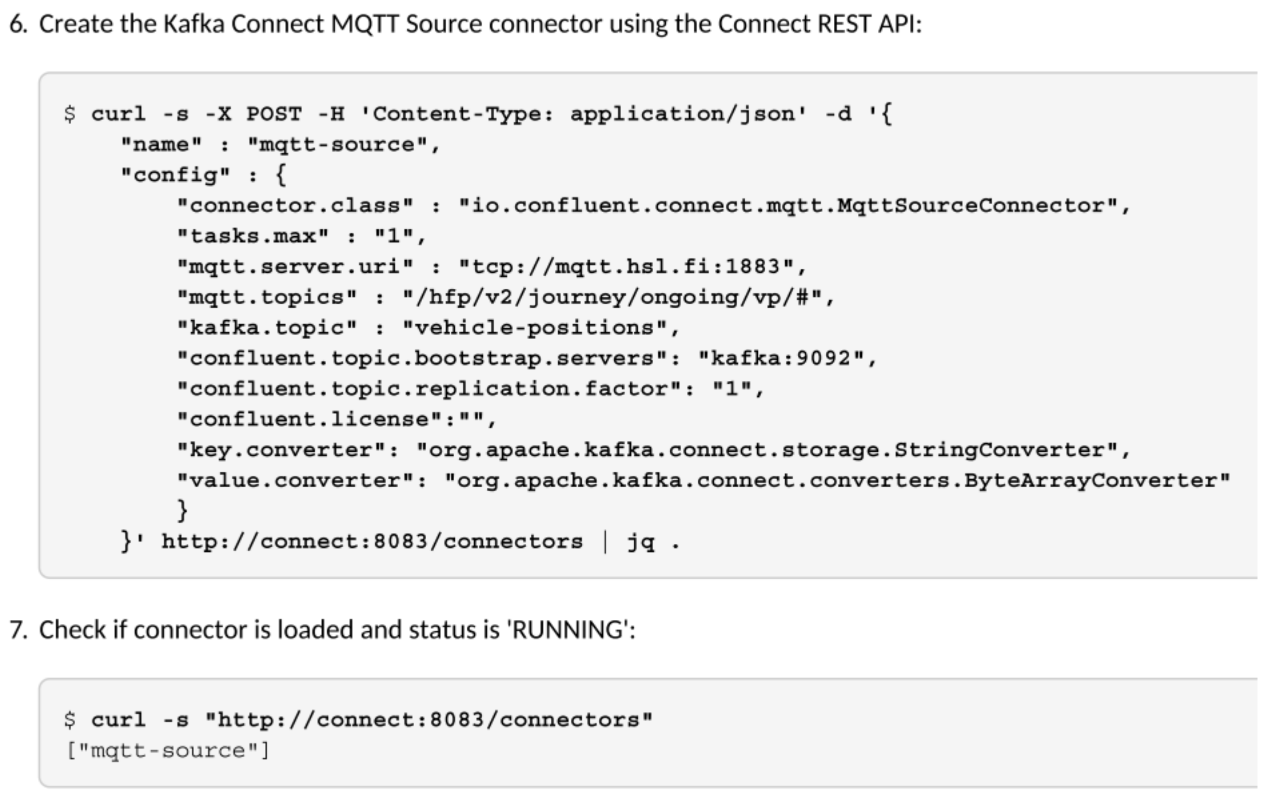
Testing the MQTT Data Source

1. Let’s install **Mosquitto**, a MQTT client:

```
$ sudo apt-get update && \
    sudo apt-get install -y software-properties-common && \
    sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa && \
    sudo apt-get install -y mosquitto-clients
```

2. To subscribe to the MQTT data provider we can use **Mosquitto’s** subscription tool as follows:

```
$ mosquitto_sub -h mqtt.hsl.fi -p 1883 -d -t "/hfp/v2/journey/ongoing/vp/#"
```



6. Create the Kafka Connect MQTT Source connector using the Connect REST API:

```
$ curl -s -X POST -H 'Content-Type: application/json' -d '{
    "name" : "mqtt-source",
    "config" : {
        "connector.class" : "io.confluent.connect.mqtt.MqttSourceConnector",
        "tasks.max" : "1",
        "mqtt.server.uri" : "tcp://mqtt.hsl.fi:1883",
        "mqtt.topics" : "/hfp/v2/journey/ongoing/vp/#",
        "kafka.topic" : "vehicle-positions",
        "confluent.topic.bootstrap.servers": "kafka:9092",
        "confluent.topic.replication.factor": "1",
        "confluent.license":"",
        "key.converter": "org.apache.kafka.connect.storage.StringConverter",
        "value.converter": "org.apache.kafka.connect.converters.ByteArrayConverter"
        }
    }' http://connect:8083/connectors | jq .
```

7. Check if connector is loaded and status is 'RUNNING':

```
$ curl -s "http://connect:8083/connectors"
["mqtt-source"]
```

(var2) Confluent MQTT proxy

18 декабря 2020 г.   14:42

1. If you haven't installed **Mosquitto** in the previous lab then follow these instructions:

```
$ sudo apt-get update && \
    sudo apt-get install -y software-properties-common && \
    sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa && \
    sudo apt-get install -y mosquitto-clients
```

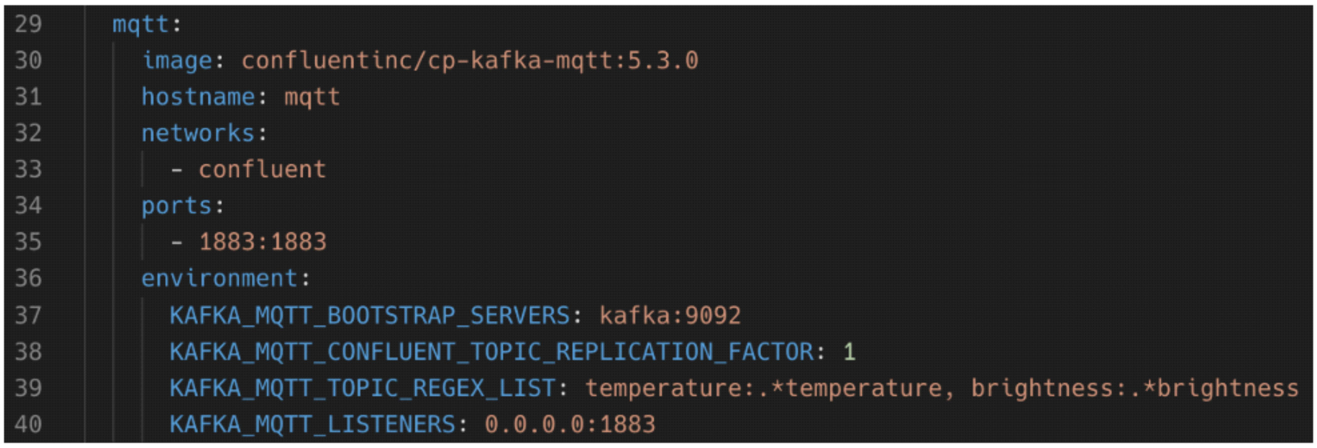
2. Navigate to the labs folder:

```
$ cd ~/confluent-dev/labs/mqtt-proxy
```

3. Run the Kafka cluster, including Kafka Connect:

```
$ docker-compose up -d
```

4. Open the file `docker-compose.yml` in your project folder and analyze the definition of the `mqtt` service:



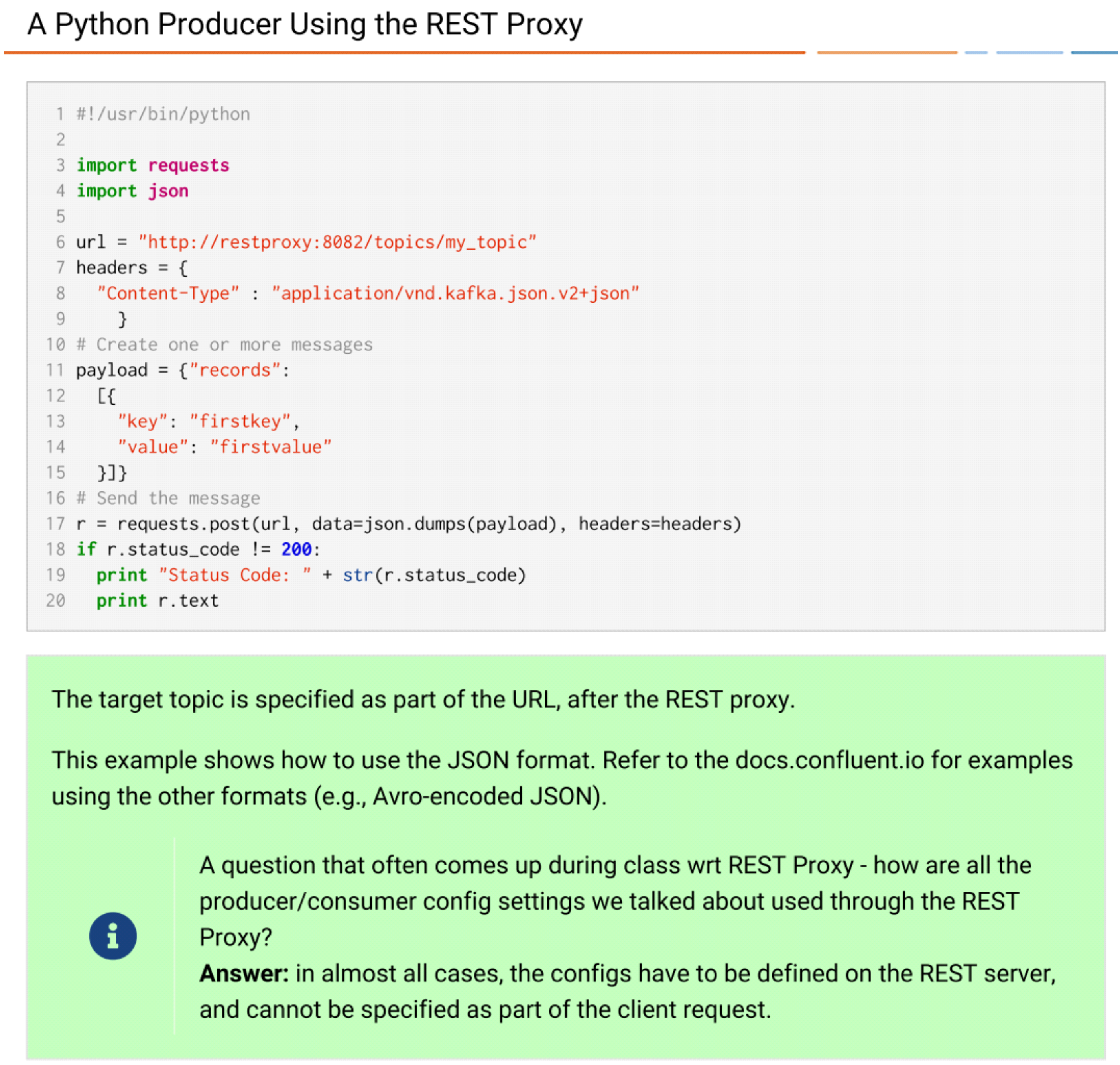
```
mqtt:
  image: confluentinc/cp-kafka-mqtt:5.3.0
  hostname: mqtt
  networks:
    - confluent
  ports:
    - 1883:1883
  environment:
    KAFKA_MQTT_BOOTSTRAP_SERVERS: kafka:9092
    KAFKA_MQTT_CONFLUENT_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_MQTT_TOPIC_REGEX_LIST: temperature:.*temperature, brightness:.*brightness
    KAFKA_MQTT_LISTENERS: 0.0.0.0:1883
```

Specifically have an eye on how the environment variable `KAFKA_MQTT_TOPIC_REGEX_LIST` is defined, which maps MQTT topics to Kafka topics. In our case every MQTT topic that ends in `temperature` is mapped to the Kafka topic `temperature`. Every MQTT topic that ends in `brightness` is mapped to the Kafka topic

* python producer example

14 декабря 2020 г. 16:02

для связи с сервером не используется какой-либо специальной библиотеки



A Python Producer Using the REST Proxy

```python
#!/usr/bin/python

import requests
import json

url = "http://restproxy:8082/topics/my_topic"
headers = {
  "Content-Type" : "application/vnd.kafka.json.v2+json"
    }
# Create one or more messages
payload = {"records":
  [{
    "key": "firstkey",
    "value": "firstvalue"
  }]}
# Send the message
r = requests.post(url, data=json.dumps(payload), headers=headers)
if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
```

The target topic is specified as part of the URL, after the REST proxy.

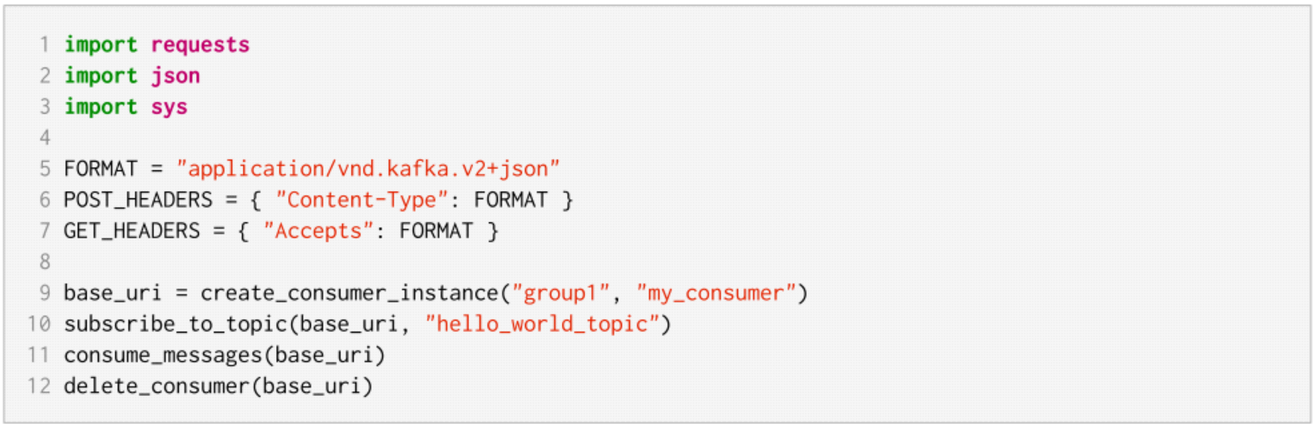
This example shows how to use the JSON format. Refer to the docs.confluent.io for examples using the other formats (e.g., Avro-encoded JSON).

A question that often comes up during class wrt REST Proxy - how are all the producer/consumer config settings we talked about used through the REST Proxy?
Answer: in almost all cases, the configs have to be defined on the REST server, and cannot be specified as part of the client request.

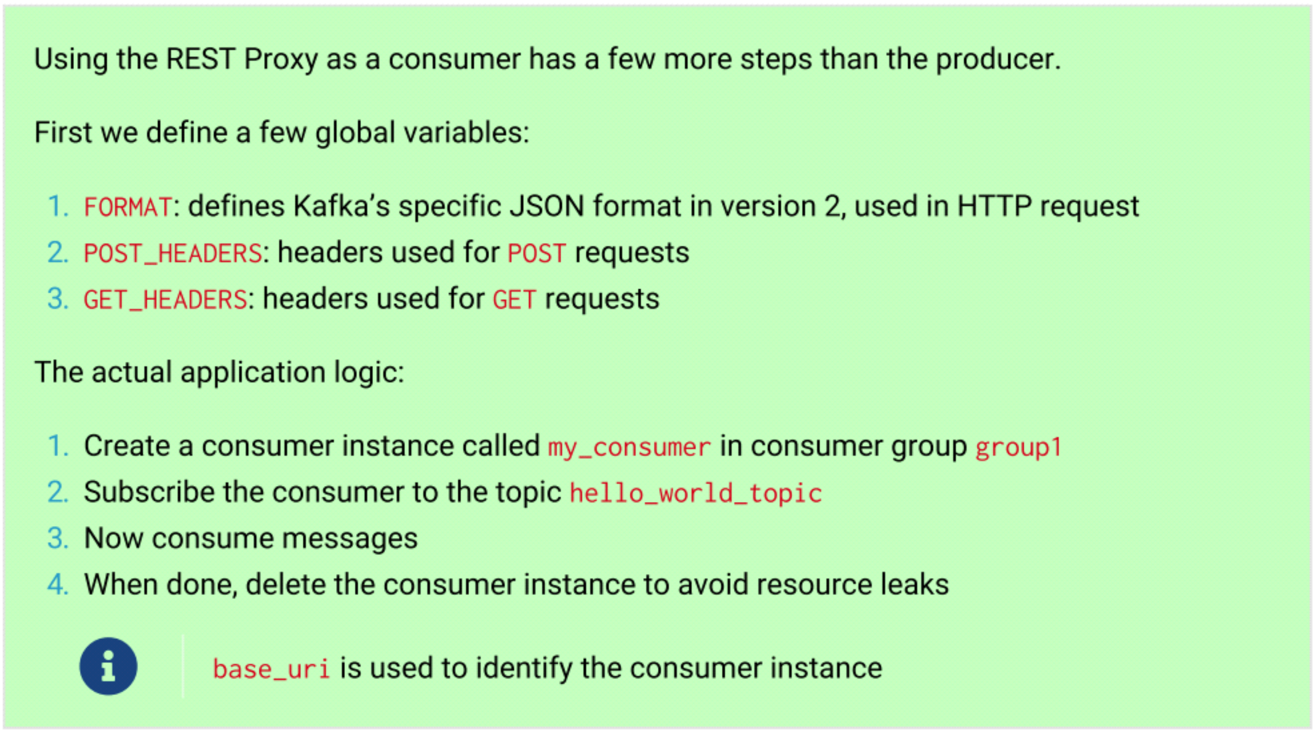
* python consumer example

14 декабря 2020 г.   20:24

Main Logic:



```python
import requests
import json
import sys

FORMAT = "application/vnd.kafka.v2+json"
POST_HEADERS = { "Content-Type": FORMAT }
GET_HEADERS = { "Accepts": FORMAT }

base_uri = create_consumer_instance("group1", "my_consumer")
subscribe_to_topic(base_uri, "hello_world_topic")
consume_messages(base_uri)
delete_consumer(base_uri)
```



Using the REST Proxy as a consumer has a few more steps than the producer.

First we define a few global variables:

1. `FORMAT`: defines Kafka's specific JSON format in version 2, used in HTTP request
2. `POST_HEADERS`: headers used for `POST` requests
3. `GET_HEADERS`: headers used for `GET` requests

The actual application logic:

1. Create a consumer instance called `my_consumer` in consumer group `group1`
2. Subscribe the consumer to the topic `hello_world_topic`
3. Now consume messages
4. When done, delete the consumer instance to avoid resource leaks

`base_uri` is used to identify the consumer instance

Creating the Consumer Instance:

```python
def create_consumer_instance(group_name, instance_name):
  url = f'http://restproxy:8082/consumers/{group_name}'
  payload = {
    "name": instance_name,
    "format": "json"
  }
  r = requests.post(url, data=json.dumps(payload), headers=POST_HEADERS)

  if r.status_code != 200:
    print ("Status Code: " + str(r.status_code))
    print (r.text)
    sys.exit("Error thrown while creating consumer")

  return r.json()["base_uri"]
```

In this part of the example, we are creating the instance of the consumer. Notice that the `url` is referencing the consumers rather than a topic name as we saw in the producer example.

1. We define the `url` to the desired consumer group
2. The payload defines among other things the instance name we want to use
3. Now we do an HTTP POST request to the `url`
4. In case of an error we stop the application with `sys.exit(…)`
5. As a last step, we return `base_uri`, which is used to identify the consumer instance

Subscribing to a topic

```python
def subscribe_to_topic(uri, topic_name):
  payload = {
    "topics": [topic_name]
  }

  r = requests.post(uri + "/subscription",
                    data=json.dumps(payload),
                    headers=POST_HEADERS)

  if r.status_code != 204:
    print("Status Code: " + str(r.status_code))
    print(r.text)
    delete_consumer(uri)
    sys.exit("Error thrown while subscribing the consumer to the topic")
```

In the body of the POST request we send the list of topics (here only a single one) that we want this consumer to subscribe to.

If there is an error, we delete the consumer and then exit the application

Consuming and processing messages:

```python
def consume_messages(uri);
  r = requests.get(base_uri + "/records", headers=GET_HEADERS, timeout=20)

  if r.status_code != 200:
    print ("Status Code: " + str(r.status_code))
    print (r.text)
    sys.exit("Error thrown while getting message")

  for message in r.json():
    if message["key"] is not None:
      print ("Message Key:" + message["key"])
      print ("Message Value:" + message["value"])
```

In a more realistic example you would loop indefinitely, since a topic is an open ended stream of data.

Deleting the consumer:

```python
def delete_consumer(uri):
  r = requests.delete(uri, headers=GET_HEADERS)

  if r.status_code != 204:
    print ("Status Code: " + str(r.status_code))
    print (r.text)
```

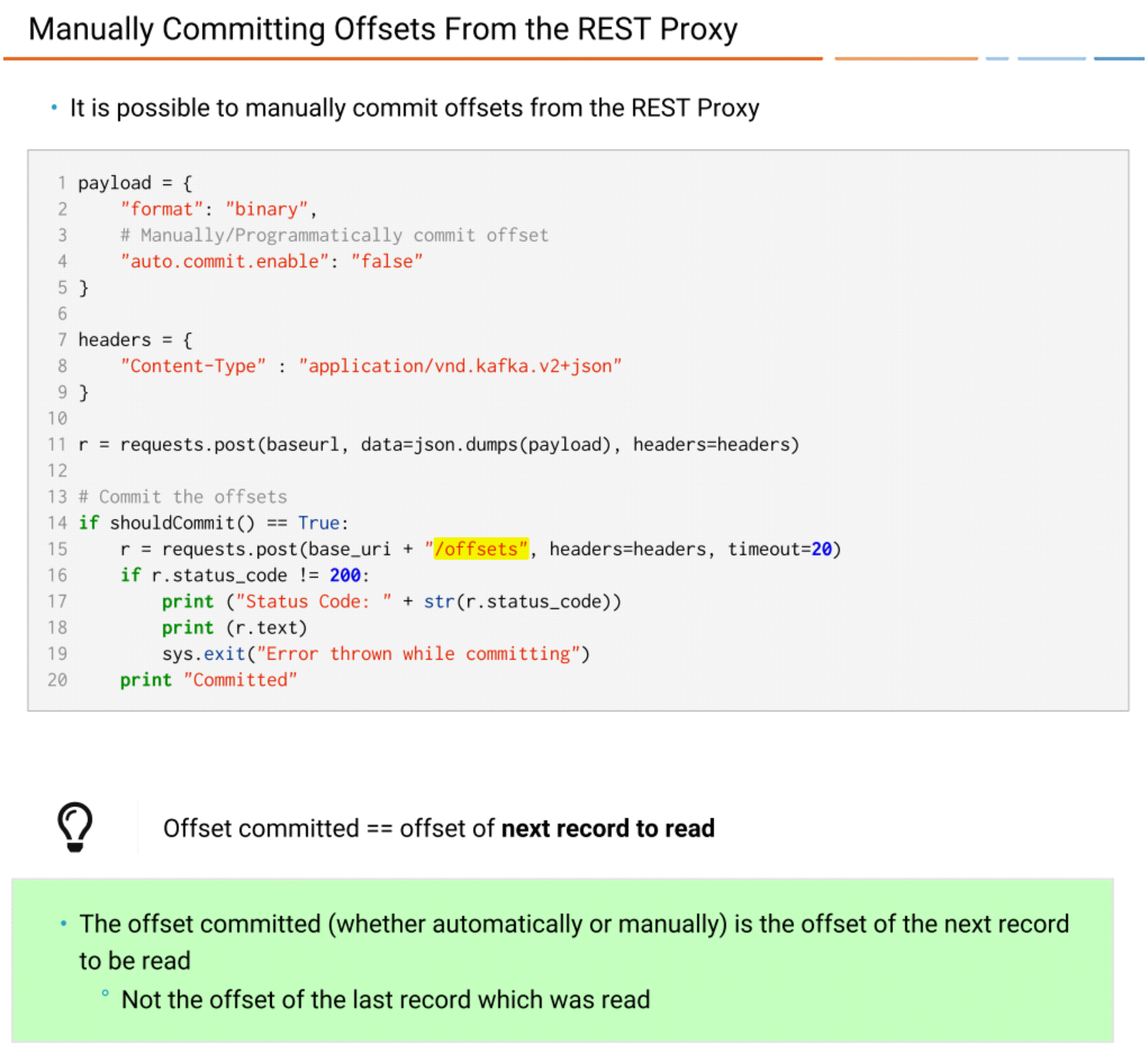
To verify that the consumer instances are removed properly, explicitly remove them using the `DELETE` call before exiting your code.

statefull call - используется четыре вызова

если не сделать deleting consumer то он отвалится по таймауту

* python consumer example: manual commit

16 декабря 2020 г. 17:33



Manually Committing Offsets From the REST Proxy

- It is possible to manually commit offsets from the REST Proxy

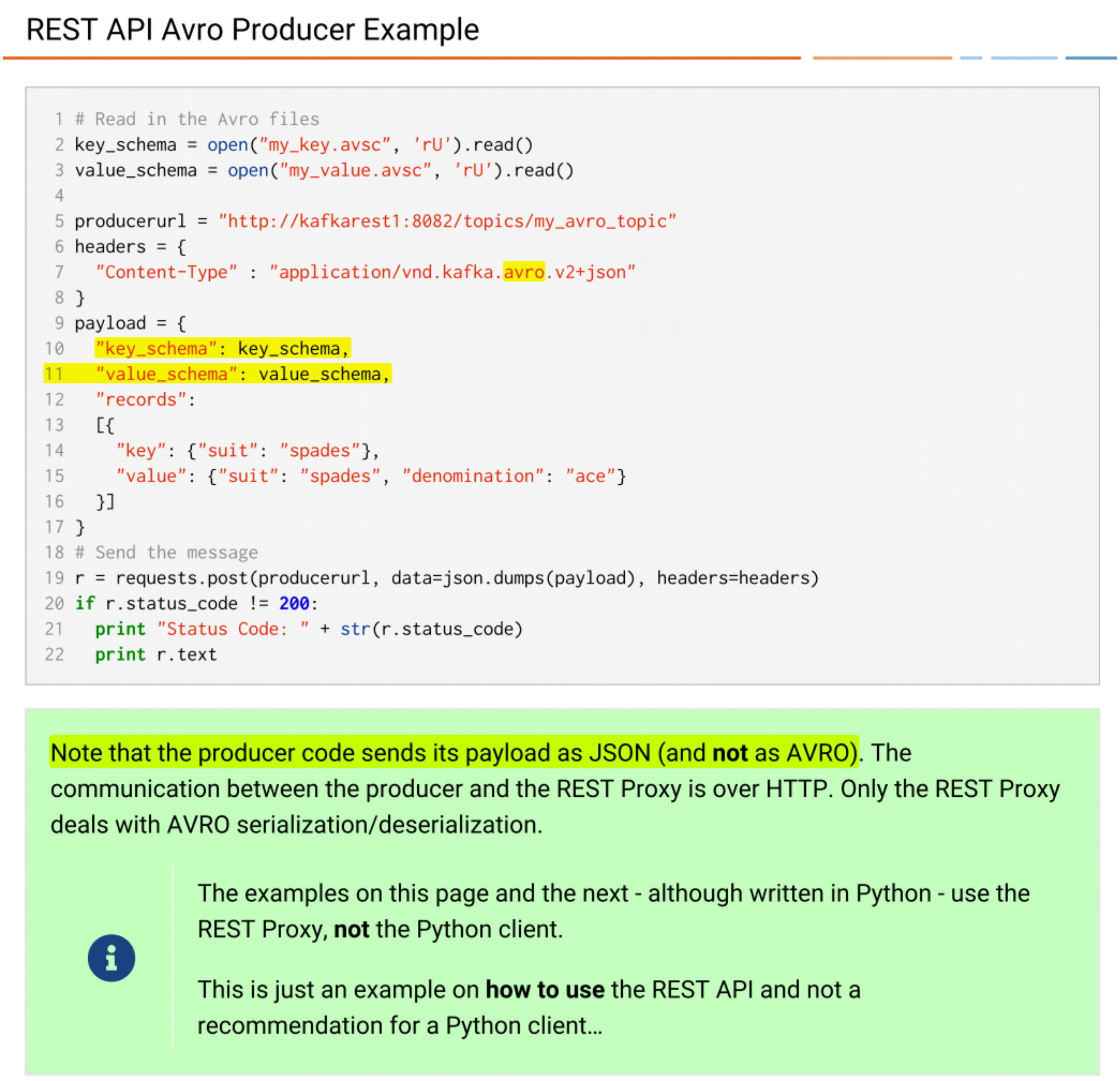
```python
payload = {
    "format": "binary",
    # Manually/Programmatically commit offset
    "auto.commit.enable": "false"
}

headers = {
    "Content-Type" : "application/vnd.kafka.v2+json"
}

r = requests.post(baseurl, data=json.dumps(payload), headers=headers)

# Commit the offsets
if shouldCommit() == True:
    r = requests.post(base_uri + "/offsets", headers=headers, timeout=20)
    if r.status_code != 200:
        print ("Status Code: " + str(r.status_code))
        print (r.text)
        sys.exit("Error thrown while committing")
    print "Committed"
```

Offset committed == offset of **next record to read**

- The offset committed (whether automatically or manually) is the offset of the next record to be read
  - Not the offset of the last record which was read

* python producer+consumer с реестром схем

17 декабря 2020 г. 21:53



REST API Avro Producer Example

```python
# Read in the Avro files
key_schema = open("my_key.avsc", 'rU').read()
value_schema = open("my_value.avsc", 'rU').read()

producerurl = "http://kafkarest1:8082/topics/my_avro_topic"
headers = {
  "Content-Type" : "application/vnd.kafka.avro.v2+json"
}
payload = {
  "key_schema": key_schema,
  "value_schema": value_schema,
  "records":
  [{
    "key": {"suit": "spades"},
    "value": {"suit": "spades", "denomination": "ace"}
  }]
}
# Send the message
r = requests.post(producerurl, data=json.dumps(payload), headers=headers)
if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
```

Note that the producer code sends its payload as JSON (and **not** as AVRO). The communication between the producer and the REST Proxy is over HTTP. Only the REST Proxy deals with AVRO serialization/deserialization.

The examples on this page and the next - although written in Python - use the REST Proxy, **not** the Python client.

This is just an example on **how to use** the REST API and not a recommendation for a Python client...

REST API Avro Consumer Example

```python
# Get the message(s) from the consumer
headers = {
  "Accept" : "application/vnd.kafka.avro.v2+json"
    }
# Request messages for the instance on the topic
r = requests.get(base_uri + "/topics/my_avro_topic", headers=headers, timeout=20)
if r.status_code != 200:
  print "Status Code: " + str(r.status_code)
  print r.text
  sys.exit("Error thrown while getting message")
# Output all messages
for message in r.json():
  keysuit = message["key"]["suit"]
  valuesuit = message["value"]["suit"]
  valuecard = message["value"]["denomination"]
  # Do something with the data
```

Don’t be surprised to see the client code deal with JSON and **not** with AVRO. AVRO serialization and deserialization is handled by the REST Proxy. Our client communicates via HTTP REST calls with the Proxy. From a consumer’s perspective AVRO is hidden away. Only the producer needs to know about AVRO, as it needs to send the schemas with the POST request to the REST Proxy.