What is Avro?

Avro is an open source project that provides data serialization and data exchange services for Apache Hadoop. These services can be used together or independently. Avro facilitates the exchange of big data between programs written in any language. With the serialization service, programs can efficiently serialize data into files or into messages. The data storage is compact and efficient. Avro stores both the data definition and the data together in one message or file.

Avro stores the data definition in JSON format making it easy to read and interpret; the data itself is stored in binary format making it compact and efficient. Avro files include markers that can be used to split large data sets into subsets suitable for Apache MapReduce processing. Some data exchange services use a code generator to interpret the data definition and produce code to access the data. Avro doesn't require this step, making it ideal for scripting languages.

A key feature of Avro is robust support for data schemas that change over time — often called schema evolution. Avro handles schema changes like missing fields, added fields and changed fields; as a result, old programs can read new data and new programs can read old data. Avro includes APIs for Java, Python, Ruby, C, C++ and more. Data stored using Avro can be passed from programs written in different languages, even from a compiled language like C to a scripting language like Apache Pig.

A Detailed Introduction to the Avro Data Format

Serialization

Let’s start with an example. Say we have a software application dealing with orders. Each order has data like the ID, date, customer ID, type, etc. This data is loaded in memory as an object/struct as per the programming language. Below are some examples of when we might want to serialize this object:

- Storing the object in a file: Suppose we want to share a list of orders with our suppliers. Without giving them access to our system, we may share a file containing orders with them. To write the orders from memory to a file would require us to serialize each order into a stream of characters/bytes. Another scenario for serializing to a file is when the orders are too numerous to all fit into memory, so the application stores them temporarily to a file and only loads those for the current page.

- Sending over a network connection: Suppose we want to share the orders with another system in our company—say the delivery system, which runs its own web service. To send an order object to that service over the network, we would have to send it as a stream of bytes (i.e., serialize it).

And what will we do with the serialized data? At some later point, we'll need to deserialize it or read it back from the stream of bytes to an object. This implies that the serialization needs to have a particular format so that we are able to deserialize it. There are already some formats in use for this (each with their own drawback):

- CSV: No support for complex data types like structs, arrays, etc.

- XML: Verbose

- JSON: Still verbose, as field names and string representation of all types take up space

Why Avro Data Format?

Why use Avro when there are already other formats present? There are quite a few reasons. Avro:

- Has a compact and fast binary data format
- Is a documented format that makes use of schemas for correctness
- Has rich data types (e.g., arrays, maps, enumerations, objects, etc.)
- Provides a container file format that is splitable into chunks for distributed processing and contains the schema along with the data
- Ships with integration with popular languages like Python, Java, C++, etc. and can work with map/dictionary-like objects

to represent records; does not need to have strongly typed classes generated from schemas

- Supports schema evolution (i.e., schema changes); for example, if a new field is added later to the schema, reading serialized data without the field won't cause a problem

Avro Data Format

The Avro format consists of two parts: the schema and the serialized data. Both need to be present when serializing/writing or deserializing/reading data.

Avro Schema

The schema specifies the structure of the serialized format. It defines the fields that the serialized format contains, their data types (whether optional or not), default values, etc. The schema itself is in JSON format. A schema can be any of the below:

- A JSON string containing a type name—e.g., "**string**" or "**int**"
- Or a JSON object that defines a new record type in the format **{ "type" : "typeName", attributes }**
- Or a JSON array specifying a union of multiple types—e.g., ["**null**", "**string**", "**int**"] is a union of three types and means that the field using this type can be either **null** OR a **string** OR an **int**

The built-in data types are as follows:

Primitive

- **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

Complex

- **record**: a type that contains named fields, each having a type, required, default, etc.
- **enum**: an enumeration of given values; other values aren't allowed
- **array**: an array of items
- **map**: a map/dictionary of key-value pairs, where the keys must be strings
- **fixed**: a fixed number of bytes
- **union**: an array containing type names; the resulting type can be any one of these types

The complex types each have different attributes (e.g., record has fields, array has items). There are some attributes common to them though.

- The **doc** attribute is used to document what the type is for.
- The **namespace** attribute helps to avoid name collisions among types with the same name. For example, there could be multiple types named "document." Each would have a different namespace (e.g., "com.company1" and "org.org2"). And we would then refer to a type by its full name to get the correct one (e.g., "com.company1.document," instead of just "document").
- The **default** attribute specifies the default value for a type.

The above is a simplified explanation of the schema. The full Avro format specification can be found here.

Example

As an example, the schema for the order data that we saw above would be as follows:

```
{

 "namespace" : "org.example",

 "type": "record",

 "name": "order",

  "fields" : [

   {"name": "orderId", "type": "long"},

   {"name": "orderDate", "type": "int", "logicalType" : "date"},

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

       {"name": "orderType", "type":{ "type" : "enum", "name" : "orderType",

       "symbols" : ["NORMAL", "INTERNAL", "CUSTOM"] }},

    {"name": "notes", "type": [ "null", "string"], "default" : null }

  ]

}
```

Here are some things to note about the above example:

- The date type is not supported out of the box and is represented as **int** that is the number of days from the start of the epoch. An extra attribute, **logicalType**, is supported so that applications handling this data may process/convert it further.

- When we use a complex type in a field, it needs to use the object syntax: **{ type:XXX name:YYY }**.

- The "**notes**" string field can have a null value. We say this by using the union of **null** and **string** for its type. It also has a default value of **null**. The default value will be used when we're reading the serialized data if this field is missing from the data.

Avro Data

The serialized data can be created in three formats/encodings. See this reference for details.

1. **JSON format:** This is the verbose, human-readable format, useful for debugging. But it's not performant for parsing and takes up more space.

2. **Binary format:** This is the compact, performant default format. Strings are encoded as UTF-8, the rest as binary. As a simplified example, consider how 2^16 = 65536 will take up five characters to encode as a string, but only two bytes as binary. Thus, the nonstring types are serialized to an optimized binary format. One important point is that this format doesn't store the field names/IDs, so the schema used to write an object is always required when reading back the serialized data in order to identify the fields.

3. **Single object encoding:** In addition to the binary format, this also contains a fingerprint or hash of the schema with which it was created. This is useful for scenarios where the schema has changed a few times. The serialized data that we're reading may belong to any of those schemas, and we want to know which data/message belongs to which schema. In this case, the fingerprint/hash can be used to find the matching schema. This could also be achieved by introducing a custom schema field in each message.

Object Container Files

Avro also provides a format for storing Avro records as a file. The records are stored in the binary format. The schema used to write the

records is also included in the file. The records are stored in blocks and can be compressed. Also, the blocks make the file easier to split, which is useful for distributed processing like Map-Reduce. This format is supported by many tools/frameworks like Hadoop, Spark, Pig, and Hive.

Writing to Avro

Many popular languages have APIs for working with Avro. Here, we're going to see an example of writing to Avro in Python. Note how the objects we're writing are just generic dictionaries and not classes with strongly typed methods like **getOrderId()**. We need to convert the date into an **int** containing the number of days since epoch, as Avro uses this for dates. At the end, we'll have printed the number for bytes used for encoding the objects.

Here we're using the **DatumWriter** to write individual records; we're not using the Avro object container file format that also contains the schema. This will be the case when we're writing individual items to a message queue or sending them as params to a service. The **DataFileWriter** should be used if we want to write an Avro object container file.

```
from avro.io import DatumWriter, DatumReader, BinaryEncoder, BinaryDecoder

import avro.schema

from io import BytesIO

from datetime import datetime

schemaStr = ”’

{ "type": "record",

"name": "order",
```

```
"fields" : [
   {"name": "orderId", "type": "long"},
   {"name": "orderDate", "type": "int", "logicalType" : "date"},
   {"name": "customerId", "type": "string"},
      {"name": "orderType", "type": { "type" : "enum", "name" : "orderType", "symbols" : ["NORMAL", "INTERNAL", "CUSTOM"] } },
   {"name": "notes", "type": [ "null", "string"], "default" : null }
   ]
}
'''

def daysSinceEpoch( dateStr) :
   inpDate = datetime.strptime( dateStr, '%Y-%m-%d')
   epochDate = datetime.utcfromtimestamp(0)
   return( inpDate - epochDate ).days


schema = avro.schema.parse( schemaStr)


wbuff = BytesIO()
avroEncoder = BinaryEncoder(wbuff)
avroWriter = DatumWriter(schema)
```

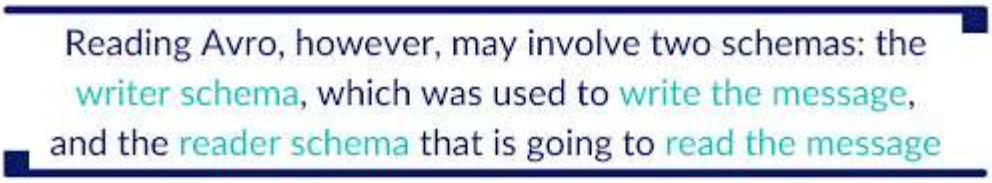
```
avroWriter.write( { "orderId":11, "orderDate":daysSinceEpoch(
"2022-01-27"), "customerId":"CST223",

   "orderType":"INTERNAL" }, avroEncoder)

avroWriter.write( { "orderId":12, "orderDate":daysSinceEpoch(
"2022-02-27"), "customerId":"MST001",

   "orderType":"NORMAL", "notes" : "Urgent" }, avroEncoder)

print( wbuff.tell())
```



Reading From Avro Data Format

Writing to Avro is straightforward. We have the schema and data to be written. Reading Avro, however, may involve two schemas: the writer schema, which was used to write the message, and the reader schema that is going to read the message. This is necessary if there are different schema versions (maybe fields have been added or removed), and so the writer's schema is different from the reader's schema. It could also happen that the reader needs to read only a subset of the fields written. Below is the code in Python, appended to the one we already have for the writer. In this case, we've used a different schema for the reader, which contains only the **orderId** and a new field, **extraInfo**. Since **extraInfo** wasn't in the written data, its default value will be used by the reader.

```
schemaStrRdr = '' { "type": "record", "name": "order", "fields" :
[ {"name": "orderId", "type": "long"}, {"name": "extraInfo",
"type": "string", "default" : "NA" } ] } '' schemaRdr =
avro.schema.parse( schemaStrRdr) rbuff =
BytesIO(wbuff.getvalue()) avroDecoder = BinaryDecoder(rbuff)
avroReader = DatumReader(schema, schemaRdr) # Use both
schemas msg = None while True : try : msg =
avroReader.read(avroDecoder) except Exception as ex: msg =
```

None print (ex) # no proper EOF handling for DatumReader if msg == None : break print(msg, rbuff.tell())

Schema Evolution

By using both the writer and reader schemas when reading data, Avro allows the schema to evolve in a limited way—for example, when adding/removing fields or when fields are changed in a way that is allowed (e.g. **int** to **long**, **float**, or **double**; **string** to and from **bytes**). We've already seen an example of this in the reader code above. Note that if a field is missing from the data being read and does not have a default value in the reader's schema, an error will be thrown. More here.

Avro Schema Registry

Given the many potential factors (e.g., that Avro messages could be consumed by many different applications, each is going to need a schema to read/write the messages, the schemas could change, and there could be multiple schema versions in use), it makes sense to keep the schemas versioned and stored in a central registry, the Schema Registry. Typically, this would be a distributed, REST API service that manages and returns schema definitions.

Why use Avro data format with Apache Kafka?

Avro is an open-source binary serialization format. But why use it with Kafka? Why not send JSON or XML messages? What benefits does it give us? This is what we'll be exploring.

Serialization

It's important to understand that records in a topic are just arrays of bytes. Kafka broker doesn't care about the type of data we're sending. To make those byte arrays useful in our applications, producers and consumers must know how to interpret them. Are we sending CSV data or JSON or XML? Producers and consumers use (de)serialization to transform data to byte arrays and back. Yet, that's only the beginning.

Schema

Whenever we use data as an integration mechanism, the schema of the data becomes very important, because it becomes the contract between producers and consumers. The teams behind producers and consumers need to agree on the following:

- Which fields are in a record
- Which fields are optional and which are mandatory
- Where and how should this be documented
- Which default values should a consumer use for the fields that are missing from the record
- How should changes to the schema be handled

Data format

Now, you could use JSON with a JSON schema or use XML with an XSD schema to describe the message format. But there's one downside with these: messages in these formats often use more space to convey the same information due to the nature of JSON and XML.

So is there a better way?

Yes. You could use Apache Avro. Avro is a data serialization format that is developed under the Apache umbrella and is suggested to be

used for Kafka messages by the creators of Apache Kafka themselves. Why?

By serializing your data in Avro format, you get the following benefits:

- Avro relies on a schema. This means every field is properly described and documented
- Avro data format is a compact binary format, so it takes less space both on a wire and on a disk
- It has support for a variety of programming languages
- in Avro, every message contains the schema used to serialize it. That means that when you're reading messages, you always know how to deserialize them, even if the schema has changed

Yet, there's one thing that makes Avro not ideal for usage in Kafka, at least not out-of-the-box, because...

Every Avro message contains the schema used to serialize the message

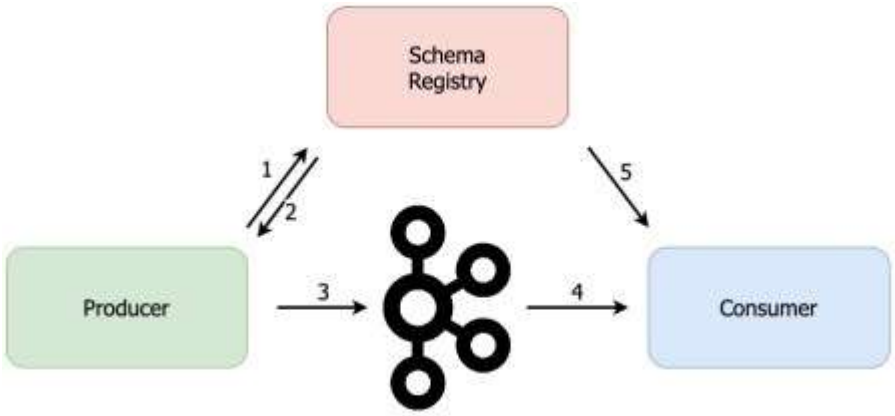
Think about this for a moment: if you plan on sending millions of messages a day to Kafka, it's a terrible waste of bandwidth and storage space to send the same schema information over and over again.

So, the way to overcome this is to...

Separate the schema from the message

That's where a Schema Registry comes into play. Schema Registry is developed by Confluent, a company behind Apache Kafka, and it provides a RESTful interface for storing and receiving Avro schemas.

Instead of sending the schema inside a Kafka record, a producer starts by checking whether schema already exists in the Schema Registry. If not, it will write the schema there (step 1 below). Then the producer will obtain the id of the schema (step 2) and send that id inside the record (step 3), saving a lot of space this way. The consumer will read the message (step 4) and then contact the Schema Registry with the schema id from the record to get the full schema (step 5) and cache it locally.



Registering and using an Avro schema

Ok, that seems nice. But that's not the only place where Schema Registry helps. Let's see what happens when you want to…

Evolve the schema

Imagine yourself in a situation where, 2 years after releasing the producer application, you decide to change the format of the message in a way that, in your opinion, doesn't break the compatibility and thus should not affect the consumers. Now you have 2 options:

- you can either be a nice person, find all the consumers, check whether the suggested changes will affect them and if so, ask them to change
- or you can do the change and wait for the mob with torches, shovels, and rakes to come your way

Assuming you chose to avoid the option that would put your picture on every train station with a monetary reward under it, what is the probable outcome of the option number one?

Those who worked in a corporate environment know the answer: a lot of planning, budgeting, negotiating and sometimes even postponing because there are more pressing issues. Getting 10s or 100s of consumers to perform the upgrade before you can continue with the changes in the producer is a sure way to an asylum.

Apache Avro and Schema Registry are coming to the rescue once again. Schema Registry allows us to enforce the rules for validating a schema compatibility when the schema is modified. If a new message

breaks the schema compatibility, a producer will reject to write the message.

This way your consumers are protected from e.g. someone suddenly changing the data type of the field from a long to a string or removing the mandatory field.

At the same time, it gives you clear guidelines about which changes to the schema are allowed. So no longer weeks or months of coordinating the change just to remove an optional field in a message.

Kafka Schema Registry & Avro: Introduction

Messages being written to and consumed from a topic can contain any data as they are byte arrays. By applying a schema it ensures that the message written to a topic by a producer can be read and understood by a consumer of the topic. In order to manage and make such a schema available, a schema registry is used that both the producer and the consumer talk utilise. One such message schema is provided Apache Avro, a serialisation framework which provides type handling and compatibility between versions.

The source code for the accompanying Spring Boot demo application is available here.

The Importance of a Schema

Schemas for Kafka messages are optional, but there are strong benefits for choosing to use them. A schema defines the fields and types of the message ensuring that only those messages that meet this definition will be understandable by the serializer or deserializer. The

message schema essentially defines the contract between the event streaming services in the same way that a REST API defines the contract for REST calls. Schemas can evolve and be versioned as message definitions change over time.

Avro Serialization

There are different options around the type of serialization to use for Kafka messages. These include String, JSON, and Protobuf, along with Avro. Each comes with their own advantages and disadvantages, from the performance of serialization, to version compatibility, and the ease of understanding and use. For example the popular JSON serialization provides human readable messages, but lacks typing and version compatibility.

Avro uses schemas for marshalling and unmarshalling binary data to and from typed records that the consumer and producer understand. The schema itself is defined in a JSON format so is human readable, and the source code representing the message can be generated from this schema.

Schemas can be defined for both message keys and for message values, as the two are serialized and deserialized independently. It is therefore possible to use a standard String Serializer then for the key, and an Avro Serializer for the main message body, for example.

Schema Registry

By sending and consuming messages via Kafka, applications remain decoupled from one another, allowing them to evolve and change

without impacting or being impacted by other applications. Utilising a message schema ensures that consumers are able to deserialize messages that have been serialized by another application with the same schema. One option for using a message schema is to include that schema in each message itself, so that the consumer can utilise it to deserialize the payload. However this increases every message size, often doubling it, and that has a direct impact on latency and throughput as more memory and CPU is required by the producer, the broker, and the consumer.

To solve this a schema registry can be used. The schema registry is an independent component that microservices talk to in order to retrieve and apply the required schemas.

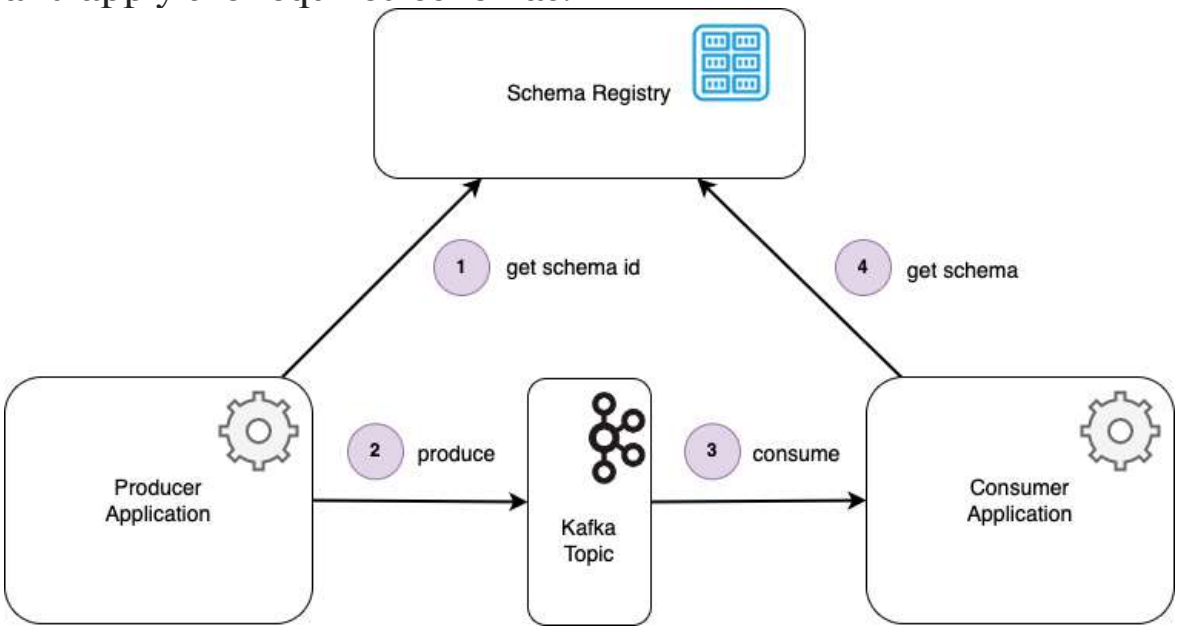


Figure 1: Producer and Consumer remain decoupled

The most popular schema registry used for Kafka applications is Confluent's Schema Registry. This can be run as a standalone service or hosted by Confluent in the cloud. It provides a REST API for administering and retrieving the schemas. New schemas are

registered with the Schema Registry, which can happen automatically as part of the serialization flow, via a REST client, or if configured as part of the CI/CD pipeline.

The upshot of using the schema registry as a common repository for the schema is that the microservices that use it remain fully decoupled. The applications are able to evolve independently, with well-defined versioned schemas ensuring the evolving messages can still be understood.

Registering Schemas

Confluent's Schema Registry uses Kafka itself as its storage backend. The Schema Registry writes the schemas that are being registered to a dedicated, compacted, topic. By default this is called **_schemas**, although can be configured using the **kafkastore.topic** Schema Registry parameter.

A new schema or version of an existing schema is registered using the Schema Registry REST API. The REST API can be called manually, or by using the Schema Registry client abstraction provided by Confluent, or triggered via the Confluent Schema Registry maven plugin. Alternatively Confluent's Kafka Avro serializer can be configured to register the new schema if it does not yet exist. The Schema Registry then consumes the schemas from the **_schemas** topic, building a copy in a local cache.

The following diagram illustrates the registration flow:

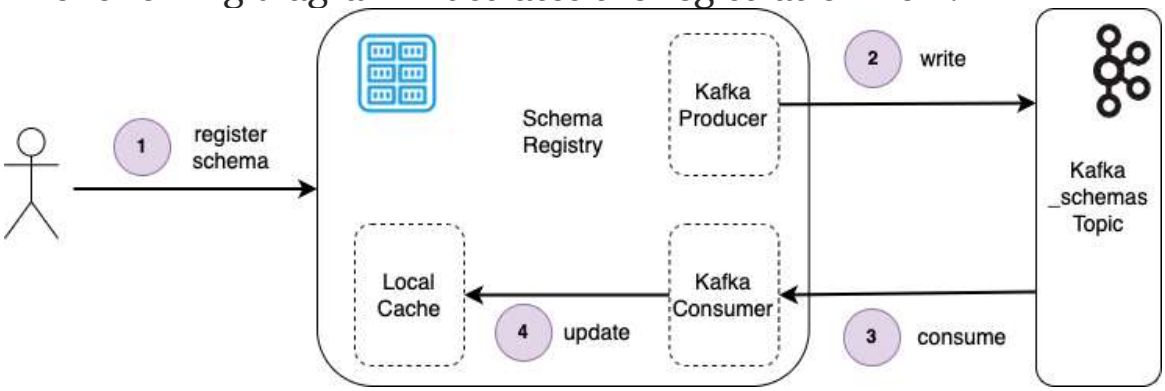


Figure 2: Registering a schema

1. An Avro schema is registered with the Schema Registry via a REST POST request.
2. The Schema Registry writes the new schema to the **_schemas** Kafka topic.
3. The Schema Registry consumes the new schema from the **_schemas** topic.
4. The local cache is updated with the new schema.

When the schema or schema Id is requested from the Schema Registry as happens during the serialization and deserialization flows, the results can be served from the local cache.

By utilizing Kafka as the backend storage for schemas, the Schema Registry therefore gets all the benefits that Kafka provides, such as resiliency and redundancy. For example, if a broker node hosting

a **_schemas** topic partition fails, a replica partition will have a copy of the data.

If the Schema Registry fails and is restarted, or a new instance of the Schema Registry is started, it begins by consuming the schemas available from the compacted **_schemas** topic to (re)build its local cache.

Serialization & Deserialization

Confluent's Kafka avro serializer library provides the **KafkaAvroSerializer** and **KafkaAvroDeserializer** classes that are responsible for performing the serialization and deserialization of message keys and values. These are registered with the consumer and producer respectively. As part of their de/serialization processing they interact with the Schema Registry as necessary, such that the developer does not need to be concerned with this complexity.

The following diagram illustrates this flow, with a producing application serializing a message, writing it to a Kafka topic, with a downstream application consuming the message and deserializing it.

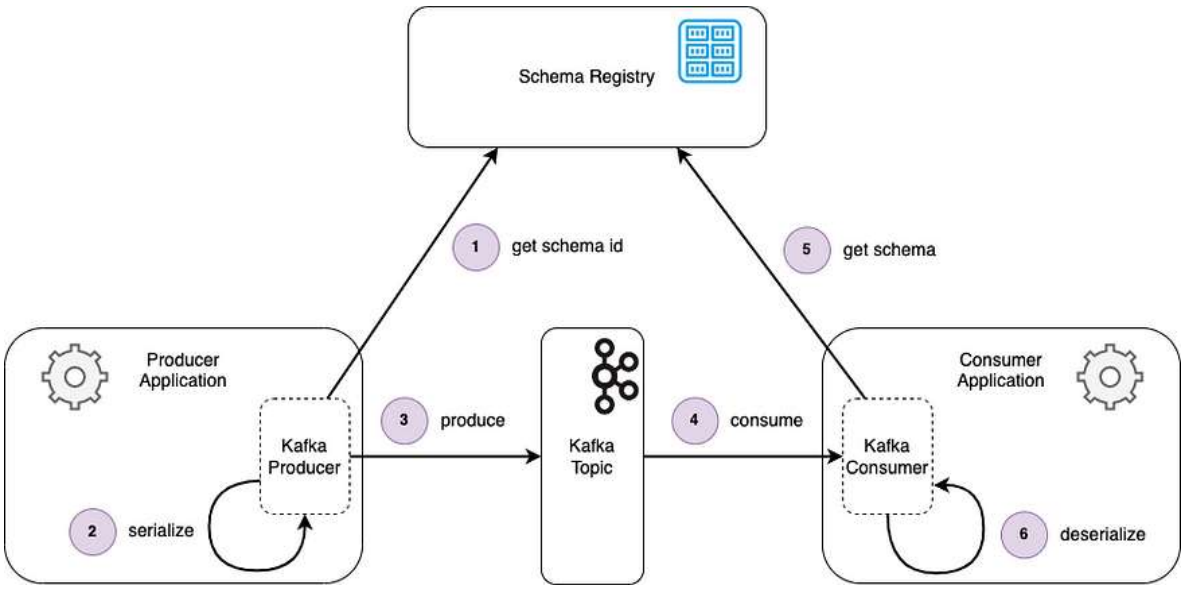


Figure 3: Serialization and deserialization

1. A message is being produced. The Kafka Avro Serializer in the producer gets the schema id associated with the schema of the message (which is obtained by reflection).
2. The message is serialized in the Avro format, verified using the retrieved schema.
3. The message is written to the Kafka topic.
4. The message is consumed from the topic by the Kafka consumer.
5. The Kafka Avro Deserializer in the consumer gets the schema Id from the message and uses this to look up the schema from the Schema Registry.
6. The message is deserialized, verified using the retrieved schema.

The following sequence diagrams break this flow out into the produce and consume sides. On the produce side, the **KafkaAvroSerializer** that is assigned to the Kafka producer

first obtains the schema from the message class by using reflection. It uses the **CachedSchemaRegistryClient**, provided by Confluent's **kafka-schema-registry-client** library, to obtain the associated schema Id. If the schema Id is not already cached locally then the client calls the Schema Registry via its REST API in order to get this Id. The schema Id is cached locally, ensuring that only on the first occasion is the REST call required for this lookup. At this point the **KafkaAvroSerializer** serializes the message to a byte array. The serialized message starts with a Confluent serialization format version number (currently always 0) which is known as the 'magic byte'. This is followed by the schema Id, followed by the rest of the data in Avro's binary encoding. This is then written to the Kafka topic.

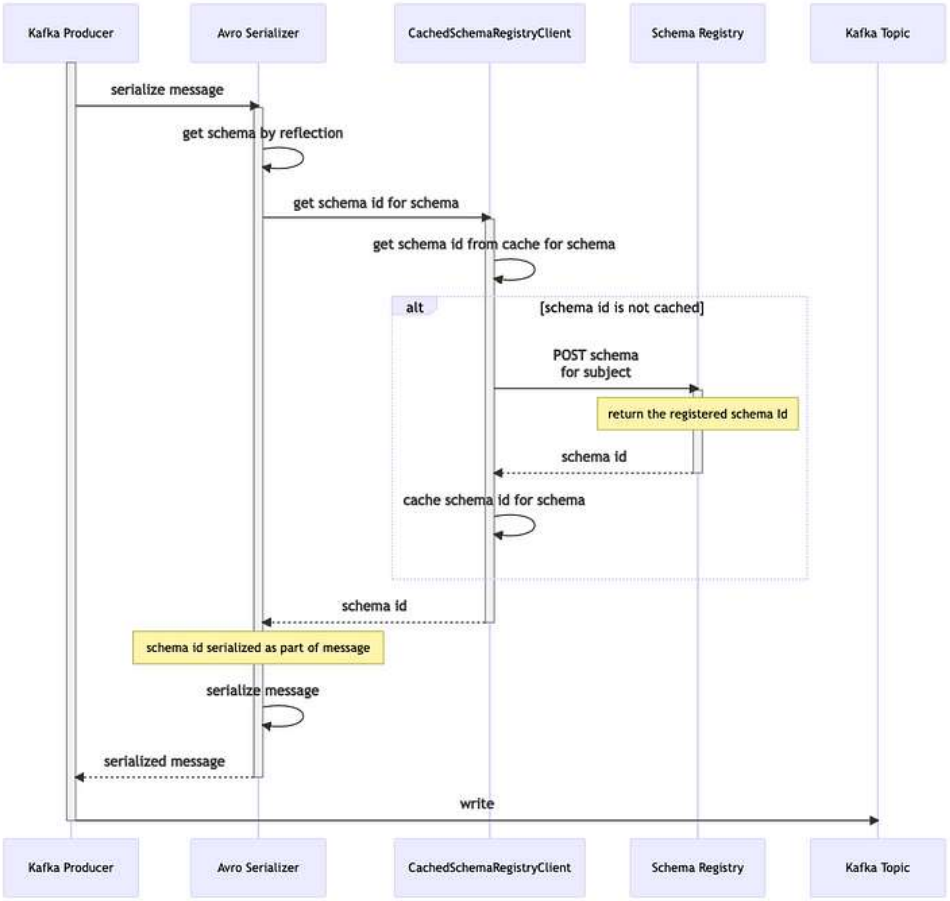


Figure 4: Producer serialization with Avro

On the consumer side, when a batch of messages is fetched from the Kafka topic and each is handed off to the Kafka consumer, the consumer delegates to the **KafkaAvroDeserializer** for the deserialization processing. It first extracts the schema Id from the message (stored in bytes 1 to 4), and uses the **CachedSchemaRegistryClient** which checks the local cache to see if the schema is already present. If not, the client performs a GET schema for the given schema Id. It saves this to its local cache for subsequent messages. It then returns the message back to the **KafkaAvroDeserializer** which can now deserialize the message to its Avro Class representation using the schema. This is returned to the consumer to continue its processing.

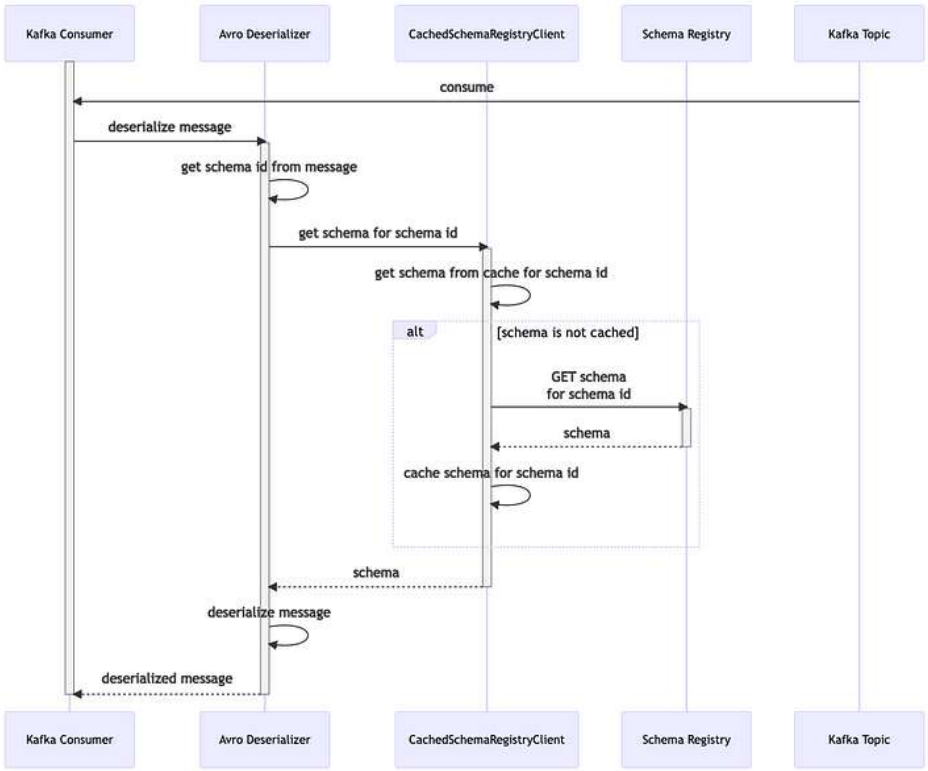


Figure 5: Consumer deserialization with Avro

While the developer does not need to write the serialization flows themselves, understanding how it works is of course important, particularly when unexpected errors occur. Beyond that however it is

necessary to understand the calls that are made to the Schema Registry when looking to mock these in integration tests. This testing will be covered in an upcoming article.

Conclusion

The addition of a schema registry such as Confluent's Schema Registry into the architecture enables applications to apply schemas to messages sent to and from Kafka. The schema registry stores and manages the schemas allowing the applications themselves to remain decoupled. Schemas essentially define the contract between messaging services. They enable messages to evolve through versioning, enabling the consumer and producer applications to evolve over time. Kafka consumers and producers can use Apache Avro to serialize and deserialize messages, querying and caching the schemas to apply from the schema registry.

Source Code

The source code for the accompanying Spring Boot demo application is available here:

https://github.com/lydtechconsulting/kafka-schema-registry-avro/tree/v1.0.0

More On Kafka Schema Registry & Avro

The following accompanying articles cover the Schema Registry and Avro:

Kafka Schema Registry & Avro: Spring Boot Demo (1 of 2): provides an overview of the companion Spring Boot application and steps to run the demo.

Kafka Schema Registry & Avro: Spring Boot Demo (2 of 2): details the Spring Boot application project structure, implementation and configuration which enables it to utilise the Schema Registry and Avro.

Kafka Schema Registry & Avro: Integration Testing: looks at integration testing the application using Spring Boot test, with the embedded Kafka broker and a wiremocked Schema Registry.

Kafka Schema Registry & Avro: Component Testing: looks at component testing the application using Testcontainers and the component-test-framework to bring up the application, Kafka, and the Schema Registry in Docker containers.