Basic Example of try using SBT.

Program 1 → create a new folder called 1 inside lab/programs and keep the following 2 files inside that.

built.sbt

```
name := "KafkaTest"
  scalaVersion := "2.11.8"
  libraryDependencies ++= Seq(
    "org.apache.kafka" % "kafka-clients" % "1.0.0")
```

SimpleProducer.java

```
import java.util.*;
  import org.apache.kafka.clients.producer.*;
  public class SimpleProducer {
    public static void main(String[] args) throws Exception{
      String topicName = "SimpleProducerTopic";
      String key = "Key1";
      String value = "Value-1";
      Properties props = new Properties();
      props.put("bootstrap.servers", "localhost:9092");
      props.put("key.serializer","org.apache.kafka.common.serialization.StringSerializer");
      props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

      Producer<String, String> producer = new KafkaProducer<>(props);

      ProducerRecord<String, String> record = new ProducerRecord<>(topicName,key,value);
      producer.send(record);
      producer.close();
      System.out.println("SimpleProducer Completed.");
    }
  }
```

Execute the command as shown below to compile your program. → sbt compile

Execute the program with → sbt run

For testing you can start a console consumer. Use below command to execute the consumer

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic
first --from-beginning
```

==========→ Avro Example with Confluent Kafka. Execute these statements within the bin folder of /confluent-4.1.1 →

1) Start Zookeeper →

   notroot@ubuntu:~/lab/software/confluent-4.1.1$ bin/zookeeper-server-start etc/kafka/zookeeper.properties

2) Start kafka server

   notroot@ubuntu:~/lab/software/confluent-4.1.1$ bin/kafka-server-start etc/kafka/server.properties

3) Execute → schema-registry-start – the property file is schema-register.properties

notroot@ubuntu:~/lab/software/confluent-4.1.1$ bin/schema-registry-start etc/schema-registry/schema-registry.properties

Check with JPS that the these daemons are running :-

notroot@ubuntu:~/lab/software/confluent-4.1.1$ jps
6695 QuorumPeerMain
6857 SchemaRegistryMain
6749 SupportedKafka
6902 Jps

Schema evolution is all about dealing with changes in your message record over time. Let's assume that you are collecting clickstream and your original schema for each click is something like this.

1. Session id - An identifier for session
2. Browser - An identifier for the browser
3. Campaign - A custom identifier for a running campaign
4. Channel - A custom identifier for the section of the site
5. Referrer - A first hit referrer (ex - facebook.com)
6. IP - An IP address from your ISP

You had this system in place for few months, and later you decided to upgrade your schema to something like this.

1. Session id - An identifier for session
2. Browser - An identifier for the browser
3. Campaign - A custom identifier for a running campaign
4. Channel - A custom identifier for the section of the site
5. Entry URL - A first hit referrer URL
6. IP - An IP address from your ISP
7. Language - An identifier for the language
8. OS ID - An identifier for the operating system

The problem starts now. If you are changing your schema, you need to create new producers because you want to send some new fields. Right? But I have two more questions.

1. Do I need to change all current producers?
   I mean, I am fine to create new producers to include additional fields. But I don't want the system to break if I am not upgrading all of them.
2. Do I need to change my existing consumers?
   Again, I am happy to create some new consumers to work with newly added fields. But my current consumers are doing good, and they have nothing to do with new attributes. I don't want to change them.

So, I don't want to change my current producers and consumers because that will be too much of work.

So, what do I want? In fact, I want to support both old and new schema simultaneously. Can I support both versions of schemas?

In a standard case, if you change your schema, you have to change everything, your producers, consumers, serializer, and deserializer. The problem doesn't end there. After making these changes, you can't read old messages because you changed the code and any attempt to read old messages using new code will raise an exception. In summary, I need to have a combination of old and new producers as well as a mix of old and new consumers. Kafka should be able to store both types of messages on the same topic. Consumers should be able to read both types of messages without any error.

That's what is a typical schema evolution problem. How do you handle this schema evolution problem in Apache Kafka?

The industry solution to handling schema evolution is to include schema information with the data. So, when someone is writing data, they write schema and data both. And when someone wants to read that data, they first read schema and then read data based on the schema. If we follow this approach, we can keep changing schema as frequently as required without worrying to change our code because we are always reading schema before reading a message.

There are pre-built and reusable serialization systems to help us and simplify the whole process of translating messages according to schemas and embedding schema information in the message record. Avro is one of them. It is the most popular serialization system for Hadoop and its ecosystem.

Avro is a data serialization system, and it offers you four things.

1. Allows you to define a schema for your data.
2. Generates code for your schema.
3. Provide APIs to serialize your data according to the schema and embed schema information in the data.
4. Provide APIs to extract schema information and deserialize your data based on the schema.

That's it. That's all a serialization system can give you.

Everything ultimately goes to an *AvroSerializer* and an *AvroDeserializer*. Producers and Consumers will only use the generated class to create data objects. Serializer and Deserializer take care of rest. So the four steps regarding Kafka implementation.

1. Define an Avro Schema for your message record.
2. Generate a source code for your Avro Schema.
3. Create a producer and use KafkaAvroSerializer.
4. Create a consumer and use KafkaAvroDeserializer.

The next property is schema registry. It's a new component developed by the confluent team.

Kafka Schema Registry

So far, we learned that *KafkaAvroSerializer* would take care of all the serialization work at producer end and *KafkaAvroDeserializer* will take care of all the deserialization work at the consumer end.

But how do they communicate with each other about the schema? The deserializer should know the schema. Without knowing the schema, it can't deserialize the raw bytes back to an object. That's where the schema registry is useful.

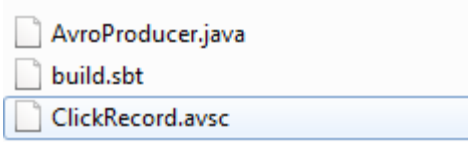
The *KafkaAvroSerializer* will store the schema details into the schema registry and include an ID of the schema into the message record.
When *KafkaAvroDeserializer* receives a message, it takes the *Schema ID* from the message and gets schema details from the registry. Once we have the schema details and message bytes, it is simple to deserialize them. That's where we use the schema registry.

Executing through SBT.

Create 2 directories

1) AvroProducer
2) AvroConsumer

In the AvroProducer folder have the following files in the:-



In the AvroConsumer have the AvroConsumer and the built.sbt and ClickRecord.avsc

ClickRecord.avsc

```
{"type": "record",
 "name": "ClickRecord",
  "fields": [
```

```
    {"name": "session_id", "type": "string"},
    {"name": "browser", "type": ["string", "null"]},
    {"name": "campaign", "type": ["string", "null"]},
    {"name": "channel", "type": "string"},
    {"name": "referrer", "type": ["string", "null"], "default": "None"},
    {"name": "ip", "type": ["string", "null"]}
  ]
}
```

==========➔

build.sbt

```
name := "AvroTest"

val repositories = Seq(
  "confluent" at "http://packages.confluent.io/maven/",
  Resolver.sonatypeRepo("public")
)

libraryDependencies ++= Seq(
"org.apache.avro" % "avro" % "1.8.1",
"io.confluent" % "kafka-avro-serializer" % "3.1.1",
"org.apache.kafka" % "kafka-clients" % "0.10.1.0"
   exclude("javax.jms", "jms")
   exclude("com.sun.jdmk", "jmxtools")
   exclude("com.sun.jmx", "jmxri")
   exclude("org.slf4j", "slf4j-simple")
)

resolvers += "confluent" at "http://packages.confluent.io/maven/"
```

==========➔

```
import java.util.*;
import org.apache.kafka.clients.producer.*;
public class AvroProducer {

   public static void main(String[] args) throws Exception{

      String topicName = "AvroClicks";
      String msg;

      Properties props = new Properties();
      props.put("bootstrap.servers", "localhost:9092,localhost:9093");
      props.put("key.serializer","org.apache.kafka.common.serialization.StringSerializer");
      props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
      props.put("schema.registry.url", "http://localhost:8081");

      Producer<String, ClickRecord> producer = new KafkaProducer <>(props);
      ClickRecord cr = new ClickRecord();
      try{
         cr.setSessionId("10001");
```

```
            cr.setChannel("HomePage");
            cr.setIp("192.168.0.1");

            producer.send(new ProducerRecord<String,
ClickRecord>(topicName,cr.getSessionId().toString(),cr)).get();

            System.out.println("Complete");
        }
        catch(Exception ex){
            ex.printStackTrace(System.out);
        }
        finally{
            producer.close();
        }

    }
}

import java.util.*;
import org.apache.kafka.clients.consumer.*;

public class AvroConsumer{

    public static void main(String[] args) throws Exception{

        String topicName = "AvroClicks";

        String groupName = "RG";
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092,localhost:9093");
        props.put("group.id", groupName);
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");
        props.put("schema.registry.url", "http://localhost:8081");
        props.put("specific.avro.reader", "true");

        KafkaConsumer<String, ClickRecord> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList(topicName));
        try{
            while (true){
                ConsumerRecords<String, ClickRecord> records = consumer.poll(100);
                for (ConsumerRecord<String, ClickRecord> record : records){
                    System.out.println("Session id="+ record.value().getSessionId()
                                + " Channel=" + record.value().getChannel()
                                + " Referrer=" + record.value().getReferrer());
                }
            }
        }catch(Exception ex){
            ex.printStackTrace();
        }
        finally{
```

```
            consumer.close();
        }
    }
}
```

==========➔

ClickRecord2.avsc

```
{"type": "record",
 "name": "ClickRecord",
 "fields": [
    {"name": "session_id", "type": "string"},
    {"name": "browser", "type": ["string", "null"]},
    {"name": "campaign", "type": ["string", "null"]},
    {"name": "channel", "type": "string"},
    {"name": "entry_url", "type": ["string", "null"], "default": "None"},
    {"name": "ip", "type": ["string", "null"]},
    {"name": "language", "type": ["string", "null"], "default": "None"},
    {"name": "os", "type": ["string", "null"],"default": "None"}
 ]
}
```

```
java -jar avro-tools-1.8.1.jar compile schema ClickRecord2.avsc .
Note: The . at the end is the final folder where the class file is created.
```

```
import java.util.*;
import org.apache.kafka.clients.producer.*;
public class ClickRecordProducerV2 {

   public static void main(String[] args) throws Exception{

      String topicName = "AvroClicks";
      String msg;

      Properties props = new Properties();
      props.put("bootstrap.servers", "localhost:9092,localhost:9093");
      props.put("key.serializer","org.apache.kafka.common.serialization.StringSerializer");
      props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
      props.put("schema.registry.url", "http://localhost:8081");

      Producer<String, ClickRecord> producer = new KafkaProducer <>(props);
      ClickRecord cr = new ClickRecord();
      try{
         cr.setSessionId("10001");
         cr.setChannel("HomePage");
         cr.setIp("192.168.0.1");
         cr.setLanguage("Spanish");
         cr.setOs("Mac");
         cr.setEntryUrl("http://facebook.com/myadd");
```

```
        producer.send(new ProducerRecord<String,
ClickRecord>(topicName,cr.getSessionId().toString(),cr)).get();

        System.out.println("Complete");
      }
      catch(Exception ex){
        ex.printStackTrace(System.out);
      }
      finally{
        producer.close();
      }

  }
}
```

Now, it's time to open three terminals.

1) Execute a consumer in one terminal. That would be an old consumer.
2) You can start the old producer and check if your consumer is able to read the message.
3) Then start a new producer in one terminal and see if the old consumer can read the new message sent by the new producer.