Hortonworks Schema Registry 3

Schema Registry Overview

Date of Publish: 2018-08-13

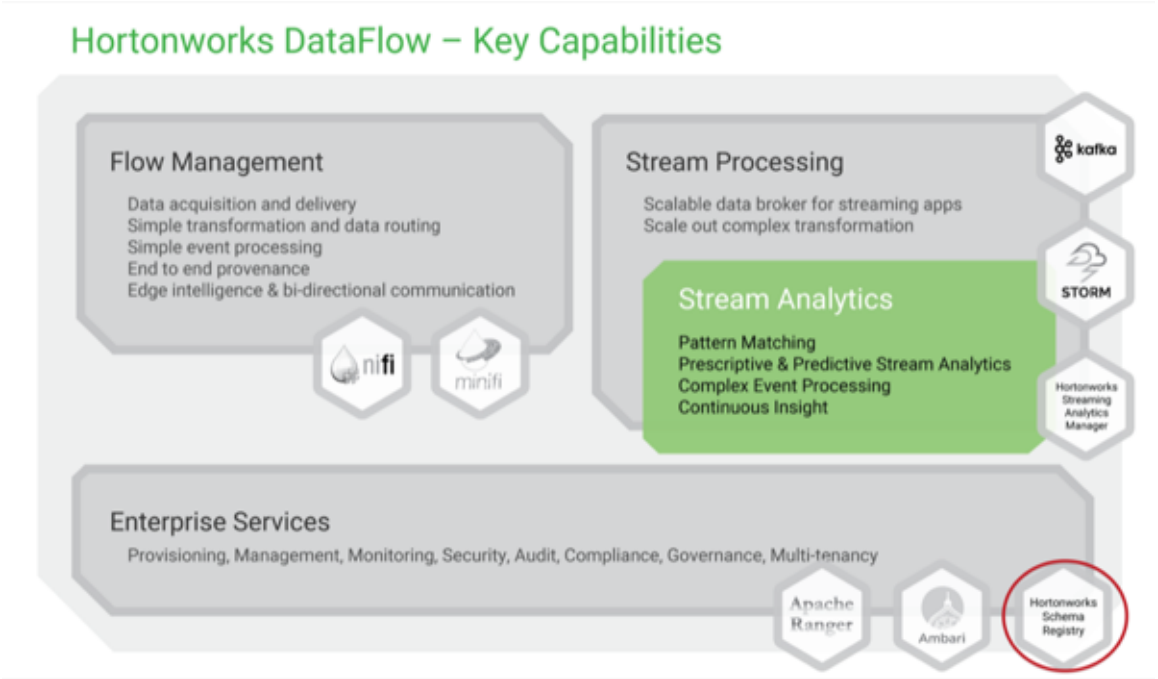
http://docs.hortonworks.com

Contents

Schema Registry Overview

The Hortonworks DataFlow Platform (HDF) provides flow management, stream processing, and enterprise services for collecting, curating, analyzing and acting on data in motion across on-premise data centers and cloud environments.

As the diagram below instructions, Hortonworks Schema Registry is part of the enterprise services that powers the HDF platform.



Schema Registry provides a shared repository of schemas that allows applications and HDF components HDF (NiFi, Storm, Kafka, Streaming Analytics Manager, and similar) to flexibly interact with each other.

Applications built using HDF often need a way to share metadata across 3 dimensions:

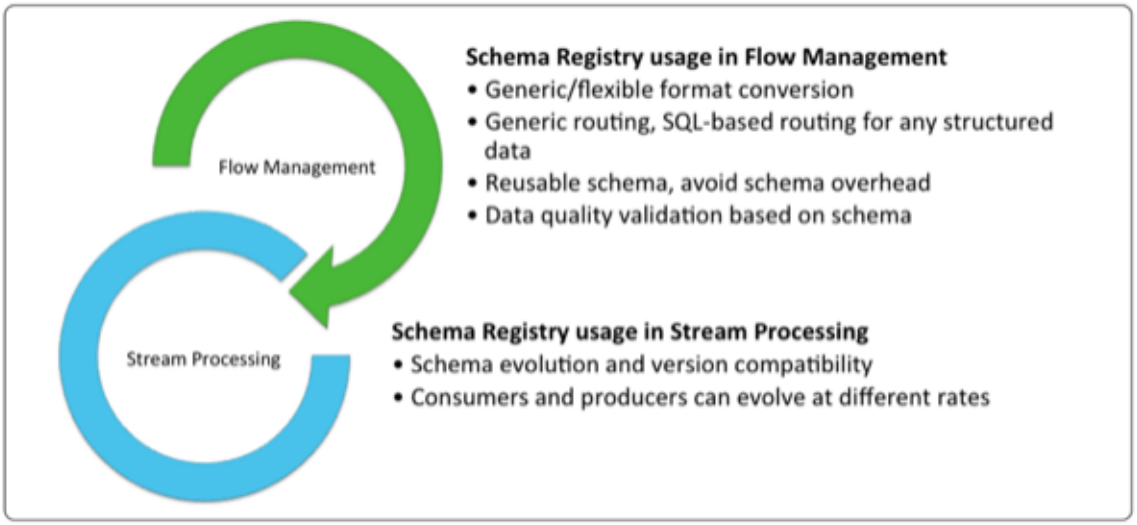
- Data format
- Schema
- Semantics or meaning of the data

The Schema Registry design principle is to provide a way to tackle the challenges of managing and sharing schemas between the components of HDF and in such a way that 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 for HDF 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

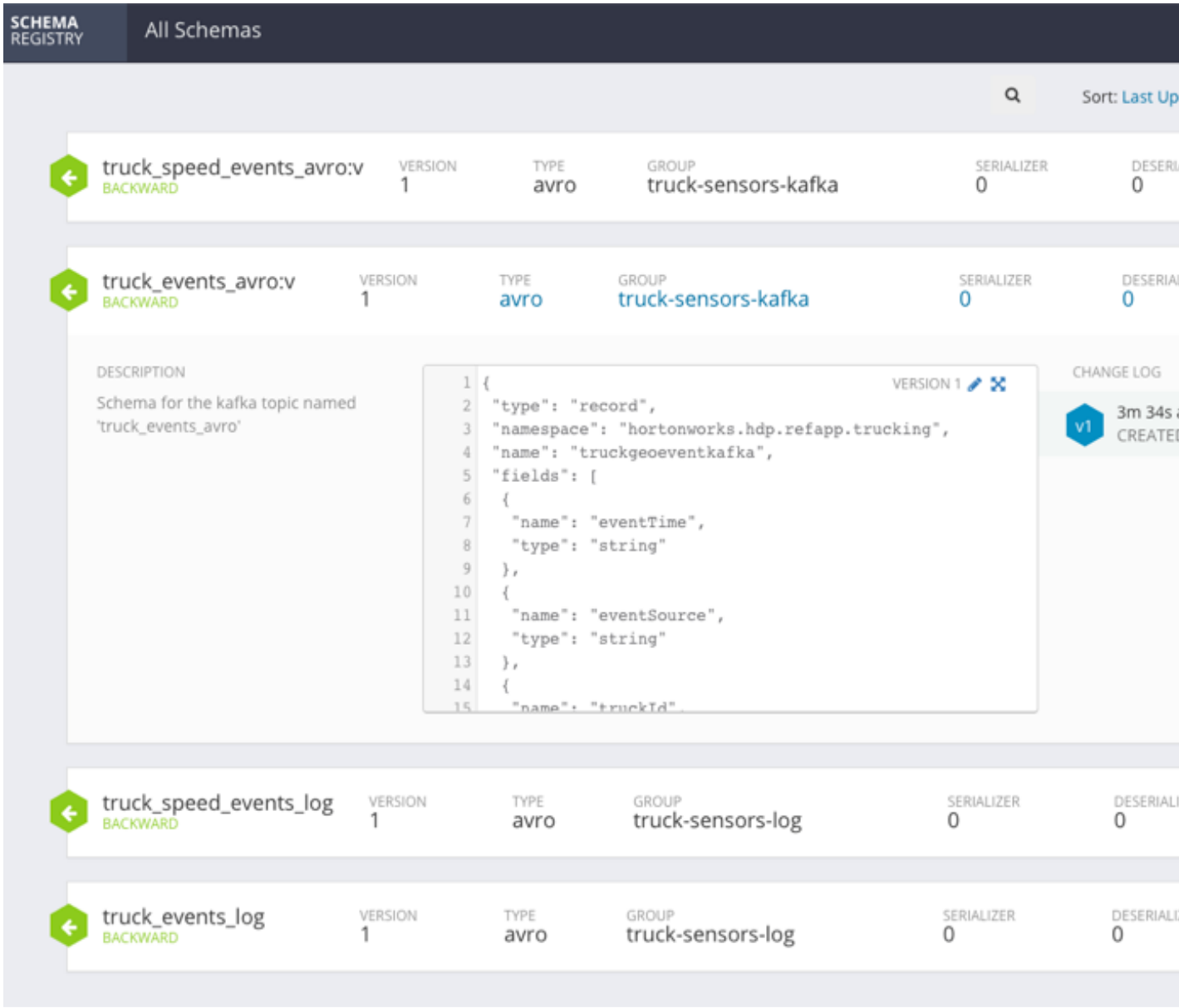
Schema Registry Usage in Flow Management



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.



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: http://localhost:9090/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/hdp/blob/master/reference-apps/iot-trucking-app/trucking-data-simulator/src/main/java/hortonworks/hdp/refapp/trucking/simulator/schemaregistry/TruckSchemaRegistryLoader.java#L48
- https://github.com/hortonworks/registry/blob/HDF-2.1.0.0/schema-registry/README.md#api-examples
- https://github.com/hortonworks/registry/blob/HDF-2.1.0.0/examples/schema-registry/avro/src/main/java/com/hortonworks/registries/schemaregistry/examples/avro/SampleSchemaRegistryClientApp.java

Kafka Serdes

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

https://github.com/hortonworks/registry/blob/HDF-2.1.0.0/examples/schema-registry/avro/src/main/java/com/hortonworks/registries/schemaregistry/examples/avro/TruckEventsKafkaAvroSerDesApp.java

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

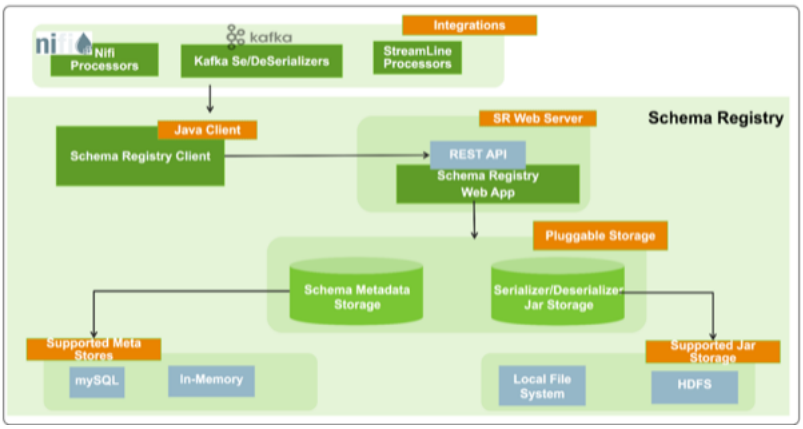
Imagine if you are using NiFi to move different types of syslog events to downstream systems. You have data movement requirements where you need to parse the syslog event to extract the event type, and route the event to a certain downstream system (different Kafka topics, for example) based on the event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

There are three integration points with HDF:

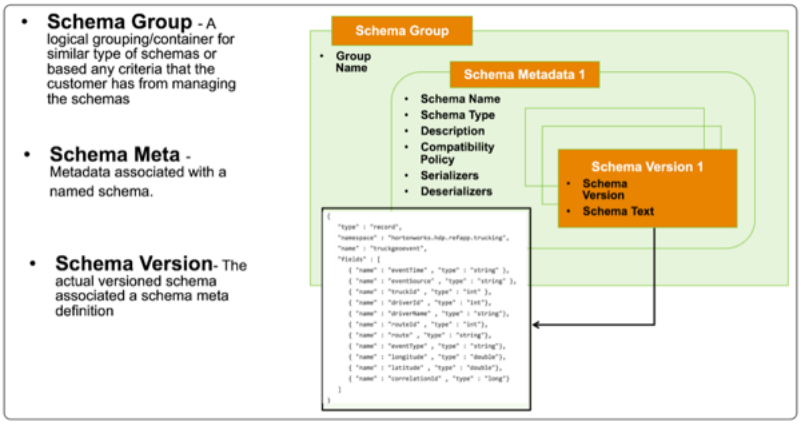
- Custom NiFi Processors – New processors and controller services in NiFi that interact with the Schema Registry.
- Kafka Serializer and Deserializer – A Kafka serializer and deserializer that uses Schema Registry. The Kafka serdes can be found on GitHub.
- Hortonworks Streaming Analytics Manager Processors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

Schema Registry Concepts

Schema Entities

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

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

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. Note: Avro is currently the only supported type. • 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.	<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>

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.
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.
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.

The default value is None.

You set the compatibility policy when you are adding a schema. Once set, you cannot change it.