Consumer

(Consumer will deserialize the data after consuming)

Producer

(he will always send serialized data to broker)

Broker

(Broker will store only the serialized data only)

Schema registry

(Schema will be stored here)

**Serialization is nothing but process of converting object data to a byte array (so that data can be travelled through a network)**

**So Apache avro is a serialization framework to serialize and deserialize the data**

References

|  |  |
| --- | --- |
| **Apache Avro**  **(even jay creeps also suggesting to use apache avro as serialization framework)**  **They also used this avro in linkedin and later they outsourced this framework to apache** | [**https://www.confluent.io/blog/avro-kafka-data/**](https://www.confluent.io/blog/avro-kafka-data/) |
|  | <https://avro.apache.org/docs/1.11.1/specification/#arrays>  [**https://avro.apache.org/docs/1.11.1/getting-started-java/**](https://avro.apache.org/docs/1.11.1/getting-started-java/) |
| **Avro producer consumer code of Udemy trainer dilip** | [**https://github.com/dilipsundarraj1/kafka-for-developers-using-schema-registry**](https://github.com/dilipsundarraj1/kafka-for-developers-using-schema-registry) |
| **Avro specific data types** | [**https://avro.apache.org/docs/1.11.1/specification/#logical-types**](https://avro.apache.org/docs/1.11.1/specification/#logical-types) |
| **Confluent schemas** | [**https://www.confluent.io/blog/schemas-contracts-compatibility/?\_ga=2.205203091.2005966880.1650190514-1685861233.1648224453&\_gac=1.154929994.1648739558.CjwKCAjwopWSBhB6EiwAjxmqDdc7q5-nqdT-Zx3DI64gxdYDjq4-Ile0txJr4rgDFYY4HAytwrpGZRoCNgUQAvD\_BwE**](https://www.confluent.io/blog/schemas-contracts-compatibility/?_ga=2.205203091.2005966880.1650190514-1685861233.1648224453&_gac=1.154929994.1648739558.CjwKCAjwopWSBhB6EiwAjxmqDdc7q5-nqdT-Zx3DI64gxdYDjq4-Ile0txJr4rgDFYY4HAytwrpGZRoCNgUQAvD_BwE) |
| **Schema registry - URLs** | [**https://docs.confluent.io/platform/current/schema-registry/develop/api.html#schemas**](https://docs.confluent.io/platform/current/schema-registry/develop/api.html#schemas) |
| **Schema evolution** | [**https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html**](https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html) |
| **Backward compatibility**  [**https://github.com/dilipsundarraj1/kafka-for-developers-using-schema-registry/tree/final?tab=readme-ov-file#backward-compatibility**](https://github.com/dilipsundarraj1/kafka-for-developers-using-schema-registry/tree/final?tab=readme-ov-file#backward-compatibility) | [**https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html#backward-compatibility**](https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html#backward-compatibility) |
|  |  |

Serialization frameworks & advantages

**There are 2** types **of serialization frameworks**

**Note:- we can send byte or Avro data to topic without having the schema registry (if schema registry is not there how can it send data as it should validate the data always with schema which is present in schema registry)**

|  |  |
| --- | --- |
| **Binary Serialization** | **Plaintext Serialization** |
| **This serializes the data to byte array** | **This serializes data to encoded text** |
| **Not Human readable** | **Human readable**  **(so far In local I practiced with normal plain text serialization that’s why I was even able to see the original data in the log file )** |
| **Final payload size will be small 🡪 (when compared to json avro will be half size lesser)**  **This is more efficient because the data is compact and less memory overhead (this is a process of reducing size of data -**  **This will remove the schema information from the payload only data will be there in payload 🡪hence final payload size will be less && data will always be checked with schema before inserting into topic, it can publish only when data is in line with schema &&**  **final message size will be small it can easily travel over network)**  Using schema id to fetch schema from schema registry  **But in avro we are sending only the data in the payload without schema, but consumer when he consumes the data he needs the schema to recreate the original message, that’s why avro will send the schema id along with the payload, consumer will go and search with the id (which was send along with the message) In the schema registry and fetch the schema with that he will recreate the message**  **Caution:- we should use the exact same schema while sending and receiving the data**  **Note:- best way is always send schema id + actual data (without the schema) as schema already stored In schema registry** | **Data is verbose and it can be inefficient**  **Where as other serialization frameworks include the schema also in the payload which** |
| • Serialization in generally faster | Serialization in slower compared to Binary Serialization |
| AVRO • ProtocolBuf • Thrift  So here AVRO is a | JSON • XML |
| Define a Schema for the data structure and make sure the data is always aligned to that schema.  So that producers can send only the data which is aligned with schema if data is not aligned with schema he cant push that data into topic |  |
| We can also communicate avro records without schema registry also |  |



Schemas & avro Data types

**If we are writing avro schema, we should write in .avsc file, so that avro plugins can understand this .avsc files and generate equivalent java classes?**

|  |  |
| --- | --- |
| **AVRO has the support for rich data structures**  **• Primitive Types**  **• String, bytes, int ,long, float, double, boolean and null**  **• Complex Types**  **• enum**  **• arrays**  **• maps**  **• record (we use this type only mostly in project , this is a complex data type like employee class object which can hold multiple data types)**  **• union**  **• fixed**  **- This type is normally used to hold multiple complex types.**  **- This is used to represent a field can hold multiple types. [String , null]**  **- This is used to represent a field can be of a fixed size, specifying the number of by** | **Ex:-**   * *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 |
| **Schema** | **Equivalent .avsc file** |
| **{**  **“empName”:”manideep”**  **}** | **{**  **“namespace”:”com.tcs.avro” 🡪 means equivalent java classes will be generated in this folder**  **“name”: “anyJavaClassName”, 🡪 Here we should give the name of the desired class**  **“type” : “record”,**  **“fields”: [**  **{“name”:”empName”, “type”:”string” }**  **]**  **}**  **Total schema file** |

Avro specific data types (logical types)

|  |  |
| --- | --- |
| **All samples are here**  **Instant/Timestamp (u can mention timestamp in both milli & micro seconds)**  **If u want to declare a java type as instant then u have to select logical type as “timestamp-millis”**  {  **"name"**: "dob",  **"type"**: {  **"type"**: "long",  **"logicalType"**: "timestamp-millis"  } } | Equivalent java types  public Employee(java.lang.Integer *id*, java.lang.*CharSequence empName*, java.time.Instant *dob*, java.nio.*ByteBuffer salary*) {  this.**id** = *id*;  this.**empName** = *empName*;  this.**dob** = *dob*.truncatedTo(java.time.temporal.*ChronoUnit*.***MILLIS***);  this.**salary** = *salary*; } |
| **Double (we already have a double but if u want to declare a data type with some precision then choose this) instead of this we can use float, double default data types**  {  **"name"**: "salary",  **"type"**: {  **"type"**: "bytes",  **"logicalType"**: "decimal",  **"precision"**: 4,  **"scale"**: 2  } } |  |
| **LocalDate**  java.time.LocalDate **dateofjoin**; | {  **"name"**: "dateofjoin",  **"type"**: {  **"type"**: "int",  **"logicalType"**: "date"   } } |

Maven code

We should add all below plugins and run the build “mvn clean install ”

|  |  |
| --- | --- |
| Add Avro plugin 🡪 to generate the java files from .avsc file  It will take the .avsc files from this source directory  It will keep the generated java files in the output directory | <plugin>  <groupId>org.apache.avro</groupId> <artifactId>avro-maven-plugin</artifactId> <version>1.9.2</version>  <configuration>  <sourceDirectory>${project.basedir}/src/main/avro/</sourceDirectory>  <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>  *<!-- These imports are very important when avsc files is being referenced in another file then we have to import those definitions else builds will be failed-->*  <imports>  <import>${project.basedir}/src/main/avro/Address.avsc</import>  <import>${project.basedir}/src/main/avro/OrderId.avsc</import>  <import>${project.basedir}/src/main/avro/OrderLineItem.avsc</import>  <import>${project.basedir}/src/main/avro/Store.avsc</import>  </imports>  <enableDecimalLogicalType>true</enableDecimalLogicalType>  <customConversions>org.apache.avro.Conversions$UUIDConversion</customConversions>  </configuration>  </plugin>  For general purpose we need compiler plugin also  <plugin>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.8.0</version> </plugin> |
| **AVRO serializer – so this is mandatory**  The generated avro java class will contain the annotations/classes of below jar  <dependency>  <groupId>org.apache.avro</groupId>  <artifactId>avro</artifactId>  <version>1.10.1</version> </dependency>  To use avro we need this main jar | |  |  | | --- | --- | | Serializer is present only in this jar  io.confluent.kafka.serializers.KafkaAvroSerializer is present in below jar  <dependency>  <groupId>io.confluent</groupId>  <artifactId>kafka-avro-serializer</artifactId>  <version>5.5.1</version> </dependency> | <repositories>  <repository>  <id>confluent</id>  <name>Confluent</name>  <url>https://packages.confluent.io/maven/</url>  </repository> </repositories>  And that serializer jar is present only in this confluent repository, so I have added it | |
|  |  |
|  |  |

In Intellij we should add a plugin called “**Apache avro IDL schema support**” 🡪 while writing .avsc file, because of this plugin u will get the auto complete feature to write those avsc files

Serializer producer-consumer code

Note: - we can happily send avro messages/ byte data **without schema registry** also

And even thou

|  |  |
| --- | --- |
| Byte[] serializer – producer code  We should use byte array serializer producer code  Properties p=new Properties(); String topicName="santu"; p.put(ProducerConfig.***CLIENT\_ID\_CONFIG***,"SDP"); p.put(ProducerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***,"localhost:9092"); p.put(ProducerConfig.***KEY\_SERIALIZER\_CLASS\_CONFIG***, StringSerializer.class.getName());  //This is going to serialize the data into byte[] p.put(ProducerConfig.***VALUE\_SERIALIZER\_CLASS\_CONFIG***, ByteArraySerializer.class.getName()); p.put(ProducerConfig.***REQUEST\_TIMEOUT\_MS\_CONFIG***,5000); p.put(ProducerConfig.***RETRIES\_CONFIG***,0);   KafkaProducer<String,byte[]> producer=new KafkaProducer<String, byte[]>(p);  byte[] byteData = *getByteData*(); ProducerRecord<String,byte[]> producerRecord=new ProducerRecord<String,byte[]>(topicName, byteData); System.***out***.println("going to send record to topic "+topicName); RecordMetadata metadata = producer.send(producerRecord).get(); System.***out***.println("record has been sent successfully to partition "+ metadata.partition()+" offset -->"+metadata.offset());  // Here we are using avro to generate the byte[]  private static byte[] getByteData() {  Greeting g=Greeting.newBuilder().setGreeting("santoor kumari").build();  byte[] byteData = new byte[0];  try {  byteData = g.toByteBuffer().array();  } catch (IOException e) {  log.error("Exception occurred while converting data into bytes array",e);  throw new RuntimeException(e);  }  return byteData; } |  |
| Byte[] Deserializer – consumer code  Properties p=new Properties(); p.put(ConsumerConfig.***GROUP\_ID\_CONFIG***,"1DSTR2"); p.put(ConsumerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***,"localhost:9092"); p.put(ConsumerConfig.***KEY\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.class); p.put(ConsumerConfig.***VALUE\_DESERIALIZER\_CLASS\_CONFIG***, ByteArrayDeserializer.class); p.put(ConsumerConfig.***AUTO\_OFFSET\_RESET\_CONFIG***,"earliest");  KafkaConsumer<String,byte[]> consumer=new KafkaConsumer<String, byte[]>(p); consumer.subscribe(*List*.*of*(MyAvroByteArrayProducer.*topicName*)); System.***out***.println("about to poll"); Instant start=Instant.*now*(); *//Loop for duration of 20 seconds instead of looping for ever* while(Duration.*between*(start,Instant.*now*()).getSeconds() <20){  ConsumerRecords<String, byte[]> records = consumer.poll(Duration.*ofSeconds*(4));  System.***out***.printf("successfully polled !!! and got %d records %n ",records.count());   for (ConsumerRecord<String,byte[]> record:records){  System.***out***.printf("key --> %s, value --> %s %n",record.key(), Greeting.*fromByteBuffer*(*ByteBuffer*.*wrap*(record.value())).getGreeting());  } } |  |

Sample code

Use KafkaAvroSerializer instead of ByteArraySerializer, Because this KafkaAvroSerializer will do the following things

1. when app started before sending 1st message it will send that message to schema registry to store the schema in that registry
2. And then in the payload it will send only the schema id +data without attaching schema

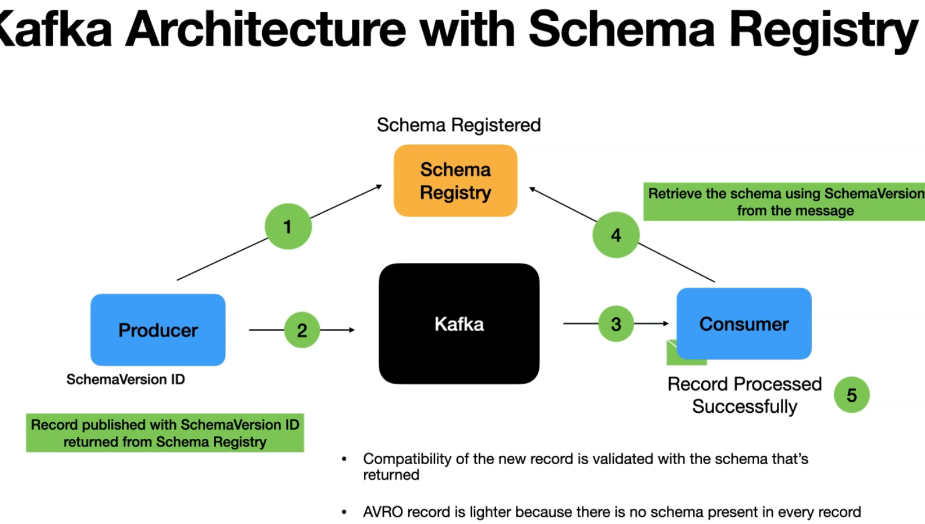
Task- Now we should send 3 data each with diff schema to same topic by starting app 3 times, - lets see if same consumer can consume all data or not? Without breaking- for every message as it internally it has schema id – for all 3 messages it should go and talk to reg and get the latest schema ??????? lets see



Seems when a consumer is continuously running - if in the existing schema if any field is removed and if that new payload is sent , then that consumer needs a restart

Schema evolution

Note:- even all the schemas will be stored in a topic called “\_schemas” as mentioned in “schema-registry.properties”



You can see all our schemas by Hitting REST endpoint

|  |  |
| --- | --- |
| Fetching the schema | http://localhost:8081/schemas/ids/4 |
| /compatibility 🡪 this is to check the compatibility between 2 schemas |  |
| /subjects (subject+topic name+version)  Ex:- subjects/topicname-value/versions | It’s a scope in which schema’s evolve  To see all the versions of a schema /topic  <http://localhost:8081/subjects/DeleteField-value/versions> |
| /config | To update the cluster level config  Compatibility is configure through config endpoint |

| **Compatibility Type** | **Changes allowed** | **Check against which schemas** | **Upgrade first** |
| --- | --- | --- | --- |
| BACKWARD | * Delete fields * Add optional fields | Last version | Consumers |
| BACKWARD\_TRANSITIVE | * Delete fields * Add optional fields | All previous versions | Consumers |
| FORWARD | * Add fields * Delete optional fields | Last version | Producers |
| FORWARD\_TRANSITIVE | * Add fields * Delete optional fields | All previous versions | Producers |
| FULL | * Add optional fields * Delete optional fields | Last version | Any order |
| FULL\_TRANSITIVE | * Add optional fields * Delete optional fields | All previous versions | Any order |
| NONE | * All changes are accepted | Compatibility checking disabled | Depends |

#### **Backward compatibility**

BACKWARD compatibility means that consumers using the new schema can read data produced with the last schema.

Backward compatibility means, assume we have 2 versions v1(old), v2 (new),

When you are **deleting** fields- first u have to **upgrade consumers with new schema**,

then consumers can consume both new and old data

if a consumer is already updated to newer version (v2), he can still consume both v1,v2 messages like java, if we upgrade java code to java 17, then still java 4 code also will mostly work (lower version of code is also supported)

ex:- consumer is still running , if producer wantedly deleted a field and if he sends the new data (where existing field is deleted) to the topic, consumers will fail,

Hence when you are **deleting** fields- first u have to **upgrade consumers with new schema**, so below exceptions are coming because of that

Reason – why they designed like that – my guess🡪 many people might be using this field, so if producer doesn’t send all of them may get null pointer exceptions as they are they are using this field

So to avoid that, they might be forcing us to update consumers 1st

Where consumer has been running but I deleted a field in the payload and then re-sent the same message, then consumer wasn’t able to process, it was able to process after restart as this restart cannot happen in production, 1st all consumers must have this new schema, then producer can send the new data (with deleted field)

When u are **adding new fields**1st we have to upgrade the **producers** with new schema- this is the most common in real time applications

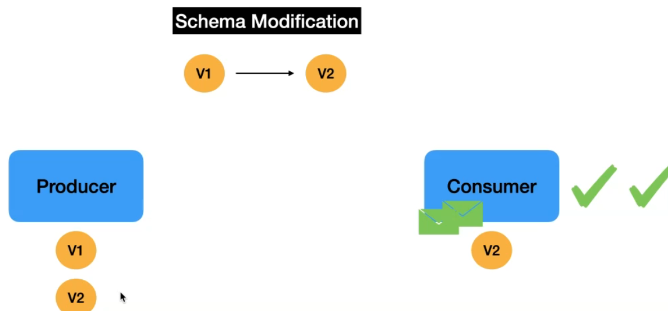
(Here you should add the default value in the schema)

Reason 🡪 why they designed like that🡪 as it is already new field, this many days consumers have already survived without that field,

so even if consumers lately upgrade also / even if they don’t upgrade also programmatically they can survive , as consumer can survive kafka also might not be forcing consumers to upgrade 1st

Kafka people may not be forcing them to upgrade consumer first

|  |  |
| --- | --- |
| Consumer is running in live-(no schema changed as its continuously running) –mandatory restart is required /1st we should update the consumer with new schema as existing field is deleted  – else it will continuously throwing exceptions  1st 10 messages produced is with OrderLineItem Data  2nd 10 messages are produced after removing this partial OrderLineItem Data | Initially consumer is stopped – started after producing 20 messages with different schema  1st 10 messages produced is with OrderLineItem Data  2nd 10 messages are produced after removing this partial OrderLineItem Data |
| Result- consumer suddenly got stopped after processing 1st message in second set  But after restarting it worked fine it consumed all those previous un-committed messages  Below is the exception occurred while processing the 1st record of new serialized message  Exception in thread "main" org.apache.kafka.common.errors.SerializationException: Error deserializing key/value for partition schema1-0 at offset 20. If needed, please seek past the record to continue consumption.  Caused by: org.apache.kafka.common.errors.SerializationException: Error deserializing Avro message for id 5  Caused by: org.apache.avro.AvroTypeException: Found com.learnavro.pojo.CoffeeOrder, expecting com.learnavro.pojo.CoffeeOrder, missing required field id | We started consumer after 20 messages are already present in topic  It consumed all 20 msgs without stopping |



So in production, if you want to delete any existing fields, 1st all consumers needs to be updated then producer should send the newer code

Let's follow the steps given in this section :

1. Let's make sure version **1.0** has **store** field in it
2. Delete the **store** field and update the version in the **schemas** module to **2.0**, and publish a 2.0 to our local maven repo.
   * gradle
     + Execute clean gradle task
     + Execute the **generateAvro** gradle task
     + Execute **publishToMavenLocal** gradle task
   * maven
     + Execute clean task
     + Execute **install** task
3. Update the consumer to use schemas module **2.0** version
4. publish the coffee-order record still with version 1.0
   * Consumer using 2.0 version should consume fine without any issues
5. Update the producer to use schemas module **2.0** version
6. publish the coffee-order record with schemas module **2.0** version
   * Consumer should consume fine without any issues
7. The subject in **coffee-orders-sr-value** should have a newer version **2.0**
   * Only the producer can create newer versions in **Schema Registry**.

Forward compatibility

This means data produced with new schema, can be read by consumers with last schema