Cloudera Runtime 7.3.2

Schema Registry Overview

Date published: 2019-08-22
Date modified: 2026-03-31



https://docs.cloudera.com/

Legal Notice

© Cloudera Inc. 2026. All rights reserved.

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

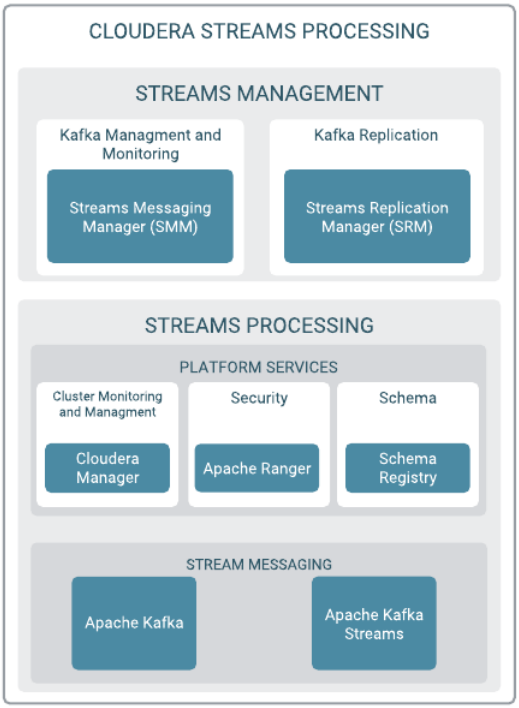
Schema Registry overview 4
Examples of interacting with Schema Registry 5
Schema Registry use cases 6
Registering and querying a schema for a Kafka topic 6
Deserializing and serializing data from and to a Kafka topic 7
Dataflow management with schema-based routing 7
Schema Registry component architecture 7
Schema Registry concepts 8
Schema entities 8
Compatibility policies 10
Importance of logical types in Avro 12

Schema Registry overview

Learn about the basic features of Schema Registry and how it integrates into Cloudera Streams Processing.

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

As displayed in the following diagram, Schema Registry is part of the enterprise services that power streams processing.



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:

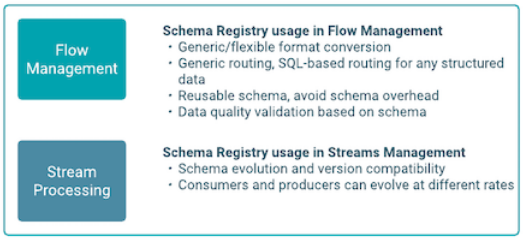
- Centralized registry

  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 displays Schema Registry usage in Flow and Streams Management:

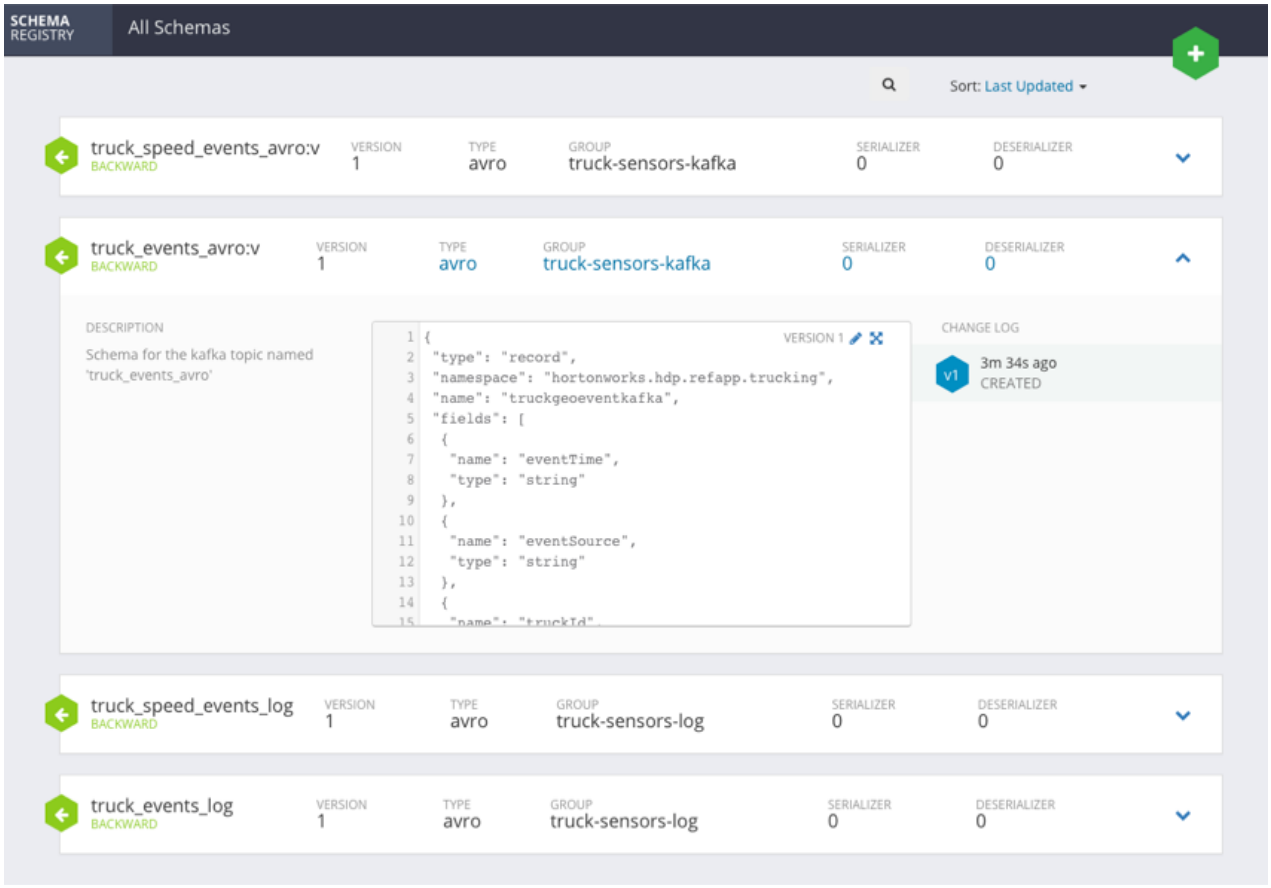


Examples of interacting with Schema Registry

Learn about different ways 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.



Schema Registry API

You can access the Schema Registry API swagger directly from the UI.

To do this, you need to append your URL with /swagger/. For example: https://localhost:7790/swagger/.

For example, you can create a schema by using Swagger, as shown in the following steps:

1. Navigate to the Swagger UI: http:<sr-host>:<sr-port>/swagger/
2. Go to the Schema section and open POST /api/v1/schemaregistry/schemas.
3. Click Try it out.

4. Enter the input for your schema in the body field.

   There is already an example input in the body. You can also edit it.
5. Click Execute.

   After the successful execution of the command, you see the newly created schema in you Schema Registry UI

For more information about the tasks you can perform through Swagger, see Cloudera Schema Registry REST API Reference.

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. To enable JSON serialization or deserialization, set the following properties:

- 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.

.NET client

See the following examples of interacting with the Schema Registry .NET client:

.NET client examples

Schema Registry use cases

Learn about different use cases of using Schema Registry.

Registering and querying a schema for a Kafka topic

Learn how to use Schema Registry to track metadata for a Kafka topic.

When Kafka is integrated into enterprise organization deployments, you typically have many different Kafka topics used by different applications 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 is designed to be a generic schema store for any type of entity or store (log files, or similar.)

Deserializing and serializing data from and to a Kafka topic

Learn how to use Schema Registry to store metadata on reading or deserializing and writing or serializing data from and to Kafka topics.

You can store metadata for the format of how data should be read and how it should be written. Schema Registry supports this use case by providing capabilities to store JAR files for serializers and deserializers, and then mapping the SerDes to the schema.

Dataflow management with schema-based routing

Learn how to use Schema Registry to support NiFi dataflow management.

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 (for example, different Kafka topics) 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 stores it into a flowfile attribute. Then NiFi uses routing processors (for example, RouteOnAttribute) 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

Learn about the component architecture of Schema Registry.

Schema Registry has the following 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 high availability and scalability.
- Pluggable storage

  Schema Registry uses the following types of storages:

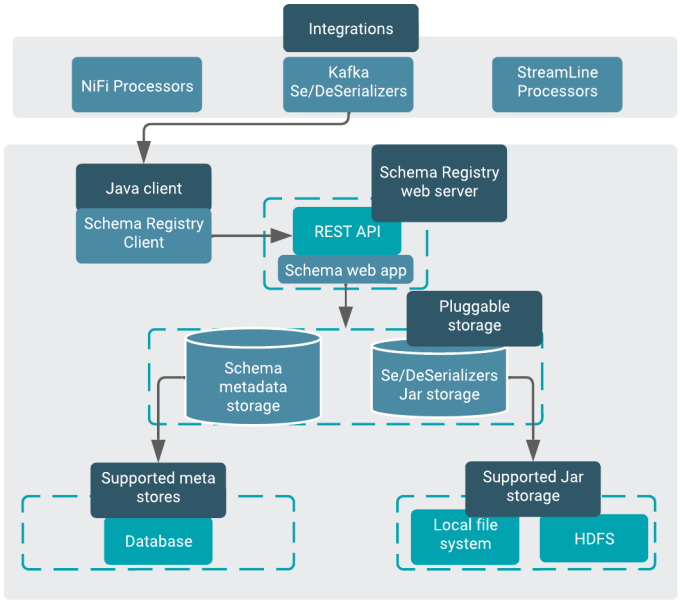
  - Schema metadata storage

    Relational store that holds the metadata for the schema entities. MySQL, PostgreSQL, and Oracle 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.

The following diagram represents the component architecture of Schema Registry.



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 or deserializer that uses Schema Registry. The Kafka SerDes can be found on GitHub.

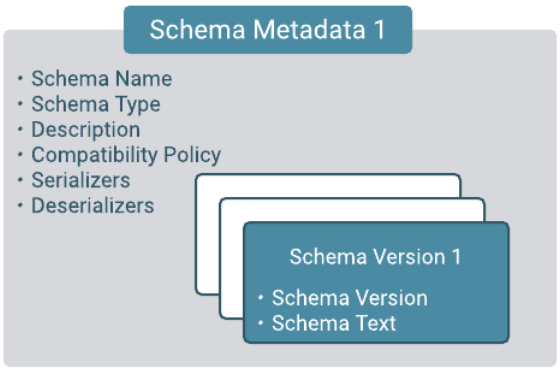
Schema Registry concepts

Learn about the basic concepts of Schema Registry.

Schema entities

Learn about the three types of schema entity types: Schema Group, Schema Metadata, Schema Version.

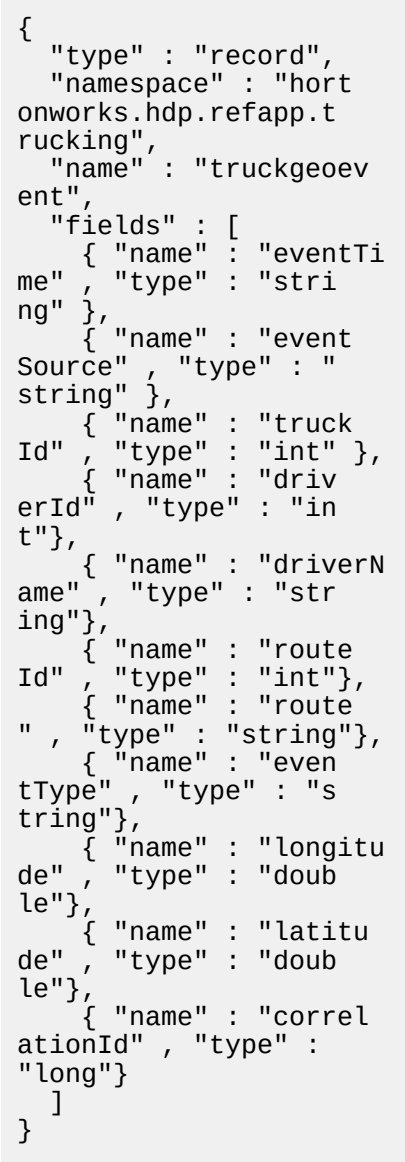
You can use Schema Registry to work with three types of schema entities. The following image shows the types of Schema entities:



The following table provides a more detailed description of the schema entity types:

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

Entity Type	Description	Example
Schema Metadata	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: • Schema Name – A unique name for each schema. Used as a key to look up schemas. • Schema Type – The format of the schema. The supported formats are Avro and JSON. • Compatibility Policy – The compatibility rules that exist when the new schemas are registered. • Serializers/Deserializers – A set of serializers and deserializers that you can upload to the registry and associate with schema metadata definitions.	• Schema Name – truck_events_avro:v • Schema Type – avro • Compatibility Policy – SchemaCompatibility.BACKWARD
Schema Version	The versioned schema associated a schema metadata definition.	

```
{
  "type" : "record",
  "namespace" : "hort
onworks.hdp.refapp.t
rucking",
  "name" : "truckgeoev
ent",
  "fields" : [
    { "name" : "eventTi
me" , "type" : "stri
ng" },
    { "name" : "event
Source" , "type" : "
string" },
    { "name" : "truck
Id" , "type" : "int" },
    { "name" : "driv
erId" , "type" : "in
t"},
    { "name" : "driverN
ame" , "type" : "str
ing"},
    { "name" : "route
Id" , "type" : "int"},
    { "name" : "route
" , "type" : "string"},
    { "name" : "even
tType" , "type" : "s
tring"},
    { "name" : "longitu
de" , "type" : "doub
le"},
    { "name" : "latitu
de" , "type" : "doub
le"},
    { "name" : "correl
ationId" , "type" :
"long"}
  ]
}
```

Compatibility policies

Learn about the compatibility policies to be able to manage versioning of schemas in Schema Registry.

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 following values:

Backward Compatibility

Indicates that a new version of a schema would be compatible with earlier versions of that schema. This means the data written from an 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 a new version of the schema can still be read with an 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 can 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

Importance of logical types in Avro

The importance of logical types in Avro format for Schema Registry can be represented with an example using a decimal logical type.

Huge numbers that cannot be stored in a long type with 8 bytes can be stored as byte arrays with decimal logical type using Avro. This means that the data will be stored as a byte array, which is flexible, but it also means that when you parse the data from Avro, you need additional information, the logical type, to interpret the raw data, the byte array, properly. By default, you will only get a byte array for this kind of field value, and you have to parse that byte array for the required logical type. Each Avro logical type has their own Java classes that they can be parsed for. Decimal logical type has BigDecimal in Java.

If you use generated classes, such as specific classes, on the classpath for Avro, the conversion happens automatically based on the types in the classes. If you do not use specific classes, Cloudera provides configuration to enable automatic conversion for logical type record fields during serialization and deserialization. Check the logical.type .conversion.enabled flag. This flag has no effect if specific.avro.reader is set to true, which means specific classes are used.

For more information, see the Logical Types section in the official Avro documentation.