Cloudera Runtime 7.2.14

Schema Registry Overview

Date published: 2019-11-08
Date modified: 2022-02-24

CLOUDERA

https://docs.cloudera.com/

Legal Notice

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

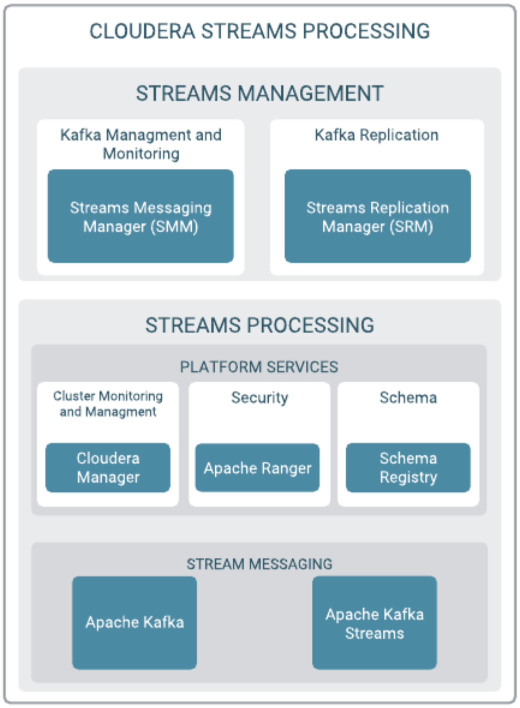
Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Schema Registry overview

As the diagram below instructs, Schema Registry is part of the enterprise services that powers streams processing.



Schema Registry provides a shared repository of schemas that allows applications to flexibly interact with each other.

Applications built often need a way to share metadata across three dimensions:

- Data format
- Schema
- Semantics or meaning of the data

The Schema Registry design principle is to provide a way to tackle the challenges of managing and sharing schemas between components. The schemas are designed to support evolution such that a consumer and producer can understand different versions of those schemas but still read all information shared between both versions and safely ignore the rest.

Hence, the value that Schema Registry provides and the applications that integrate with it are the following:

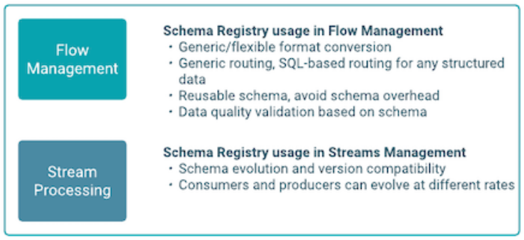
- Centralized registry

  Provide reusable schema to avoid attaching schema to every piece of data
- Version management

  Define relationship between schema versions so that consumers and producers can evolve at different rates
- Schema validation

  Enable generic format conversion, generic routing and data quality

The following image shows Schema Registry usage in Flow and Streams Management:



Schema Registry usage in Flow Management

- Generic/flexible format conversion
- Generic routing, SQL-based routing for any structured data
- Reusable schema, avoid schema overhead
- Data quality validation based on schema

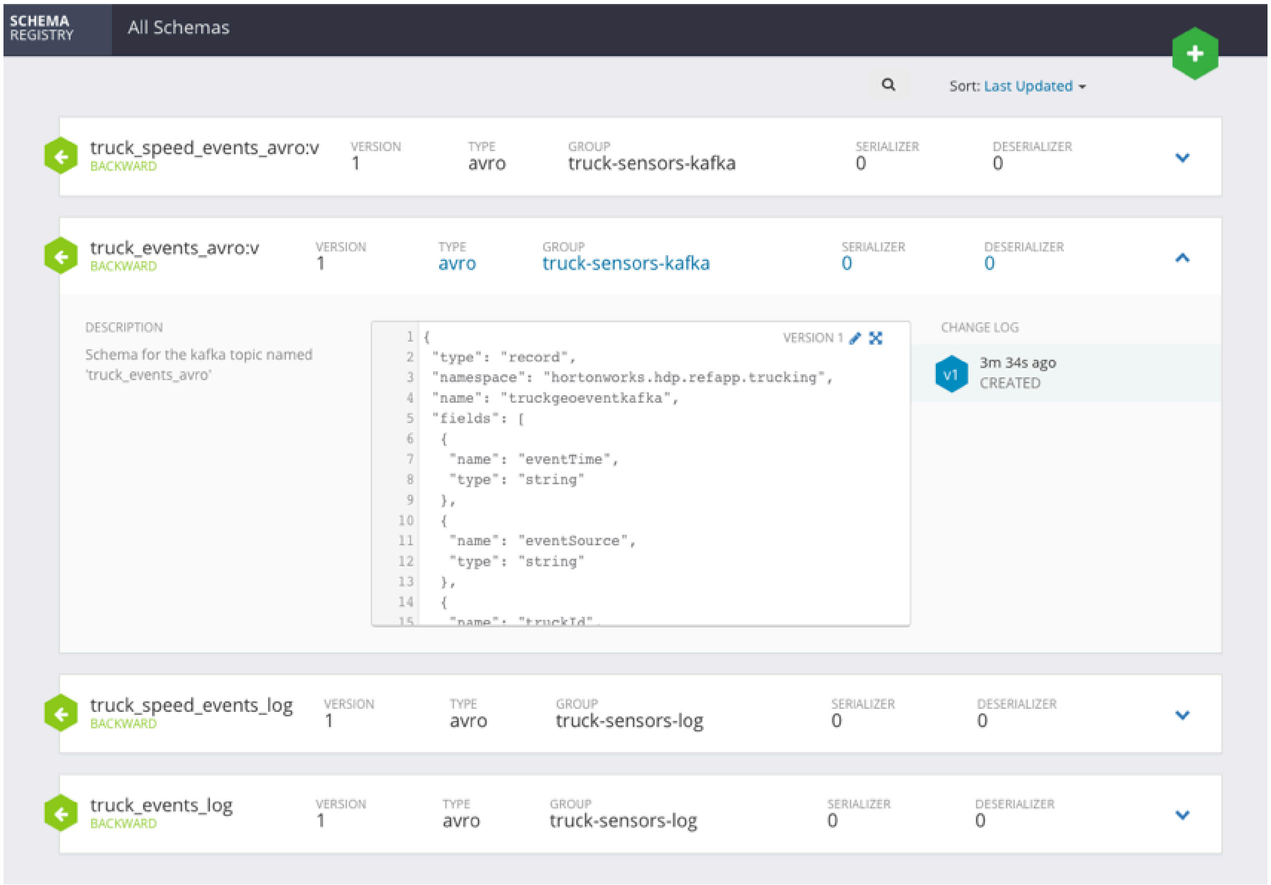
Schema Registry usage in Streams Management

- Schema evolution and version compatibility
- Consumers and producers can evolve at different rates

Examples of interacting with Schema Registry

Schema Registry UI

You can use the Schema Registry UI to create schema groups, schema metadata, and add schema versions.



truck_speed_events_avro:v BACKWARD — VERSION 1 — TYPE avro — GROUP truck-sensors-kafka — SERIALIZER 0 — DESERIALIZER 0

truck_events_avro:v BACKWARD — VERSION 1 — TYPE avro — GROUP truck-sensors-kafka — SERIALIZER 0 — DESERIALIZER 0

DESCRIPTION

Schema for the kafka topic named 'truck_events_avro'

VERSION 1

```
{
 "type": "record",
 "namespace": "hortonworks.hdp.refapp.trucking",
 "name": "truckgeoeventkafka",
 "fields": [
  {
   "name": "eventTime",
   "type": "string"
  },
  {
   "name": "eventSource",
   "type": "string"
  },
  {
   "name": "truckId",
```

CHANGE LOG

v1 3m 34s ago CREATED

truck_speed_events_log BACKWARD — VERSION 1 — TYPE avro — GROUP truck-sensors-log — SERIALIZER 0 — DESERIALIZER 0

truck_events_log BACKWARD — VERSION 1 — TYPE avro — GROUP truck-sensors-log — SERIALIZER 0 — DESERIALIZER 0

Schema Registry API

You can access the Schema Registry API Swagger documentation directly from the UI.

To do this, append your URL with: /swagger/

For example: https://localhost:7790/swagger/

Java Client

You can review the following GitHub repositories for examples of how to interact with the Schema Registry Java Client:

- https://github.com/georgevetticaden/cdf-ref-app/blob/master/csp-trucking-schema/src/main/java/cloudera/cdf/csp/schema/refapp/trucking/schemaregistry/TruckSchemaRegistryLoader.java#L62
- https://github.com/hortonworks/registry/blob/0.9.0/examples/schema-registry/avro/src/main/java/com/hortonworks/registries/schemaregistry/examples/avro/SampleSchemaRegistryClientApp.java

Java and Scala

See the following examples of using schema related API:

https://github.com/hortonworks/registry/blob/HDF-3.4.1.0-5-tag/examples/schema-registry/avro/src/main/java/com/hortonworks/registries/schemaregistry/examples/avro/SampleSchemaRegistryClientApp.java

https://github.com/hortonworks/registry/blob/HDF-2.1.0.0/schema-registry/README.md

Kafka Serdes

See the following example of using the Schema Registry Kafka Serdes:

https://github.com/hortonworks/registry/blob/0.9.0/examples/schema-registry/avro/src/main/java/com/hortonworks/registries/schemaregistry/examples/avro/KafkaAvroSerDesApp.java

Schema Registry also supports serializing objects as JSON. In order to enable JSON serialization or deserialization, please set the following property:

- value.serializer=com.hortonworks.registries.schemaregistry.serdes.json.kafka.KafkaJsonSerial
- value.deserializer=com.hortonworks.registries.schemaregistry.serdes.json.kafka.KafkaJsonDeserializer

Jackson 2 is used for serialization, so any Java object which can be processed with this library is also going to be processed by Schema Registry.

Schema Registry use cases

With a basic understanding of Schema Registry, the below sections walks through common use cases for Schema Registry.

Use case 1: Registering and querying a Schema for a Kafka topic

When Kafka is integrated into enterprise organization deployments, you typically have many different Kafka topics used by different apps and users. With the adoption of Kafka within the enterprise, some key questions that often come up are the following:

- What are the different events in a given Kafka topic?
- What do I put into a given Kafka topic?
- Do all Kafka events have a similar type of schema?
- How do I parse and use the data in a given Kafka topic?

While Kafka topics do not have a schema, having an external store that tracks this metadata for a given Kafka topic helps to answer these common questions. Schema Registry addresses this use case.

One important point to note is that Schema Registry is not just a metastore for Kafka. Schema Registry was designed to be generic schema store for any type of entity or store (log files, or similar.)

Use case 2: Reading/deserializing and writing/serializing data from and to a Kafka topic

In addition to storing schema metadata, another key use case is to store metadata for the format of how data should be read and how it should be written. Schema Registry supports this use case as well by providing capabilities to store JAR files for serializers and deserializers and then mapping the serdes to the schema.

Use case 3: Dataflow management with schema-based routing

Imagine if you are using NiFi to move different types of syslog events to downstream systems. You have data movement requirements where you need to parse the syslog event to extract the event type, and route the event to a certain downstream system (different Kafka topics, for example) based on the event type.

Without Schema Registry, NiFi uses regular expressions or other utilities to parse the event type value from the payload and store into a flowfile attribute. Then NiFi uses routing processors (RouteOnAttribute, for example) to use the parsed value for routing decisions. If the structure of the data changes considerably, this type of extract and routing pattern is brittle and requires frequent changes.

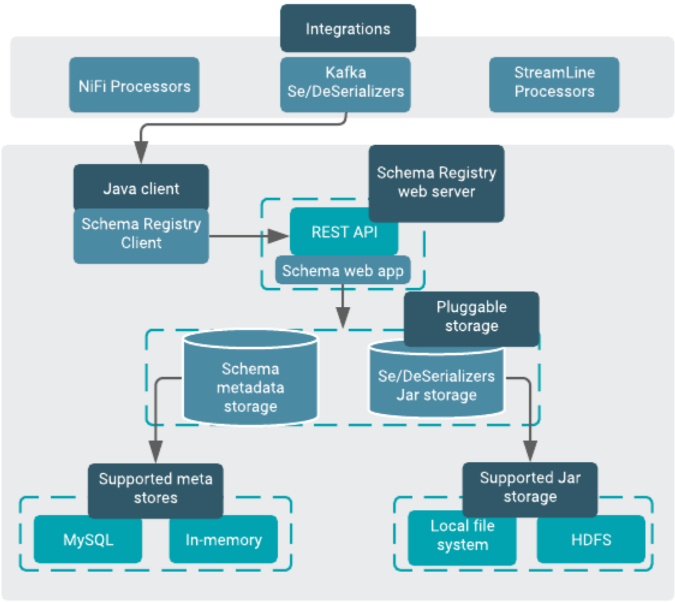
With the introduction of Schema Registry, NiFi queries the registry for schema and then retrieves the value for a certain element in the schema. In this case, even if the structure changes, as long as compatibility policies are adhered to, NiFi's extract and routing rules do not change. This is another common use case for Schema Registry.

Schema Registry component architecture

The below diagram represents the component architecture of Schema Registry.

Schema Registry has three main components:

- Registry web server – Web Application exposing the REST endpoints you can use to manage schema entities. You can use a web proxy and load balancer with multiple Web Servers to provide HA and scalability.
- Pluggable storage – Schema Registry uses the following two types of storages:
  - Schema Metadata Storage – Relational store that holds the metadata for the schema entities. Inn-memory storage (for development purposes) and mySQL databases are supported.
  - Serdes Storage – File storage for the serializer and deserializer jars. Local file system and HDFS storage are supported. Local file system storage is the default.
- Schema Registry Client – A java client that components can use to interact with the RESTful services.



There are two integration points:

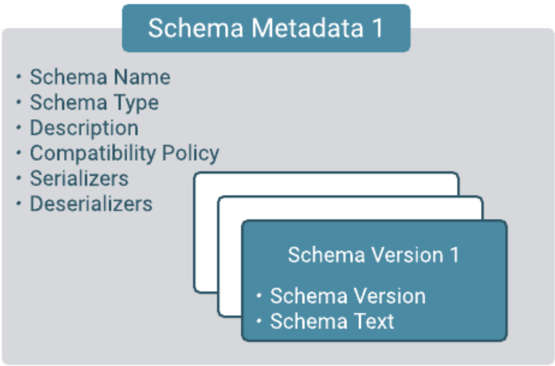
- Custom NiFi Processors – New processors and controller services in NiFi that interact with the Schema Registry.
- Kafka Serializer and Deserializer – A Kafka serializer and deserializer that uses Schema Registry. The Kafka serdes can be found on GitHub.

Schema Registry concepts

Schema entities

You can use Schema Registry to work with three types of schema entities:

The following image shows the types of Schema entities:



This table provides a more detailed description of the schema entities:

Table 1: Schema entity types

Entity Type	Description	Example
Schema Group	A logical grouping of similar schemas. A Schema Group can be based on any criteria you have for managing schemas. Schema Groups can have multiple Schema Metadata definitions.	• Group Name – truck-sensors-log • Group Name – truck-sensors-kafka

<table>
<tr><th>Entity Type</th><th>Description</th><th>Example</th></tr>
<tr><td>Schema Metadata</td><td>Metadata associated with a named schema. A metadata definition is applied to all the schema versions that are assigned to it.
Key metadata elements include:<ul><li>Schema Name – A unique name for each schema. Used as a key to look up schemas.</li><li>Schema Type – The format of the schema.
Note: Avro is currently the only supported type.</li><li>Compatibility Policy – The compatibility rules that exist when the new schemas are registered.</li><li>Serializers/Deserializers – A set of serializers and deserializers that you can upload to the registry and associate with schema metadata definitions.</li></ul></td><td><ul><li>Schema Name – truck_events_avro:v</li><li>Schema Type – avro</li><li>Compatibility Policy – SchemaCompatibility.BACKWARD</li></ul></td></tr>
<tr><td>Schema Version</td><td>The versioned schema associated a schema metadata definition.</td><td><pre>{
  "type" : "record",
  "namespace" : "hortonworks.hdp.refapp.trucking",
  "name" : "truckgeoevent",
  "fields" : [
    { "name" : "eventTime" , "type" : "string" },
    { "name" : "eventSource" , "type" : "string" },
    { "name" : "truckId" , "type" : "int" },
    { "name" : "driverId" , "type" : "int"},
    { "name" : "driverName" , "type" : "string"},
    { "name" : "routeId" , "type" : "int"},
    { "name" : "route" , "type" : "string"},
    { "name" : "eventType" , "type" : "string"},
    { "name" : "longitude" , "type" : "double"},
    { "name" : "latitude" , "type" : "double"},
    { "name" : "correlationId" , "type" : "long"}
  ]
}</pre></td></tr>
</table>

Compatibility policies

A key Schema Registry feature is the ability to version schemas as they evolve. Compatibility policies are created at the schema metadata level, and define evolution rules for each schema.

After a policy has been defined for a schema, any subsequent version updates must honor the schema’s original compatibility, otherwise you experience an error.

Compatibility of schemas can be configured with any of the below values:

Backward Compatibility

Indicates that new version of a schema would be compatible with earlier version of that schema. That means the data written from earlier version of the schema, can be deserialized with a new version of the schema.

When you have a Backward Compatibility policy on your schema, you can evolve schemas by deleting portions, but you cannot add information. New fields can be added only if a default value is also provided for them.

Forward Compatibility

Indicates that an existing schema is compatible with subsequent versions of the schema. That means the data written from new version of the schema can still be read with old version of the schema.

When you have a Forward Compatibility policy on your schema, you can evolve schemas by adding new information, but you cannot delete existing portions.

Full Compatibility

Indicates that a new version of the schema provides both backward and forward compatibilities.

None

Indicates that no compatibility policy is in place.

Note: The default value is Backward. You set the compatibility policy when you are adding a schema. Once set, you cannot change it.

Validation level

Validation level limits the scope of the compatibility check when you add a new version of the schema. It checks compatibility only against the latest version or all previous versions based on the validation level value configured. Validation level only makes sense when coupled with compatibility. When compatibility is set to None then the validation is also ignored.

The validation level cannot be set from the Schema Registry UI. You can set it through the Swagger REST API. For example,

```
SchemaMetadata.Builder builder = new SchemaMetadata.Builder("Blah")
        .type("avro")
        .schemaGroup("Kafka")
        .description("test")
        .compatibility(SchemaCompatibility.BACKWARD)
        .validationLevel(SchemaValidationLevel.ALL);
```

When using the Schema Registry UI, the validation level defaults to ALL.

The supported validation levels are as follows:

- All

  Schemas are compatible across multiple versions, for example, with Backward Compatibility, data written with version 1 can be read with version 7 too. All is the default validation level.
- Latest

  There is no transient compatibility, only the latest version is used, for example, with Backward Compatibility, data written with version 1 can only be read with version 2, not with version 7.

Allowed schema changes for different compatibilities

The following table presents a summary of the types of schema changes allowed for the different compatibility types and validation levels, for a given subject.

Compatibility types	Validation levels	Changes allowed	Check against which schemas	Upgrade first
Backward	Latest	• Delete fields • Add optional fields	Last version	Consumers
Backward	All	• Delete fields • Add optional fields	All previous versions	Consumers
Forward	Latest	• Add fields • Delete optional fields	Last version	Producers
Forward	All	• Add fields • Delete optional fields	All previous versions	Producers
Full	All	• Add optional fields • Delete optional fields	Last version	Any order
None	All	• All changes are accepted	Compatibility checking disabled	Depends