**Kafka Message Serialization with Apache Avro using Schema Registry**

Apache Kafka allows an application to send messages, stores them on a central cluster, and allows these messages to be received by applications that process them. The Kafka cluster stores messages in categories called topics.

Apache Avro is a data serialization system. It provides a compact, fast, binary data format and relies on JSON schemas to describe data structures.

In order to serialize the data and then to interpret it, both the sending and receiving applications must have access to a schema that describes the binary format.

A schema registry lets us store schema information in a textual format (typically JSON) and makes that information accessible to various applications that need it to receive and send data in binary format. A schema is referenceable as a tuple consisting of:

* A subject that is the logical name of the schema
* The schema version
* The schema format, which describes the binary format of the data

This post will provide steps to send/ receive Avro format messages to/ from a Kafka topic using Spring Cloud Schema Registry.

Spring Cloud Schema Registry uses an H2 database by default, but server can be used with other databases by providing appropriate datasource configuration.

Schema Registry Server API

The Schema Registry Server API consists of the following operations:

* POST / - Registering a New Schema
* GET /{subject}/{format}/{version} - Retrieving an Existing Schema by Subject, Format, and Version
* GET /{subject}/{format} - Retrieving an Existing Schema by Subject and Format
* GET /schemas/{id} - Retrieving an Existing Schema by ID
* DELETE /{subject}/{format}/{version} - Deleting a Schema by Subject, Format, and Version
* DELETE /schemas/{id} - Deleting a Schema by ID
* DELETE /{subject} - Deleting a Schema by Subject

We assume that Apache Kafka is installed and running.

Here are the steps:

1. Create a Spring Cloud Schema Registry Server project.
2. Create a Spring Boot Project to send and consume Kafka messages using Avro serialization.
3. Create a Kafka topic to publish messages.
4. Create custom data type to exchange messages using Avro Schema.
5. Create a Schema Provider
6. Create Avro Message Serializer and Deserializer.
7. Create Schema Store.
8. Create a Producer to generate messages.
9. Create a Consumer to consume messages.

1. **Create a Spring Cloud Schema Registry Server project**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-schema-server</artifactId>

<version>2.0.1.RELEASE</version>

</dependency>

1. **Create a Spring Boot Project to send and consume Kafka messages using Avro serialization**

Create a Spring Starter Project with web and kafka dependencies.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

<!-- Avro dependency -->

<dependency>

<groupId>org.apache.avro</groupId>

<artifactId>avro</artifactId>

<version>1.10.0</version>

</dependency>

We will also configure the avro-maven-plugin to run the 'schema' goal on all schema’s that are found in the /src/main/resources/avro/ location as shown below.

<plugin>

<groupId>org.apache.avro</groupId>

<artifactId>avro-maven-plugin</artifactId>

<version>1.10.0</version>

<executions>

<execution>

<phase>generate-sources</phase>

<goals>

<goal>schema</goal>

</goals>

<configuration>

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

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

</configuration>

</execution>

</executions>

</plugin>

This generates the classes which contain the schemas and a few Builder methods to construct the domain objects.

1. **Create a Kafka topic to publish messages**

Create a topic named ‘events-topic’ in Kafka to publish messages.

1. **Create custom data type to exchange messages using Avro Schema**

eventmessage\_v1\_1.avsc:

{"namespace": "com.example.domain",

"type": "record",

"name": "EventMessage",

"fields": [

{"name": "identifier", "type": "string"},

{"name": "desc", "type": "string"}

]

}

Generated Class: com.example.domain.EventMessage.java

1. **Create a Schema Provider**

**public** **interface** SchemaProvider **extends** AutoCloseable {

**public** VersionedSchema get(**int** id);

**public** VersionedSchema get(String schemaName, **int** schemaVersion);

**public** VersionedSchema getMetadata(Schema schema);

}

1. **Create Avro Message Serializer and Deserializer**

**6.1 Serializing Generic Data**

We need schema to convert the message to ByteArray for transmission from the producer to the broker.

* Each record has a getSchema method. But finding out the identifier from the schema might be time consuming. It is more efficient to set the schema at initialization time. This may be done directly by identifier or by name and version.
* With the schema in hand, we need to store schema identifier in our message. We will save this information on the first four bytes.
* We will use datumWriter to encode the message.

public class KafkaAvroSerializer<T extends GenericContainer> implements Serializer<T> {

private SchemaProvider schemaProvider;

@Override

public void configure(Map<String, ?> configs, boolean isKey) {

schemaProvider = SchemaUtils.getSchemaProvider(configs);

}

@Override

public byte[] serialize(String topic, T data) {

try (ByteArrayOutputStream stream = new ByteArrayOutputStream()) {

VersionedSchema schema = getSchema(data, topic);

writeSchemaId(stream, schema.getId());

writeSerializedAvro(stream, data, schema.getSchema());

return stream.toByteArray();

} catch (IOException e) {

throw new RuntimeException("Serialization Exception: ", e);

}

}

private void writeSchemaId(ByteArrayOutputStream stream, int id) throws IOException {

try (DataOutputStream os = new DataOutputStream(stream)) {

os.writeInt(id);

}

}

private void writeSerializedAvro(ByteArrayOutputStream stream, T data, Schema schema) throws IOException {

BinaryEncoder encoder = EncoderFactory.get().binaryEncoder(stream, null);

DatumWriter<T> datumWriter = new GenericDatumWriter<>(schema);

datumWriter.write(data, encoder);

encoder.flush();

}

private VersionedSchema getSchema(T data, String topic) {

return schemaProvider.getMetadata( data.getSchema());

}

@Override

public void close() {

try {

schemaProvider.close();

} catch (Exception e) {

throw new RuntimeException(e);

}

}

}

* 1. **Deserializing Generic Data**

In order to deserialize the message, we need schema it is serialized with. But, we can also use a different schema to deserialize, if the two schemas are compatible with each other. While initializing the deserializer, we fetch the desired schemas using schema name and version and store them to use for deserialization.

public class KafkaAvroDeserializer implements Deserializer<GenericData.Record> {

private Map<String, VersionedSchema> readerSchemasByName;

private SchemaProvider schemaProvider;

@Override

public void configure(Map<String, ?> configs, boolean isKey) {

schemaProvider = SchemaUtils.getSchemaProvider(configs);

readerSchemasByName = SchemaUtils.getVersionedSchemas(configs, schemaProvider);

}

@Override

public GenericData.Record deserialize(String topic, byte[] data) {

try (ByteArrayInputStream stream = new ByteArrayInputStream(data)) {

int schemaId = readSchemaId(stream);

VersionedSchema writerSchema = schemaProvider.get(schemaId);

VersionedSchema readerSchema = readerSchemasByName.get(writerSchema.getName());

GenericData.Record avroRecord = readAvroRecord(stream, writerSchema.getSchema(), readerSchema.getSchema());

return avroRecord;

} catch (IOException e) {

throw new RuntimeException(e);

}

}

private int readSchemaId(InputStream stream ) throws IOException {

try(DataInputStream is = new DataInputStream(stream)) {

return is.readInt();

}

}

private GenericData.Record readAvroRecord(InputStream stream, Schema writerSchema, Schema readerSchema) throws IOException {

DatumReader<Object> datumReader = new GenericDatumReader<>(writerSchema, readerSchema);

BinaryDecoder decoder = DecoderFactory.get().binaryDecoder(stream, null);

GenericData.Record record = new GenericData.Record(readerSchema);

datumReader.read(record, decoder);

return record;

}

@Override

public void close() {

try {

schemaProvider.close();

} catch (Exception e) {

throw new RuntimeException(e);

}

}

}

**6.3 Serializing specific records**

Classes generated from Avro Schema have embedded schema, they are generated from. When we use such classes to send messages, we can use these schemas at runtime to serialize the message.

Classes generated from Avro Schema have embedded schema, they are generated from. When we use such classes to send messages, we can use these schemas at runtime to serialize the message.

public class KafkaSpecificRecordSerializer<T extends SpecificRecord> implements Serializer<T> {

public static final String KEY\_RECORD\_CLASSNAME = "key.record.classname";

public static final String VALUE\_RECORD\_CLASSNAME = "value.record.classname";

private int writerSchemaId;

@Override

public void configure(Map<String, ?> configs, boolean isKey) {

String className = configs.get(isKey ? KEY\_RECORD\_CLASSNAME : VALUE\_RECORD\_CLASSNAME).toString();

try (SchemaProvider schemaProvider = SchemaUtils.getSchemaProvider(configs)) {

Class<?> recordClass = Class.forName(className);

Schema writerSchema = new SpecificData(recordClass.getClassLoader()).getSchema(recordClass);

this.writerSchemaId = schemaProvider.getMetadata(writerSchema).getId();

} catch (Exception e) {

throw new RuntimeException(e);

}

}

@Override

public byte[] serialize(String topic, T data) {

try (ByteArrayOutputStream stream = new ByteArrayOutputStream()) {

writeSchemaId(stream, writerSchemaId);

writeSerializedAvro(stream, data);

return stream.toByteArray();

} catch (IOException e) {

throw new RuntimeException("Could not serialize data", e);

}

}

private void writeSchemaId(ByteArrayOutputStream stream, int id) throws IOException {

try( DataOutputStream os = new DataOutputStream(stream)) {

os.writeInt(id);

}

}

private void writeSerializedAvro(ByteArrayOutputStream stream, T data) throws IOException {

BinaryEncoder encoder = EncoderFactory.get().binaryEncoder(stream, null);

DatumWriter<T> datumWriter = new SpecificDatumWriter<>(data.getSchema());

datumWriter.write(data, encoder);

encoder.flush();

}

@Override

public void close() {

// nothing to do

}

}

**6.4 Deserializing specific records**

For deserialization, the reader schema can be found out from the class itself. Deserialization logic becomes simpler, because reader schema is fixed at configuration time and does not need to be looked up by schema name.

**public** **class** KafkaSpecificRecordDeserializer<T **extends** SpecificRecord> **implements** Deserializer<T>

{

**public** **static** **final** String ***KEY\_RECORD\_CLASSNAME*** = "key.record.classname";

**public** **static** **final** String ***VALUE\_RECORD\_CLASSNAME*** = "value.record.classname";

**private** SchemaProvider schemaProvider;

**private** Schema readerSchema;

@Override

**public** **void** configure(Map<String, ?> configs, **boolean** isKey) {

String className = configs.get(isKey ? ***KEY\_RECORD\_CLASSNAME*** : ***VALUE\_RECORD\_CLASSNAME***).toString();

**try** {

schemaProvider = SchemaUtils.*getSchemaProvider*(configs);

Class<?> recordClass = Class.*forName*(className);

**this**.readerSchema = **new** SpecificData(recordClass.getClassLoader()).getSchema(recordClass);

} **catch** (Exception e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** T deserialize(String topic, **byte**[] data) {

**try** (ByteArrayInputStream stream = **new** ByteArrayInputStream(data)) {

**int** schemaId = readSchemaId(stream);

VersionedSchema writerSchema = schemaProvider.get(schemaId);

**return** readAvroRecord(stream, writerSchema.getSchema(), readerSchema);

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

}

**private** **int** readSchemaId(InputStream stream) **throws** IOException {

**return** **new** DataInputStream(stream).readInt();

}

**private** T readAvroRecord(InputStream stream, Schema writerSchema, Schema readerSchema) **throws** IOException {

DatumReader<T> datumReader = **new** SpecificDatumReader<>(writerSchema, readerSchema);

BinaryDecoder decoder = DecoderFactory.*get*().binaryDecoder(stream, **null**);

**return** datumReader.read(**null**, decoder);

}

@Override

**public** **void** close() {

**try** {

schemaProvider.close();

} **catch** (Exception e) {

**throw** **new** RuntimeException(e);

}

}

}

1. **Create a Schema Store**

We want to create a store, such that we can quickly look for schemas. That’s why we use an In-Memory store for look up. We will create a schema registry client to communicate with Schema Registry.

**public** **class** InMemorySchemaStore **implements** SchemaStore {

**private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(InMemorySchemaStore.**class**);

**private** **final** Map<Integer, VersionedSchema> schemasById = **new** ConcurrentHashMap<>();

**private** **final** Map<SchemaNameWithVersion, VersionedSchema> schemasByNameAndVersion = **new** ConcurrentHashMap<>();

**private** **final** Map<String, VersionedSchema> schemasByParsingForm = **new** ConcurrentHashMap<>();

**private** SchemaRegistryClient service;

**public** InMemorySchemaStore(SchemaRegistryClient service) {

**super**();

**this**.service = service;

}

@Override

**public** **void** add(VersionedSchema schema) {

schemasById.put(schema.getId(), schema);

schemasByNameAndVersion.put(**new** SchemaNameWithVersion(schema.getName(), schema.getVersion()), schema);

schemasByParsingForm.put(SchemaNormalization.*toParsingForm*(schema.getSchema()), schema);

}

@Override

**public** VersionedSchema get(**int** id) {

VersionedSchema versionedSchema = schemasById.get(id);

**if** (versionedSchema == **null**) {

addSchemaFromRegistry(id);

versionedSchema = schemasById.get(id);;

}

**if** (versionedSchema == **null**) {

**throw** **new** RuntimeException("Could not find version with id=" + id);

}

**return** versionedSchema;

}

**public** **void** addSchemaFromRegistry(**int** id) {

VersionedSchema versionedSchema = **null**;

**try** {

SchemaResponse r = service.fetch(id);

versionedSchema = **new** VersionedSchema(r.getId(), r.getSubject().toString(), r.getVersion(),

**new** Schema.Parser().parse(r.getDefinition().toString()));

add(versionedSchema);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

@Override

**public** VersionedSchema get(String schemaName, **int** schemaVersion) {

VersionedSchema versionedSchema = schemasByNameAndVersion.get(**new** SchemaNameWithVersion(schemaName, schemaVersion));

**if** (versionedSchema == **null**) {

addSchemaFromRegistry(schemaName, schemaVersion);

versionedSchema = get(schemaName, schemaVersion);

}

**if** (versionedSchema == **null**) {

**throw** **new** RuntimeException("Could not find version with name=" + schemaName + " and version=" + schemaVersion);

}

**return** versionedSchema;

}

@Override

**public** VersionedSchema getMetadata(Schema schema) {

String parsingForm = SchemaNormalization.*toParsingForm*(schema);

VersionedSchema versionedSchema = schemasByParsingForm.get(parsingForm);

**if** (versionedSchema == **null**) {

addSchemaFromRegistry(schema);

versionedSchema = schemasByParsingForm.get(parsingForm);

}

**if** (versionedSchema == **null**) {

**throw** **new** RuntimeException("Could not find metadata for schema.\nParsing form: " + parsingForm);

}

**return** versionedSchema;

}

SchemaRegistryClient: -

**public** **class** SchemaRegistryClient {

**public** **static** **final** Log ***logger*** = LogFactory.*getLog*(SchemaRegistryClient.**class**);

**private** OkHttpClient client = **new** OkHttpClient();

**private** RestTemplate restTemplate;

**private** String endpoint = "http://localhost:8990";

**public** SchemaRegistryClient() {

**super**();

}

**public** SchemaRegistryClient(String endpoint) {

**super**();

**this**.setEndpoint(endpoint);

}

**protected** String getEndpoint() {

**return** **this**.endpoint;

}

**public** **void** setEndpoint(String endpoint) {

**if**(endpoint!=**null** && !endpoint.trim().isEmpty())**this**.endpoint = endpoint;

}

**protected** RestTemplate getRestTemplate() {

**return** **this**.restTemplate;

}

**public** SchemaResponse register(SchemaRegistrationRequest req) {

***logger***.info(req.getDefinition());

//POST A NEW SCHEMA

Map<String, String> map = **new** HashMap<>();

map.put("subject", req.getSubject());

map.put("format", req.getFormat());

map.put("definition", req.getDefinition());

RestTemplate restTemplate = **new** RestTemplate();

ResponseEntity<Map> responseEntity = restTemplate.postForEntity(

**this**.endpoint,

map, Map.**class**);

**if** (responseEntity.getStatusCode().is2xxSuccessful()) {

SchemaResponse registrationResponse = **new** SchemaResponse();

Map<String, Object> responseBody = (Map<String, Object>) responseEntity.getBody();

registrationResponse.setId((Integer) responseBody.get("id"));

registrationResponse.setSubject(req.getSubject());

registrationResponse.setVersion((Integer) responseBody.get("version"));

registrationResponse.setFormat(responseBody.get("format").toString());

registrationResponse.setDefinition(responseBody.get("definition").toString());

***logger***.info(registrationResponse.getId() + registrationResponse.getSubject().toString()

+ registrationResponse.getVersion() + registrationResponse.getFormat() + registrationResponse.getDefinition());

**return** registrationResponse;

}

**throw** **new** RuntimeException(

"Failed to register schema: " + responseEntity.toString());

}

**public** SchemaResponse fetch(**int** id) **throws** Exception {

Request request = **new** Request.Builder()

.url(**this**.endpoint + "/schemas/" + id)

.build();

SchemaResponse schemaResponse = **new** ObjectMapper().readValue(

client.newCall(request).execute().body().string(), SchemaResponse.**class**);

**return** schemaResponse;

}

**public** SchemaResponse fetch(SchemaReference schemaReference) **throws** Exception {

Request request = **new** Request.Builder()

.url(**this**.endpoint

+ "/" + schemaReference.getSubject() + "/" + schemaReference.getFormat()

+ "/v" + schemaReference.getVersion()

)

.build();

String s = client.newCall(request).execute().body().string();

***logger***.info(s);

SchemaResponse schemaResponse = **new** ObjectMapper().readValue(

client.newCall(request).execute().body().string(), SchemaResponse.**class**);

**return** schemaResponse;

}

**public** List<SchemaResponse> fetchAll(SchemaReference schemaReference) **throws** Exception {

Request request = **new** Request.Builder()

.url(**this**.endpoint

+ "/" + schemaReference.getSubject() + "/" + schemaReference.getFormat()

).build();

String s = client.newCall(request).execute().body().string();

***logger***.info(s);

List<SchemaResponse> schemaResponses = **new** ObjectMapper().readValue(

client.newCall(request).execute().body().string(), **new** TypeReference<List<SchemaResponse>>(){});

**return** schemaResponses;

}

}

**public** **class** SchemaRegistryStore<T> {

**private** SchemaRegistryClient service;

**private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(SchemaRegistryStore.**class**);

**public** SchemaRegistryStore(SchemaRegistryClient service) {

**super**();

**this**.service = service;

}

**public** VersionedSchema add(T record) **throws** Exception {

VersionedSchema rec = (VersionedSchema)record;

SchemaRegistrationRequest req = **new** SchemaRegistrationRequest();

req.setSubject(rec.getName());

req.setFormat(RegistrySchemaProvider.***FORMAT\_AVRO***);

req.setDefinition(rec.getSchema().toString());

SchemaResponse s = service.register(req);

rec.setId(s.getId());

**return** rec;

}

**public** **void** close() **throws** Exception {

}

}

**public** **class** RegistrySchemaProvider **implements** SchemaStore {

**public** **static** **final** String ***FORMAT\_AVRO***= "avro";

**public** **static** **final** String ***ENDPOINT\_CONF*** = "schema.registry.endpoint";

**public** **static** **final** **class** RegistrySchemaProviderFactory **implements** SchemaProviderFactory {

@Override

**public** SchemaProvider getProvider(Map<String, ?> config) **throws** Exception {

SchemaRegistryClient service = **new** SchemaRegistryClient();

**if**(config.get(***ENDPOINT\_CONF***)!=**null**)service.setEndpoint(config.get(***ENDPOINT\_CONF***).toString());

InMemorySchemaStore cache = **new** InMemorySchemaStore(service);

SchemaRegistryStore<VersionedSchema> store = **new** SchemaRegistryStore<>(service);

**return** **new** RegistrySchemaProvider(cache, store);

}

}

**private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(RegistrySchemaProvider.**class**);

**private** **final** InMemorySchemaStore cache;

**private** **final** SchemaRegistryStore<VersionedSchema> store;

**public** RegistrySchemaProvider(InMemorySchemaStore cache, SchemaRegistryStore<VersionedSchema> store) {

**this**.store = store;

**this**.cache = cache;

}

@Override

**public** VersionedSchema get(**int** id) {

**return** cache.get(id);

}

@Override

**public** VersionedSchema get(String schemaName, **int** schemaVersion) {

**return** cache.get(schemaName, schemaVersion);

}

@Override

**public** VersionedSchema getMetadata(Schema schema) {

**return** cache.getMetadata(schema);

}

**public** Collection<VersionedSchema> getAllSchemas() {

**return** cache.getAllSchemas();

}

**public** **void** add(VersionedSchema schema) **throws** Exception {

VersionedSchema s = store.add(schema);

}

@Override

**public** **void** close() **throws** Exception {

cache.close();

**try** {

store.close();

} **catch** (Exception e) {

***logger***.error("Could not close store.", e);

}

}

}

1. **Create a Producer to generate messages**

* Configure the producer to use our serializer
* Configure the Serializer to use our Avro generated class
* Configure the Schema Provider to use our SchemaRegistery

Map<String, Object> producerProps = **new** HashMap<>();

producerProps.put(ProducerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, ***BOOTSTRAP\_SERVERS***);

producerProps.put(ProducerConfig.***ACKS\_CONFIG***, "-1");

producerProps.put(ProducerConfig.***KEY\_SERIALIZER\_CLASS\_CONFIG***, IntegerSerializer.**class**.getName());

producerProps.put(ProducerConfig.***VALUE\_SERIALIZER\_CLASS\_CONFIG***, KafkaSpecificRecordSerializer.**class**.getName());

producerProps.put(KafkaSpecificRecordSerializer.***VALUE\_RECORD\_CLASSNAME***, EventMessage.**class**.getName());

producerProps.put(SchemaUtils.***SCHEMA\_PROVIDER\_FACTORY\_CONFIG***, RegistrySchemaProvider.RegistrySchemaProviderFactory.**class**.getName());

producerProps.put(RegistrySchemaProvider.***ENDPOINT\_CONF***, ***SCHEMA\_REGISTRY\_ENDPOINT***);

KafkaProducer<Integer, EventMessage> producer = **new** KafkaProducer<>(producerProps);

**for** (EventMessage msg : **new** EventMessage[] {

**new** EventMessage("EventMessage1", "EventMessage, One"),

**new** EventMessage("EventMessage2", "EventMessage, Two"),

**new** EventMessage("EventMessage3", "EventMessage, Three")

})

producer.send(**new** ProducerRecord<>(***EVENT\_TOPIC***, msg.getIdentifier().hashCode(), msg)).get();

1. **Create a Consumer to consume messages**