**What is and Why a schema registry?**

Confluent Schema Registry provides a RESTful interface for developers to define standard schemas for their events, share them across the organization and safely evolve them in a way that is backward compatible and future proof.

Schema Registry validates compatibility and warns about possible issues. This allows developers to add and remove fields independently to move faster. New schemas and versions are automatically registered and automatically validated.

If you use incompatible schema accidentally? That’s just an exception for the producer to handle. Incompatible data will never make it into Apache Kafka. Schema Registry avoids producers from pushing message that are bad in to topics. This saves a lot of headache for down-stream consumer.

Why Schema Registry? These are the answers:

* DEPLOY RELIABLY: Let your developers focus on deploying applications freely while protecting your Apache Kafka deployment from breaking changes. If you change data format, the application is able to handle that or to reject that data (so there is no “crash” of the data application because the data validation).
* EVOLVE QUICKLY: Need to add a new column to a downstream database? You don’t need an involved change process and at least 5 meetings to coordinate 20 teams. Validate your changes as an integrated part of the development process, reducing coordination overhead and ensuring information integrity in the environment.

In breve, lo schema registry aiuta nel processo di VALIDAZIONE e COMPATIBILITà dei dati. Infatti se un producer manda un dato a kafka con uno schema che non esiste, Kafka non accetta I dati e quindi è come se facesse una validazione dei dati. Inoltre se hai diverse verioni del prodotto (v1, v2, v3, etc...) in cui cambi il data schema, lo schema registry è in grado di capire queste diverse version dei dati.

>> Background: problemi nel data serialization

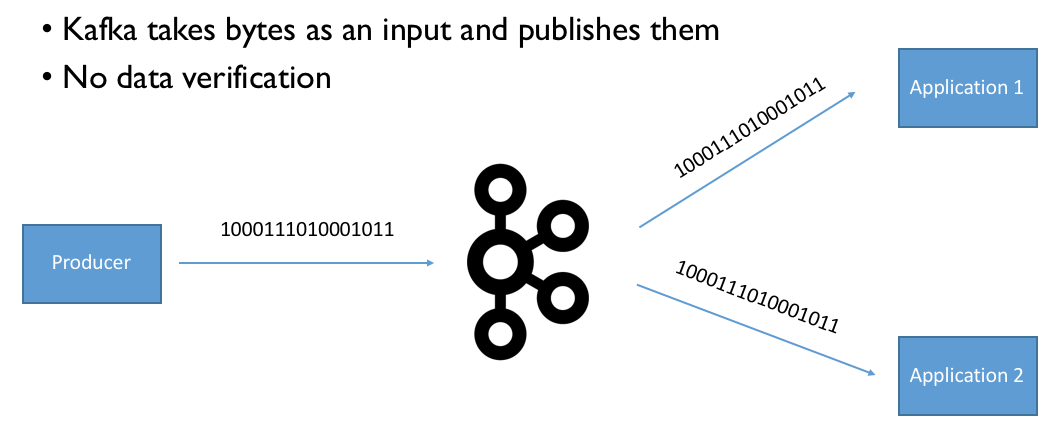
Kafka reads streams of bytes! Obviously, your data has meaning beyond bytes, so your consumers need to parse it and later on interpret it.

Typycal errors due to data parsing are:

* The field you’re looking for doesn’t exist anymore
* ‎The type of the field has changed (e.g. what used to be a String is now an Integer)

What are our options to prevent and overcome these issues?

* Catch exception on parsing errors. Your code becomes ugly and very hard to maintain. {{ BAD solution }}
* ‎Never ever change the data producer and triple check your producer code will never forget to send a field. That’s what most companies do. But after a few key people quit, all your “safeguards” are gone. {{ BAD solution }}
* Adopt a data format and enforce rules that allow you to perform schema evolution while guaranteeing not to break your downstream applications. {{ COOL and GOOD solution }}

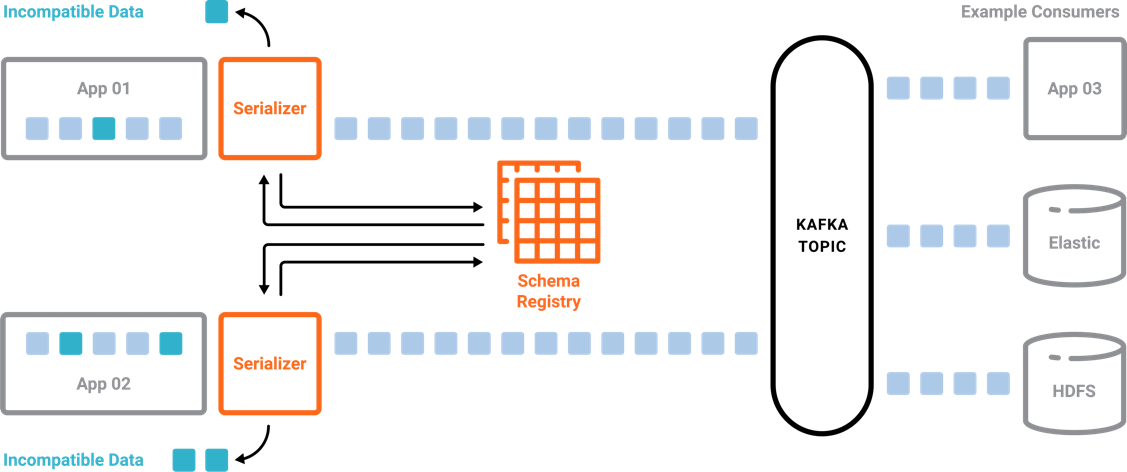


Per mandare dati over the network, I dati devono essere serializzati, cioè deve essere usato un metodo di encoding per trasformare I dati in bytes.

Formats like JSON lack a strictly defined format (it has not specified the structure, the type and the meaning of the data), which has two significant drawbacks:

* Data consumers may not understand data producers: The lack of structure makes consuming data in these formats more challenging because fields can be arbitrarily added or removed, and data can even be corrupted. This drawback becomes more severe the more applications or teams across an organization begin consuming a data feed: if an upstream team can make arbitrary changes to the data format at their discretion, then it becomes very difficult to ensure that all downstream consumers will (continue to) be able to interpret the data. What’s missing is a “**contract**” (cf. schema below) for data between the producers and the consumers, similar to the contract of an API.
* Overhead and verbosity: They are verbose because field names and type information have to be explicitly represented in the serialized format, despite the fact that are identical across all messages.

>> Descrizione di un generic esempio



Kafka legge streams di bytes! Quindi qualunque cosa Kafka riceve, lo manderà al consumers che si sono sottoscritti su di uno specific topic SENZA preoccuparsi del “significato” o “validazione” dei dati.

Il problema è quindi come evitare di salvare nei kafka topics e di mandare ai consumatori dati che non sono in grado di interpretare e di utilizzare? In altre parole, come poter fare la validazione dei dati senza che il consumatore debba crashare o ricevere (e poi scartare) dati che non è in grado di usare?

Lo schema registry è in grado di aiutarci in questo! Qui sotto è spiegato il processo descritto nella figura! E’ importante che **tutti I PUBLISHERs nell’applicazione prima chiamino lo schema registry**, e poi effettivamente pubblichino I dati su kafka.

01. The serializer places a call to the schema registry, to see if it has a format for the data the application wants to publish. If it does, schema registry passes that format to the application’s serializer, which uses it to filter out incorrectly formatted messages.

02. After checking the schema is authorized, it’s automatically serialized and there’s no effort you need to put into it. The message will, as expected, be delivered to the Kafka topic.

03. Your consumers will handle deserialization, making sure your data pipeline can quickly evolve and continue to have clean data. You simply need to have all applications call the schema registry when publishing.

>> Avro Schema and Serializer

There are different cross-language serialization libraries. These libraries include Avro, Thrift, and Protocol Buffers. The advantage of having a schema is that it clearly specifies the structure, the type and the meaning (through documentation) of the data. With a schema, data can also be encoded more efficiently. In particular, we recommend Avro which is supported in Confluent Platform.

Apache Avro™ is a **data serialization system (nella foto sopra, Avro è il blocco di “serializer”)**. Avro provides:

* Avro has support for primitive types ( int, string, long, bytes, etc…), complex types (enum, arrays, unions, optionals), logical types (dates, timestamp-millis, decimal), and data record (name and namespace). All the types you’ll ever need.
* Avro has support for embedded documentation. Although documentation is optional, in my workflow I will reject any Avro Schema PR (pull request) that does not document every single field, even if obvious. By embedding documentation in the schema, you reduce data interpretation misunderstandings, you allow other teams to know about your data without searching a wiki, and you allow your devs to document your schema where they define it. It’s a win-win for everyone.
* Avro schemas are defined using JSON. Because every developer knows or can easily learn JSON, there’s a very low barrier to entry.
* An Avro object contains the schema and the data. The data without the schema is an invalid Avro object. That’s a big difference with say, CSV, or JSON.
* You can make your schemas evolve over time. Apache Avro has a concept of projection which makes evolving schema seamless to the end user.

An Avro schema defines the data structure in a JSON format.

The following is an example Avro schema that specifies a user record with two fields: name and favorite\_number of type string and int, respectively.

{"namespace": "example.avro",

"type": "record",

"name": "user",

"fields": [

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

{"name": "favorite\_number", "type": "int"}

]

}

You can then use this Avro schema, for example, to serialize a Java object (POJO) into bytes, and deserialize these bytes back into the Java object.

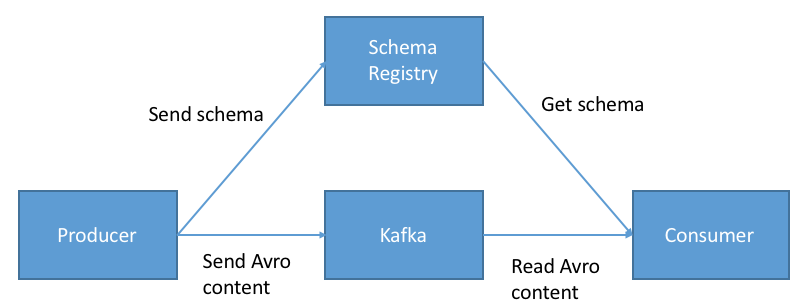
Using Avro, you have the opportunity to not include the full data schema in all the messages, but you can just include the schema\_id. In this case it will be the Avro deserializer component in the consumer process who will handle the validation and deserialization process. It will read the schema from the schema registry using the schema\_id, and apply the information contained in the schema registry in roder to deserialize the data. So, looking at the example above, it will “cast” the filed “favorite\_number” as an integer.

Where the schemas are stored? In kafka itself! :-D

The default Kafka topic in which schema\_id, schema\_data and other metadata are stored in **\_schemas** topic.

A Schema Registry instance (cioè il processo “./bin/schema-registry” che lancio sul terminale) therefore both produces and consumes messages under the \_schemastopic. It produces messages to the log when, for example, new schemas are registered under a subject, or when updates to compatibility settings are registered. Schema Registry consumes from the \_schemas log in a background thread, and updates its local caches on consumption of each new \_schemas message to reflect the newly added schema or compatibility setting. Updating local state from the Kafka log in this manner ensures durability, ordering, and easy recoverability.

The architecture which include the kafka-schema-registry, will look like this:



As you can see, your producers and consumers still talk to Kafka, but now they also talk to your Schema Registry.

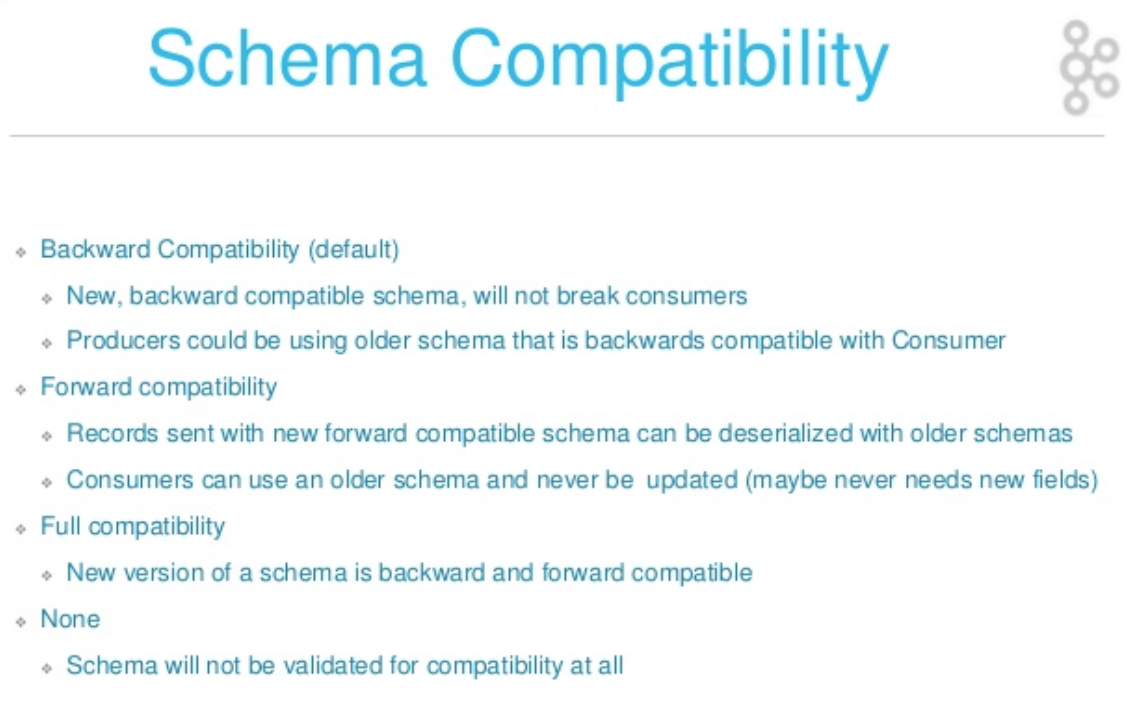
The engineering beauty of this architecture is that now, your Producers use a new Serializer, provided courtesy of Confluent, named the KafkaAvroSerializer. Upon producing Avro data to Kafka, the following will happen (simplified version):

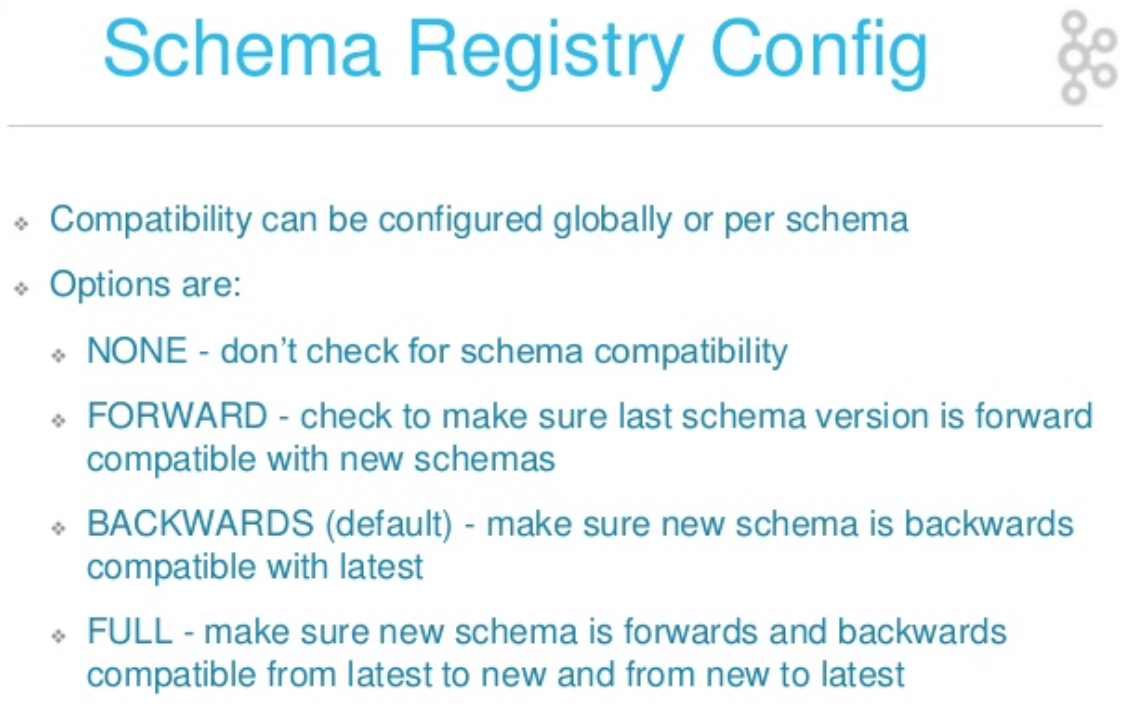
* Your producer will check if the schema is available is in the Schema Registry. If not available, it will register and cache it
* The Schema Registry will verify if the schema is either the same as before or a valid evolution. If not, it will return an exception and the KafkaAvroSerializer will crash your producer. Better safe than sorry
* If the schema is valid and all checks pass, the producer will only include a reference to the Schema (the Schema ID) in the message sent to Kafka, not the whole schema. The advantage of this is that now, your messages sent to Kafka are much smaller!

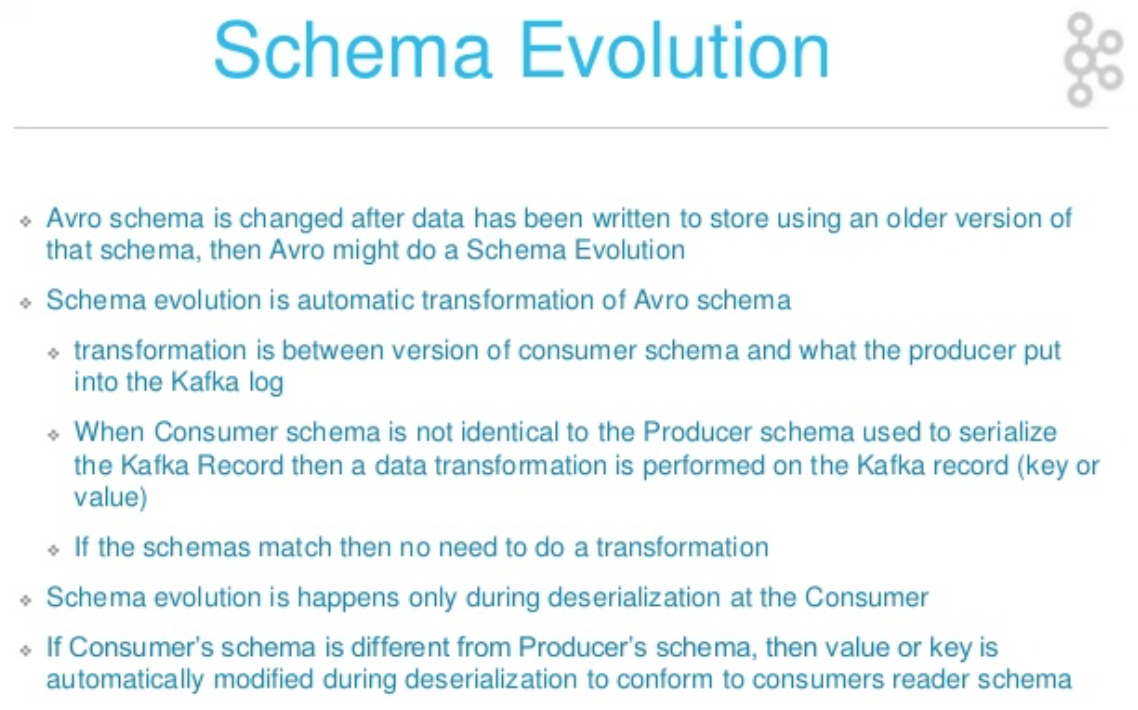
Links:

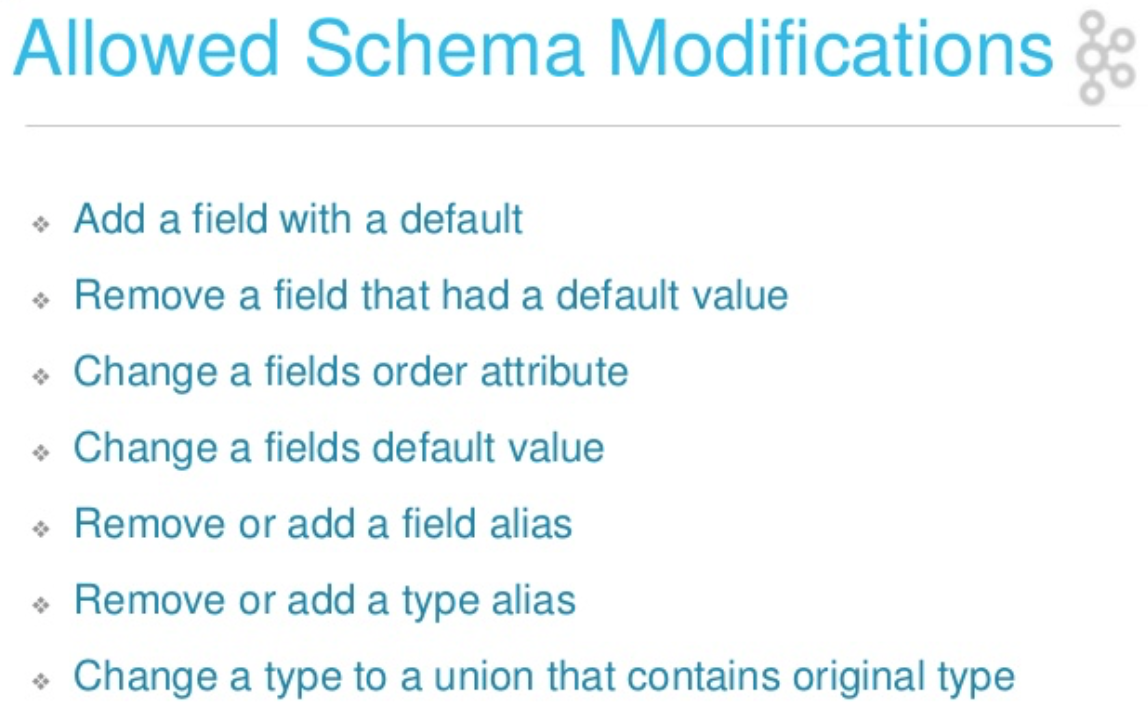
* <https://docs.confluent.io/current/schema-registry/docs/index.html>
* <https://medium.com/@stephane.maarek/introduction-to-schemas-in-apache-kafka-with-the-confluent-schema-registry-3bf55e401321>
* <https://www.confluent.io/blog/getting-started-with-kafka-in-node-js-with-the-confluent-rest-proxy/> => avro in nodejs

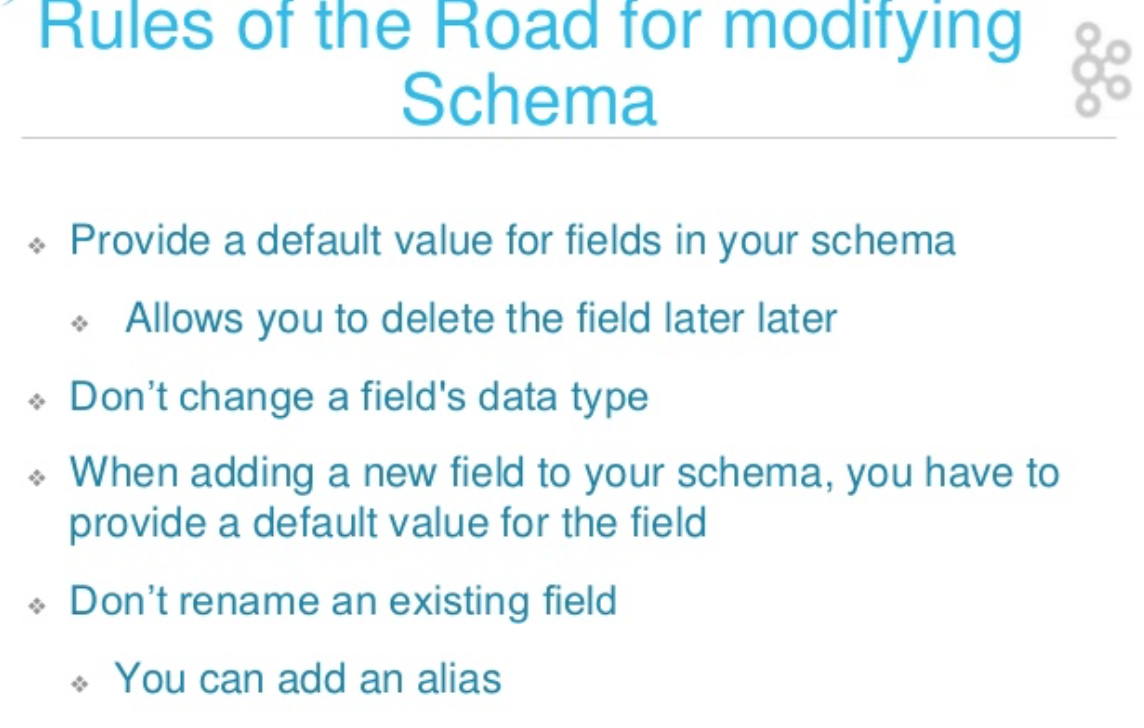
**Types of compatibility in Schema Registry and How/When to use the compatibility?**



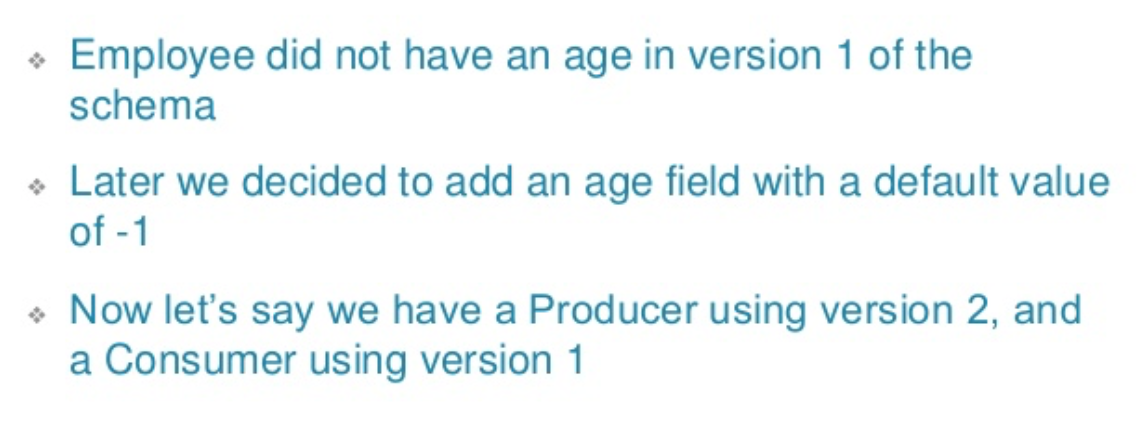


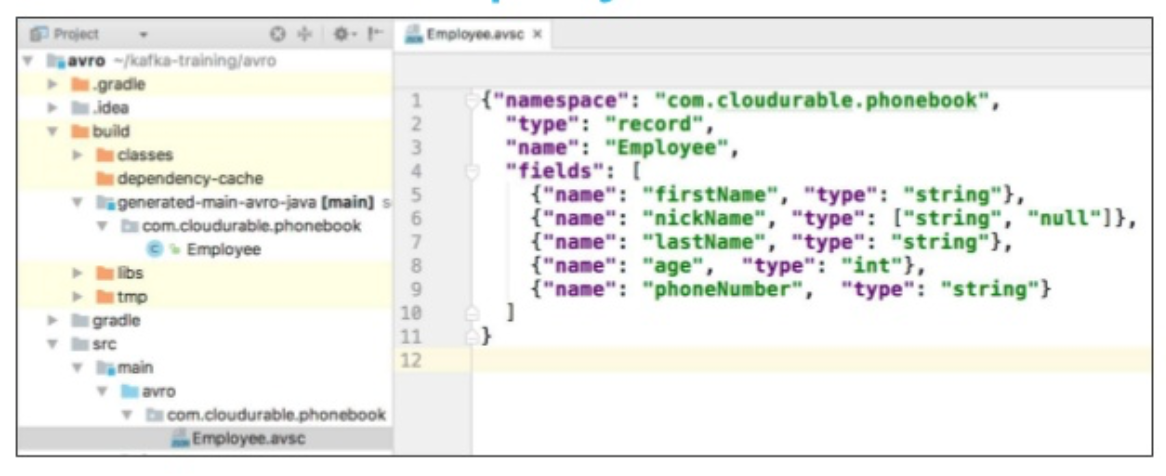




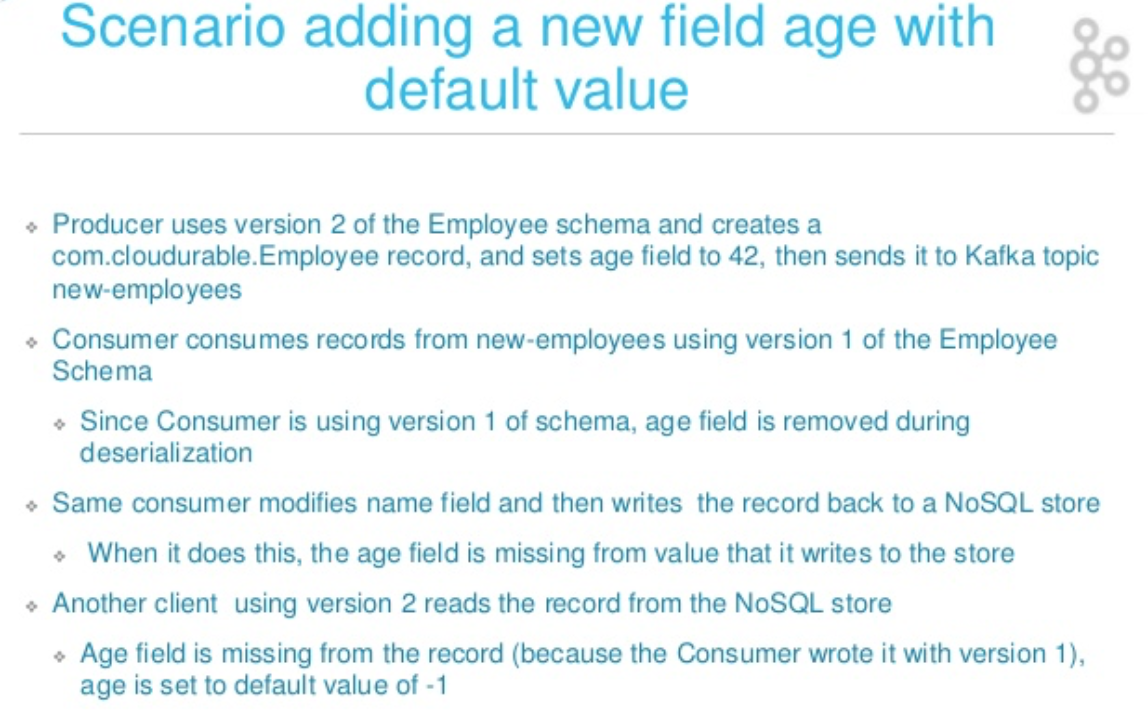


>> Example





>> Resolution of the Example



Links:

* <http://cloudurable.com/blog/kafka-avro-schema-registry/index.html> => Why compatibility and how to use it? VERY GOOD article

**Benefit of Schema Registry in IoT scenario?**

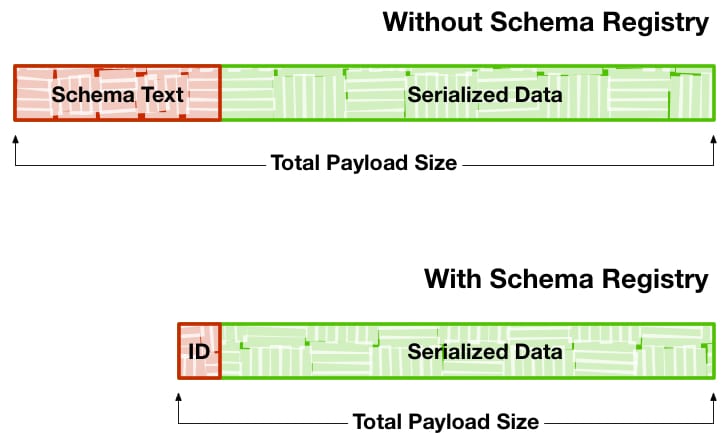
Schema Registry provides a centralized repository for schemas and metadata, allowing services to flexibly interact and exchange data with each other without the challenge of managing and sharing schemas between them.

Schema Registry has support for multiple underlying schema representations (Avro, JSON, etc.) and is able to store a schema’s corresponding serializer and deserializer.

>> Smaller Payload

Typically, when serializing data for transmission using schemas, the actual schema (text) needs to be transmitted with the data. This results in an increase of payload size.

Using Schema Registry, all schemas are registered with a central system. Data producers no longer need to include the full schema text with the payload, but instead only include the ID of that schema, also resulting in speedier serialization.



>> Differing Schema for Differing Devices

Consider the case where thousands of medical devices are reading the vitals of patients and relaying information back to a server.

The services and applications in your pipeline are expecting data using a specific format and fields that these medical devices use.

What about when medical devices from a different vendor are added to the system? Data in a different format carrying a different set of fields would typically require updates to the different components of your data pipeline.

Schema Registry enables generic format conversion and generic routing, allowing you to build a resilient pipeline able to handle data in different format with varying sets of fields.

>> Schema Evolution for New Devices Release

Following the use-case above, consider the case when the software in some of the medical devices you are collecting data from is updated. Some devices now collect new data points, while other devices report to same limited number of fields as before. Similarly, consider when the processing step in the pipeline is altered to output data with fewer or more fields than its previous version. Typically, for either of these cases, the rest of your pipeline would need to be updated to handle these changes.

With Schema Registry, the different components in your architecture (IoT devices, routing logic, processing nodes, etc.) can evolve at different rates. Components can change the shape of its data while Schema Registry handles the translation from one schema to another, ensuring compatibility with downstream services.

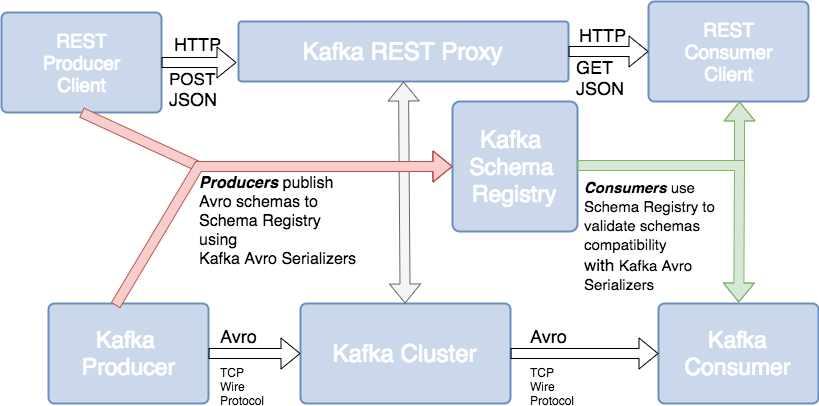
Link (IoT + Kafka):

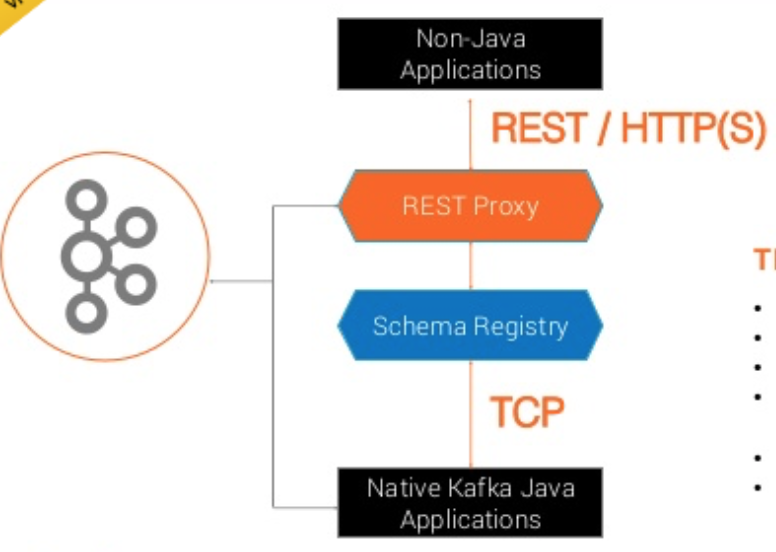
* <https://www.slideshare.net/KaiWaehner/iot-integration-with-mqtt-and-apache-kafka>
* <https://www.landoop.com/blog/2017/01/mqtt-kafka-connect-with-ais-data/>

**Schema Registry in Nodejs?**

To use a schema registry there are two possible ways:

* or using producer/consumer with integrate libraries able to handle data schema
* Use a Kafka REST Proxy as interface to communicate both with Kafka Broker/Cluster and the schema registry (NB: the picture below is not actually correct, because REST Producer Client can’t communicate directly with Kafka Schema Registry)





To use the first way, you need to use the following nodejs module:

<https://www.npmjs.com/package/kafka-avro>

To use the second way, follow the following tutorial:

<https://www.confluent.io/blog/getting-started-with-kafka-in-node-js-with-the-confluent-rest-proxy/>

and the following nodejs module:

<https://github.com/confluentinc/kafka-rest-node>

**What is Confluent?**

**What is API Streams?**

**What is KSQL?**

Links:

* <https://medium.com/@stephane.maarek/the-kafka-api-battle-producer-vs-consumer-vs-kafka-connect-vs-kafka-streams-vs-ksql-ef584274c1e>

**How to implement the scenario using Docker?**