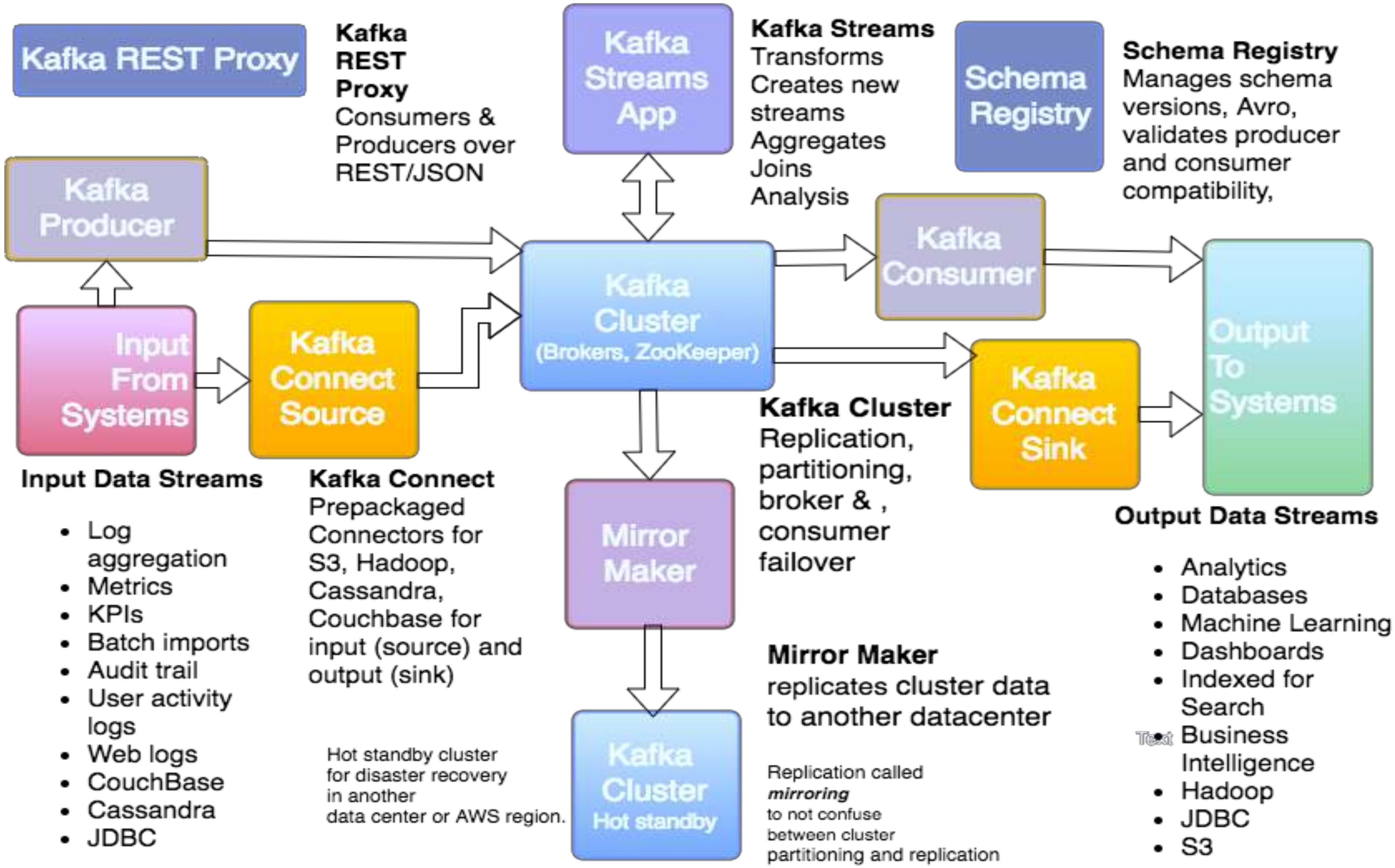
Kafka Ecosystem

Kafka Universe

Tos



Kafka REST Proxy
Consumers & Producers over REST/JSON

Kafka Streams
Transforms
Creates new streams
Aggregates
Joins
Analysis

Schema Registry
Manages schema versions, Avro, validates producer and consumer compatibility,

Kafka Cluster
Replication, partitioning, broker & , consumer failover

Input Data Streams

- Log aggregation
- Metrics
- KPIs
- Batch imports
- Audit trail
- User activity logs
- Web logs
- CouchBase
- Cassandra
- JDBC

Kafka Connect
Prepackaged Connectors for S3, Hadoop, Cassandra, Couchbase for input (source) and output (sink)

Hot standby cluster
for disaster recovery
in another
data center or AWS region.

Mirror Maker
replicates cluster data to another datacenter

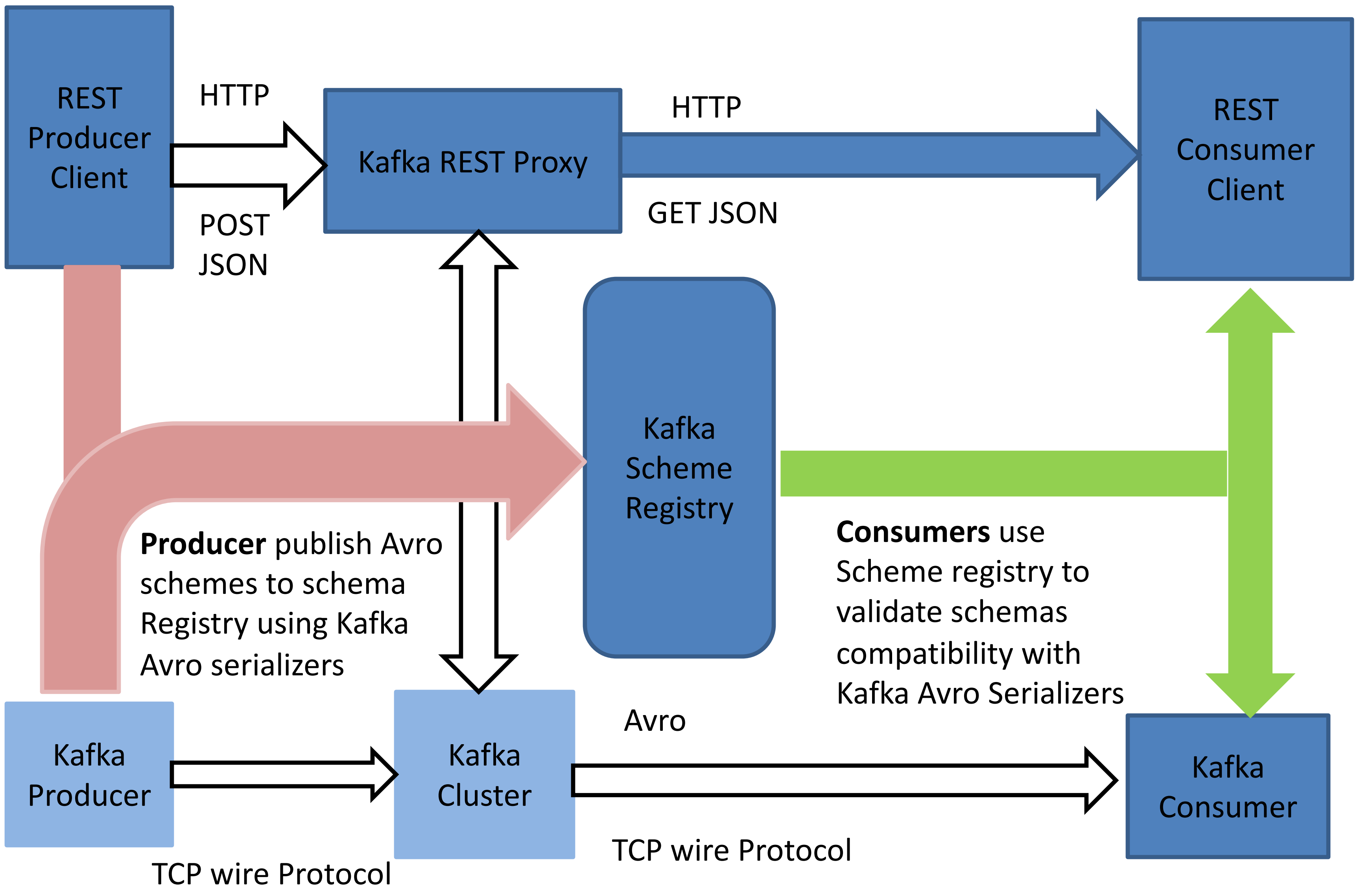
Replication called
mirroring
to not confuse
between cluster
partitioning and replication

Output Data Streams

- Analytics
- Databases
- Machine Learning
- Dashboards
- Indexed for Search
- Business Intelligence
- Hadoop
- JDBC
- S3

Kafka REST Proxy and Schema Registry

Tos



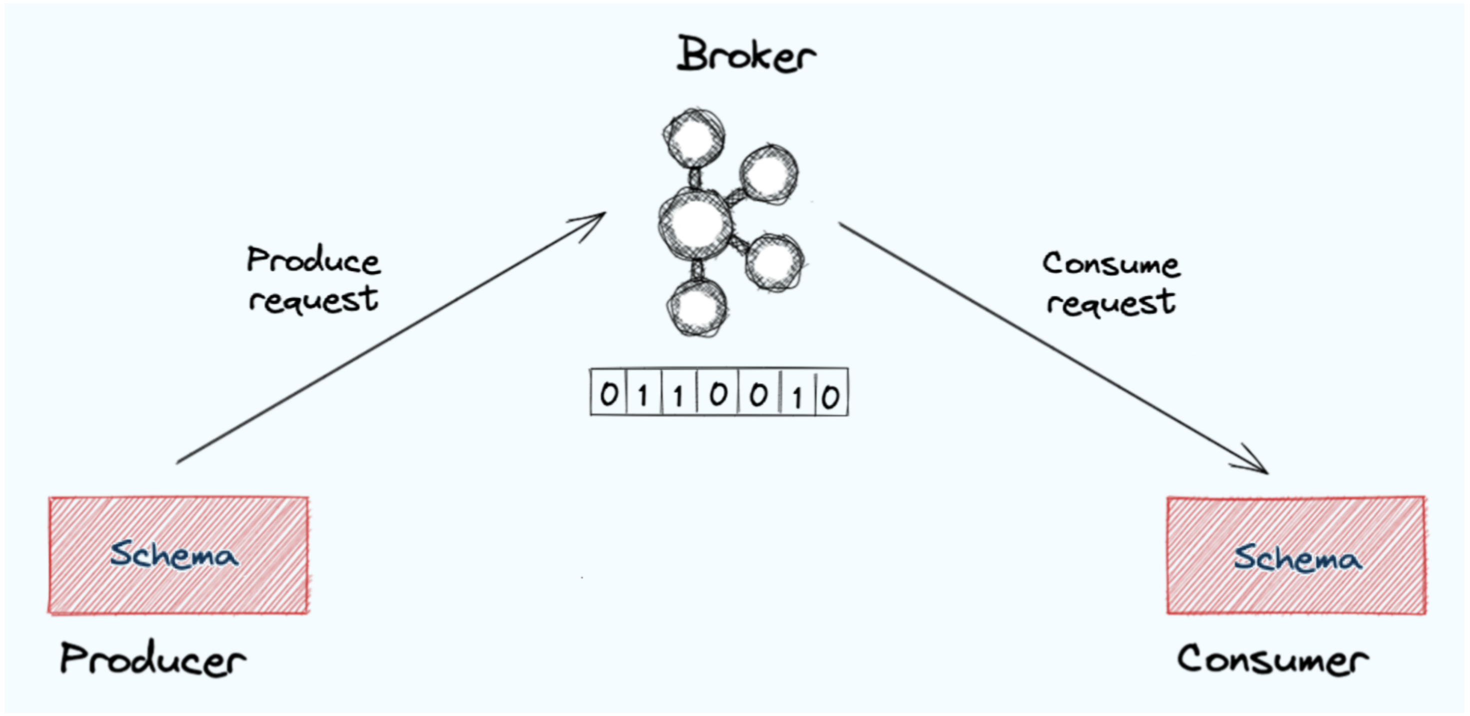
Tos

Schema Registry

Confluent Schema Registry

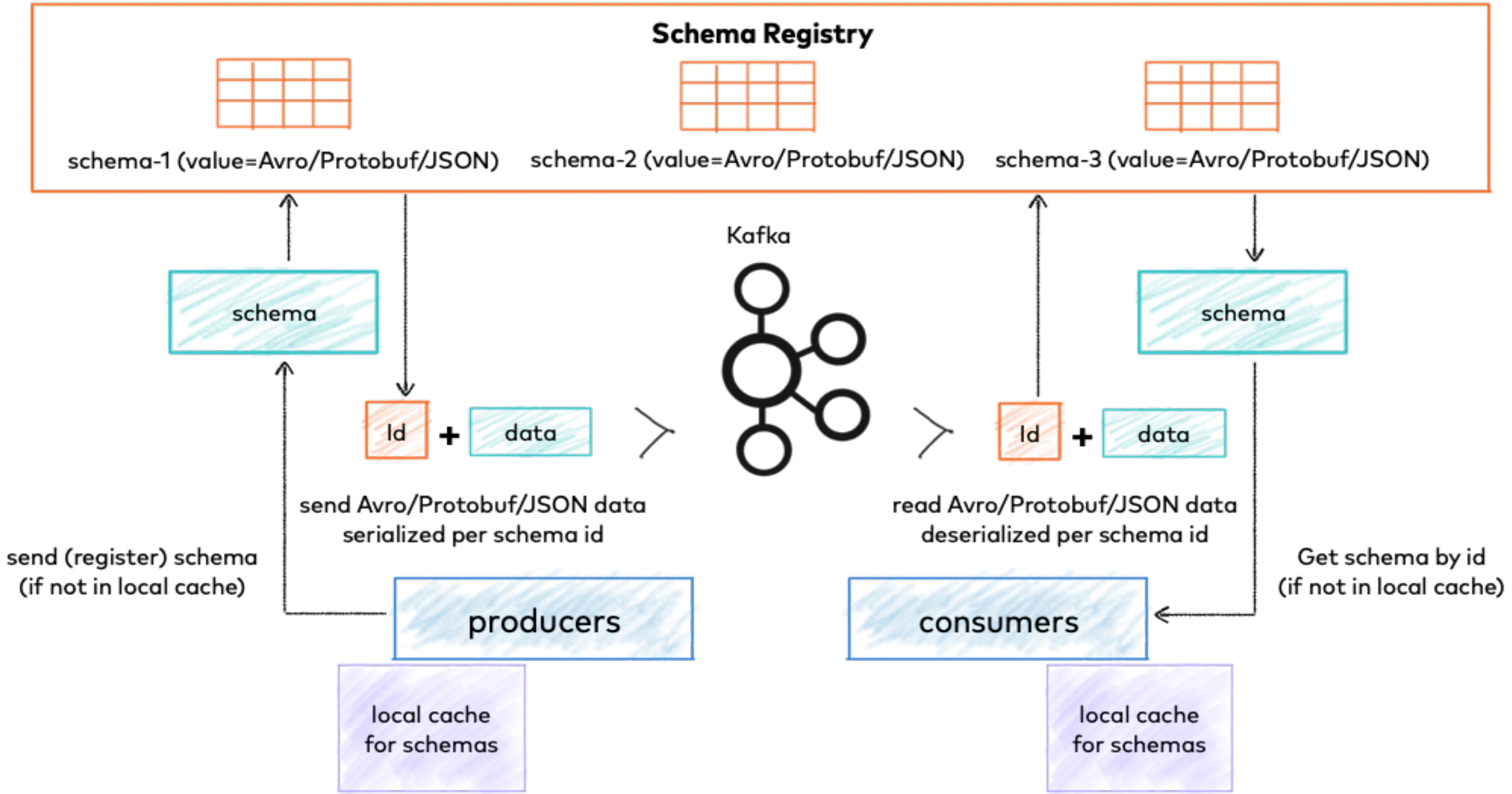
Confluent Schema Registry provides a serving layer for your metadata.
It provides a RESTful interface for storing and retrieving your Avro®, JSON Schema, and Protobufschemas.
It stores a versioned history of all schemas based on a specified subject name strategy, provides multiple compatibility settings and allows evolution of schemas according to the configured compatibility settings and expanded support for these schema types.
It provides serializers that plug into Apache Kafka® clients that handle schema storage and retrieval for Kafka messages that are sent in any of the supported formats.

Schema Is the Contract



If either your producer or consumer fails to meet the rules established by the contract, there needs to be a consequence.
This contract is codified in something called a schema.
A schema is a set of rules that establishes the format of the messages being sent.
It outlines the structure of the message, the names of any fields, what data types they contain, and any other important details.
This schema is a contract between the two applications

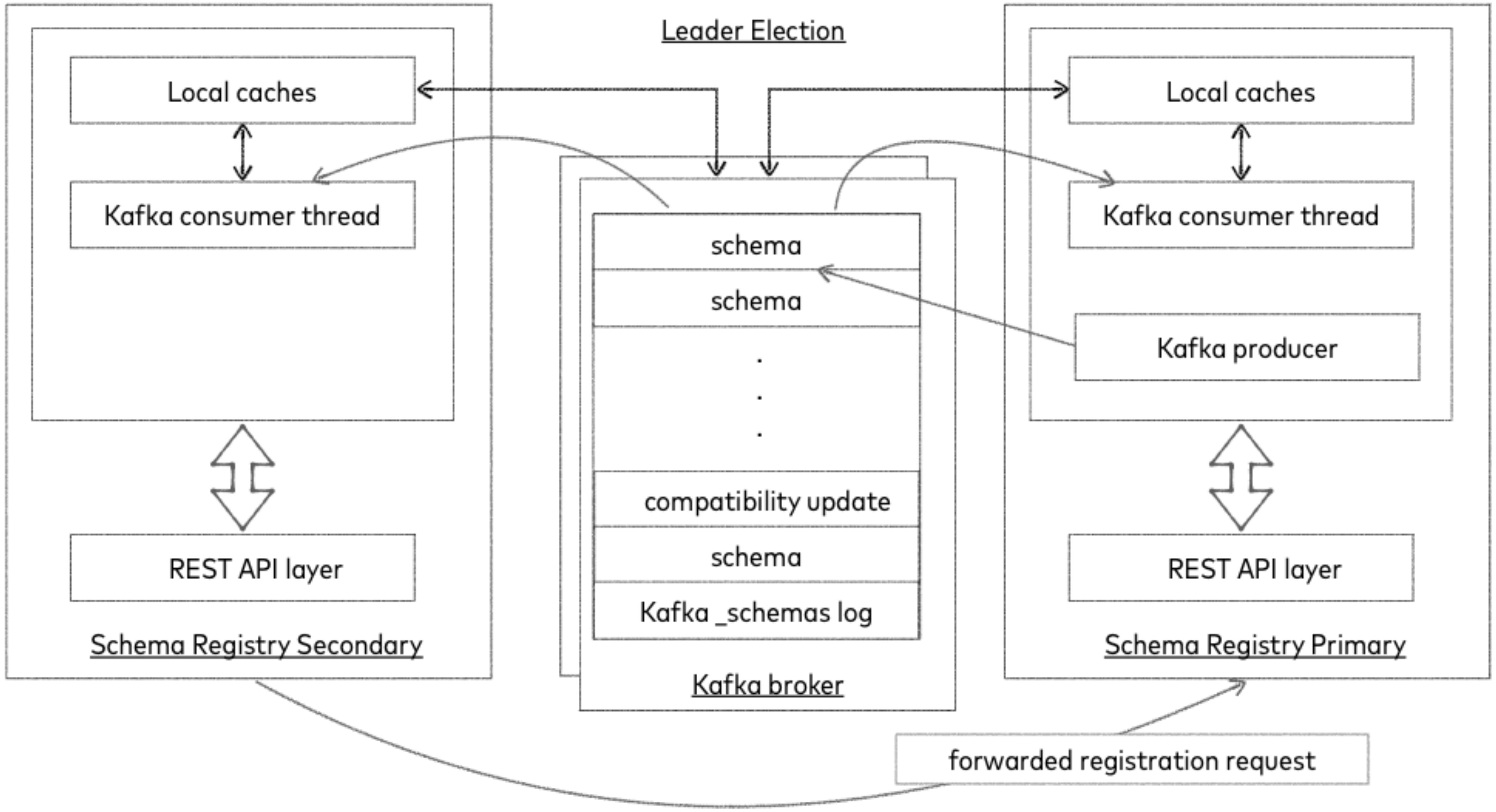
Confluent Schema Registry


Schema Registry
schema-1 (value=Avro/Protobuf/JSON)
schema-2 (value=Avro/Protobuf/JSON)
schema-3 (value=Avro/Protobuf/JSON)
Kafka
schema
schema
Id
+
data
Id
+
data
send Avro/Protobuf/JSON data serialized per schema id
read Avro/Protobuf/JSON data deserialized per schema id
send (register) schema (if not in local cache)
Get schema by id (if not in local cache)
producers
consumers
local cache for schemas
local cache for schemas

Schema Registry

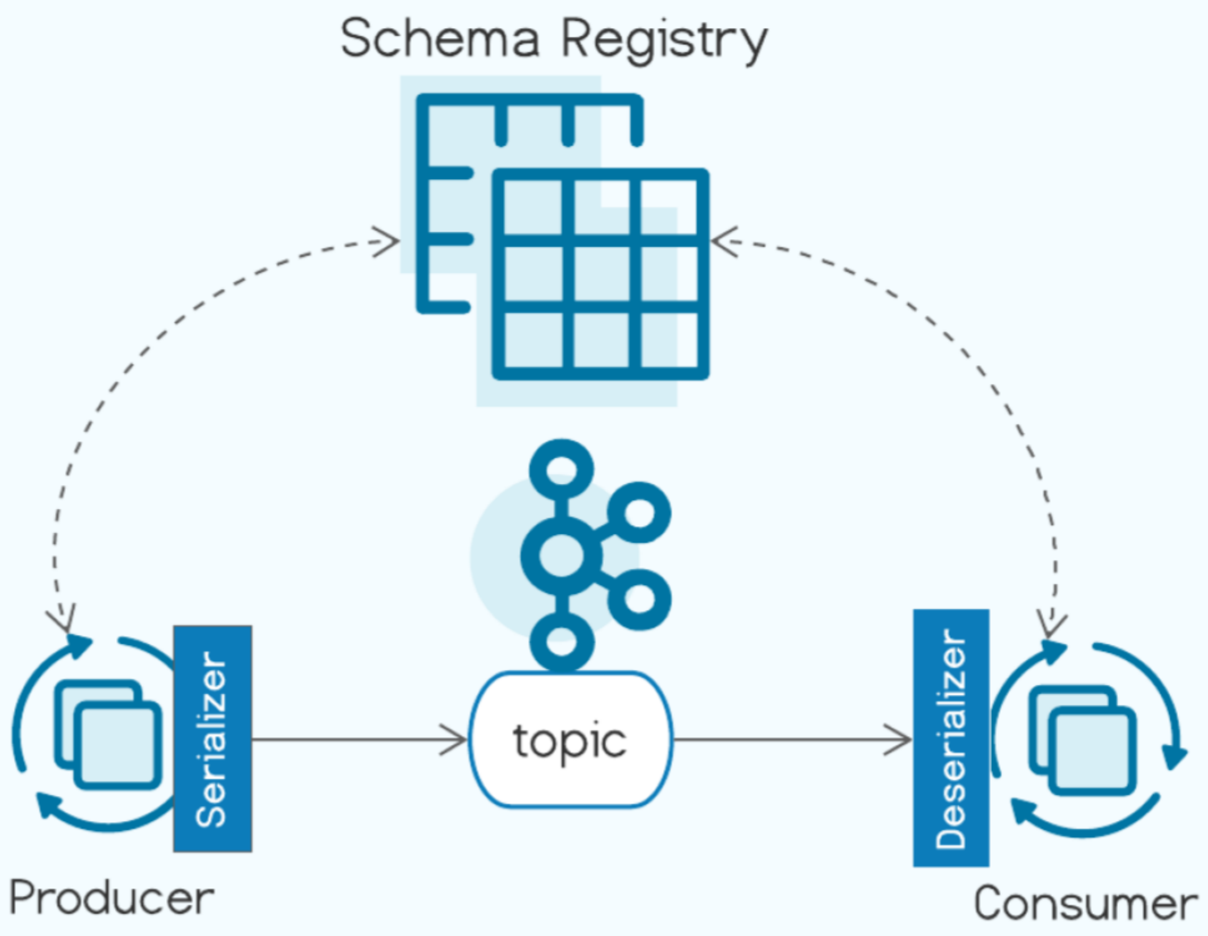
The schema registry is a service that records the various schemas and their different versions as they evolve.
Producer and consumer clients retrieve schemas from the schema registry via HTTPS, store them locally in cache, and use them to serialize and deserialize messages sent to and received from Kafka.
This schema retrieval occurs only once for a given schema and from that point on the cached copy is relied upon.

Confluent Schema Registry - **Single Primary Architecture**



Confluent Schema Registry - **Single Primary Architecture**

- **Versioned** schema repository
- **Safe** schema evolution
- **Resilient** data pipelines
- **Enhanced** data integrity
- **Reduce** storage and computation
- **Discover** your data
- **Cost-efficient** ecosystem



Understanding the Schema Registry Workflow

The workflow of Schema Registry includes:

- Writing a schema file
- Adding schema files to a project
- Leveraging schema tools and plugins—Maven and Gradle
- Configuring Schema Registry plugins
- Generating the model objects from a schema
- Locating and exploring the generated files

Schema file - Protobuf

```
syntax = "proto3";
package io.confluent.developer.proto;
option java_outer_classname = "PurchaseProto";

message Purchase {
  string item = 1;
  double total_cost = 2;
  string customer_id = 3;
}
```

- The first line declares the version of Protocol Buffers to use. This course uses version 3.
- The ***package*** declaration creates a name space for proto files to prevent name clashes. It ends up being the package name in the generated Java class unless you specify an ***option java_package*** in the proto file.
- The ***option java_outer_classname*** field describes the name of the generated java file
- Protocol Buffers define the ***message*** in the schema as an inner class in the generated file

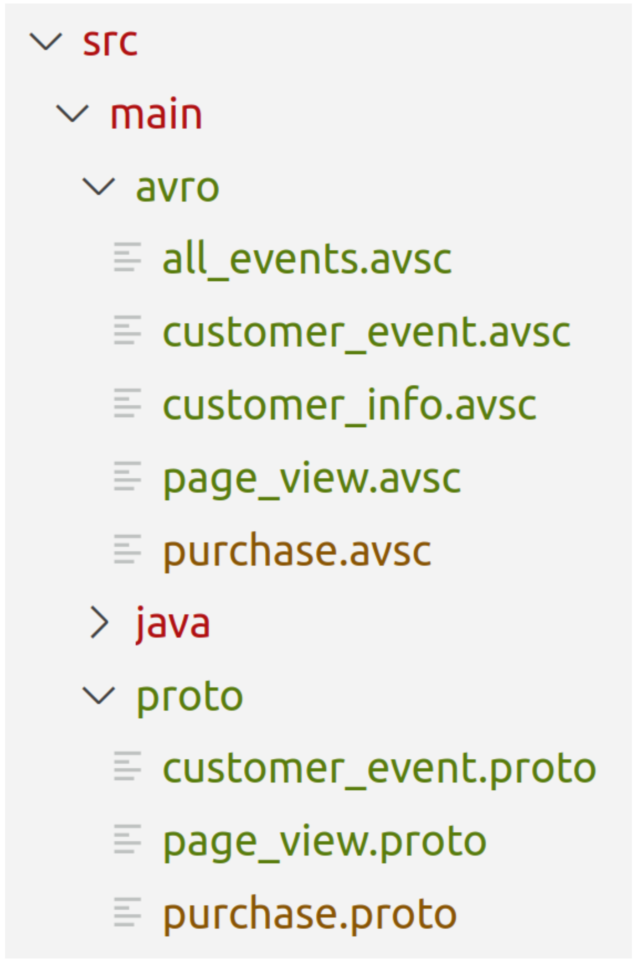
Schema file - Avro

```
{
  "type":"record",
  "namespace": "io.confluent.developer.avro",
  "name":"Purchase",
  "fields": [
    {"name": "item", "type":"string"},
    {"name": "total_cost", "type": "double"},
    {"name": "customer_id", "type": "string"}
  ]
}
```

- The ***namespace*** field serves the same purpose as in the Protobuf file, preventing name collisions. The namespace also becomes the package name for the generated Java file.
- Fields are declared in a ***JSON*** array.

Schema Files in a Project

- The Avro schema files go in the ***src/main/avro*** directory and ***Protobuf*** files land in ***src/main/proto***.

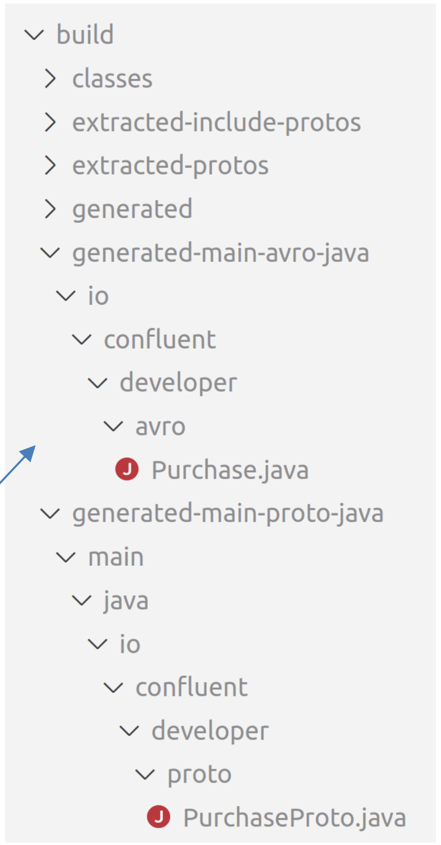


```
src
  main
    avro
      all_events.avsc
      customer_event.avsc
      customer_info.avsc
      page_view.avsc
      purchase.avsc
    java
    proto
      customer_event.proto
      page_view.proto
      purchase.proto
```

Once you've written your schema files you will want to generate the model/data objects from the schema files. Fortunately, there are plugins available, either Maven or Gradle, that can integrate into your local build.

```
schemaRegistry {
    url = Confluent Cloud SR endpoint
    credentials {
        //characters up to the ':' in the basic.auth.user.info property
        username = <username>
        // password is everything after ':' in the basic.auth.user.info property
        password = <password>
    }
    // Possible types are ["JSON", "PROTOBUF", "AVRO"]
    register {
        subject('avro-purchase', 'src/main/avro/purchase.avsc', 'AVRO')
        subject('proto-purchase', 'src/main/proto/purchase.proto', 'PROTOBUF')
    }
}
```

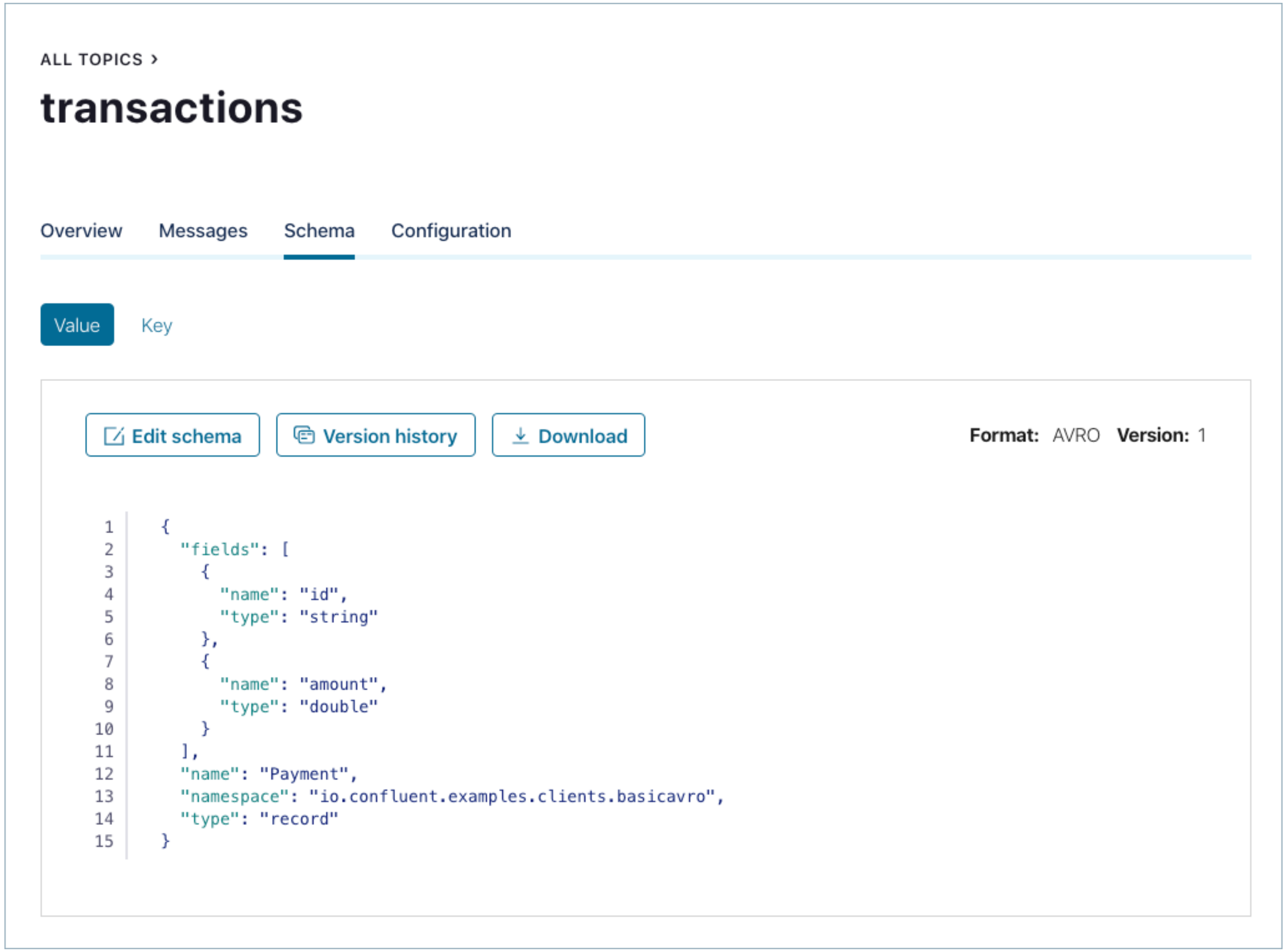
Plugins to generate files



```
build
  classes
  extracted-include-protos
  extracted-protos
  generated
  generated-main-avro-java
    io
      confluent
        developer
          avro
            Purchase.java
  generated-main-proto-java
    main
      java
        io
          confluent
            developer
              proto
                PurchaseProto.java
```

Generated Class files

Confluent Schema Registry - **Single Primary Architecture**



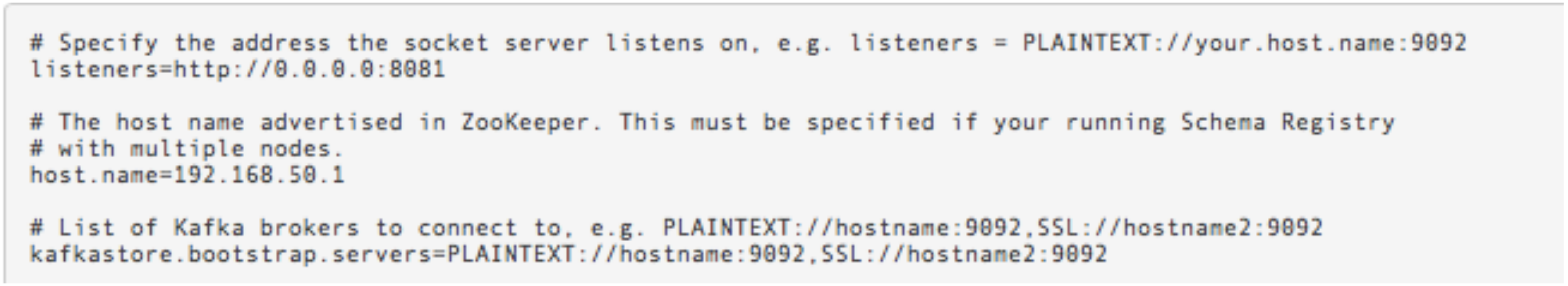
Format: AVRO **Version:** 1

```
{
  "fields": [
    {
      "name": "id",
      "type": "string"
    },
    {
      "name": "amount",
      "type": "double"
    }
  ],
  "name": "Payment",
  "namespace": "io.confluent.examples.clients.basicavro",
  "type": "record"
}
```

Take note of the version, schema ID, and the fields.

Confluent Schema Registry - **Single Primary Architecture**

etc/schema-registry/schema-registry.properties

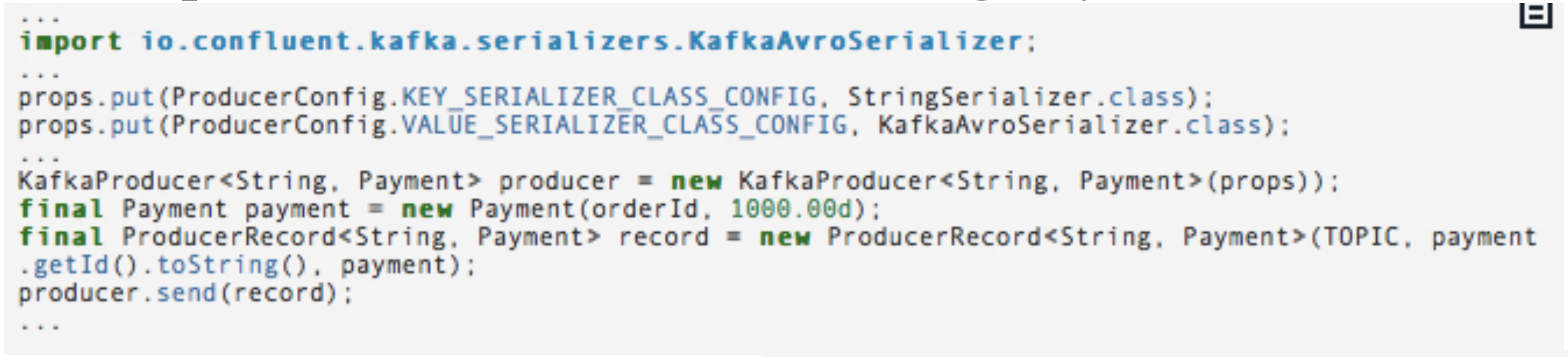


```
# Specify the address the socket server listens on, e.g. listeners = PLAINTEXT://your.host.name:9092
listeners=http://0.0.0.0:8081

# The host name advertised in ZooKeeper. This must be specified if your running Schema Registry
# with multiple nodes.
host.name=192.168.50.1

# List of Kafka brokers to connect to, e.g. PLAINTEXT://hostname:9092,SSL://hostname2:9092
kafkastore.bootstrap.servers=PLAINTEXT://hostname:9092,SSL://hostname2:9092
```

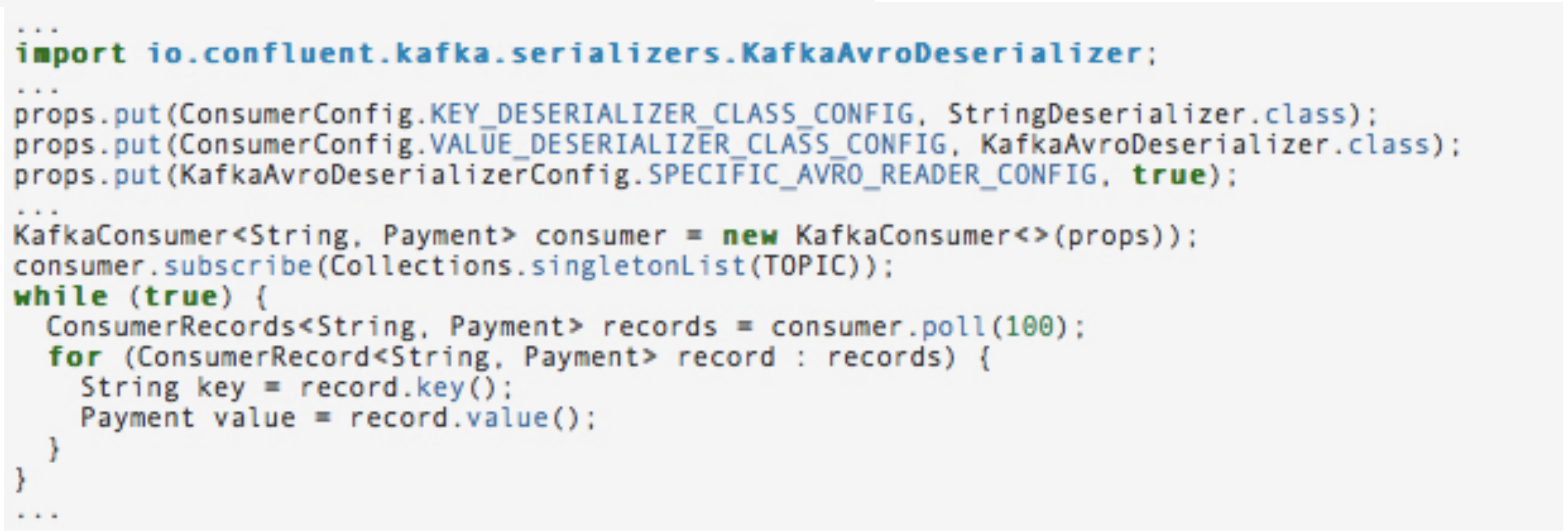
Configuring Schema Registry API

Kafka applications using Avro data and Schema Registry need to specify at least two configuration parameters:

- Avro serializer or deserializer
- Properties to connect to Schema Registry



```java
...
import io.confluent.kafka.serializers.KafkaAvroSerializer;
...
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
...
KafkaProducer<String, Payment> producer = new KafkaProducer<String, Payment>(props));
final Payment payment = new Payment(orderId, 1000.00d);
final ProducerRecord<String, Payment> record = new ProducerRecord<String, Payment>(TOPIC, payment
.getId().toString(), payment);
producer.send(record);
...
```



```java
...
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
...
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class);
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, true);
...
KafkaConsumer<String, Payment> consumer = new KafkaConsumer<>(props));
consumer.subscribe(Collections.singletonList(TOPIC));
while (true) {
  ConsumerRecords<String, Payment> records = consumer.poll(100);
  for (ConsumerRecord<String, Payment> record : records) {
    String key = record.key();
    Payment value = record.value();
  }
}
...
```

Schema Management

Tos

Effective schema management requires:

- Schema IDs
- Schema registration
- Schema versioning
- Viewing and retrieving schemas

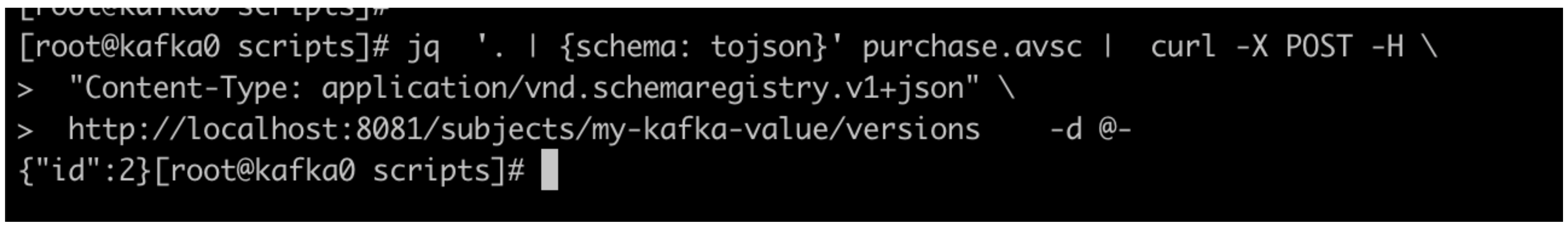
Register an Avro Schema – REST API

Tos

```
jq  '. | {schema: tojson}' purchase.avsc |  curl -X POST -H \
 "Content-Type: application/vnd.schemaregistry.v1+json" \
 http://localhost:8081/subjects/my-kafka-value/versions    -d
@-
```

purchase.avsc

```
{
  "type":"record",
  "namespace": “tos.developer.avro",
  "name":"Purchase",
  "fields": [
   {"name": "item", "type":"string"},
   {"name": "total_cost", "type": "double" },
   {"name": "customer_id", "type": "string"}
  ]
}
```



```
[root@kafka0 scripts]# jq  '. | {schema: tojson}' purchase.avsc |  curl -X POST -H \
>  "Content-Type: application/vnd.schemaregistry.v1+json" \
>  http://localhost:8081/subjects/my-kafka-value/versions    -d @-
{"id":2}[root@kafka0 scripts]#
```

#dnf install jq

Schema Registration

Tos

When you register a schema you need to provide the ***subject-name*** and the ***schema*** itself.
The s***ubject-name*** is the ***name-space*** for the schema, almost like a key when you use a hash-map.
The standard naming convention is ***topic-name-key*** or ***topic-name-value***.
Once Schema Registry receives the ***schema***, it assigns it a unique ID number and a version number.
The first time you register a schema for a given ***subject-name***, it is assigned a version of 1.