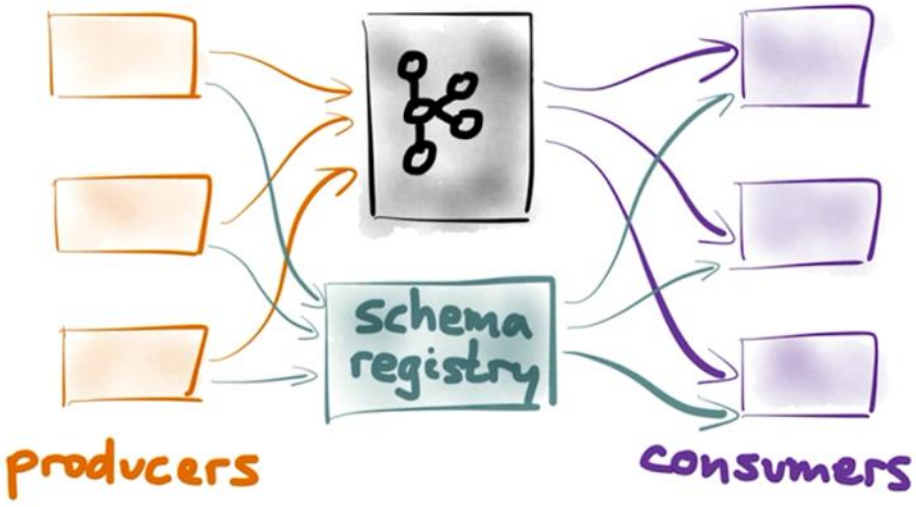
_общее

30 ноября 2020 г. 20:28



кафка не проверяет(это делает реестр схем) схему сообщений, именно поэтому она такая быстрая

- на уровне самой кафки есть только биты-байты и нет какой либо их интерпретации

- What if the Kafka Brokers were verifying the messages they receive?

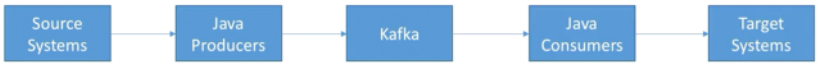
- It would break what makes Kafka so good:
  - Kafka doesn't parse or even read your data (no CPU usage)
  - Kafka takes bytes as an input without even loading them into memory (that's called zero copy)
  - Kafka distributes bytes
  - As far as Kafka is concerned, it doesn't even know if your data is an integer, a string etc.

- The Schema Registry has to be a separate components
- Producers and Consumers need to be able to talk to it
- The Schema Registry must be able to reject bad data
- A common data format must be agreed upon
  - It needs to support schemas
  - It needs to support evolution
  - It needs to be lightweight
- Enter… the Confluent Schema Registry
- And Apache Avro as the data format.

- Utilizing a schema registry has a lot of benefits
- BUT it implies you need to
  - Set it up well
  - Make sure it's highly available
  - Partially change the producer and consumer code
- Apache Avro as a format is awesome but has a learning curve
- The schema registry is free and open sourced, created by Confluent (creators of Kafka)
- As it takes time to setup, we won't cover the usage in this course

- без схемы обычно используем JSON
- схема необходима например когда используем AVRO

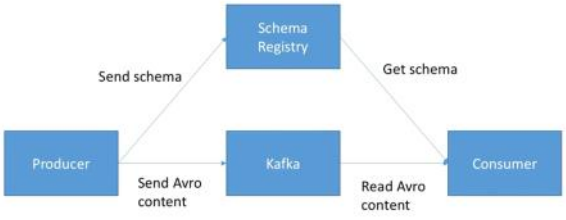
Pipeline without Schema Registry



сама схема не передается в сообщениях от продьюсера консьюмеру (поэтому и нужен реестр схем)

Confluent Schema Registry Purpose

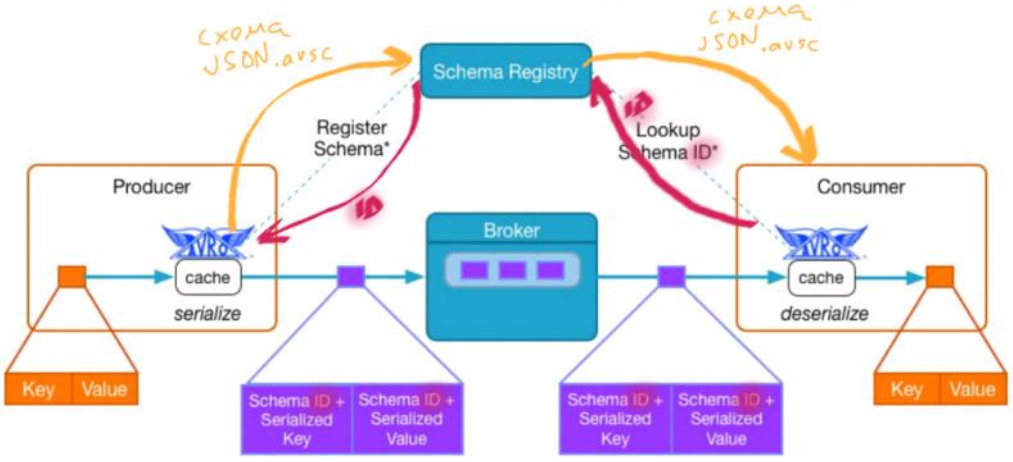
- Store and retrieve schemas for Producers / Consumers
- Enforce Backward / Forward / Full compatibility on topics
- Decrease the size of the payload of data sent to Kafka



кеш ускоряет работу со схемами так как они часто не меняются, поэтому они есть в локальном кеше на консьюмере/продьюсере

Schema Registration and Data Flow

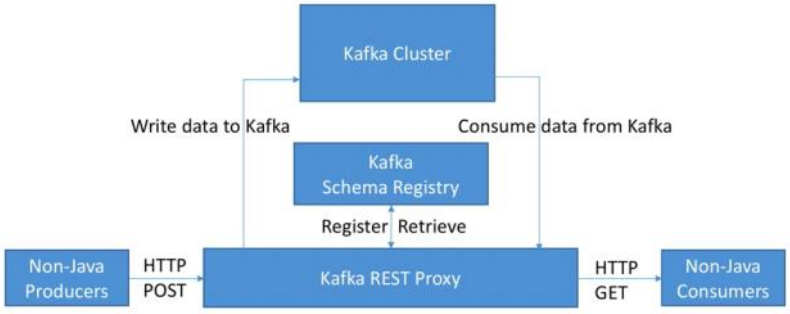
- The message key and value can be independently serialized
- Producers serialize data and prepend the schema ID
- Consumers use the schema ID to deserialize the data
- Schema Registry communication is only on the first message of a new schema
  - Producers and Consumers cache the schema/ID mapping for future messages



Traditionally, the full JSON schema is included with any data that is created with the associated data types. In a Kafka environment, this could significantly increase the size of the key and/or value of a message and so is unacceptable. The Schema Registry solves this issue by storing schemas in a special topic and identifying them by a ID number. That ID is included with the keys and values instead of the JSON schema.

Note the * in the diagram next to "Register Schema" and "Lookup Schema ID". This is a reminder that this happens only on new schemas/IDs. Otherwise, previously used schemas are cached locally on the client.

Kafka Ecosystem: Confluent REST Proxy

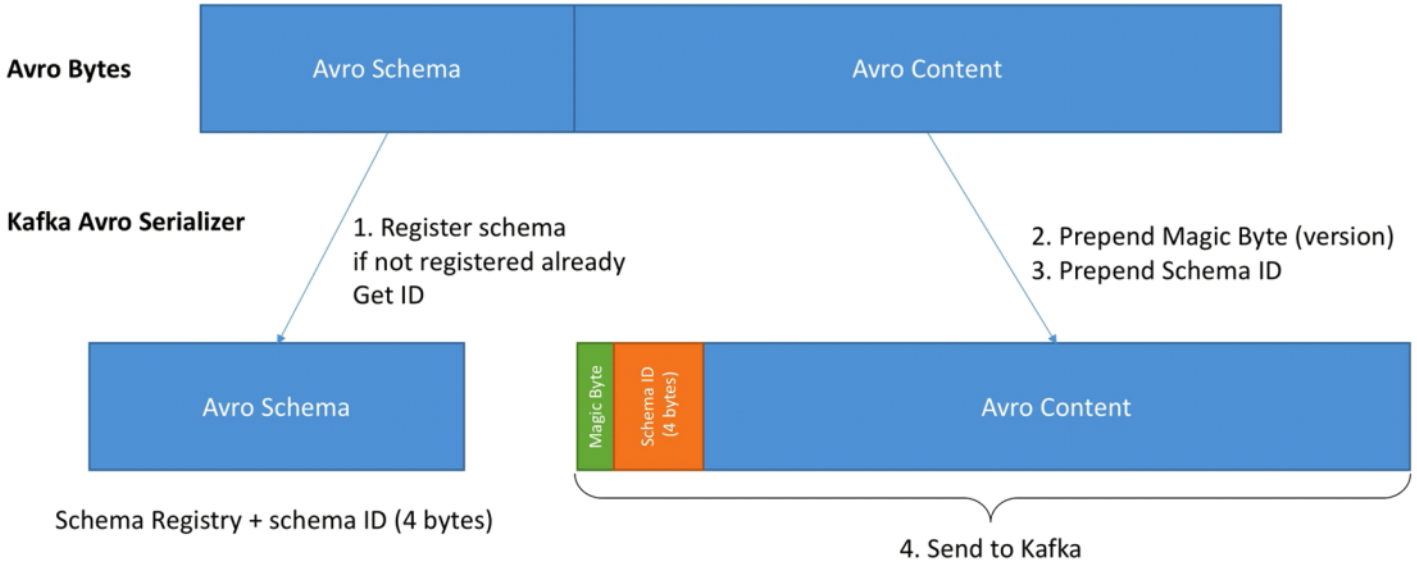


проверка схем замедляет кафку

- проверка схем может отклнить плохие данные не прошедшие проверку

схема регистрируется автоматически в реестре схем при первом обращении к реестру

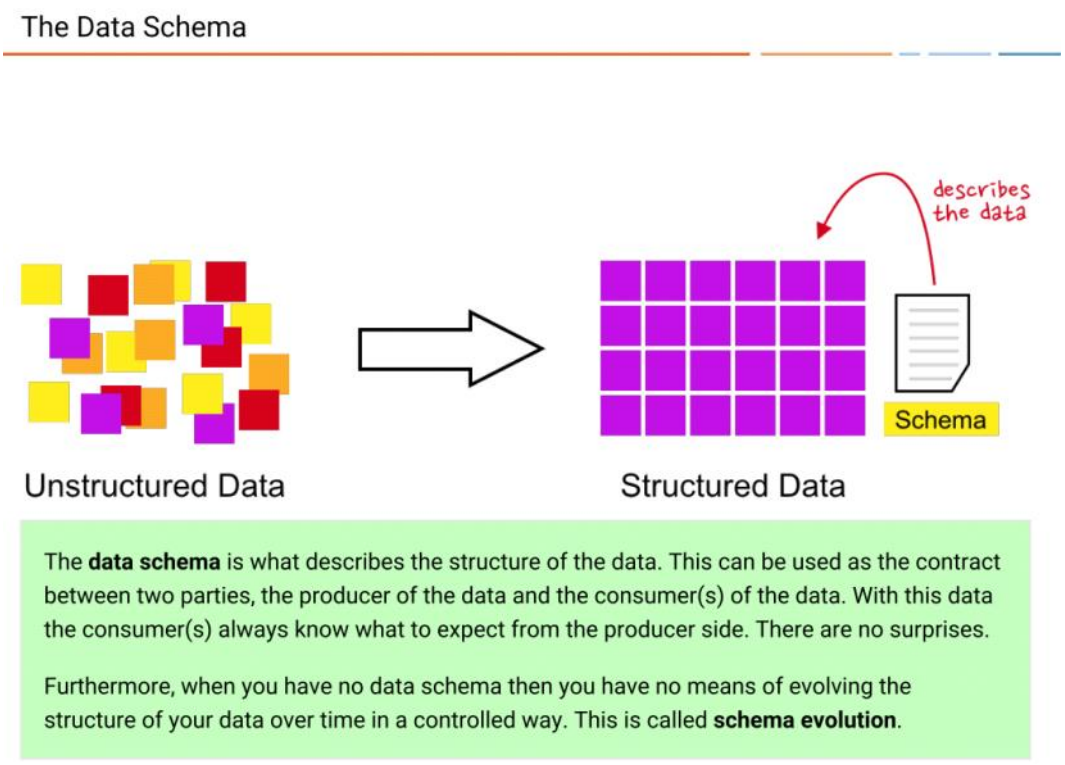
- те мне явно регистрировать ее ненадо, либа на клиенте сама ее зарегит и сэволюционирует



- We now better understand how the schema registry works
  - It externalises the schema (the schema doesn't live in Kafka)
  - It reduces the message size by a lot as the schema is never sent
  - If the schema registry is not available, this could break producers / consumers

kafka schema registry

5 декабря 2020 г. 20:26



schema registry - это единый каталог всех форматов данных (все контрактов, наподобие UDDI-registry)

What Is Confluent Schema Registry?

- Confluent Schema Registry provides centralized management of schemas
  - Stores a versioned history of all schemas
  - Provides a RESTful interface for storing and retrieving Avro schemas
  - Checks schemas and throws an exception if data does not conform to the schema
  - Allows evolution of schemas according to the configured compatibility setting
- Sending the Avro schema with each message would be inefficient
  - Instead, a globally unique ID representing the Avro schema is sent with each message
- The Schema Registry stores schema information in a special Kafka topic
- The Schema Registry is accessible both via a REST API and a Java API
  - There are also command-line tools, `kafka-avro-console-producer` and `kafka-avro-console-consumer`

Schema Registry is a component that provides a management and lookup service for schemas. It provides a RESTful interface for storing and retrieving Avro schemas. It stores a versioned history of all schemas, provides multiple compatibility settings and allows evolution of schemas according to the configured compatibility setting. It provides serializers that plug into some of the Kafka clients (Java, Python, and .NET) that handle schema storage and retrieval for Kafka messages that are sent in the Avro format.

A special Kafka topic (default name is "_schemas") is required and can be reconfigured with the `kafkastore.topic` parameter.

Spanning multiple data centers with your Schema Registry provides additional protection against data loss and improved latency. The recommended multi-datacenter deployment designates one datacenter as “master” and all others as “slaves”.

Confluent GUI tool

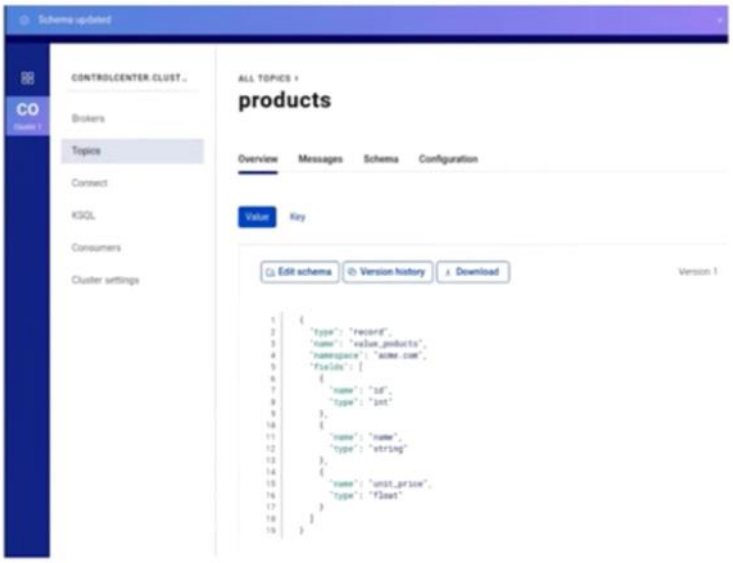
Confluent Schema Registry Operations

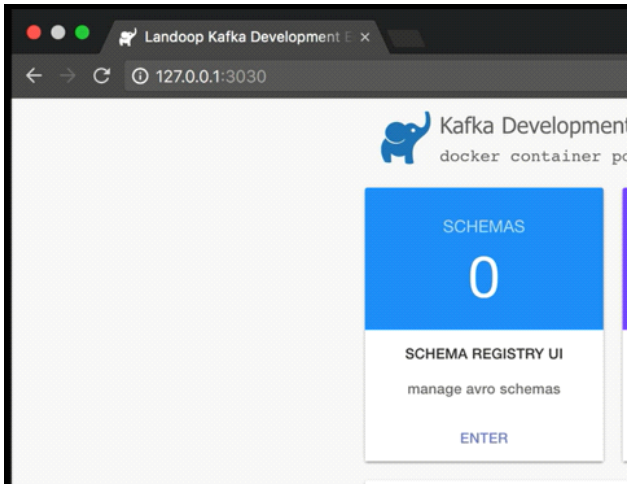
- We can add schemas
- We can retrieve a schema
- We can update a schema
- We can delete a schema (as of 3.3.0).
- All of this through a REST API

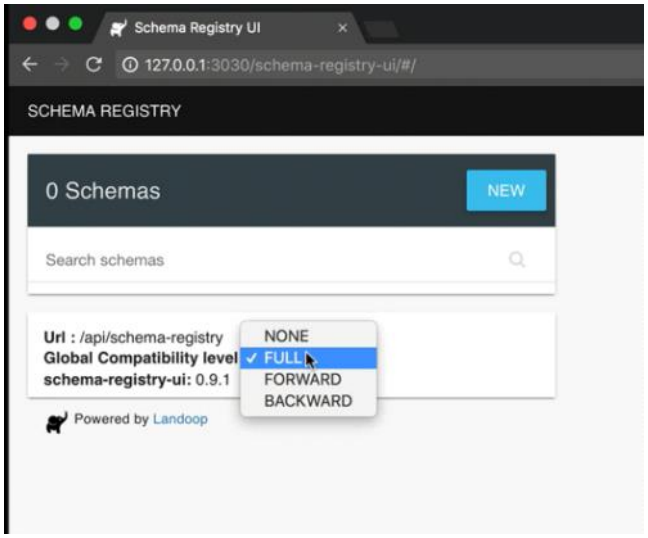
- Schemas can be applied to key and / or values!

Viewing Schemas in Confluent Control Center

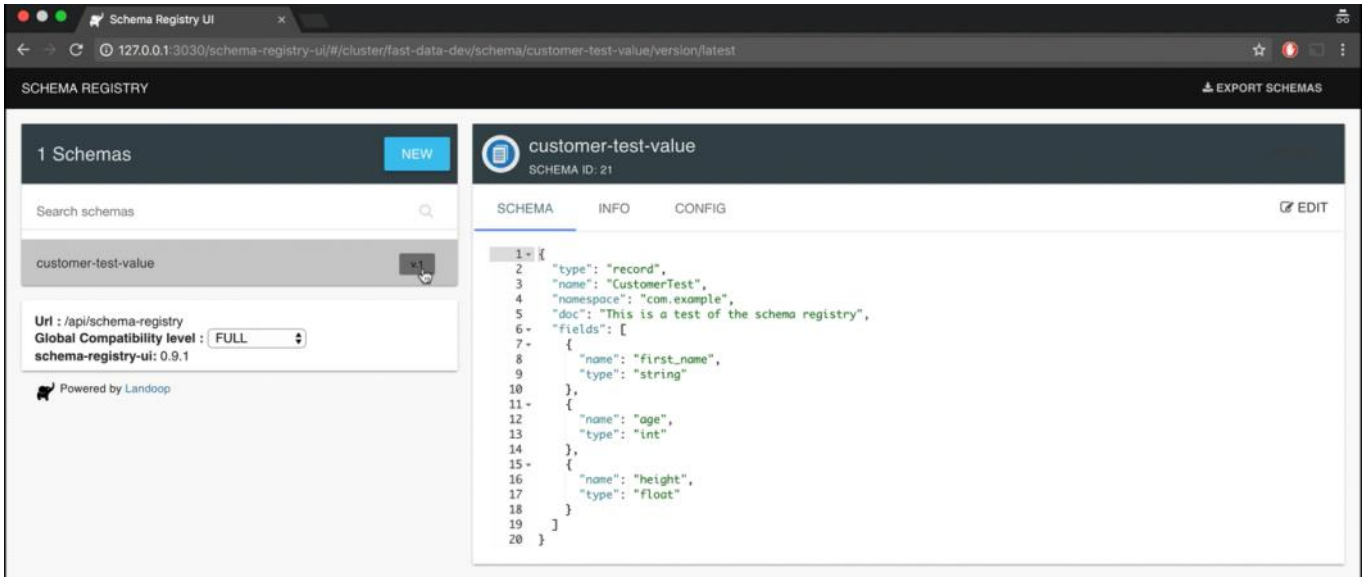
Viewing Schemas in Confluent Control Center

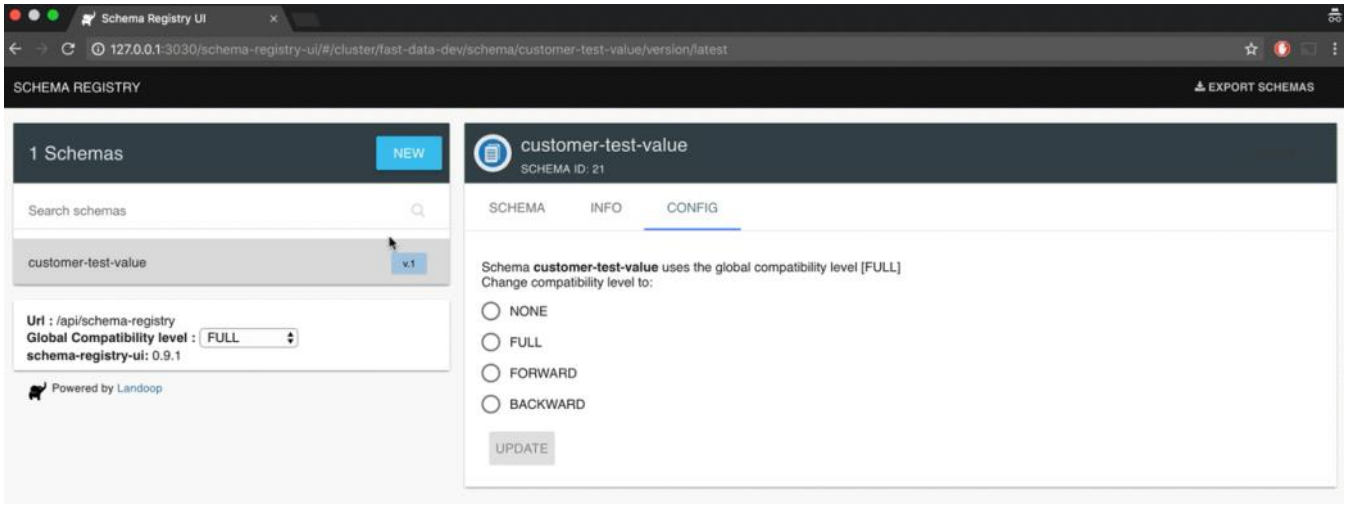




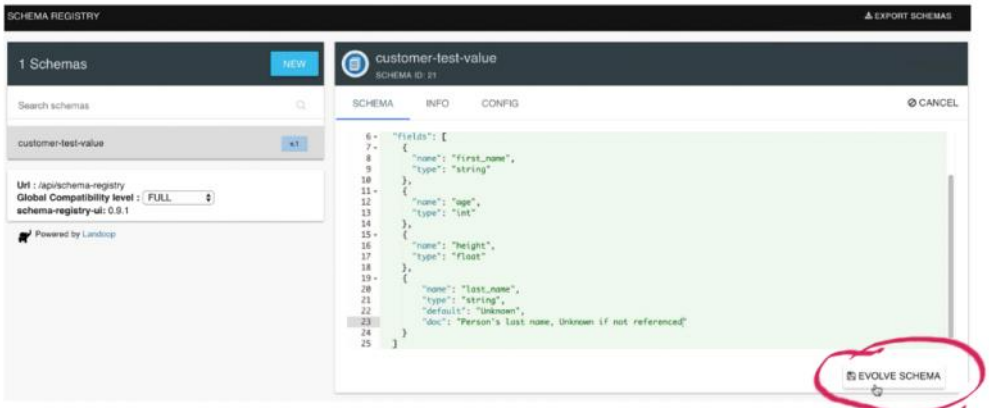


(1) как добавить новую схему в реестр

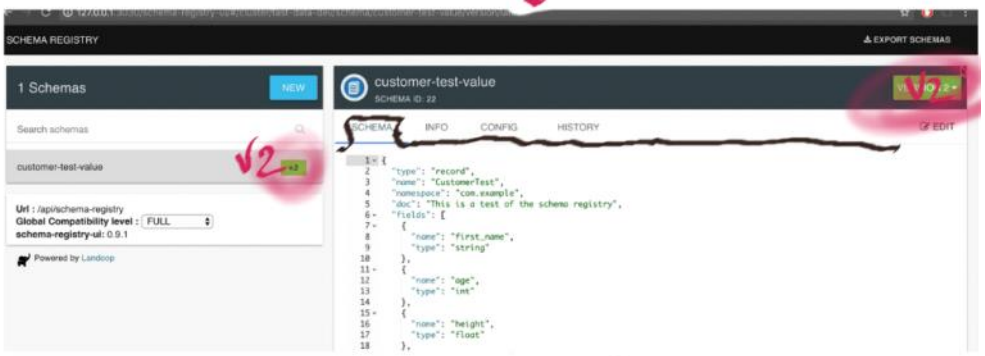




(2) как заэволюционировать схему (те поменять на новую версию)


SCHEMA REGISTRY
EXPORT SCHEMAS
1 Schemas
NEW
Search schemas
customer-test-value
v1
Url : /api/schema-registry
Global Compatibility level : FULL
schema-registry-ui: 0.9.1
Powered by Landoop
customer-test-value
SCHEMA ID: 21
SCHEMA
INFO
CONFIG
CANCEL
"fields": [
{
"name": "first_name",
"type": "string"
},
{
"name": "age",
"type": "int"
},
{
"name": "height",
"type": "float"
},
{
"name": "last_name",
"type": "string",
"default": "Unknown",
"doc": "Person's last name, Unknown if not referenced"
}
]
EVOLVE SCHEMA

evolve


SCHEMA REGISTRY
EXPORT SCHEMAS
1 Schemas
NEW
Search schemas
customer-test-value
v2
Url : /api/schema-registry
Global Compatibility level : FULL
schema-registry-ui: 0.9.1
Powered by Landoop
customer-test-value
SCHEMA ID: 22
SCHEMA
INFO
CONFIG
HISTORY
EDIT
{
"type": "record",
"name": "CustomerTest",
"namespace": "com.example",
"doc": "This is a test of the schema registry",
"fields": [
{
"name": "first_name",
"type": "string"
},
{
"name": "age",
"type": "int"
},
{
"name": "height",
"type": "float"
},


Search schemas
customer-test-value
v2
Url : /api/schema-registry
Global Compatibility level : FULL
schema-registry-ui: 0.9.1
Powered by Landoop
SCHEMA
INFO
CONFIG
HISTORY
Version 2 (Schema ID: 22)
},
{
"name": "last_name",
"type": "string",
"doc": "Person's last name, Unknown if not referenced",
"default": "Unknown"
Version 1 (Schema ID: 21)
{
"type": "record",
"name": "CustomerTest",
"namespace": "com.example",
"doc": "This is a test of the schema registry",
"fields": [
{
"name": "first_name",
"type": "string"

CLI

6 декабря 2020 г. 15:08

AVRO Command-line Tools

AVRO Command-line Tools

Producer

```
$ kafka-avro-console-producer \
    --broker-list ${YOUR_BOOTSTRAP_SERVER} \
    --property schema.registry.url=schemaregistry1:8081 \
    --topic my_avro_topic \
    --property value.schema="${MY_AVRO_SCHEMA}"
```

Consumer

```
$ kafka-avro-console-consumer \
    --bootstrap-server broker1:9092 \
    --property schema.registry.url=schemaregistry1:8081 \
    --from-beginning \
    --topic my_avro_topic
```

In the producer example, MY_AVRO_SCHEMA is an environment variable that contains the schema. It may defined like this:

```
$ MY_AVRO_SCHEMA='{
    "type":"record",
    "namespace": "example.avro",
    "name":"product",
    "fields":[{"name":"id",         "type":"int"},
              {"name":"name",       "type":"string"},
              {"name":"unit_price","type":"float"}]
  }'
```

пример эволюции схемы

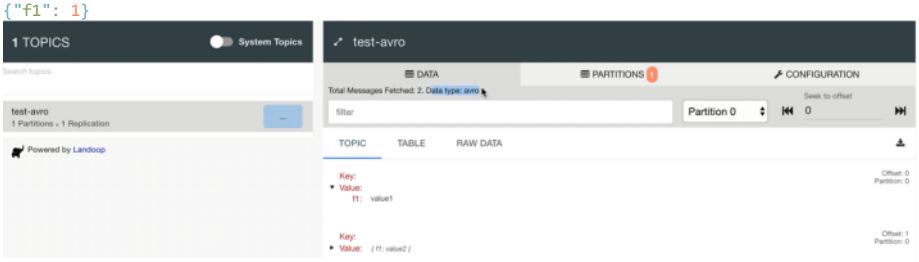
```
~/confluent-3.3.0 bin/camus-run
camus-config*                       kafka-delete-records*               kafka-streams-application-reset*
camus-run*                          kafka-mirror-maker*                 kafka-topics*
confluent*                          kafka-preferred-replica-election*   kafka-verifiable-consumer*
connect-distributed*                kafka-producer-perf-test*           kafka-verifiable-producer*
connect-standalone*                 kafka-reassign-partitions*          schema-registry-run-class*
kafka-acls*                         kafka-replay-log-producer*          schema-registry-start*
kafka-avro-console-consumer*        kafka-replica-verification*         schema-registry-stop*
kafka-avro-console-producer*        kafka-rest-run-class*               schema-registry-stop-service*
kafka-broker-api-versions*          kafka-rest-start*                   support-metrics-bundle*
kafka-configs*                      kafka-rest-stop*                    windows/
kafka-console-consumer*             kafka-rest-stop-service*            zookeeper-security-migration*
kafka-console-producer*             kafka-run-class*                    zookeeper-server-start*
kafka-consumer-groups*              kafka-server-start*                 zookeeper-server-stop*
kafka-consumer-offset-checker*      kafka-server-stop*                  zookeeper-shell*
kafka-consumer-perf-test*           kafka-simple-consumer-shell*
```

```
# Use a docker image to have access to all the binaries right away:
docker run -it --rm --net=host confluentinc/cp-schema-registry:3.3.1 bash
# Then you can do
kafka-avro-console-consumer
```

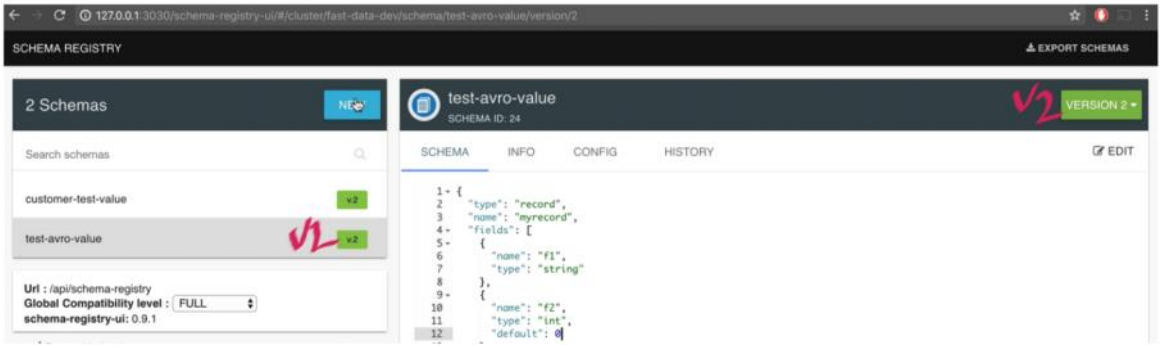
```
#!/bin/bash
# change 127.0.0.1 by your Docker ADV_HOST
# Get the command line from:
docker run --net=host -it confluentinc/cp-schema-registry:3.3.0 bash

# Or download the Confluent Binaries at https://www.confluent.io/download/
# And add them to your PATH
# Produce a record with one field
kafka-avro-console-producer \
    --broker-list 127.0.0.1:9092 --topic test-avro \
    --property schema.registry.url=http://127.0.0.1:8081 \
    --property value.schema='{"type":"record","name":"myrecord","fields":[{"name":"f1","type":"string"}]}'
```

```
повлем в топик тестовые данные
{"f1": "value1"}
{"f1": "value2"}
```

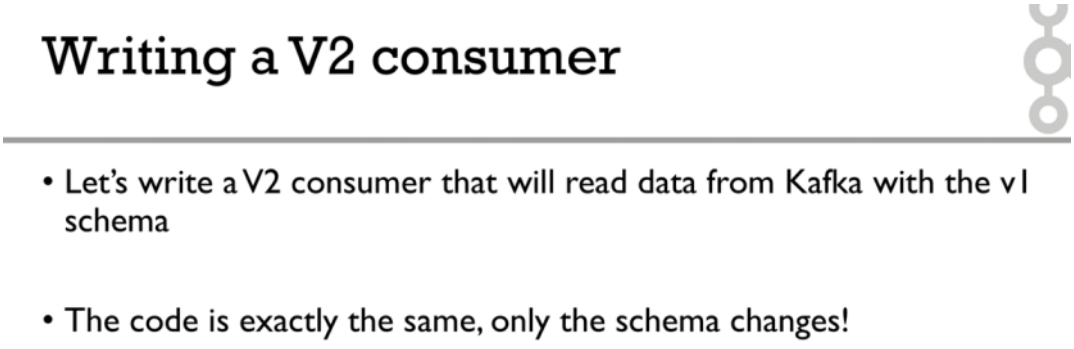
```
{"f1": "value3"}
# let's trigger an error:
{"f2": "value4"}
# let's trigger another error:
{"f1": 1}
```




```
# Consume the records from the beginning of the topic:
kafka-avro-console-consumer --topic test-avro \
    --bootstrap-server 127.0.0.1:9092 \
    --property schema.registry.url=http://127.0.0.1:8081 \
    --from-beginning
в консьюмере схему указывать ненадо

# Produce some errors with an incompatible schema (we changed to int) - should produce a 409
kafka-avro-console-producer \
    --broker-list localhost:9092 --topic test-avro \
    --property schema.registry.url=http://127.0.0.1:8081 \
    --property value.schema='{"type":"int"}'
     при повторной отправке сообщений схему указывать ненадо тк она уже автоматически зарегестрирована

# Some schema evolution (we add a field f2 as an int with a default)
kafka-avro-console-producer \
    --broker-list localhost:9092 --topic test-avro \
    --property schema.registry.url=http://127.0.0.1:8081 \
    --property value.schema='{"type":"record","name":"myrecord","fields":[{"name":"f1","type":"string"},{"name": "f2", "type": "int", "default": 0}]}'
     а если я всетки укажу другу схему при отправке то она автоматически сэволюционирует и автоматически зарегестрируется как v2
```




```
{"f1": "evolution", "f2": 1 }

# Consume the records again from the beginning of the topic:
kafka-avro-console-consumer --topic test-avro \
    --bootstrap-server localhost:9092 \
    --from-beginning \
    --property schema.registry.url=http://127.0.0.1:8081
```



как вручную зарегестрировать схему через CURL

- Other clients use the Schema Registry REST API to manually pre-register schemas and use the IDs in the requests

```
$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    --data '{"schema": "{\"type\": \"string\"}"}' \
    http://schemaregistry1:8081/subjects/<topic-name>-key/versions

$ curl -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    --data '{"schema": "{\"type\": \"string\"}"}' \
    http://schemaregistry1:8081/subjects/<topic-name>-value/versions

$ curl -X GET http://schemaregistry1:8081/schemas/ids/1
```

<topic-name> is the name of the topic where the schema is used.

Schema Registry is supported with other clients, but requires the registration and ID lookups to be done manually through the REST interface.

TopicNameStrategy- по дефолту на топике только одна схема для ключей и одна схема для значений сообщения

- но можно сделать так что в одном топике будут гулять сообщения разных типов данных (те с разными схемами)

Subject Naming Strategies

Configurations	Naming Strategies
• key.subject.name.strategy • value.subject.name.strategy	• TopicNameStrategy (default) • RecordNameStrategy • TopicRecordNameStrategy

Configurations

- key.subject.name.strategy
- value.subject.name.strategy

3 possible settings are available for each of the above config property:

- TopicNameStrategy (default): <subject-name> = <topic>-key|<topic>-value
- RecordNameStrategy: <subject-name> = <type>
  This setting also allows any number of event types in the same topic
- TopicRecordNameStrategy: <subject-name> = <topic>-<type>
  This setting also allows any number of event types in the same topic, and further constrains the compatibility check to the current topic only

Where <topic> is the topic name and <type> is the fully qualified Avro record type name

Alternatively users can create **different Topics** for different schemas.

как явно задать compatibility mode для схем на топике

Different Versions of Schemas in the Same Topic

- Different versions of schemas in the same Topic can be Backward, Forward, or Full compatible
  - Default is `BACKWARD`
- If they are neither, set compatibility to `NONE`
  - Code has no assumptions on schema as long as it is valid Avro
  - Code has full burden to read and process data
- Configuring compatibility
  - Use REST API

```
$ curl -X PUT -i -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    --data '{"compatibility": "NONE"}' \
    http://schemaregistry1:8081/config/my_topic
```

Some people call a Topic that has multiple schemas a "fat" Topic. For a detailed discussion of when to use this approach, and the use of **custom subject naming strategies**, refer to: https://www.confluent.io/blog/put-several-event-types-kafka-topic/

When Confluent's Avro serializer registers a schema in the registry, it does so under a subject name. By default, that subject is `<topic>-key` for message keys and `<topic>-value` for message values. The schema registry then checks the mutual compatibility of all schemas that are registered under a particular subject.

schema-on-read approach

4 января 2021 г. 12:19

schema on read - означает что мы runtime проверяем схему сообщений
late-bound approach

- The pain of long schema migrations is one of the telltale criticisms of the relational era. But the reality is that evolving schemas are a fundamental attribute of the way data ages. The main difference between then and now is that late-bound/ schema-on-read approaches allow many incompatible data schemas to exist in the same table, topic, or the like at the same time. This pushes the problem of translating the format—from old to new—into the application layer, hence the name “schema on read.”

преимущество в том что: мы можем перейти на новую версию схемы без миграции данных

- Schema on read turns out to be useful in a couple of ways. In many cases recent data is more valuable than older data, so programs can move forward without migrating older data they don’t really care about. This is a useful, pragmatic solution used broadly in practice, particularly with messaging

1) PLAINTEXT

17 декабря 2020 г. 18:22

как вариант можно вообще НЕ использовать схему на проектах

- но тогда нельзя будет делать runtime проверку ошибок в формате поступающих сообщений
- тогда получается что у операции нет контракта

The Need for a More Complex Serialization System

So far, all our data has been **plain text**...

PROS

- Supported by most programming languages
- **Easy to inspect** files for debugging

CONS

- Data is stored **inefficiently**
- Non-text data must be converted to strings
  - No type checking is performed
  - It is inefficient to convert binary data to strings
- No **schema evolution**

There are a few pros and cons when using a purely text based data format. One of the biggest drawbacks is the lack of a well defined **data schema**. We're going to talk about this in detail on the next slide.

2) AVRO

5 декабря 2020 г. 20:38

Avro: An Efficient Data Serialization System

- Avro is an Apache OSS project
- Provides data serialization
- Data is defined with a self-describing schema
- Supported by many programming languages, including Java
- Provides a data structure format
- Supports code generation of data types
- Provides a container file format
- Avro data is binary, so stores data efficiently
- Type checking is performed at write time


```
{
 "namespace": "model",
 "type": "record",
 "name": "SimpleCard",
 "fields": [
     {
       "name": "suit",
       "type": "string",
       "doc" : "The suit of the card"
     },
     {
       "name": "denomination",
       "type": "string",
       "doc" : "The card number"
     }
 ]
}
```

From clouddurable.com: "Apache Avro is a data serialization system. Avro provides data structures, binary data format, container file format to store persistent data, and provides RPC capabilities. Avro does not require code generation to use and integrates well with JavaScript, Python, Ruby, C, C#, C++ and Java. Avro gets used in the Hadoop ecosystem as well as by Kafka."

неймспейс авро схемы должен совпадать с именем java пакета
src/main/avro - путь для avro-схемы
.avsc расширение для файла со схемой

- By default, the schema definition is placed in `src/main/avro`
  - File extension is `.avsc`
- The `namespace` is the Java package name, which you will import into your code

AVRO-сообщения и AVRO-схемы в итоговом виде представляются в виде JSON (хоть и передаются внутри бинарно)

Name	Description	Java equivalent
`boolean`	True or false	`boolean`
`int`	32-bit signed integer	`int`
`long`	64-bit signed integer	`long`
`float`	Single-precision floating-point number	`float`
`double`	Double-precision floating-point number	`double`
`string`	Sequence of Unicode characters	`java.lang.CharSequence`
`bytes`	Sequence of bytes	`java.nio.ByteBuffer`
`null`	The absence of a value	`null`

Avro Data Types (Complex)

Name	Description
`record`	A user-defined field comprising one or more simple or complex data types, including nested records
`enum`	A specified set of values
`union`	Exactly one value from a specified set of types
`array`	Zero or more values, each of the same type
`map`	Set of key/value pairs; key is always a `string`, value is the specified type
`fixed`	A fixed number of bytes
`logical`	Avro primitive or complex type with extra attributes to represent a derived type

Avro Primitive Types

- Primitive Types are the support base types
  - null: no value
  - boolean: a binary value
  - int: 32-bit signed integer
  - long: 64-bit signed integer
  - float: single precision (32-bit) IEEE 754 floating-point number
  - double: double precision (64-bit) IEEE 754 floating-point number
  - bytes: sequence of 8-bit unsigned bytes
  - string: unicode character sequence

type: record свой кастомный тип данных

`record` is the most important of these, as we will see

The complex types can be nested (e.g., a union of arrays, a map of enums).

Example for the use of `union`:

```
{
   "type" : "record",
   "namespace" : "tutorialspoint",
   "name" : "empdetails ",
   "fields" :
   [
      { "name" : "experience", "type": ["int", "null"] }, { "name" : "age", "type": "int" }
   ]
}
```

In this case the field `experience` can either be undefined (`null`) or an integer.

Avro Record Schemas

- Avro Record Schemas are defined using JSON
- It has some common fields:
  - Name: Name of your schema
  - Namespace: (equivalent of package in java)
  - Doc: Documentation to explain your schema
  - Aliases: Optional other names for your schema
  - Fields
    - Name: Name of your field

- Fields
  - Name: Name of your field
  - Doc: Documentation for that field
  - Type: Data type for that field (can be a primitive type)
  - Default: Default Value for that field

default дефолтные и null значения

Default and Null Values

Default Value

```
{
  "name": "suit_type",
  "type" : {
    "type" : "enum",
    "name" : "Suit",
    "symbols" : ["SPADES", "HEARTS",
                 "DIAMONDS", "CLUBS"],
    "default": "SPADES"
  },
}
```

Null Value

```
{
  "name" : "experience",
  "type": ["int", "null"]
}
```

- The field `suit_type`, if missing in the source data, will be assigned the **default** value of `SPADES`
- The value of the field `experience` can be either undefined (`null`) of of type `int`

перечисление

Enums

- These are for fields you know for sure that their values can be enumerated.

- Example: Customer status
  - Bronze
  - Silver
  - Gold

{ "type": "enum", "name": “CustomerStatus", "symbols" : [“BRONZE", “SILVER", “GOLD"] }

- Note: once an enum is set, changing the enum values is forbidden if you want to maintain compatibility.

```
{
  "type": "record",
  "name": "Test",
  "namespace": "com.acme",
  "fields": [
    {
      "name": "name",
      "type": "string"
    },
    {
      "name": "suit_type",
      "type" : {
        "type" : "enum",
        "name" : "Suit",
        "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
      },
      "doc" : "The suit of the card"
},
```

To allow for `null` values define the type of the enum like this:

```
"type": [
  "null",
  {
    "type": "enum",
    "name": "Suit",
    "symbols": ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
  }
]
```

Note that we define the value of `type` as an array including `null`.

массив

Arrays

- Arrays are a way for you to represent a list of undefined size of items that all share the same schema.

- Example: Customer Emails (multiple emails)
- ["john.doe@gmail.com","jon92@hotmail.com"]

```
{"type": "array", "items": "string"}
```

- Note: the schema can be anything you want so you can use any existing schemas for it

Array

```
{
  "namespace": "example.avro",
  "name": "Parent",
  "type":"record",
  "fields":[
    {
      "name":"children",
      "type":{
        "type": "array",
        "items":{
          "name":"Child",
          "type":"record",
          "fields":[
            {
              "name":"name",
              "type":"string"
            }
          ]
        }
      }
    }
  ]
}
```

мапа

Maps

- Maps are a way to define a list of keys and values, where the keys are strings.

- Example: secrets questions
  - "What's your favourite colour?": "green"
  - "Where were you born?": "Paris"
  - "Name of first bet?": "Mr Snuggles"

{"type": "map", "values": "string"}

- Note: don't store secrets in Avro. This is just to illustrate the concepts of maps

Map

```
{
  "namespace": "example.avro",
  "type": "record",
  "name": "Log",
  "fields": [
    {"name": "ip", "type": "string"} ,
    {"name": "timestamp", "type": "string"},
    {"name": "message", "type": "string"},
    {
      "name": "additional",
      "type": {
        "type": "map",
        "values": "string"
       }
     }
  ]
}
```

Note that the `map` type is a complex type and could handle such things as optional fields in data structured like this:

```
{ "ip": "172.18.80.109", "timestamp": "2015-09-17T23:00:18.313Z", "message":"blahblahblah"}
{ "ip": "172.18.80.112", "timestamp": "2015-09-17T23:00:08.297Z", "message":"blahblahblah",
"microseconds": 223}
{ "ip": "172.18.80.113", "timestamp": "2015-09-17T23:00:08.299Z", "message":"blahblahblah",
"thread":"http-apr-8080-exec-1147"}
```

where now the fields `microseconds` and `thread` are not mentioned explicitly...

Unions

- Unions can allow a field value to take different types.
- Example: [“string”, “int”, “boolean”]

- If defaults are defined, the default must be of the type of the first item in the union (so in this case, “string”).
- The most common use case for unions is to define an optional value.

```
{"name": “middle_name“, "type": [“null”, “string”], “default”: null]
```

- Note: for the default don’t write “null”, write null

```
{
  "type": "record",
  "namespace": "com.example",
  "name": "Customer",
  "fields": [
    { "name": "first_name", "type": "string" },
    { "name": "middle_name", "type": ["null", "string"], "default": null },
    { "name": "last_name", "type": "string" },
    { "name": "age", "type": "int" },
    { "name": "height", "type": "int" },
    { "name": "weight", "type": "int" },
    { "name": "automated_email", "type": "boolean", "default": true },
    { "name": "customer_emails", "type": "array", "items": "string", "default": [] },
    { "name": "customer_address", "type": "CustomerAddress" }
  ]
}]
```

my custom avro type

```
[
  "type": "record",
  "namespace": "com.example",
  "name": "CustomerAddress",
  "fields": [
    { "name": "address", "type": "string" },
    { "name": "city", "type": "string" },
    { "name": "postcode", "type": ["int", "string"] },
    { "name": "type", "type": "enum", "symbols": ["PO BOX", "RESIDENTIAL", "ENTERPRISE"]
  ]
},
```

логический тип - это как бы уточнение основного типа (те говорит о том как

интерпретировать основной тип данных)

Logical Data Types

Decimal

```
{
  "type": "bytes",
  "logicalType": "decimal",
  "precision": 4,
  "scale": 2
}
```

Date

```
{
  "type": "int",
  "logicalType": "date"
}
```

Timestamp (ms precision)

```
{
  "type": "long",
  "logicalType": "timestamp-millis"
}
```

- The `decimal` logical type represents an arbitrary-precision signed `decimal` number of the form `unscaled × 10^(-scale)`
  - `scale`, a JSON integer representing the scale (optional). If not specified the scale is 0.
  - `precision`, a JSON integer representing the (maximum) precision of decimals stored in this type
- A `date` logical type annotates an Avro `int`, where the `int` stores the number of days from the unix epoch, 1 January 1970 (ISO calendar).
- A `timestamp-millis` logical type annotates an Avro `long`, where the long stores the number of milliseconds from the unix epoch, **1 January 1970 00:00:00.000 UTC**.

Logical Types

- Avro has a concept of logical types used to give more meaning to already existing primitive types.
- The most commonly used are:
- decimals (bytes – see next lecture)
- date (int) – number of days since unix epoch (Jan 1st 1970)
- time-millis (long) - number of milliseconds after midnight, 00:00:00.000.
- timestamp-millis (long) - the number of milliseconds from the unix epoch, 1 January 1970 00:00:00.000 UTC

- To use a logical type, just add "logicalType":"time-millis" to the field name and it will help avro schema processors to inter a specific type.

- Example: Customer Signup timestamp:

```
{"name": "signup_ts", "type": "long", "logicalType": "timestamp-millis"}
```

- Note: logical types are new (1.7.7), not fully supported by all languages and don't play nicely with unions. Be careful when using them!

Fixed

- Feld mit einer fixen Anzahl von Bytes
- Beispiel: Übertragen von Binärdaten

```
{"type":"fixed", "name": "BData", "size":1024}
```

3) PROTOBUF

17 декабря 2020 г. 18:48

confluent schema registry поддерживат не только AVRO но также и PROTOBUF, JSON_schema

https://www.confluent.io/blog/confluent-platform-now-supports-protobuf-json-schema-custom-formats/

Для сериализации данных широко используют Avro — это основанный на строках, то есть строковый, формат хранения данных в Hadoop. Он хранит схему в формате JSON, облегчая ее чтение и интерпретацию любой программой. Сами данные лежат в двоичном формате, компактно и эффективно.

… evolution of data

5 декабря 2020 г. 19:47

- First CSV:

rownum	column1	column2	column3	column4	column5	column6
row1	John	Doe	25	John.doe	true	OK
row2	Mary	Poppins	sixty	Mary.pop	yes	OK
row3	Tom	Cruise	45	Tom.Cru		

- Advantages:
  - Easy to parse
  - Easy to read
  - Easy to make sense of
- Disadvantages:
  - The data types of elements has to be inferred and is not a guarantee
  - Parsing becomes tricky when data contains commas
  - Column names may or may not be there

- Relational table definitions add types:

```
CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

- Advantages:
  - Data is fully typed
  - Data fits in a table
- Disadvantages:
  - Data has to be flat
  - Data is stored in a database, and data definition will be different for each database

- JSON format can be shared across the network!

```
{
        "id": "0001",
        "type": "donut",
        "name": "Cake",
        "image":
                {
                        "url": "images/0001.jpg",
                        "width": 200,
                        "height": 200
                },
        "thumbnail":
                {
                        "url": "images/thumbnails/0001.jpg",
                        "width": 32,
                        "height": 32
                }
}
```

  - Advantages:
    - Data can take any form (arrays, nested elements)
    - JSON is a widely accepted format on the web
    - JSON can be read by pretty much any language
    - JSON can be easily shared over a network
  - Disadvantages:
    - Data has no schema enforcing
    - JSON Objects can be quite big in size because of repeated keys

- Avro is defined by a schema (schema is written in JSON)

- To get started, you can view Avro as JSON with a schema attached to it

```
{
    "type": "record",
    "name": "userInfo",
    "namespace": "my.example",
    "fields": [{
            "name": "username",
            "type": "string",
            "default": "NONE"
        },
        {
            "name": "age",
            "type": "int",
            "default": -1
        },
        {
            "name": "address",
            "type": {
                "type": "record",
                "name": "mailing_address",
                "fields": [{
                        "name": "street",
                        "type": "string",
                        "default": "NONE"
                    },
                    {
                        "name": "city",
                        "type": "string",
                        "default": "NONE"
                    }
                ]
            },
            "default": {}
        }
    ]
}
```

  - Advantages:
    - Data is fully typed
    - Data is compressed automatically (less CPU usage)
    - Schema (defined using JSON) comes along with the data
    - Documentation is embedded in the schema
    - Data can be read across any language
    - Schema can evolve over time, in a safe manner (schema evolution)
  - Disadvantages:
    - Avro support for some languages may be lacking (but the main ones is fine)
    - Can't "print" the data without using the avro tools (because it's compressed and serialised)

сначала схему или сначала код

17 декабря 2020 г. 18:55

top-down подход является более предпочтительным, те когда сначала есть схема (и потом уже по ней может генериться код)

Avro Schemas

- Avro schemas define the **structure** of your data
- Schemas are represented in **JSON** or **IDL** format
- Avro has three different ways of creating records:

Generic	1. Manually create data type 2. Manually create schema
Reflection	1. Manually create data type 2. Generate schema from code
Specific	1. Manually write schema 2. Generate code to include in your program *Most common way to use Avro classes*

For Avro schemas to be useful in your environment, you must have data types in your programming language of choice and the schema to describe it in a JSON format for compatibility reasons (more on that later).

Schemas can also be defined in an IDL-based format. In fact, many customers use that option instead of JSON. https://avro.apache.org/docs/1.8.2/idl.html

IDL, short for **Interactive Data Language**, is a programming language used for data analysis.

What is the difference between the three ways of creating records?

- Generic: Manually create both the data type (Java class) and the schema (*.avsc file)
- Reflection: Manually create the data type and then generate a schema from that code
- Specific: Manually write the schema and then generate the code to include in your program

floats vs decimal

6 декабря 2020 г. 12:41

floats для математиков а decimal для денег

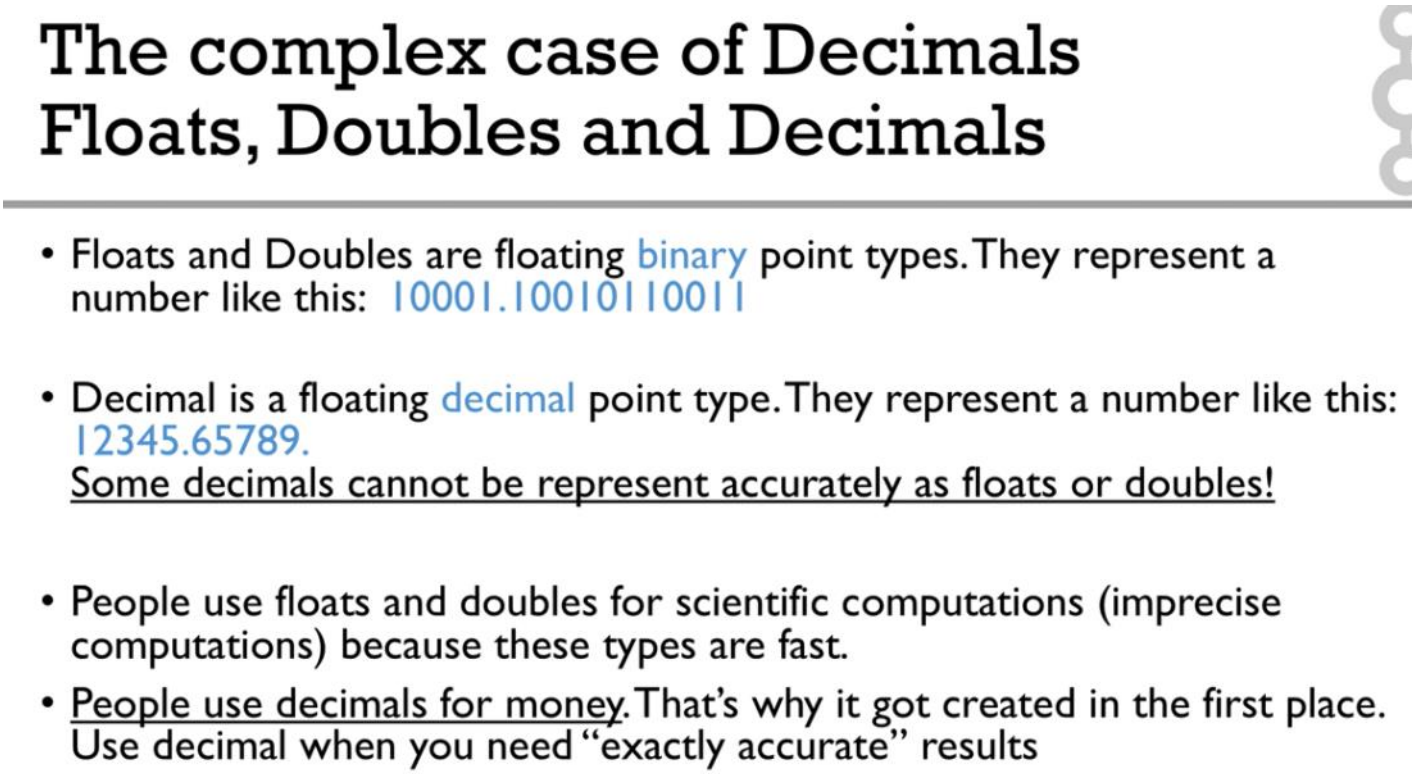


схема <--> объект

6 декабря 2020 г. 12:46

Generic Record

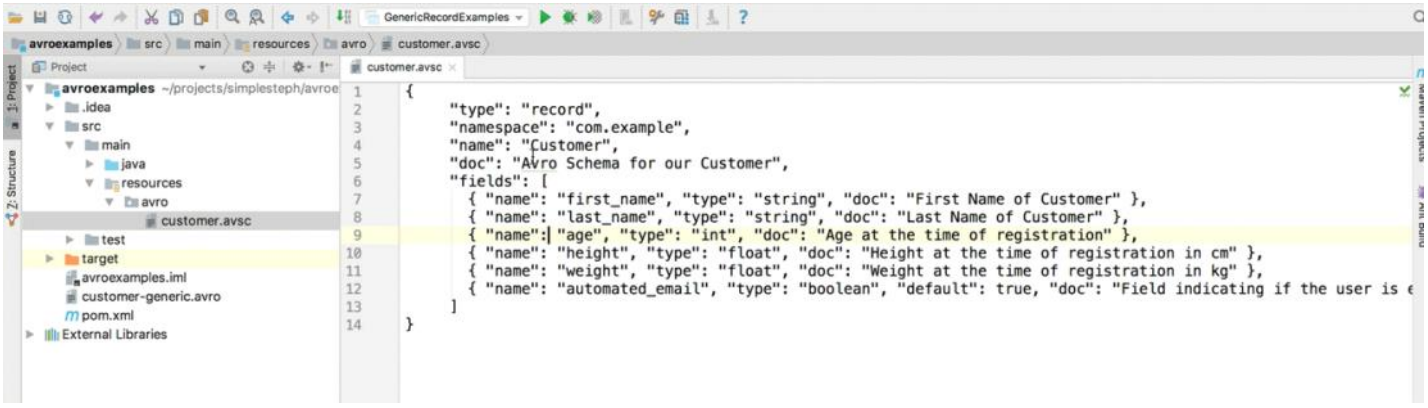
- A GenericRecord is used to create an avro object from a schema, the schema being referenced as:
  - A file
  - A string
- It's not the most recommended way of creating Avro objects because things can fail at runtime, but it is the most simple way.
- We'll learn to:
  - Create a Generic Record
  - Write it to a file
  - Read it from a file
  - Read the Generic Record

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

    // step 0: define schema
    Schema.Parser parser = new Schema.Parser();
    Schema schema = parser.parse( s: "{\n" +
            "     \"type\": \"record\",\n" +
            "     \"namespace\": \"com.example\",\n" +
            "     \"name\": \"Customer\",\n" +
            "     \"doc\": \"Avro Schema for our Customer\",     \n" +
            "     \"fields\": [\n" +
            "       { \"name\": \"first_name\", \"type\": \"string\", \"doc\": \"First Name of Customer\" },\n" +
            "       { \"name\": \"last_name\", \"type\": \"string\", \"doc\": \"Last Name of Customer\" },\n" +
            "       { \"name\": \"age\", \"type\": \"int\", \"doc\": \"Age at the time of registration\" },\n" +
            "       { \"name\": \"height\", \"type\": \"float\", \"doc\": \"Height at the time of registration in cm\" },\n" +
            "       { \"name\": \"weight\", \"type\": \"float\", \"doc\": \"Weight at the time of registration in kg\" },\n" +
            "       { \"name\": \"automated_email\", \"type\": \"boolean\", \"default\": true, \"doc\": \"Field indicating if t
            "     ]\n" +
            "}");

    // step 1: create a generic record
    GenericRecordBuilder customerBuilder = new GenericRecordBuilder(schema);
    customerBuilder.set("first_name", "John");
    customerBuilder.set("last_name", "Doe");
    customerBuilder.set("age", 25);
    customerBuilder.set("height", 170f);
    customerBuilder.set("weight", 80.5f);
    customerBuilder.set("automated_email", false);
    GenericData.Record customer = customerBuilder.build();

    System.out.println(customer);
```

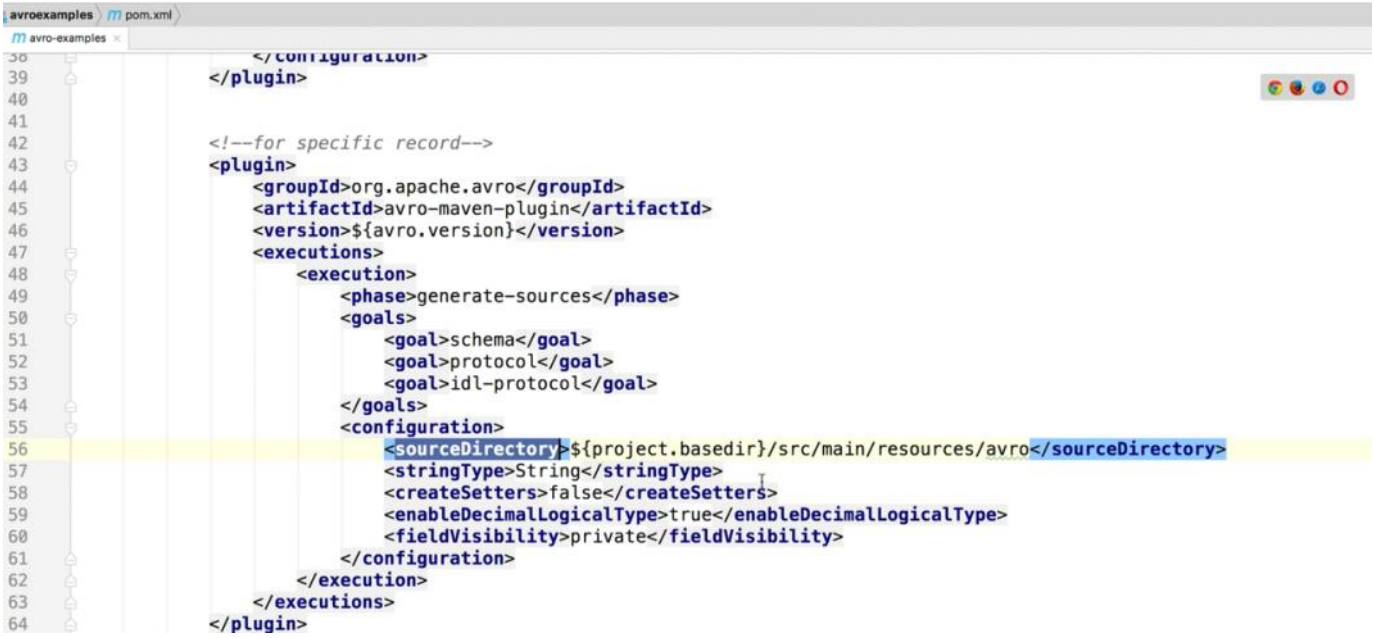
Specific Record

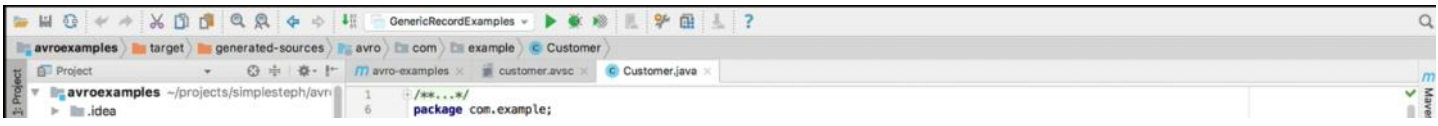
- A SpecificRecord is also an Avro object, but it is obtained using code generation from an Avro schema
- There are different plugins for different build tools (gradle, maven, sbt) etc, but in our example, we'll use the official code generation tool shipping with Avro: Maven

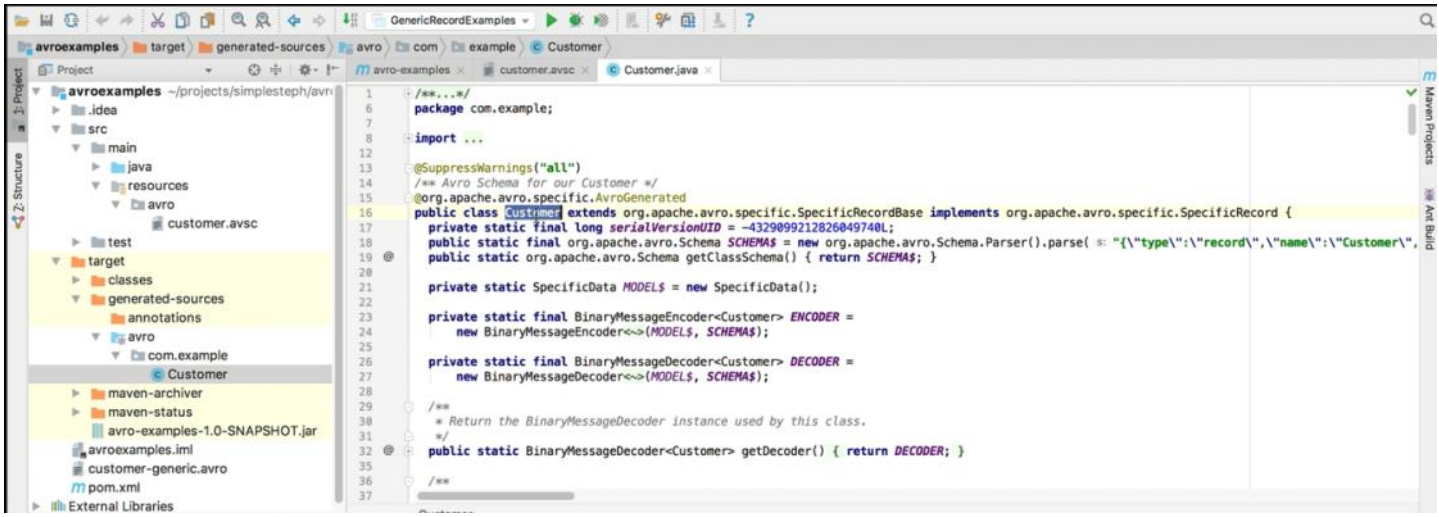
- We'll perform the exact same tasks as the previous lecture, but all using a SpecificRecord now

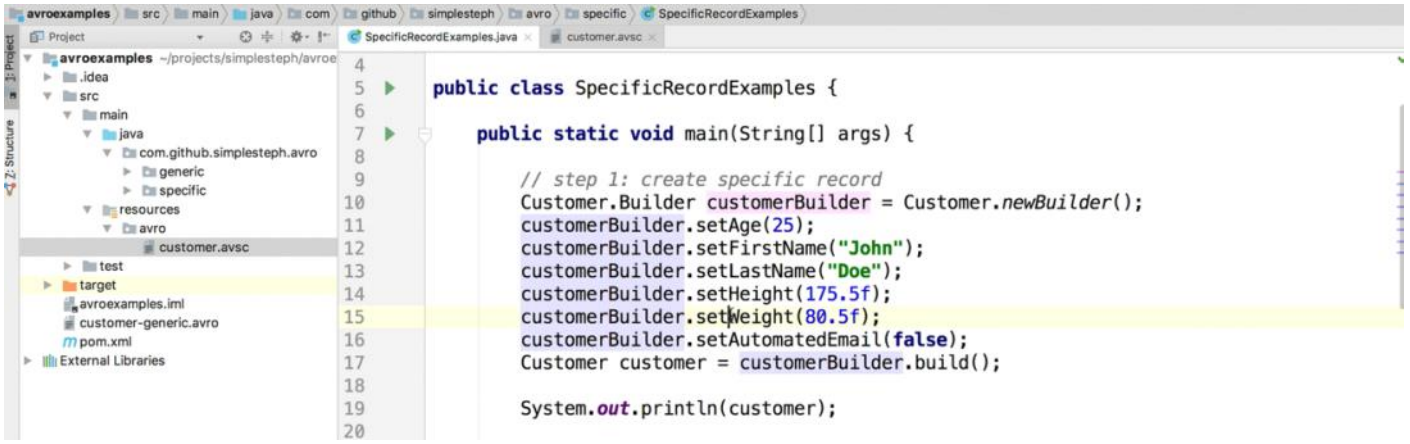


на основе схемы avro классы сгенерятся автоматически









Using the Avro Tools

- It is possible to read avro files using the avro tools commands
- These are very handy when we want to display (print) data to our command line for a quick analysis of a content of an Avro file
- Let's learn how to use the Avro tools right now!

```
# put this in any directory you like
wget http://central.maven.org/maven2/org/apache/avro/avro-tools/1.8.2/avro-tools-1.8.2.jar
# run this from our project folder. Make sure ~/avro-tools-1.8.2.jar is your actual avro tools location
java -jar ~/avro-tools-1.8.2.jar tojson --pretty customer-generic.avro
java -jar ~/avro-tools-1.8.2.jar tojson --pretty customer-specific.avro
# getting the schema
java -jar ~/avro-tools-1.8.2.jar getschema customer-specific.avro
```

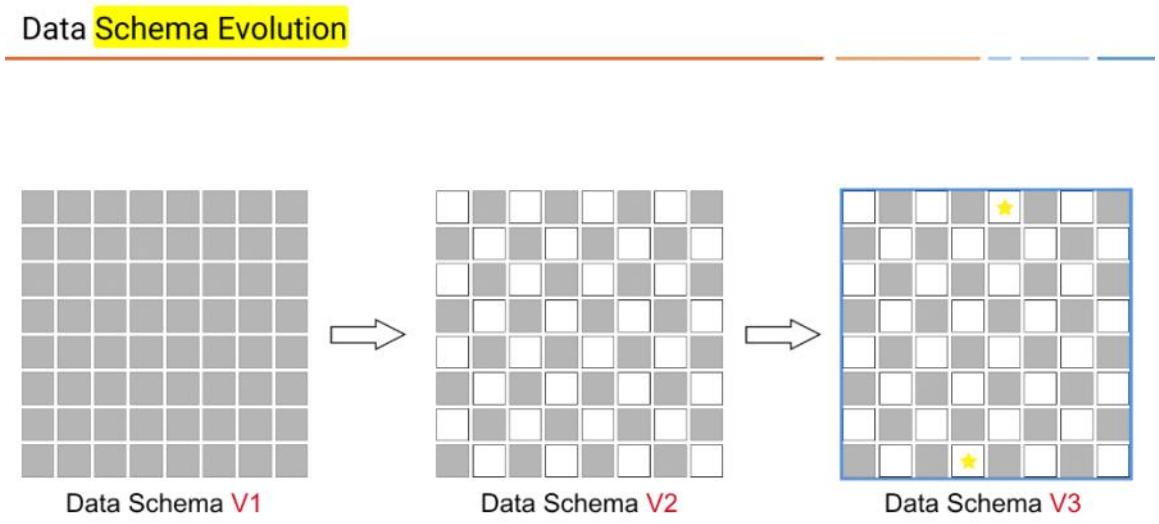
Avro Reflection

- You can use Reflection in order to build Avro schemas from your class
- This is a less common scenario but still a valid one! It's useful when you want to add some classes to your Avro objects.


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

    // here we use reflection to determine the schema
    Schema schema = ReflectData.get().getSchema(ReflectedCustomer.class);
    System.out.println("schema = " + schema.toString( pretty: true));
```

schema evolution

6 декабря 2020 г. 13:53



The image on the slide conceptually describes what goes on in "real-life". A structure (or schema) is conceived (v1). Over time new (business) requirements come into play and the structure/schema has to be adapted to the situation (v2). But that might not be the end, further refinements are required, resulting in further modifications (v3).

As business changes or more applications want to leverage the same data, there is a need for the existing data structures to evolve. We need what's called a **schema evolution**.

In asynchronous event-driven architecture, the data format is the API, and having support for schema evolution is as important for preserving compatibility as API compatibility is in RPC/request-response architecture. With event-first design, the data becomes the API which, like any production system, needs to support change and evolution (i.e., Avro or Protobuf)

backward - означает, что если в клиент_v2 пришло сообщение_v1 "**из прошлого**", то подставится дефолтное значение (которое должно быть в соотвествии с v2)
forward - означает что если в клиент_v1 пришло сообщение_v2 "**из будущего**", то клиент проигнорирует новые поля (появившиеся только в v2)

Schema Evolution

- Avro schemas may evolve as updates to code happen
  - Schema Registry allows schema evolution
- We often want compatibility between schemas

Compatibility	Description
Backward	Code with a **new version** of the schema: • can read data written in the **old** schema • assumes **default values** if fields are not provided
Forward	Code with **previous versions** of the schema: • can read data written in a **new** schema • **ignores** new fields
Full	Forward and Backward

- **Backward compatibility example:** schema is written with fields (A, B); on the other side, schema is expecting to read fields (A, B, C); code reads (A, B) and assumes default value for C
- **Forward compatibility example:** schema is written with fields (A, B, C); on the other side, schema is expecting to read fields (A, B); code reads (A, B) and ignores C
- **Forward** & **Backward**: Only check compatibility against **previous** schema version
- **Forward transient:** Check compatibility against **all** previous schema versions
- **Backward transient:** Check compatibility against **all** previous schema versions

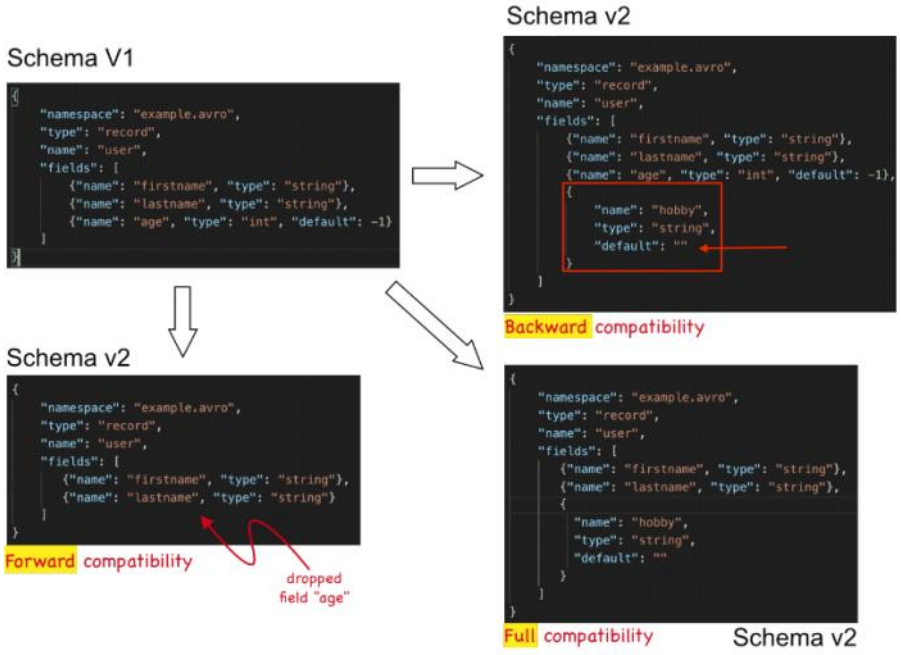
ТИПЫ ЭВОЛЮЦИИ

Table 6.1 The different ways that an application's events can evolve

Level	Change	Backward compatible
Schema	Define a new aggregate type	Yes
Remove aggregate	Remove an existing aggregate	No
Rename aggregate	Change the name of an aggregate type	No
Aggregate	Add a new event type	Yes
Remove event	Remove an event type	No
Rename event	Change the name of an event type	No
Event	Add a new field	Yes
Delete field	Delete a field	No
Rename field	Rename a field	No
Change type of field	Change the type of a field	No

реестр схем автоматически будет осуществлять проверки схем
рахных версий (и можно явно задать режим)

Compatibility Examples



The various possible compatibility modes for schema evolution are defined as follows:

- BACKWARD: (default) consumers using the new schema can read data written by producers using the latest registered schema
- BACKWARD_TRANSITIVE: consumers using the new schema can read data written by producers using all previously registered schemas
- FORWARD: consumers using the latest registered schema can read data written by producers using the new schema
- FORWARD_TRANSITIVE: consumers using all previously registered schemas can read data written by producers using the new schema
- FULL: the new schema is forward and backward compatible with the latest registered schema
- FULL_TRANSITIVE: the new schema is forward and backward compatible with all previously registered schemas
- NONE: schema compatibility checks are disabled

BACKWARD compartibility - рекомендуется придерживаться по дефолту

We recommend keeping the default BACKWARD compatibility.

The supported schema compatibility settings mentioned here are more an SR concept, not settings within Avro itself!

Schema Evolution High Level

- There are 4 kinds of schema evolution:
  1. Backward: a backward compatible change is when a **new** schema can be used to read **old** data
  2. Forward: a forward compatible change is when an **old** schema can be used to read **new** data
  3. Full: which is both **backward** and **forward**
  4. Breaking: which is none of those

1. Backward Compatible

- Backward: a backward compatible change is when a **new** schema can be used to read **old** data

```
{"namespace": "com.github.simplesteph",
  "type": "record", "name": "Customer",
    "version" :  "1",
    "fields": [
        {"name": "firstName", "type": "string"},
        {"name": "lastName", "type": "string"}
    ]
}
```

```
{"namespace": "com.github.simplesteph",
  "type": "record", "name": "Customer",
    "version" : "2",
    "fields": [
        {"name": "firstName", "type": "string"},
        {"name": "lastName", "type": "string"},
        {"name": "phoneNumber", "type": "string", "default":
"000-0000-000"}
    ]
}
```

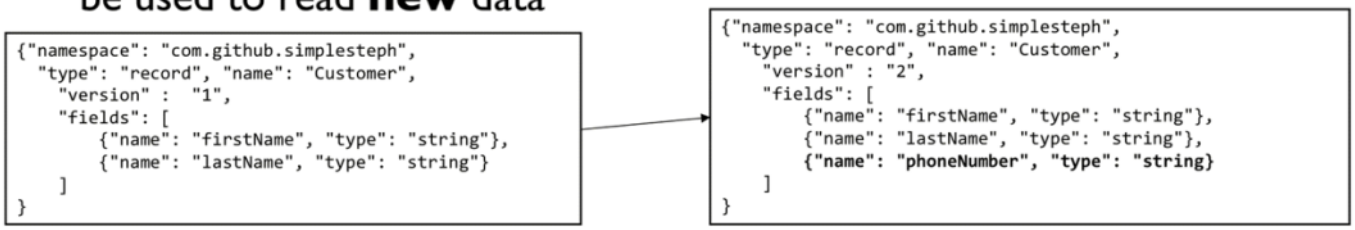
- We can read old data with the new schema, thanks to a **default** value. In case the field doesn't exist, Avro will use the **default**
- We want backwards when we want to successfully perform queries (Hive-SQL for example) over old and new data using a new schema.

2. Forward compatible

2. Forward: a forward compatible change is when an **old** schema can

2. Forward compatible

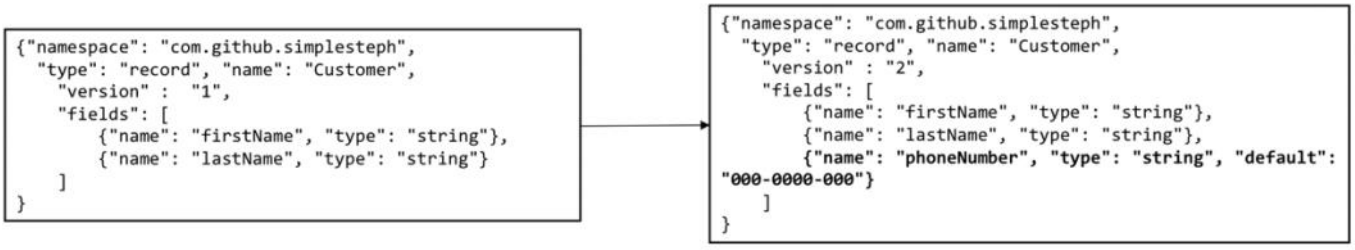
2. Forward: a forward compatible change is when an **old** schema can be used to read **new** data



- We can read new data with the old schema, Avro will just **ignore** new fields. Deleting fields without defaults is not forward compatible
- We want forward compatible when we want to make a data stream evolve without changing our downstream consumers

3. Fully Compatible

3. Full: which is both **backward** and **forward**



- Only add fields with defaults
- Only remove fields that have defaults
- When writing your schema changes, most of the time you want to target full compatibility *(and it's not too hard, is it?)*

Schema Evolution
4. Not compatible

Here are examples of changes that are NOT compatible:

- Adding / Removing elements from an Enum
- Changing the type of a field (string => int for example)
- Renaming a required field (field without default)

Schema Evolution
4. Not compatible

Here are examples of changes that are NOT compatible:

- Adding / Removing elements from an Enum
- Changing the type of a field (string => int for example)
- Renaming a required field (field without default)

Don't do that.

Don't

Advice when writing an Avro schema

1. Make your primary key required
2. Give default values to all the fields that could be removed in the future
3. Be very careful when using Enums as they can't evolve over time.
4. Don't rename fields. You can add aliases instead (other names)
5. When evolving a schema, ALWAYS give default values
6. When evolving a schema, NEVER delete a required field

Summary on Compatibility changes

- Two patterns
- Write a forward compatible change (very common)
  - => update your producer to V2, you won't break your consumers
  - => take your time to update your consumers to V2
- Write a backward compatible change (less common)
  - => update all consumers to V2, you will still be able to read v1 producer data
  - => when all are updated, update producer to V2

не удаляйте поля которые имеют значения по умолчанию
не меняйте тип поля (например с маассива на мапу)

Inkompatibilität

- Folgende Änderungen führen zu inkompatiblen Schemas
  - Hinzufügen / entfernen von Enum - Elementen
  - Ändern des Typs eines Feldes (Array → Map)
  - Umbenennen eines Feldes (ohne Default oder Alias)

avro in python

5 декабря 2020 г. 20:43

https://habr.com/ru/post/346698/
https://habr.com/ru/post/530422/
https://habr.com/ru/post/492312/
https://habr.com/ru/company/mailru/blog/504952/

Изначально схема данных в кафке описывалась с помощью авро. В настоящее время добавлена поддержка протобафа.

Integration with Schema Registry

- Java clients (producer, consumer)
- KSQL
- Kafka Streams
- Kafka Connect
- Confluent REST Proxy
- Non-Java clients based on `librdkafka` (except Go)

- **Java** based Kafka clients directly integrate with SR, that is it supports automated schema registration and lookup
- **KSQL** supports topics whose record values are encoded in AVRO out of the box
- **Kafka Streams** applications are written in Java and thus support the SR
- **Kafka Connect** directly integrates with SR via simple configuration settings
- **REST Proxy** also integrates with SR via simple configuration settings
- `librdkafka`: Currently Confluent supports direct access to SR for Python, .NET and C/C++. ~~Go is not yet offering direct SR support.~~

… avro in java

6 декабря 2020 г. 15:37

пример продьюсера

Java Avro Producer example

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
// Configure serializer classes
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
          KafkaAvroSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
          KafkaAvroSerializer.class);
// Configure schema repository server
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
          "http://schemaregistry1:8081");
// Create the producer expecting Avro objects
KafkaProducer<Object, Object> avroProducer = new KafkaProducer<Object, Object>(props);
// Create the Avro objects for the key and value
CardSuit suit = new CardSuit("spades");
SimpleCard card = new SimpleCard("spades", "ace");
// Create the ProducerRecord with the Avro objects and send them
ProducerRecord<Object, Object> record = new
ProducerRecord<Object, Object>("my_avro_topic", suit, card);
avroProducer.send(record);
```

Why is ProducerRecord typed <Object,Object> on this slide if consumer.poll() returns <CardSuit,SimpleCard> on the next slide?

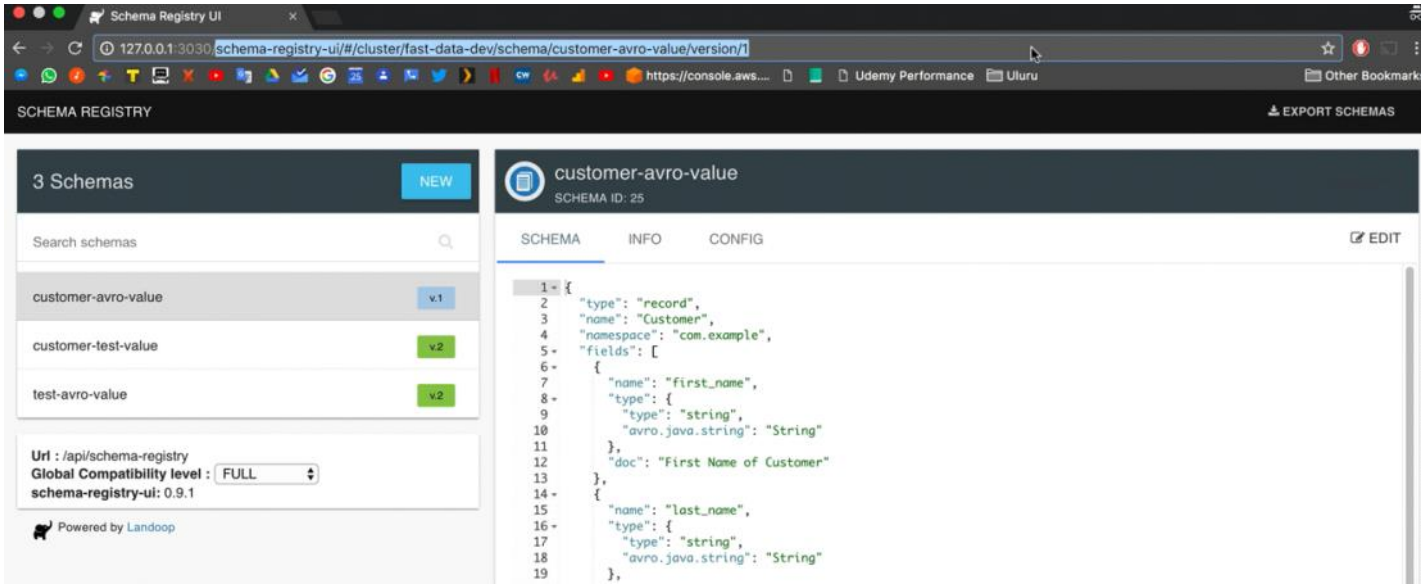
This illustrates that the Producer can send different types of Avro objects to different topics without having to instantiate different Producers for each topic and type of object. However on the Consumer side in this example, the Consumer is subscribing to a single topic, which is only going to have one type of Avro object, so it can be more specific in the typing. You can set the types to be more specific in the Producer, but it has to be done consistently throughout - in the KafkaProducer declaration and in the ProducerRecord declaration.

```
public static void main(String[] args) {
    Properties properties = new Properties();
    properties.setProperty("bootstrap.servers", "127.0.0.1:9092");
    properties.setProperty("acks", "1");
    properties.setProperty("retries", "10");

    properties.setProperty("key.serializer", StringSerializer.class.getName());
    properties.setProperty("value.serializer", KafkaAvroSerializer.class.getName());
    properties.setProperty("schema.registry.url", "http://127.0.0.1:8081");

    KafkaProducer<String, Customer> kafkaProducer = new KafkaProducer<String, Customer>(properties);
    String topic = "customer-avro";

    Customer customer = Customer.newBuilder()
            .setFirstName("John")
            .setLastName("Doe")
            .setAge(26)
            .setHeight(185.5f)
            .setWeight(85.6f)
            .setAutomatedEmail(false)
            .build() ;

    ProducerRecord<String, Customer> producerRecord = new ProducerRecord<String, Customer>(
            topic, customer
    );

    kafkaProducer.send(producerRecord, new Callback() {
```

автосгенерили класс из avro схемы

заполнили класс

при первой отправке сообщения схема сама автоматически появится в реестре схем



пример консьюмера

Java Avro Consumer Example

```
1  public class CardConsumer {
2    public static void main(String[] args) {
3      Properties props = new Properties();
4      props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
5      props.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
6      props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
7                KafkaAvroDeserializer.class);
8      props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
9                KafkaAvroDeserializer.class);
10     props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
11               "http://schemaregistry1:8081");
12     props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
13
14     KafkaConsumer<CardSuit, SimpleCard> consumer = new KafkaConsumer<>(props);
15     consumer.subscribe(Arrays.asList("my_avro_topic"));
16
17     while (true) {
18       ConsumerRecords<CardSuit, SimpleCard> records = consumer.poll(Duration.ofMillis(100));
19         for (ConsumerRecord<CardSuit, SimpleCard> record : records) {
20           System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(),
21                             record.key().getSuit(), record.value().getCard());
22
23         }
24     }
25   }
26 }
```

Note that in this code and the producer code on the previous page, there do not appear to be any methods which communicate directly with the Schema Registry. The interaction with the Schema Registry is abstracted into the `KafkaAvroSerializer` and `KafkaAvroDeserializer`.

Line 12 is required for the preferred behavior of "specific" code generation. If this line is omitted, the default of "generic" is used instead.

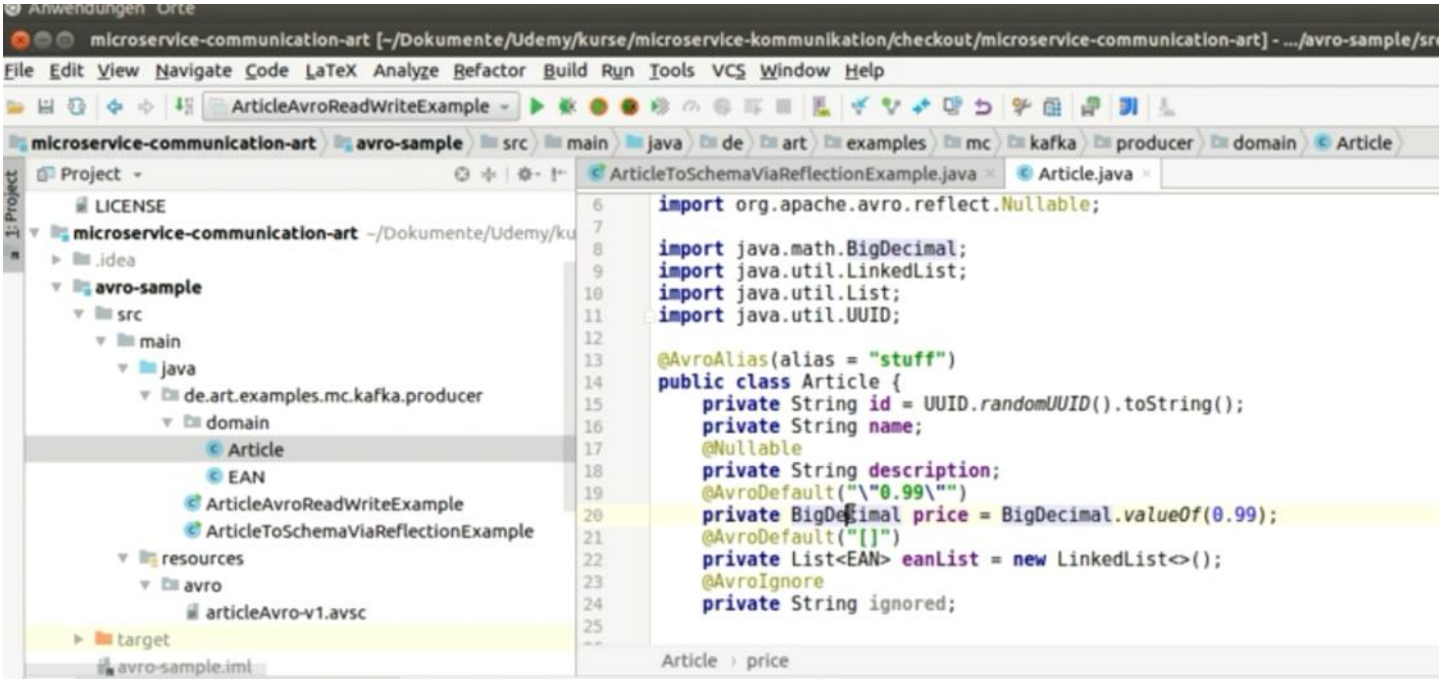
the `CardSuit` and `SimpleCard` types shown in these examples could possibly be generated from different schema versions, with compatibility enforced performed by the serializer on the Producer side before data is sent to Kafka (and on to Consumers).

```
public static void main(String[] args) {
    Properties properties = new Properties();
    properties.setProperty("bootstrap.servers", "127.0.0.1:9092");
    properties.setProperty("group.id", "my-avro-consumer");
    properties.setProperty("enable.auto.commit", "false");
    properties.setProperty("auto.offset.reset", "earliest");

    properties.setProperty("key.deserializer", StringDeserializer.class.getName());
    properties.setProperty("value.deserializer", KafkaAvroDeserializer.class.getName());
    properties.setProperty("schema.registry.url", "http://127.0.0.1:8081");
    properties.setProperty("specific.avro.reader", "true");


    KafkaConsumer<String, Customer> consumer = new KafkaConsumer<String, Customer>(proper
    String topic = "customer-avro";

    consumer.subscribe(Collections.singleton(topic));

    System.out.println("Waiting for data...");

    while(true){
        ConsumerRecords<String,Customer> records = consumer.poll( timeout: 500);
```

… avro in java (annotated)

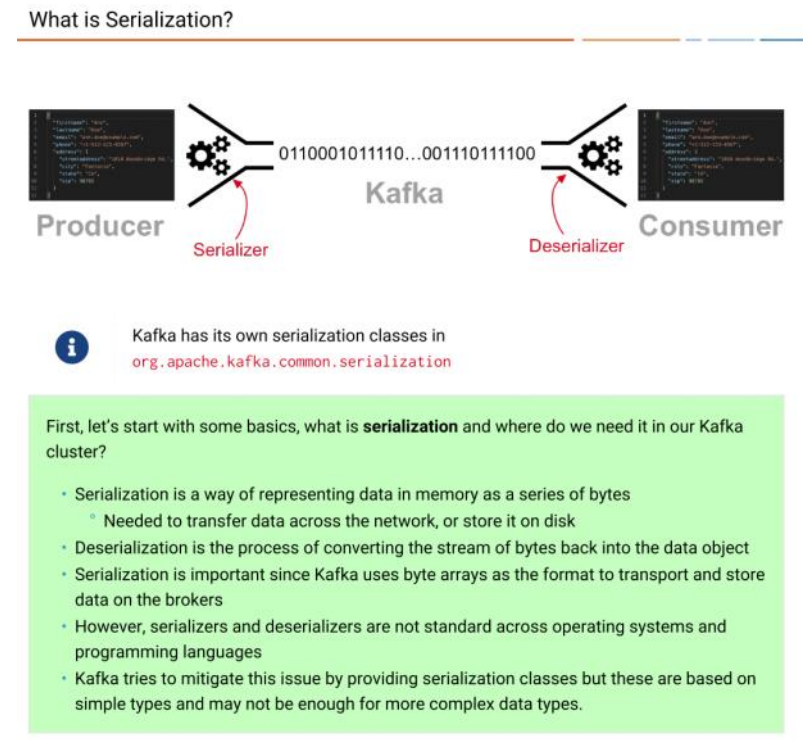
7 декабря 2020 г. 21:43

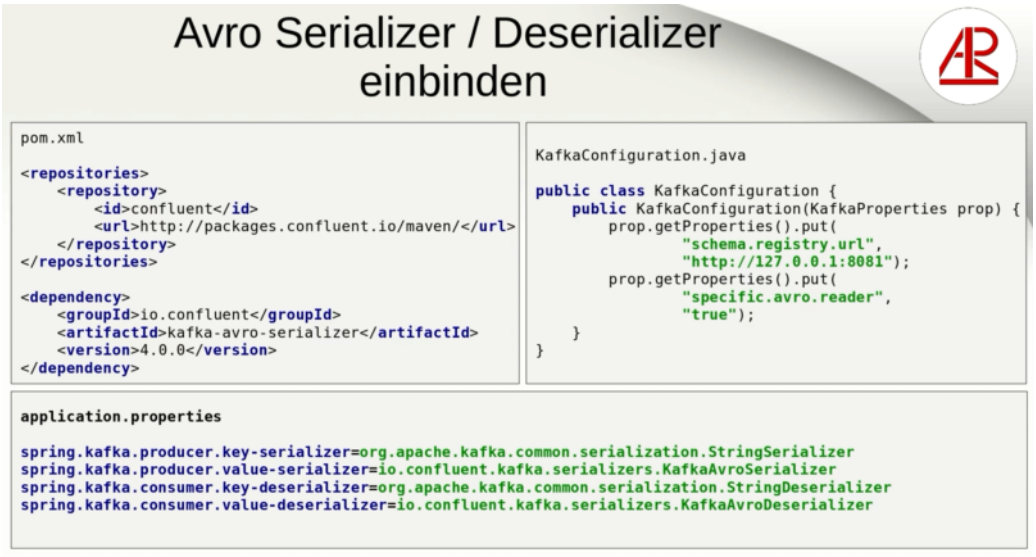
разметка аннотациями как всегда проще

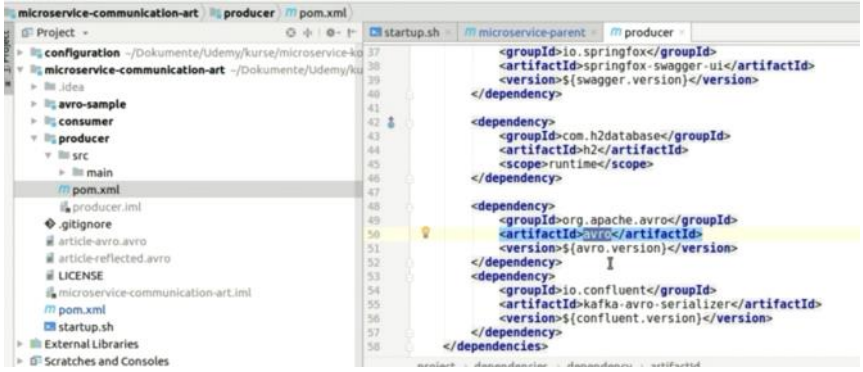


… spring serializer/deserializer

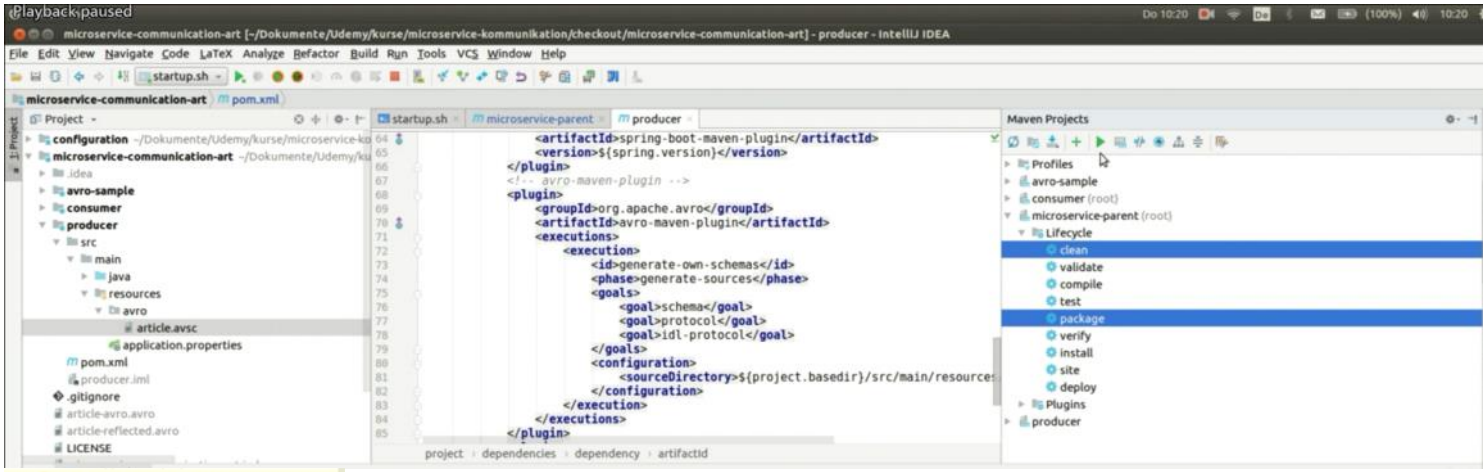
7 декабря 2020 г. 21:51

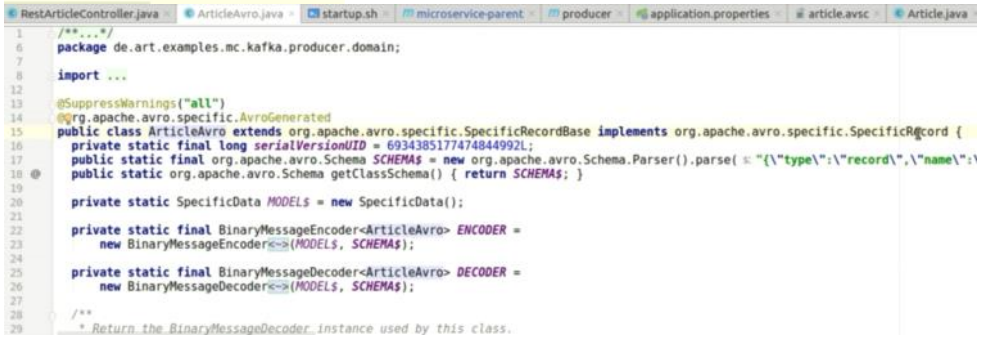


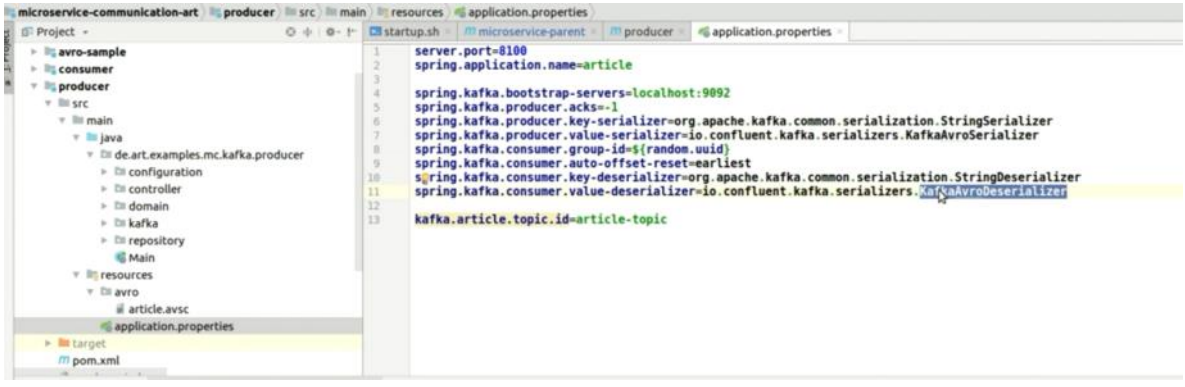


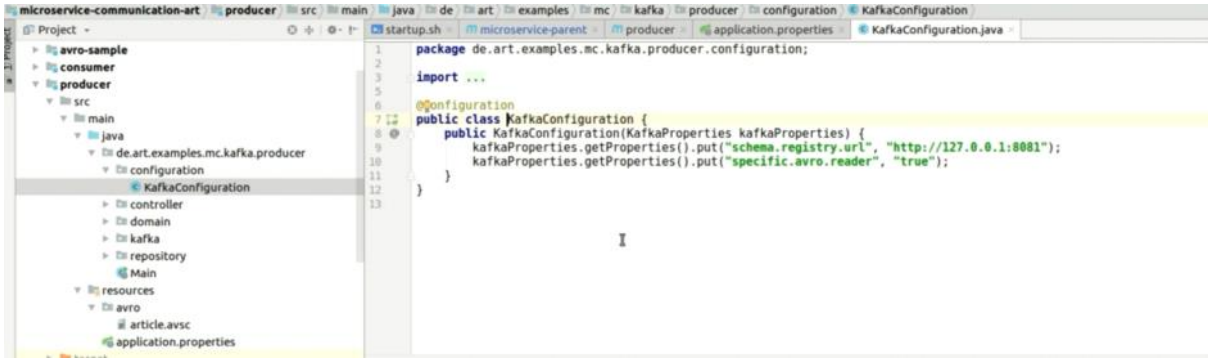


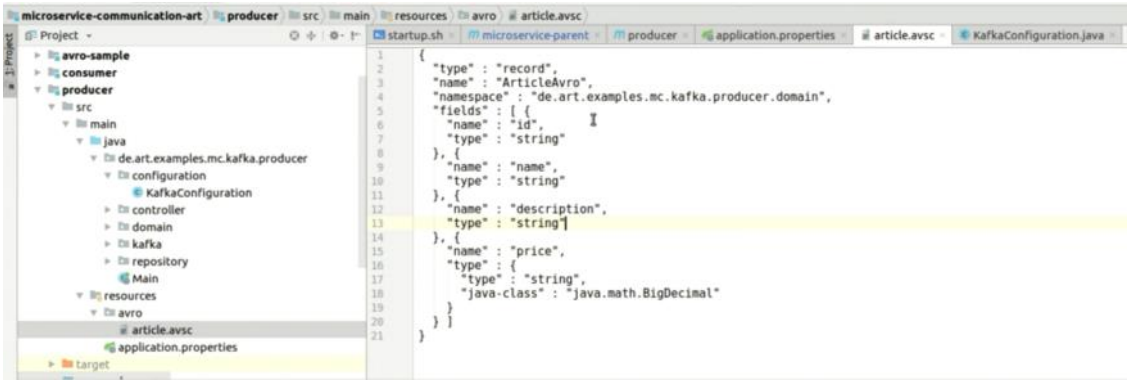
автосгенерим джава-класс из схемы

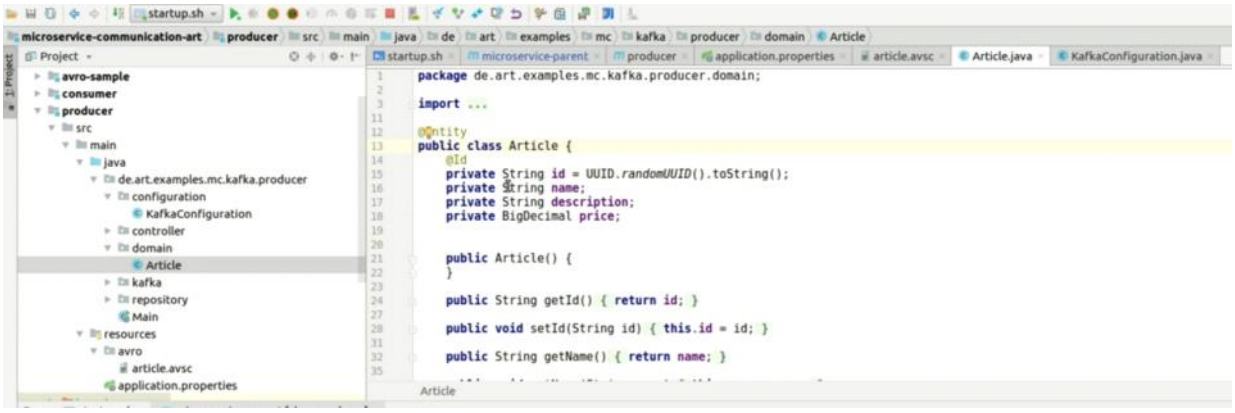

Playback paused
microservice-communication-art [~/Dokumente/Udemy/kurse/microservice-kommunikation/checkout/microservice-communication-art] - producer - IntelliJ IDEA
File Edit View Navigate Code LaTeX Analyze Refactor Build Run Tools VCS Window Help
Project
configuration ~/Dokumente/Udemy/kurse/microservice-ku
microservice-communication-art ~/Dokumente/Udemy/ku
.idea
avro-sample
consumer
producer
src
main
java
resources
avro
article.avsc
application.properties
pom.xml
producer.iml
.gitignore
article-avro.avro
article-reflected.avro
LICENSE
startup.sh
microservice-parent
producer
<artifactId>spring-boot-maven-plugin</artifactId>
<version>${spring.version}</version>
</plugin>
<!-- avro-maven-plugin -->
<plugin>
<groupId>org.apache.avro</groupId>
<artifactId>avro-maven-plugin</artifactId>
<executions>
<execution>
<id>generate-own-schemas</id>
<phase>generate-sources</phase>
<goals>
<goal>schema</goal>
<goal>protocol</goal>
<goal>idl-protocol</goal>
</goals>
<configuration>
<sourceDirectory>${project.basedir}/src/main/resources
</configuration>
</execution>
</executions>
</plugin>
project dependencies dependency artifactId
Maven Projects
Profiles
avro-sample
consumer (root)
microservice-parent (root)
Lifecycle
clean
validate
compile
test
package
verify
install
site
deploy
Plugins
producer
target
classes
generated-sources
annotations
avro
de.art.examples.mc.kafka.producer.domain
ArticleAvro
maven-archiver


RestArticleController.java
ArticleAvro.java
startup.sh
microservice-parent
producer
application.properties
article.avsc
Article.java
/**...*/
package de.art.examples.mc.kafka.producer.domain;
import ...
@SuppressWarnings("all")
@org.apache.avro.specific.AvroGenerated
public class ArticleAvro extends org.apache.avro.specific.SpecificRecordBase implements org.apache.avro.specific.SpecificRecord {
private static final long serialVersionUID = 6934385177474844992L;
public static final org.apache.avro.Schema SCHEMA$ = new org.apache.avro.Schema.Parser().parse("{\"type\":\"record\",\"name\":
public static org.apache.avro.Schema getClassSchema() { return SCHEMA$; }
private static SpecificData MODEL$ = new SpecificData();
private static final BinaryMessageEncoder<ArticleAvro> ENCODER =
new BinaryMessageEncoder<>(MODEL$, SCHEMA$);
private static final BinaryMessageDecoder<ArticleAvro> DECODER =
new BinaryMessageDecoder<>(MODEL$, SCHEMA$);
/**
* Return the BinaryMessageDecoder instance used by this class.


microservice-communication-art producer src main resources application.properties
Project
avro-sample
consumer
producer
src
main
java
de.art.examples.mc.kafka.producer
configuration
controller
domain
kafka
repository
Main
resources
avro
article.avsc
application.properties
target
pom.xml
startup.sh
microservice-parent
producer
application.properties
server.port=8100
spring.application.name=article
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.producer.acks=-1
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=io.confluent.kafka.serializers.KafkaAvroSerializer
spring.kafka.consumer.group-id=${random.uuid}
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=io.confluent.kafka.serializers.KafkaAvroDeserializer
kafka.article.topic.id=article-topic


microservice-communication-art producer src main java de art examples mc kafka producer configuration KafkaConfiguration
Project
avro-sample
consumer
producer
src
main
java
de.art.examples.mc.kafka.producer
configuration
KafkaConfiguration
controller
domain
kafka
repository
Main
resources
avro
article.avsc
application.properties
startup.sh
microservice-parent
producer
application.properties
KafkaConfiguration.java
package de.art.examples.mc.kafka.producer.configuration;
import ...
@Configuration
public class KafkaConfiguration {
public KafkaConfiguration(KafkaProperties kafkaProperties) {
kafkaProperties.getProperties().put("schema.registry.url", "http://127.0.0.1:8081");
kafkaProperties.getProperties().put("specific.avro.reader", "true");
}
}


microservice-communication-art
producer
src
main
resources
avro
article.avsc
Project
avro-sample
consumer
producer
src
main
java
de.art.examples.mc.kafka.producer
configuration
KafkaConfiguration
controller
domain
kafka
repository
Main
resources
avro
article.avsc
application.properties
target
startup.sh
microservice-parent
producer
application.properties
article.avsc
KafkaConfiguration.java
{
"type" : "record",
"name" : "ArticleAvro",
"namespace" : "de.art.examples.mc.kafka.producer.domain",
"fields" : [{
"name" : "id",
"type" : "string"
}, {
"name" : "name",
"type" : "string"
}, {
"name" : "description",
"type" : "string"
}, {
"name" : "price",
"type" : {
"type" : "string",
"java-class" : "java.math.BigDecimal"
}
}]
}


startup.sh
microservice-communication-art
producer
src
main
java
de
art
examples
mc
kafka
producer
domain
Article
Project
avro-sample
consumer
producer
src
main
java
de.art.examples.mc.kafka.producer
configuration
KafkaConfiguration
controller
domain
Article
kafka
repository
Main
resources
avro
article.avsc
application.properties
startup.sh
microservice-parent
producer
application.properties
article.avsc
Article.java
KafkaConfiguration.java
package de.art.examples.mc.kafka.producer.domain;
import ...
@Entity
public class Article {
@Id
private String id = UUID.randomUUID().toString();
private String name;
private String description;
private BigDecimal price;
public Article() {
}
public String getId() { return id; }
public void setId(String id) { this.id = id; }
public String getName() { return name; }
Article

confluent community license

6 декабря 2020 г. 19:28

https://habr.com/ru/company/1cloud/blog/472964/

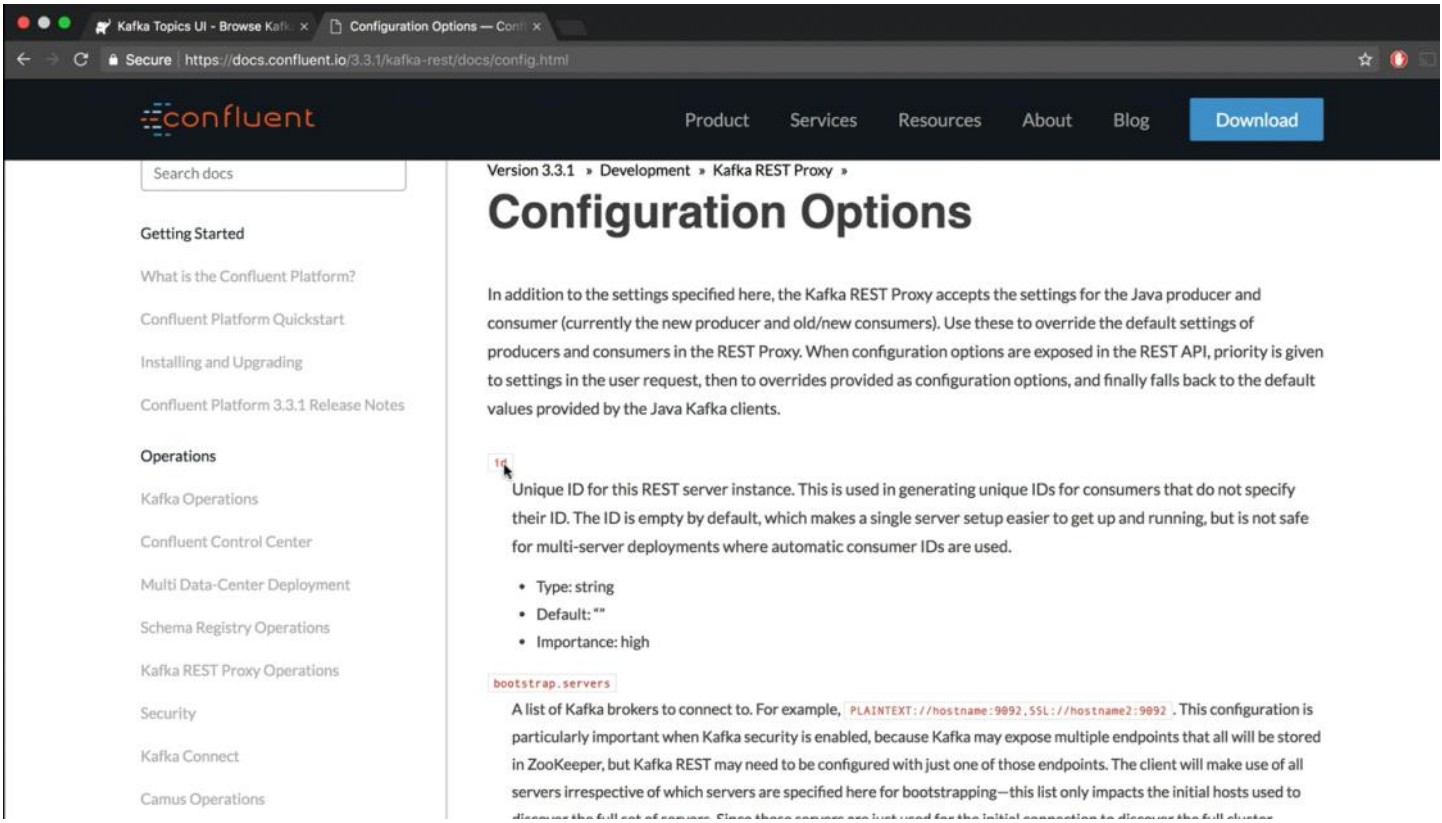
… как поставить rest proxy

6 декабря 2020 г. 20:40

REST Proxy Installation and Scaling

- To install the REST Proxy in production go through the documentation at: https://docs.confluent.io/current/kafka-rest/docs/config.html

- You can scale the REST Proxy by having multiple of them behind a load balancer for example



что делать при нарушениях совместимости

22 декабря 2020 г. 0:17

если требуется сделать новую версию , которая будет абсолютно несовместима с предыдущей то лучше для нее сделать абсолютно новый топик (и натравить на него новую версию микросервисов)

https://www.confluent.io/blog/apache-kafka-for-service-architectures/

How to Break Backwards Compatibility

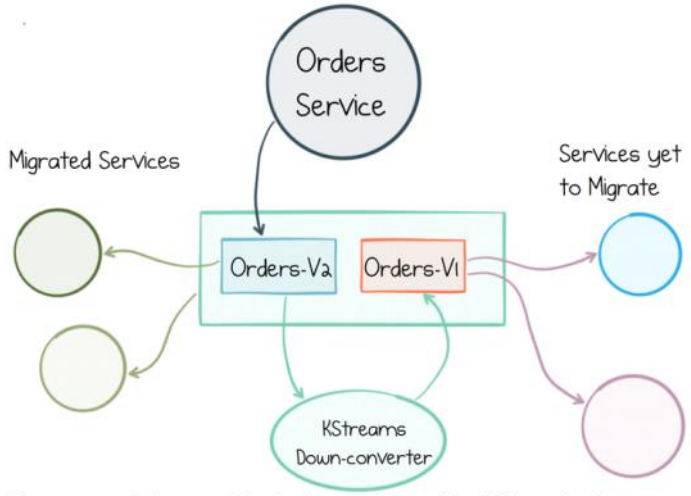
Most of the time, when we're evolving and releasing our services, we maintain backwards compatibility between schemas. This keeps releases simple. Periodically however message formats will need to be changed in incompatible ways, most commonly when a whole schema needs to be reworked.

Say we want to change the way an Order is modelled. Rather than being a single element, Cancels, Returns and Reorders are all to be modelled in the same schema. This kind of change would likely break backwards compatibility because the structure of the whole schema has to be changed.

The most common approach for handling breaking changes like these is to create two topics: orders-v1 and orders-v2, for messages with the old and new schemas respectively. Assuming Orders are mastered by the Orders Service, this gives us a couple of options:

1. The Orders service can dual-publish in both schemas at the same time.
2. We can add a process that down-converts from orders-v2 topic to orders-v1 topic. A simple KStreams job is typically used to do this kind of down-conversion.

Both approaches achieve the same goal: to give services a window in which they can upgrade, but the KStreams method has the advantage that it's also easy to port any historic messages from the v1 to the v2 topics.



The same data coexists in two topics, with different schemas, so there is a window during which services can upgrade.

Services continue in this dual-topic mode until all services have fully migrated to the v2 topic, at which point the v1 topic can be archived or deleted as appropriate.

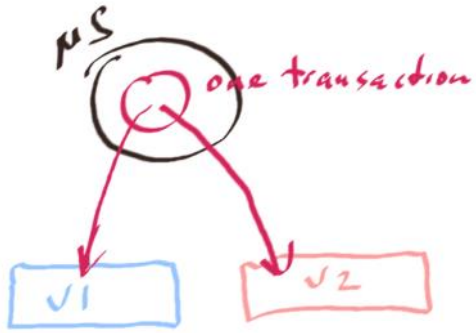
As an aside, in the last post we discussed applying the single writer principle. One of the reasons for doing this is it makes schema upgrades simpler. If we had three different services writing Orders it would be much harder to schedule a non-backward compatible upgrade without a conjoined release.

общее для четырех подходов ниже

- Все четыре подхода достигают одной и той же цели: предоставить сервисам окно, в котором они могут обновляться.
- Службы продолжают работать в этом dual-topic режиме до тех пор, пока не будут полностью перенесены в тему v2, после чего тема v1 может быть заархивирована или удалена при необходимости.
- Кроме того, мы обсудили принцип единственного писателя в главе 11 . Одна из причин применения этого подхода состоит в том, что он упрощает обновление схемы. Если бы у нас было три разных заказа на написание сервисов, было бы намного сложнее запланировать обновление без обратной совместимости без объединенного выпуска.

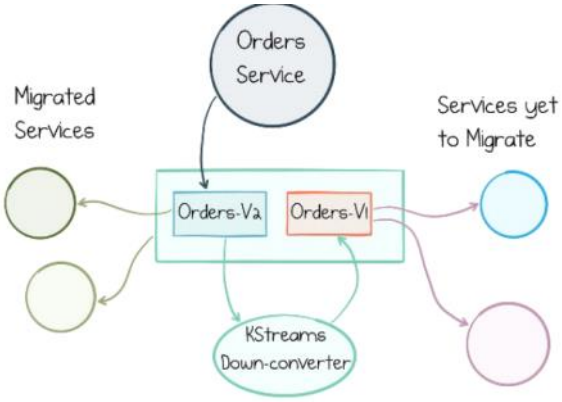
(1) микросервис может публиковать в v1 тему и в v2(новую чистую тему) одновременно

- The orders service can dual-publish in both schemas at the same time, to two topics, using Kafka's transactions API to make the publication atomic. (Thisapproach doesn't solve back-population so isn't appropriate for topics used for long-term storage.)



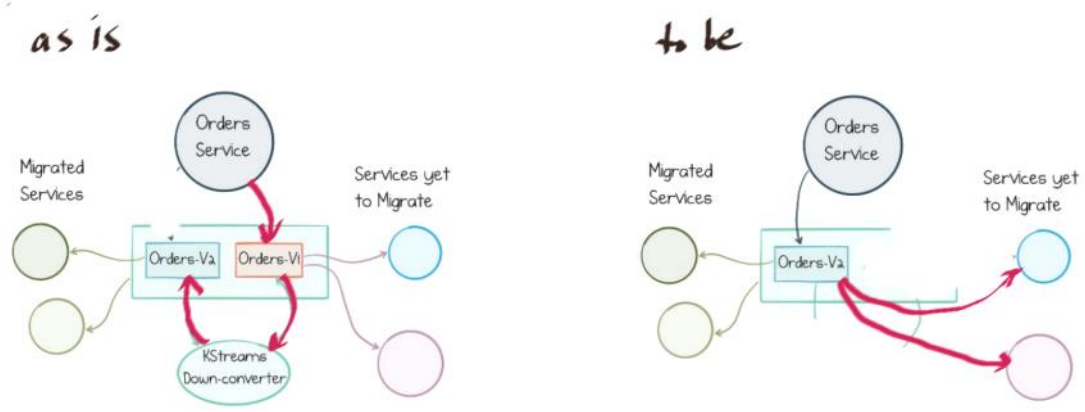
(2) микросервис использует v2, добавлен конвертер v2->v1 для отсталых клиентов

- The orders service can be repointed to write to orders-v2. A Kafka Streams job is added to down-convert from the orders-v2 topic to the orders-v1 for backward compatibility. (This also doesn't solve back-population.) See Figure 13-2.



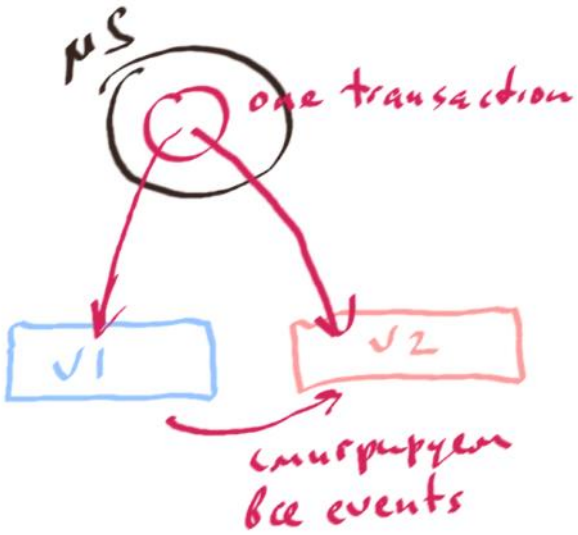
(3) микросервис использует v1, добавлен конвертер v1->v2 для перехода клиентов на новую версию

- The orders service continues to write to orders-v1. A Kafka Streams job is added that up-converts from orders-v1 topic to orders-v2 topic until all clients have upgraded, at which point the orders service is repointed to ordersv2. (This approach handles back-population.)



(4) миграция всех событий темы v1 в тему v2

The orders service can migrate its dataset internally, in its own database, then republish the whole view into the log in the orders-v2 topic. It then continues to write to both orders-v1 and orders-v2 using the appropriate formats. (This approach handles back-population.)



согласовывать между различными microservice teams переход на новую версию топика через gitlab

- Лучший подход, который я видел для этого, - использовать GitHub. Это хорошо работает, потому что (а) схемы являются кодом и должны контролироваться версиями по тем же причинам, что и код, и (б) GitHub позволяет разработчикам предлагать изменения и поднимать запрос на вытягивание (PR), против которого они могут кодировать, пока они построить и протестировать их систему. разное заинтересованные стороны могут просматривать, комментировать и утверждать. После достижения консенсуса PR может быть объединен, и новая схема может быть развернута. Именно этот процесс для надежного достижения (и аудита) консенсуса по изменению, не препятствуя чрезмерно прогрессу разработчика, делает этот подход наиболее полезным вариантом.

dead letter queue - когда консьюмер не может прочитать поврежденные сообщения, мы их помещаем в отдельную очередь ошибочних сообщений

- Traditional messaging systems often include a related concept called a dead letter queue,which is used to hold messages thatcan’t be sent, for example, because they cannot be routed to a destination queue. The concept doesn’t apply in the same way to Kafka, but it is possible for consumersto not be able to read a message, either for semantic reasons or due to the message payload being invalid (e.g., the CRC check failing, on read, as the message has become corrupted).
- Некоторые разработчики предпочитают создавать собственный тип очереди недоставленных сообщений в отдельной теме. Если потребитель не может прочитать сообщение по какой-либо причине, оно помещается в эту очередь ошибок, чтобы можно было продолжить обработку. Позже очередь ошибок может быть повторно обработана .