Apache Kafka

Kafka –Serialization

Every Kafka Streams application would use a bunch of Java types (Custom Java Objects), and you must provide Serdes (Serializer/Deserializer) for those types to materialize them whenever it is necessary.

It is not only source and sink processor that requires a Serdes. There are a bunch of other operations such as table(), through(), groupBy() that need a Serdes.

Kafka Streams application is mainly dealing with data records in a variety of formats. A simple example or a toy application can manage with 4-5 types.

However, in a real-life scenario, a complex real-time data processing requirement can quickly scale up to hundreds of unique record formats. We want to be able to define a message schema using some simple schema definition language.

Then we want our IDE or the build tool to generate Java Class definition from the schema definition automatically. There are many ways and several tools to help you achieve this.

However, there are two alternatives that I can suggest for our Kafka Streams application.

1. JSON schema to POJO
2. Avro Schema to POJO

JSON Schema to POJO

Problem – POJO from JSON

We want to be able to define a Kafka message schema using simple JSON language and use it to generate a Java class definition automatically. We also wish to the generated class definition to be well annotated to support serialization and deserialization using a simple JSON Serializer/Deserializer. The ability to quickly create serializable Java classes for a custom Kafka message format is a valuable technique that can save a lot of time and simplify the overall Kafka Streams development effort.

```
{ "type": "object", "javaType": "com.weshopifyapp.types.Notification", "properties": { "InvoiceNumber": { "type": "string" }, "CustomerCardNo": { "type": "string" }, "TotalAmount": { "type": "number" }, "EarnedLoyaltyPoints": { "type": "number" } } }
```

Apache Kafka

Solution – POJO from JSON

Generating Java objects from a schema definition is straightforward. Let us follow a step by step process to achieve it.

Step-1: Add jsonschema2pojo-maven-plugin to our maven project. The below figure shows the required pom.xml content.

```
<!-- Json Schema to POJO plugin-->
<plugin>
  <groupId>org.jsonschema2pojo</groupId>
  <artifactId>jsonschema2pojo-maven-plugin</artifactId>
  <version>0.5.1</version>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
```

```
        <configuration>

          <sourceDirectory>${project.basedir}/src/main/resources/
schema/</sourceDirectory>

          <outputDirectory>${project.basedir}/src/main/java/</
outputDirectory>

          <generateBuilders>true</generateBuilders>


        </configuration>

      </execution>

    </executions>

</plugin>
```

Apache Kafka

Step-2: Add commons-lang dependency to our maven project. The below figure shows the required pom.xml content.

```
<!--Apache commons-->

<dependency>


<groupId>commons-lang</groupId>

<artifactId>commons-lang</artifactId>


<version>2.6</version>

</dependency>
```

Step-3: Create a **resources/schema/notification.json** file and define a schema using schema definition language detailed specified by **www.jsonschema2pojo.org**

We can get more details about the schema definition language and available constructs at https://github.com/joelittlejohn/jsonschema2pojo/wiki/Reference.

Defining record schema is simple. The below figure shows a schema for the Notification message. Execute maven compile, and it will automatically generate a Java class in our maven project. That is all. We can quickly start using the Java object in our Kafka Streams application.

Avro Schema to POJO

Avro is pretty much a standard and very well accepted by the community for Bigdata projects. If we prefer to use Avro, we can quickly generate POJO from an Avro schema with the same kind of simplicity that we learned for JSON.

Problem-POJO from AVRO:

We want to be able to define a Kafka message schema using simple AVRO schema definition language and use it to generate a Java class definition automatically. We also wish to the generated class definition to be well annotated to support serialization and deserialization using an AVRO Serializer/Deserializer.

The ability to quickly create serializable Java classes for a custom Kafka message format is a valuable technique that can save a lot of time and simplify the overall Kafka Streams development effort. For this example, we want to redefine the schema definition shown in below figure using Avro schema definition language.

Apache Kafka

Solution – POJO from AVRO:

Generating Java objects from an Avro schema definition is simple. Let us follow a step by step process to achieve it.

Step-1: Add Avro dependency to our maven project. The below figure shows the necessary pom.xml

content.

```
<!-- Avro dependency-->
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro</artifactId>
  <version>${avro.version}</version>
</dependency>
```

Step-2:

Add avro-maven-plugin to our maven project. The below figure shows the required pom.xml content.

```
<!-- Maven Avro plugin for generating pojo-->
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>${avro.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
      </goals>
      <configuration>
        <sourceDirectory>${project.basedir}/src/main/resources/
schema/</sourceDirectory>
        <outputDirectory>${project.basedir}/src/main/java/</
outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Apache Kafka

Step-3: Create a **resources/schema/notification.avsc** file and define a schema using schema definition language described by Avro specification.

we can get more details of the specification at http://avro.apache.org/docs/current/spec.html.

Defining record schema is simple, the following figure shows a schema for Notification class.

```
{
 "namespace": "com.weshopifyapp.types",
 "type": "record",
 "name": "Notification",
 "fields": [
  {
   "name": "InvoiceNumber",
   "type": [
    "null",
    "string"
   ]
  },
  {
   "name": "CustomerCardNo",
   "type": [
    "null",
```

Apache Kafka

```
      "string"
    ]
  },
  {
    "name": "TotalAmount",
    "type": [
      "null",
      "double"
    ]
  },
  {
    "name": "EarnedLoyaltyPoints",
    "type": [
      "null",
      "int"
    ]
  }
 ]
}
```


Apache Kafka

```
{
  "namespace": "com.weshopifyapp.types",
  "type": "record",
  "name": "Notification",
  "fields": [
    {
      "name": "InvoiceNumber",
      "type": [
        "null",
        "string"
      ]
    },
    {
      "name": "CustomerCardNo",
      "type": [
        "null",
        "string"
      ]
    },
    {
      "name": "TotalAmount",
      "type": [
        "null",
        "double"
      ]
    },
    {
      "name": "EarnedLoyaltyPoints",
      "type": [
        "null",
        "int"
      ]
    }
  ]
}
```

Step-4: Execute maven compile, and it will automatically generate a Java class in our maven project. That is all. We can quickly start using the Java object in your Kafka Streams application.