# Contents

1 Installation ................................................. 3  
1.1 CentOS/RedHat ........................................ 3  
1.2 OS X .................................................. 4  
1.3 Running Schema Registry with a load balancer .......... 5  

2 SchemaRegistry ............................................. 7  
2.1 Compatibility ........................................... 7  

3 Quickstart .................................................. 9  
3.1 Installation ............................................ 9  
3.2 Running Kafka Example ................................. 9  
3.3 Download and Start Apache Kafka ...................... 9  
3.4 Run producer to register schema and send data .......... 11  
3.5 Kafka Producer Integration with SchemaRegistry ....... 11  
3.6 Run consumer to retrieve schema and deserialize the messages 12  
3.7 Kafka Consumer Integration with SchemaRegistry ...... 12  

4 API examples ............................................... 15  
4.1 Using schema related APIs ................................ 15  
4.2 Default serializer and deserializer APIs ................. 16  
4.3 Using serializer and deserializer related APIs .......... 16  
  4.3.1 Uploading jar file .................................. 16  
  4.3.2 Register serializer and deserializer ................. 17  
  4.3.3 Map serializer/deserializer with a schema .......... 17  
  4.3.4 Marshal and unmarshal using the registered serializer and deserializer for a schema 17  

5 Examples ................................................... 19  
5.1 Running Kafka Producer with AvroSerializer ............ 19  
5.2 To run the producer in Secure cluster: ................ 19  
5.3 Running Kafka Consumer with AvroDeserializer ........ 20  
5.4 To run the consumer in Secure cluster: ................. 20  

6 Running registry and streamline web-services securely .... 23  
6.1 SPNEGO ................................................ 23  
6.2 SSL ................................................... 24  

7 Serialization/Deserialization protocol ..................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Message Formats</td>
<td>27</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Message Format 1</td>
<td>27</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Message Format 2</td>
<td>27</td>
</tr>
<tr>
<td>7.2</td>
<td>Confluent protocol</td>
<td>28</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Serialization</td>
<td>28</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Deserialization</td>
<td>28</td>
</tr>
<tr>
<td>7.3</td>
<td>Schema version id as long protocol</td>
<td>28</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Serialization</td>
<td>28</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Deserialization</td>
<td>29</td>
</tr>
<tr>
<td>7.4</td>
<td>Schema version id as int protocol</td>
<td>29</td>
</tr>
<tr>
<td>7.5</td>
<td>Schema metadata id and version protocol</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Roadmap</td>
<td>31</td>
</tr>
<tr>
<td>8.1</td>
<td>Schema Registry</td>
<td>31</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Schema Improvements</td>
<td>31</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Collaboration</td>
<td>31</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Security</td>
<td>32</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Integration</td>
<td>32</td>
</tr>
<tr>
<td>8.1.5</td>
<td>Operations</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>Competition</td>
<td>33</td>
</tr>
</tbody>
</table>
1.1 CentOS/RedHat

1. Install Java

```
# wget --no-cookies --no-check-certificate --header "Cookie: gpw_e24=http%3A%2F
   →download.oracle.com/otn-pub/java/jdk/8u111-b14/jdk-8u111-linux-x64.tar.gz"
# tar xzf jdk-8u111-linux-i586.tar.gz
# cd /opt/jdk1.8.0_111/
# alternatives --install /usr/bin/java java /opt/jdk1.8.0_111/bin/java 2
# alternatives --config java
```

2. Download latest Schema Registry binaries from here https://github.com/hortonworks/registry/releases

3. edit $REGISTRY_HOME/conf/registry-env.sh, add the following

```
export JAVA_HOME=/opt/jdk1.8.0_111/
```

4. Setup Database

**4.1 Mysql**

**Install Mysql**

```
sudo yum install mysql-server
sudo mysql-server start
```
Configure Mysql

```bash
# set root password
sudo mysql_secure_installation
```

Setup Database

```bash
mysql -u root -p
cREATE USER 'registry_user'@'localhost' IDENTIFIED BY 'registry_password';
GRANT ALL PRIVILEGES ON schema_registry.* TO 'registry_user'@'localhost' WITH
   GRANT OPTION;
commit;
```

5. Configure registry.yaml

```bash
cp conf/registry-mysql-example.yaml conf/registry.yaml
```

Edit the following section to add appropriate database and user settings

```yaml
storageProviderConfiguration:
  providerClass: "com.hortonworks.registries.storage.impl.jdbc.JdbcStorageManager"
  properties:
    db.type: "mysql"
    queryTimeoutInSecs: 30
    db.properties:
      dataSourceClassName: "com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
      dataSource.url: "jdbc:mysql://localhost/schema_registry"
      dataSource.user: "registry_user"
      dataSource.password: "registry_password"
```

6. Run bootstrap scripts

```bash
$REGISTRY_HOME/bootstrap/bootstrap-storage.sh
```

7. Start the registry server

```bash
sudo ./bin/registry start
```

1.2 OS X

1. Download latest Schema Registry binaries from here [https://github.com/hortonworks/registry/releases](https://github.com/hortonworks/registry/releases)
2. edit `$REGISTRY_HOME/conf/registry-env.sh`, add the following

```bash
export JAVA_HOME=$(/usr/libexec/java_home -v 1.8)
```
3. Setup Database

3.1 Mysql
Install Mysql

```bash
brew install mysql
launchctl load ~/Library/LaunchAgents/homebrew.mxcl.mysql.plist
export MYSQL_PATH=/usr/local/Cellar/mysql/5.6.27
export PATH=$PATH:$MYSQL_PATH/bin
```

Configure Mysql

```bash
mysqladmin -u root password 'yourpassword'
mysql -u root -p
```

Setup Database

```sql
mysql -u root -p
create database schema_registry;
CREATE USER 'registry_user'@'localhost' IDENTIFIED BY 'registry_password';
GRANT ALL PRIVILEGES ON schema_registry.* TO 'registry_user'@'localhost' WITH
    GRANT OPTION;
commit;
```

4. Configure registry.yaml

```bash
cp conf/registry-mysql-example.yaml conf/registry.yaml
```

Edit the following section to add appropriate database and user settings

```yaml
storageProviderConfiguration:
    providerClass:"com.hortonworks.registries.storage.impl.jdbc.JdbcStorageManager"
    properties:
        db.type: "mysql"
        queryTimeoutInSecs: 30
        db.properties:
            dataSourceClassName: "com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
            dataSource.url: "jdbc:mysql://localhost/schema_registry"
            dataSource.user:"registry_user"
            dataSource.password: "registry_password"
```

5. Run bootstrap scripts

```bash
$REGISTRY_HOME/bootstrap/bootstrap-storage.sh
```

6. Start the registry server

```bash
sudo ./bin/registry start
```

1.3 Running Schema Registry with a load balancer

One or more schema registry instances can be put behind a load balancer for reverse proxying, in that case appropriate schema registry url

must be mentioned in the load balancer’s configuration file. For example, in case of Apache mod proxy the VirtualHost tag in the configuration file should be edited out with the following

1.3. Running Schema Registry with a load balancer
<VirtualHost *:80>
  <Proxy balancer://mycluster>
    BalancerMember http://127.0.0.1:9090 <!-- First schema registry server -->
    BalancerMember http://127.0.0.2:9090 <!-- Second schema registry server -->
  </Proxy>
</VirtualHost>

ProxyPreserveHost On

ProxyPass / balancer://mycluster/ ProxyPassReverse / balancer://mycluster/

In case of serializers and deserializers the parameter “schema.registry.url” can be pointed to the loader balancer’s url or it can be a list of schema registry servers with “schema.registry.client.url.selector” set to one of “FailoverUrlSelector”, “LoadBalanced-FailoverUrlSelector” or “RoundRobinUrlSelector”. The parameter “schema.registry.client.url.selector” defines the retry strategy in the case the currently picked schema registry server from the list of schema registry servers is not reachable.
CHAPTER 2

SchemaRegistry

SchemaRegistry provides a central repository for a message’s metadata. A schema specifies the message structure and type. Schema Registry will allow us to store these schemas efficiently and provides a pluggable serializer/deserializer interfaces and run-time provision of serializer/deserializer implementations based on incoming messages. Schema registry will also enable reuse/discovery/authoring/collaboration related to schemas.

Each Schema is mainly represented with metadata like

- **name** - Name of the schema which is unique across the schema registry.
- **type** - Represents the type of schema. For ex Avro, ProtoBuf, Json etc
- **schemaGroup** - Group of schemas in which this schema belongs to. It can be like Kafka, Hive, Spark or system log etc
- **compatibility** - Compatibility between different versions of the schema.
- **description** - Description about the different versions of a schema.

Each of these schemas can evolve with multiple versions. Each version of the Schema can have

- **schemaText** - Textual representation of schema
- **description** - Description about this version

2.1 Compatibility

Compatibility of different versions of a schema can be configured with any of the below values

- **Backward** - It 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.
- **Forward** - It 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** - It indicates that a new version of the schema provides both backward and forward compatibilities.
- **None** - There is no compatibility between different versions of a schema.
CHAPTER 3

Quickstart

3.1 Installation

1. Download the latest release from here

2. Registry server can be started with in-memory store or a persistent store like mysql. To setup with mysql please follow the instructions here.

3. To start with in-memory store.

   ```bash
   cp $REGISTRY_HOME/conf/registry-inmemory-example.yaml $REGISTRY_HOME/conf/registry.yaml
   # start the server in fore-ground
   $REGISTRY_HOME/bin/registry-server-start conf/registry.yaml
   # To start in daemon mode
   sudo ./bin/registry start
   
   4. Access the UI at http://host.name:9090

3.2 Running Kafka Example

SchemaRegistry makes it very easy to integrate with Kafka, Storm and Nifi and any other systems. We’ve an example code on how to integrate with kafka here.

To run this example, follow the steps below

3.3 Download and Start Apache Kafka

1. Download kafka 0.10.0.1 or higher from here.

2. $KAFKA_HOME/bin/zookeeper-server-start.sh config/zookeeper.properties
Fig. 1: SchemaRegistry UI
3. $KAFKA_HOME/bin/kafka-server-start.sh config/server.properties
4. $KAFKA_HOME/bin/kafka-topics.sh –zookeeper localhost:2181 –topic truck_events_stream –partitions 1 –replication-factor 1 –create

3.4 Run producer to register schema and send data

1. cd $REGISTRY_HOME/examples/schema-registry/avro
2. To send messages to topic “truck_events_stream”
   ```
   java -jar avro-examples-0.1.0-SNAPSHOT.jar -d data/truck_events.csv -p data/kafka-˓
   → _producer.props -sm -s data/truck_events.avsc
   ```

3.5 Kafka Producer Integration with SchemaRegistry

1. Any client integration code must make a dependency on schema-registry-serdes
   ```
   <dependency>
   <groupId>com.hortonworks.registries</groupId>
   <artifactId>schema-registry-serdes</artifactId>
   <dependency>
   ```
2. For KafkaProducer, user need to add the following config
   ```
   config.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
   config.put(SchemaRegistryClient.Configuration.SCHEMA_REGISTRY_URL.name(), props.˓
   → .get(SchemaRegistryClient.Configuration.SCHEMA_REGISTRY_URL.name()));
   config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.˓
   → getName());
   config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.˓
   → getName());
   ```

Important settings from the above are schema.registry.url:

This should be set to where the registry server is running ex: http://localhost:9090/api/v1

key.serializer: StringSerializer is used in the above example.

value.serializer: com.hortonworks.registries.schemaregistry.serdes.avro.kafka.KafkaAvroSerializer is used in the above example. This serializer has integration with schema registry. It will take the producer config and retrieves schema.registry.url and the topic name to find out the schema. If there is no schema defined it will publish a first version of that schema.

3. For KafkaProducer, to save the schema version information in the Record Header, user need to include the following config:
   ```
   config.put(KafkaAvroSerializer.STORE_SCHEMA_VERSION_ID_IN_HEADER, "true");
   config.put(KafkaAvroSerde.KEY_SCHEMA_VERSION_ID_HEADER_NAME, "key.schema.version.id");
   → // optional
   config.put(KafkaAvroSerde.VALUE_SCHEMA_VERSION_ID_HEADER_NAME, "value.schema.version.˓
   → _id"); // optional
   ```

store.schema.version.id.in.header By default, this is set to ‘false’ to maintain backward compatibility. User needs to enable it to save the schema version information in the header.
key_schema_version_id_header_name Configurable header name to save the Record Key schema version information. This configuration is applicable only when key.serializer is set to ‘KafkaAvroSerializer’.

value_schema_version_id_header_name Configurable header name to save the Record Value schema version information. This configuration is applicable only when value.serializer is set to ‘KafkaAvroSerializer’.

### 3.6 Run consumer to retrieve schema and deserialize the messages

1. cd $REGISTRY_HOME/examples/schema-registry/avro
2. To consume messages from topic “truck_events_stream”
   
   ```
   java -jar avro-examples-0.5.0-SNAPSHOT.jar -cm -c data/kafka-consumer.props
   press ctrl + c to stop
   ```

### 3.7 Kafka Consumer Integration with SchemaRegistry

1. Any client integration code must make a dependency on schema-registry-serdes

   ```
   <dependency>
   <groupId>com.hortonworks.registries</groupId>
   <artifactId>schema-registry-serdes</artifactId>
   </dependency>
   ```

2. For KafkaConsumer, user need to add the following to config

   ```
   config.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
   config.put(SchemaRegistryClient.Configuration.SCHEMA_REGISTRY_URL.name(), props.get(SCHEMA_REGISTRY_URL));
   config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
   config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class.getName());
   ```

   Important settings from the above are

   **schema.registry.url**: This should be set to where the registry server is running ex: http://localhost:9090/api/v1

   **key.deserializer**: `StringDeserializer` is used in the above example.

   **value.deserializer**: `com.hortonworks.registries.schemaregistry.serdes.avro.kafka.KafkaAvroDeserializer` is used in the above example.

   This deserializer tries to find schema.id in the message payload.

   3. For KafkaConsumer, to retrieve the schema version information from the Record Header, user may have to include the below configs, if they were supplied in the KafkaProducer.

   ```
   config.put(KafkaAvroSerde.KEY_SCHEMA_VERSION_ID_HEADER_NAME, "key.schema.version.id"); // optional
   config.put(KafkaAvroSerde.VALUE_SCHEMA_VERSION_ID_HEADER_NAME, "value.schema.version.id"); // optional
   ```

   KafkaAvroDeserializer tries to find schema.id from the message header. If it’s not available, tries to find the schema.id from the message payload. So, a topic can contain messages that can hold version information (schema.id) either in the header / payload.
If the deserializer finds schema.id, makes a call to schema registry to fetch the avro schema. If it doesn’t find schema.id it falls back to using topic name to fetch a schema.
API examples

4.1 Using schema related APIs

Below set of code snippets explain how SchemaRegistryClient can be used for - registering new versions of schemas - fetching registered schema versions - registering serializers/deserializers - fetching serializer/deserializer for a given schema

```java
String schema1 = getSchema("/device.avsc");
SchemaMetadata schemaMetadata = createSchemaMetadata("com.hwx.schemas.sample-" + 
  System.currentTimeMillis());

// registering a new schema
SchemaIdVersion v1 = schemaRegistryClient.addSchemaVersion(schemaMetadata, new 
  SchemaVersion(schema1, "Initial version of the schema"));
LOG.info("Registered schema [{}] and returned version [{}]", schema1, v1);

// adding a new version of the schema
String schema2 = getSchema("/device-next.avsc");
SchemaVersion schemaInfo2 = new SchemaVersion(schema2, "second version");
SchemaIdVersion v2 = schemaRegistryClient.addSchemaVersion(schemaMetadata, 
  schemaInfo2);
LOG.info("Registered schema [{}] and returned version [{}]", schema2, v2);

// adding same schema returns the earlier registered version
SchemaIdVersion version = schemaRegistryClient.addSchemaVersion(schemaMetadata, 
  schemaInfo2);
LOG.info("Received version [{}] for schema metadata [{}]", version, schemaMetadata);

// get a specific version of the schema
String schemaName = schemaMetadata.getName();
SchemaVersionInfo schemaVersionInfo = schemaRegistryClient.getSchemaVersionInfo(new 
  SchemaVersionKey(schemaName, v2.getVersion()));
LOG.info("Received schema version info [{}] for schema metadata [{}]", 
  schemaVersionInfo, schemaMetadata);
```

(continues on next page)
// get latest version of the schema
SchemaVersionInfo latest = schemaRegistryClient.
    getLatestSchemaVersionInfo(schemaName);
LOG.info("Latest schema with schema key [{}] is : [{}]", schemaMetadata, latest);

// get all versions of the schema
Collection<SchemaVersionInfo> allVersions = schemaRegistryClient.
    getAllVersions(schemaName);
LOG.info("All versions of schema key [{}] is : [{}]", schemaMetadata, allVersions);

// finding schemas containing a specific field
SchemaFieldQuery md5FieldQuery = new SchemaFieldQuery.Builder().name("md5").build();
    findSchemasByFields(md5FieldQuery);
LOG.info("Schemas containing field query [{}] : [{}]", md5FieldQuery,
    md5SchemaVersionKeys);

SchemaFieldQuery txidFieldQuery = new SchemaFieldQuery.Builder().name("txid").build();
    findSchemasByFields(txidFieldQuery);
LOG.info("Schemas containing field query [{}] : [{}]", txidFieldQuery,
    txidSchemaVersionKeys);

// deleting a schema metadata and all data associated with it including versions, branches, etc
schemaRegistryClient.deleteSchema(schemaName);

4.2 Default serializer and deserializer APIs.

Default serializer and deserializer for a given schema provider can be retrieved with the below APIs.

```java
// for avro,
AvroSnapshotSerializer serializer = schemaRegistryClient.
    getDefaultSerializer(AvroSchemaProvider.TYPE);
AvroSnapshotDeserializer deserializer = schemaRegistryClient.
    getDefaultDeserializer(AvroSchemaProvider.TYPE);
```

4.3 Using serializer and deserializer related APIs

Registering serializer and deserializer is done with the below steps

- Upload jar file which contains serializer and deserializer classes and its dependencies
- Register serializer/deserializer
- Map serializer/deserializer with a registered schema
- Fetch Serializer/Deserializer and use it to marshal/unmarshal payloads.

4.3.1 Uploading jar file
String serdesJarName = "/serdes-examples.jar";
InputStream serdesJarInputStream = SampleSchemaRegistryApplication.class.getResourceAsStream(serdesJarName);
if (serdesJarInputStream == null) {
    throw new RuntimeException("Jar " + serdesJarName + " could not be loaded");
}
String fileId = schemaRegistryClient.uploadFile(serdesJarInputStream);

4.3.2 Register serializer and deserializer

String simpleSerializerClassName = "org.apache.schemaregistry.samples.serdes.SimpleSerializer";
String simpleDeserializerClassName = "org.apache.schemaregistry.samples.serdes.SimpleDeserializer";
SerDesPair serializerInfo = new SerDesPair("simple-serializer-deserializer",
"simple serializer and deserializer",
fileId,
simpleSerializerClassName,
simpleDeserializerClassName);
Long serDesId = schemaRegistryClient.addSerDes(serializerInfo);

4.3.3 Map serializer/deserializer with a schema

// map serializer and deserializer with schema key
// for each schema, one serializer/deserializer is sufficient unless someone want to
// maintain multiple implementations of serializers/deserializers
String schemaName = ...
schemaRegistryClient.mapSchemaWithSerDes(schemaName, serializerId);

4.3.4 Marshal and unmarshal using the registered serializer and deserializer for a schema

SnapshotSerializer<Object, byte[], SchemaMetadata> snapshotSerializer = ...
    getSnapshotSerializer(schemaMetadata);
String payload = "Random text: " + new Random().nextLong();
byte[] serializedBytes = snapshotSerializer.serialize(payload, schemaMetadata);

SnapshotDeserializer<byte[], Object, Integer> snapshotDeserializer = ...
    getSnapshotDeserializer(schemaMetadata);
Object deserializedObject = snapshotDeserializer.deserialize(serializedBytes, null);
CHAPTER 5

Examples

SchemaRegistry comes with examples of integration into Kafka. You can find the code here.

5.1 Running Kafka Producer with AvroSerializer

1. Login into one of the Kafka broker hosts
2. bin/kafka-topics.sh --zookeeper zk:2181 --topic truck_events_stream --partitions 2 --replication-factor 1 --create
3. On registry host; cd /usr/hdf/2.1.0.0-164/registry/examples/schema-registry/avro/
4. Edit data/kafka-producer.props

<table>
<thead>
<tr>
<th>topic=truck_events_stream</th>
<th>bootstrap.servers=kafka_host1:6667,kafka_host2:6667</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema.registry.url=http://registry_host:9090/api/v1</td>
<td>security.protocol=PLAINTEXT</td>
</tr>
<tr>
<td>ignoreInvalidMessages=true</td>
<td></td>
</tr>
</tbody>
</table>

5. java -jar avro-examples-0.5.0-SNAPSHOT.jar -d data/truck_events_json -p data/kafka-producer.props -sm -s data/truck_events.avsc
6. The above command will register truck_events schema in data/truck_events.avsc into registry and ingests 200 messages into topic “truck_events_stream”

5.2 To run the producer in Secure cluster:

1. Issue ACLs on the topic you are trying to ingest
2. create kafka topic bin/kafka-acls.sh –authorizer kafka.security.auth.SimpleAclAuthorizer –authorizer-properties
   zookeeper.connect=pm-eng1-cluster2.field.hortonworks.com:2181 –add
   –allow-principal User:principal_name –allow-host “*” –operation All –topic truck_events_stream
   2.1 make sure you replace “principal_name” with the username you are trying to ingest
3. edit data/kafka-producer.props, add security.protocol=SASL_PLAINTEXT

| topic=truck_events_stream  
| bootstrap.servers=kafka_host1:6667,kafka_host2:6667  
| schema.registry.url=http://registry_host:9090/api/v1  
| security.protocol=SASL_PLAINTEXT  
| key.serializer=org.apache.kafka.common.serialization.StringSerializer  
| value.serializer=com.hortonworks.registries.schemaregistry.serdes.avro.kafka.KafkaAvroSerializer |

4. Create following kafka_client_jaas.conf to pass to the Kafka Producer's JVM

KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required
  useTicketCache=true
  renewTicket=true
  service_name="kafka";
}

In the above config we are expecting Kafka brokers running with principal "kafka".

5. kinit -kt your.keytab principal@EXAMPLE.COM. Make sure you gave ACLs to the principal refer to [2]

6. java -Djava.security.auth.login.config=/etc/kafka/conf/kafka_client_jaas.conf -jar avro-examples-0.5.0-SNAPSHOT.jar -d data/truck_events_json -p data/kafka-producer.props -sm -s data/truck_events.avsc

5.3 Running Kafka Consumer with AvroDeserializer

1. Edit data/kafka-consumer.props

| topic=truck_events_stream  
| bootstrap.servers=localhost:9092  
| schema.registry.url=http://localhost:9090/api/v1  
| security.protocol=PLAINTEXT  
| key.deserializer=org.apache.kafka.common.serialization.StringDeserializer  
| value.deserializer=com.hortonworks.registries.schemaregistry.serdes.avro.kafka.KafkaAvroDeserializer  
| group.id=truck_group  
| auto.offset.reset=earliest |

2. java -jar avro-examples-0.5.0-SNAPSHOT.jar -c data/kafka-consumer.props -cm -s data/truck_events.avsc

5.4 To run the consumer in Secure cluster:

1. Edit data/kafka-consumer.props

| topic=truck_events_stream  
| bootstrap.servers=pm-eng1-cluster4.field.hortonworks.com:6667  
| schema.registry.url=http://pm-eng1-cluster4.field.hortonworks.com:7788/api/v1 |
security.protocol=SASL_PLAINTEXT
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=com.hortonworks.registries.schemaregistry.serdes.avro.kafka.KafkaAvroDeserializer
group.id=truck_group
auto.offset.reset=earliest

2. Create following kafka_client_jaas.conf to pass to the Kafka Producer’s JVM

```java
KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required
  useTicketCache=true
  renewTicket=true
  serviceName="kafka";
}
```

In the above config we are expecting Kafka brokers running with principal "kafka".

3. kinit -kt your.keytab principal@EXAMPLE.COM. Make sure you gave ACLs to the principal refer to [2]

4. java -Djava.security.auth.login.config=/etc/kafka/conf/kafka_client_jaas.conf -jar avro-examples-0.5.0-SNAPSHOT.jar -c data/kafka-consumer.props -cm

5.4. To run the consumer in Secure cluster:
Running registry and streamline web-services securely

6.1 SPNEGO

This module is intended to be used by registry and streamline web-services so that they can enable http client authentication via SPNEGO. The supported authentication mechanism is Kerberos. The code for this module has been borrowed from the hadoop-auth(2.7.3) module in Hadoop project and slightly modified. The reasons for doing so are to avoid having a dependency on hadoop-auth module which brings in some other modules, avoid conflicts with other versions of hadoop-auth module and having more control over the changes needed in future. Some text for this document has been borrowed from SECURITY.md of Apache Storm.

By default, registry and streamline web-services are running with authentication disabled and therefore anyone can access the web-services from ui/client as far as they know the url and can access the web-server from the client machine. To enable client authentication, webservice needs to add a servlet filter from this module. The webservice module will need to declare a dependency on this module. One way of adding a servlet filter in code is as follows.

```java
List<ServletFilterConfiguration> servletFilterConfigurations = registryConfiguration.
    getServletFilters();
if (servletFilterConfigurations != null && !servletFilterConfigurations.isEmpty()) {
    for (ServletFilterConfiguration servletFilterConfiguration: servletFilterConfigurations) {
        try {
            FilterRegistration.Dynamic dynamic = environment.servlets().
                addFilter(servletFilterConfiguration.getClassName(), (Class<? extends Filter>)
                Class.forName(servletFilterConfiguration.getClassName()));
            dynamic.setInitParameters(servletFilterConfiguration.getParams());
            dynamic.addMappingForUrlPatterns(EnumSet.allOf(DispatcherType.class),
            true, "/*");
        } catch (Exception e) {
            LOG.error("Error registering servlet filter ",
            servletFilterConfiguration);
            throw new RuntimeException(e);
        }
    }
}
```
In the above code, ServletFilterConfiguration is a Java object representing the servlet filter specified using the registry YAML file as shown in the example below. However the general idea is that one needs to add com.hortonworks.registries.auth.server.AuthenticationFilter for enabling authentication.

The filter configuration is passed using the params property in the YAML file, as follows:

```yaml
servletFilters:
  - className: "com.hortonworks.registries.auth.server.AuthenticationFilter"
    params:
      type: "kerberos"
      kerberos.principal: "HTTP/web-service-host.com"
      kerberos.keytab: "/path/to/keytab"
      kerberos.name.rules: "RULE:[2:$1@$0](\[jt\]t@[.*EXAMPLE.COM]s/.*/$MAPRED_USER/\)
                   RULE:[2:$1@$0](\[nd\]n@[.*EXAMPLE.COM]s/.*/$HDFS_USER/DEFAULT"
      token.validity: 36000
```

The servlet filter uses the principal `HTTP/{hostname}` to login (hostname must be the host where the web-service runs). Make sure that principal is created as part of Kerberos setup.

Once configured, the user must do `kinit` on client side using the principal declared before accessing the web-service via the browser or some other client. This principal also needs to be created first during Kerberos setup.

Here’s an example on how to access the web-service after the setup above:

```bash
```

1. Firefox: Go to `about:config` and search for `network.negotiate-auth.trusted-uris` double-click to add value “http://”.

2. Google-chrome: start from command line with:
   ```bash
google-chrome --auth-server-whitelist="*web-service-hostname" --auth-negotiate-delegate-whitelist="*"*
```

3. IE: Configure trusted websites to include “web-service-hostname” and allow negotiation for that website.

**Caution:** In AD MIT Kerberos setup, the key size is bigger than the default UI jetty server request header size. If using MIT Kerberos with jetty server, make sure you set HTTP header buffer bytes to 65536.

### 6.2 SSL

This section talks about enabling SSL for Registry Server. Below steps mention about how to generate self signed certificates and use them with Registry Server.

1. Run the following to create a self-signed entry in the keystore.jks. The alias selfsigned can be anything you want.
   ```bash
   # keytool -genkey -alias selfsigned -keyalg RSA -keystore keystore.jks -keysize 2048
   ```

2. Export the certificate to selfsigned.crt with:
   ```bash
   # keytool -export -alias selfsigned -file selfsigned.crt -keystore keystore.jks
   ```

3. Import that certificate into your cacerts, the default Java keystore. You may need to do this as root, or with sudo.
   Go to the `/jre/lib/security` directory, and run:
   ```bash
   # keytool -import -trustcacerts -alias selfsigned -file selfsigned.crt -keystore <path_to_java>/cacerts
   ```
4. Registry config for the server can be configured like below.

```yaml
server:
  applicationConnectors:
    - type: https
      port: 8443
      keyStorePath: ./conf/keystore.jks
      keyStorePassword: test12
      validateCerts: false
      validatePeers: false
  adminConnectors:
    - type: https
      port: 8444
      keyStorePath: ./conf/keystore.jks
      keyStorePassword: test12
      validateCerts: false
      validatePeers: false
```

5. When you start the server, you can access via https on the port 8443.
Serializer and Deserializer are used to marshal and unmarshal messages according to a given schema by adding schema version information either in the Message Header [or] along with the payload. There are different protocols and/or formats in how schema version are added with the Message. This framework allows to build custom ser/des protocols by users.

### 7.1 Message Formats

In Kafka v0.11.0.0, Record Header support is introduced. With this support, user can optionally save the schema information in the header rather than prepending it in the body of the message. (See Message Format 2)

#### 7.1.1 Message Format 1

|header:| |body:<version_info><payload>| This is the default format to maintain backward compatibility. Prepends the schema version information along with the user payload in the body of the Message.

#### 7.1.2 Message Format 2

|header:<version_info>| |body:<payload>| Adds the schema version information in the Message Header and keeps the user payload as it’s in the body of the Message.

```java
public byte[] serialize(String topic, Headers headers, Object data) {
    final MessageAndMetadata message = serializer.serialize(data, schemaMetadata);
    headers.add("value.schema.version.id", message.metadata());
    return message.payload();
}
```
7.2 Confluent protocol

Protocol-id: 0

7.2.1 Serialization

version-info: version identifier as integer type, so it would be of 4 bytes.

payload:

if schema type is byte array then write byte array as it is.
else follow the below process to serialize the payload.

if the given payload is of specific-record type then use specific datum writer along with binary encoder
else use generic datum writer along with binary encoder.

7.2.2 Deserialization

version-info: version identifier as integer type, so it would be of 4 bytes.

Get the respective avro schema for the given schema version id which will be writer schema. User Deserializer API can take any schema version which can be treated as reader schema.

payload:

if schema type is byte array then read byte array as it is.
else follow the below process to deserialize the payload.

if the given payload is of specific-record type then use specific datum writer
else use generic datum writer along with binary encoder.

if user api asks to read it as specific-record then use specific datum reader passing both writer and reader schemas
else use generic datum reader passing both writer and reader schemas.

Java implementation is located at serialization/deserialization and protocol.

7.3 Schema version id as long protocol

Protocol-id: 2

7.3.1 Serialization

version-info: long value which represents schema version id, viz 8 bytes

payload:

if schema type is byte array then write byte array as it is.
else follow the below process to serialize the payload.

if the given payload is of specific-record type then use specific datum writer along with binary encoder.
else use generic datum writer along with binary encoder.
7.3.2 Deserialization

version-info: long value which represents schema version id, viz 8 bytes

Get the respective avro schema for the given schema version id which will be writer schema. User Deserializer API can take any schema version which can be treated as reader schema.

payload:
  
  if schema type is byte array  then read byte array as it is
  
  else if schema type is string  then generate UTF-8 string from the remaining bytes
  
  else  follow the below process to deserialize the payload
  
  if the given payload is of specific-record type  then use specific datum writer along with binary encoder.
  
  else  use generic datum writer along with binary encoder.

  if user api asks to read it as specific-record  then use specific datum reader passing both writer and reader schemas.

  else  use generic datum reader passing both writer and reader schemas.

Java implementation is located at serialization/deserialization and protocol.

7.4 Schema version id as int protocol

Protocol-id: 3

This protocol’s serialization and deserialization of payload process is similar to Schema version id as long protocol except the schema version id is treated as int and it falls back to long when it is more than max integer value.

Java implementation is located at serialization/deserialization and protocol.

7.5 Schema metadata id and version protocol

Protocol-id: 1

This protocol’s serialization and deserialization of payload process is similar to Schema version id as long protocol except the version info contains both schema metadata id and version number.

version-info: <metadata-id><version>

metadata-id: long value which represents schema metadata id, viz 8 bytes

version: int value of version, viz 4 bytes

Java implementation is located at serialization/deserialization and protocol.
We are working on making Registry a true Central Metadata Repository that can serve Schemas, ML Docs, Service Discovery and Application Configs. Here is our roadmap that we are working for next versions. As always, as the community needs evolve, this list may be reprioritized and changes may be made. We greatly appreciate the community feedback on this roadmap.

### 8.1 Schema Registry

#### 8.1.1 Schema Improvements

1. Rich Data types

   With Avro and other formats, Users get standard types including primitives and in Avro’s case Record etc. With Rich Data types we are looking to add Rich-type decorator document on top of Schema. This Decorator doc will define how to semantically interpret the schema fields. Rich-types can be both system types and user-defined types and come from a hierarchical taxonomy such as String -> Date -> Specific Date format.

   For example if you have Latitude, Longitude fields in the schema, decorator doc can have a field saying Location = [Latitude, Longitude], similarly if the schema has date field, decorator can have a field saying Date = [“DD/MM/YYYY”]. This will give much better integration of schemas in application and an important context around each field. The context can be used for data discovery, and also to maintain tools related to such data types such as validation tools, conversion etc.

#### 8.1.2 Collaboration

1. Notifications

   Allow users to subscribe to a schema of interest for any version changes. The changes to a schema will be updated via Email.

2. Schema Review & Management
Currently, we allow users to programmatically register a schema and it will be immediately visible to all the clients. With this option, we will allow schema to be staged & reviewed. Once approved will be available to production users.

3. Audit Log

Record all the usages of a schema and add Heartbeat mechanism on client side. This will allow us to show case all the clients that are using a particular Schema. With this option, users will be able to see how many clients are currently using the schema and which version they depend upon.

4. Schema Life Cycle Management

Schemas are immutable, once created they can be extended but cannot be deleted. But there are often cases where the clients are not any more using schemas and having archiving option will allow users to keep the schema not available and still be in the system, if in case they need to refer or put back in production.

5. Improved UI

We will continue to improve and provide a better collaboration and management of Schemas through UI. This helps in authoring, discovering, monitoring and managing the schemas.

8.1.3 Security

1. Authentication

As part of 0.3 release, we added SPNEGO auth to authenticate users. As part of authentication improvements we will add SSL and OAUTH 2.0.

2. Authorization

Implement Schema & Sub-Schema level authorization. This will allow users to fine-tune the control of Schema on who is allowed to access/edit etc..

8.1.4 Integration

1. Multi-lang Client support for Registry

SchemaRegistryClient can only be used in Java based applications. We are working on adding support of Python and in future extend that to other languages.

2. Pluggable Listeners

Allows users to write a simple plugin to fetch metadata from other stores such as Hive Metastore, Confluent Schema Registry etc..

3. Converters

SchemaRegistry uses Avro as the format with an option of extending it to other formats such as Protobuf etc. With converters we provide an option for users to convert their csv, xml, json to Registry format such as Avro, Protobuf etc..

8.1.5 Operations

1. Cross-Colo Mirroring

Mirroring will allow users to have a independent registry cluster per data center and allow syncing of schemas. This will allow users to keep the registry servers closer to their clients and keep the schemas in-sync across the clusters.
Schema Registry is only one of the types of registries this project supports. On the same foundation, other registries such as MLRegistry, Rules, ApplicationConfigs etc. can be built. The foundation offers StorageManager, REST API structure and a Versioned Entity store upon which other modules are built.

Specifically in SchemaRegistry, we are providing a solution that can capture various formats of Schema and provide pluggable serializer/deserializers for schemas.

Any clients such as Storm, Spark, Nifi, Kafka and other projects can easily use the schema-registry-client to query schema registry to serialize/deserialize the messages.

A frequently asked question is, how Schema Registry from this project is different from Confluent Schema Registry. While we don’t have a clear roadmap of Confluent Schema Registry, here are the current capabilities of both. You may want to look at future roadmap.
## Features

<table>
<thead>
<tr>
<th>Features</th>
<th>Registry</th>
<th>Confluent Schema Registry</th>
</tr>
</thead>
<tbody>
<tr>
<td>REST API for Schema Management</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Supported Schema Types</td>
<td>AVRO (0.3), PROTOBUF, JSON, extensible…</td>
<td>AVRO only</td>
</tr>
<tr>
<td>Pluggable Serializer/Deserializer</td>
<td>Yes</td>
<td>No, uses Avro parser</td>
</tr>
<tr>
<td>Storage</td>
<td>Pluggable Storage (Mysql, Postgres)</td>
<td>Uses Kafka as storage</td>
</tr>
<tr>
<td>Registry Client to support multi platform</td>
<td>Registry client to interact with schema registry</td>
<td>No. Developed for Kafka producer/consumer APIs only</td>
</tr>
<tr>
<td>HA</td>
<td>No Master, Multi Webserver deployment</td>
<td>Single Master Architecture Depends on zookeeper</td>
</tr>
<tr>
<td>UI</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Collaboration</td>
<td>In future- See Roadmap</td>
<td>?</td>
</tr>
<tr>
<td>Search</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>