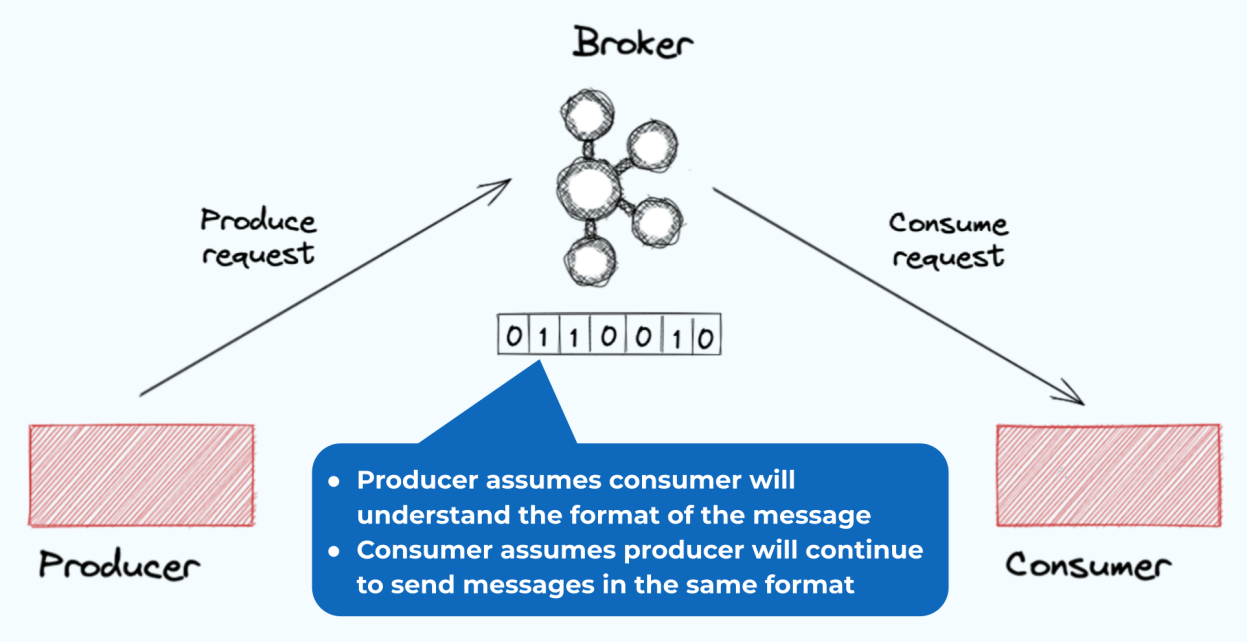
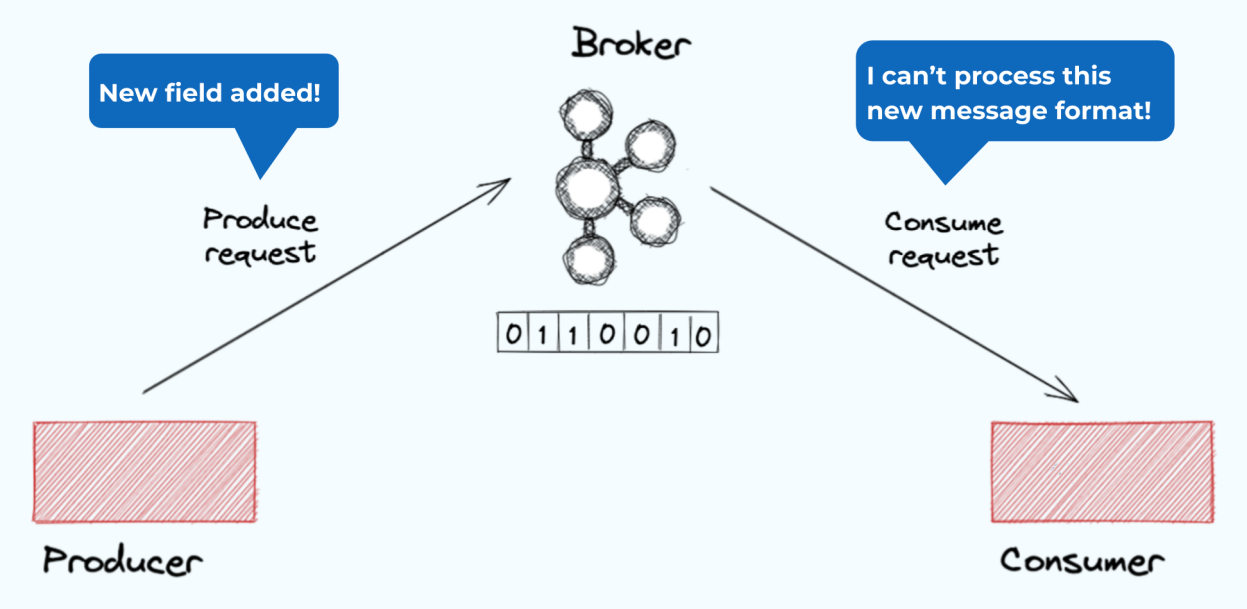
Kafka - Schema Registry

Key Concepts

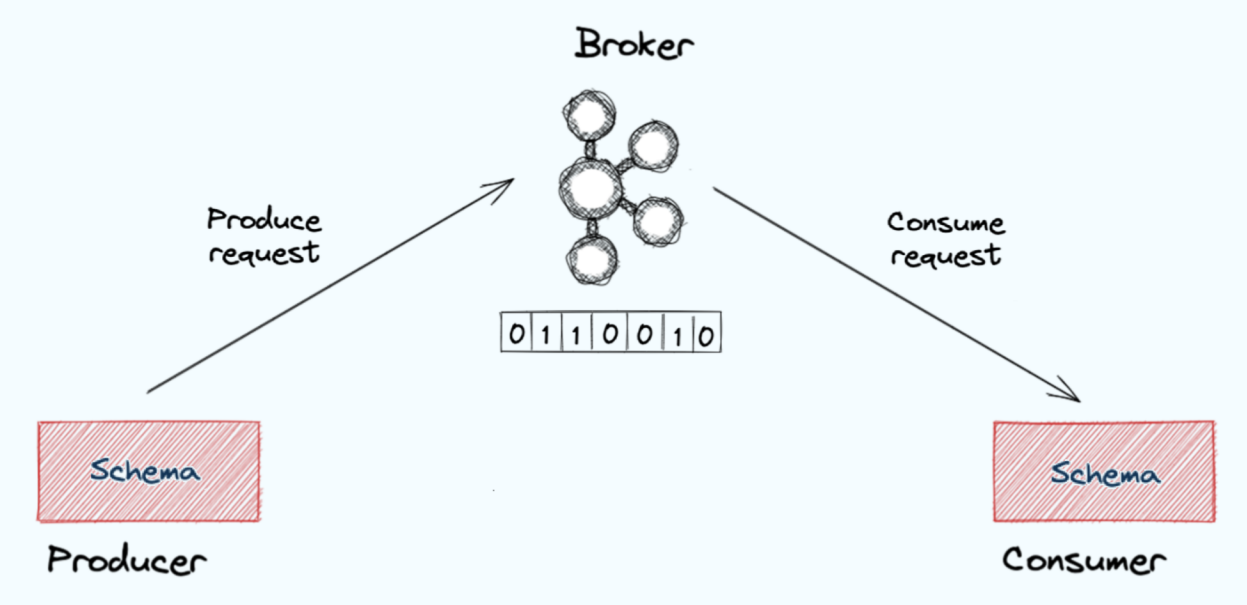
Implied Contract Between Kafka Applications



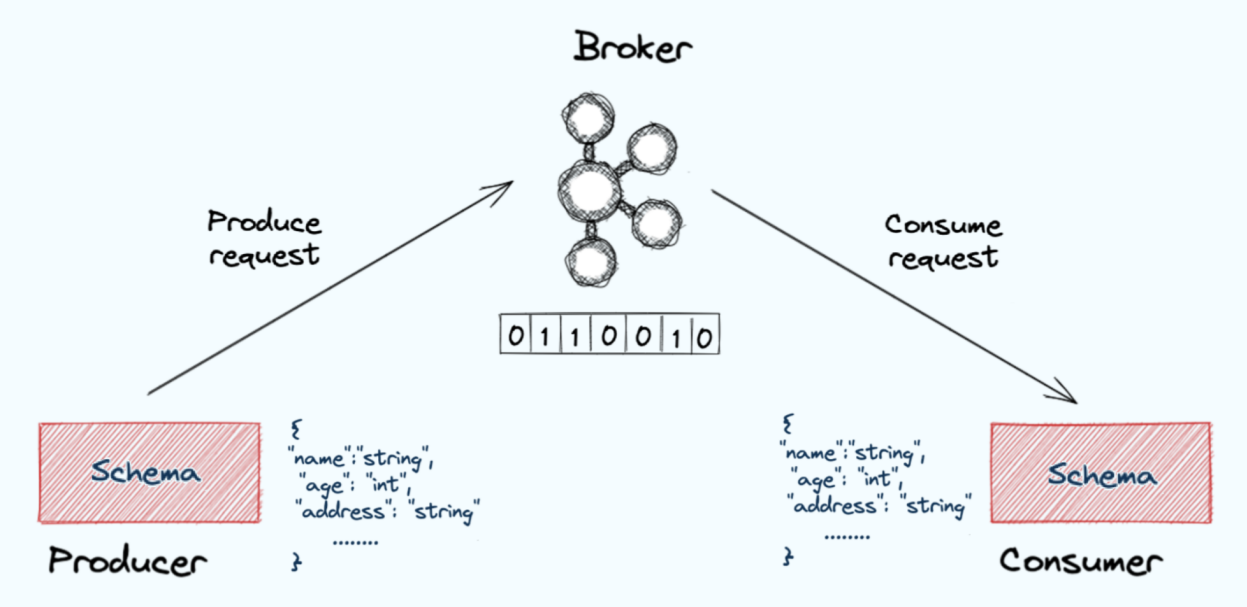
Change Always Happens



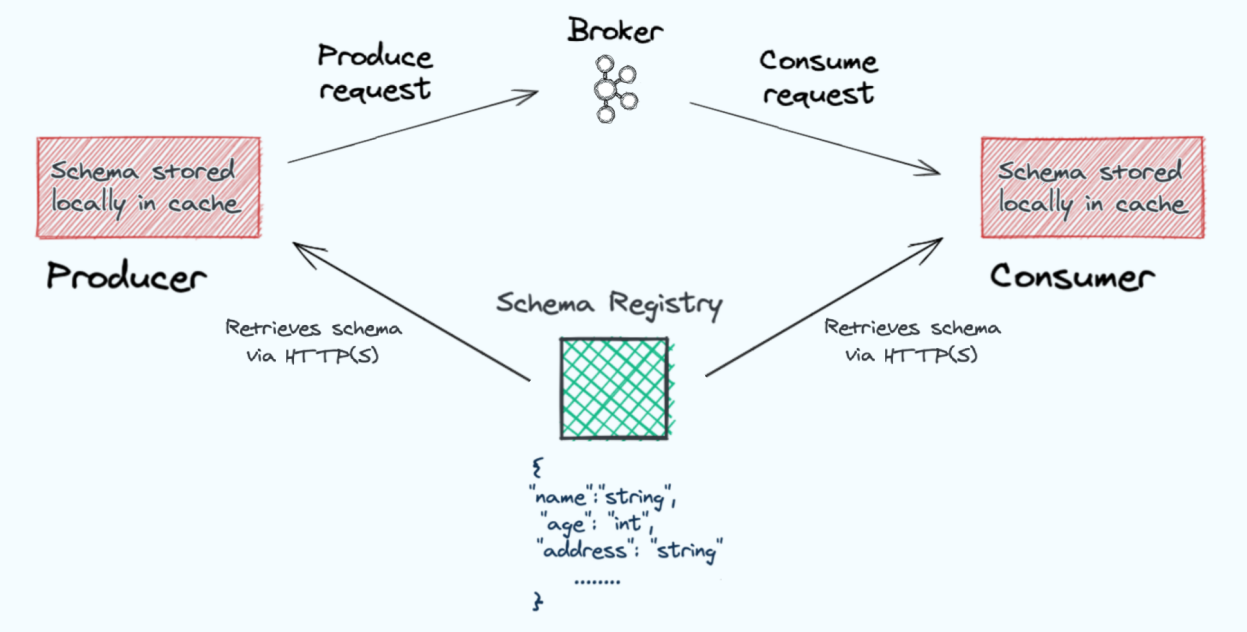
The Schema Is the Contract



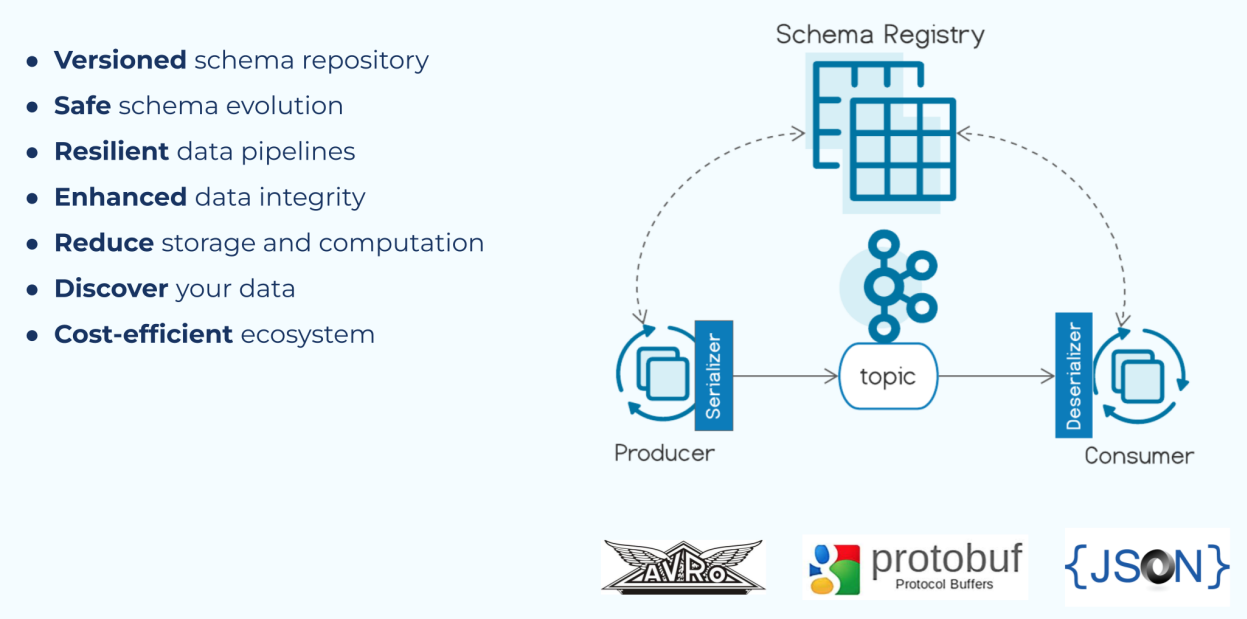
Schemas Establish the Format of the Message



Schema Registry Is the Contract Arbitrator



Confluent Schema Registry



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

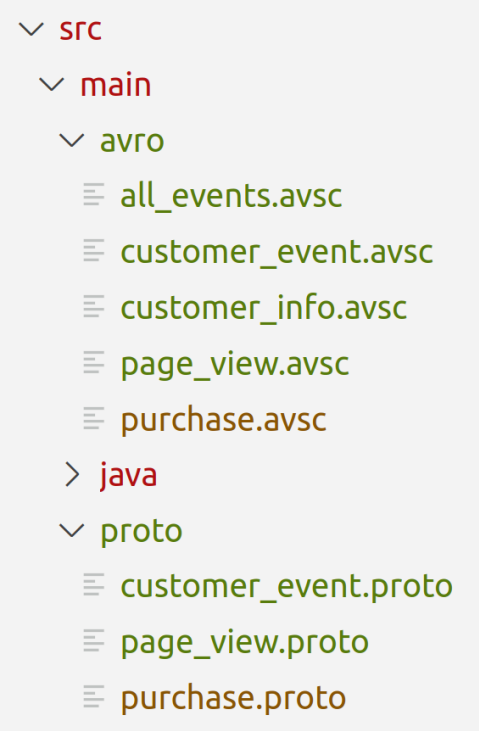
Writing a Schema File – Protobuf

```
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;
}
```

Writing a 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"}
  ]
}
```

Placing Schema Files in a Project



Available Plugins

Avro – Gradle and Maven plugins

- https://github.com/davidmc24/gradle-avro-plugin
- https://mvnrepository.com/artifact/org.apache.avro/avro-maven-plugin

Protobuf – Gradle and Maven plugins

- https://www.xolstice.org/protobuf-maven-plugin/
- https://github.com/google/protobuf-gradle-plugin

Plugin Configuration

```
plugins {
  ... other plugin definitions

  id "com.google.protobuf" version "0.8.15"
  id "com.github.imflog.kafka-schema-registry-gradle-plugin" version "1.1.1"
  id "com.github.davidmc24.gradle.plugin.avro" version "1.0.0"
}
```

Protobuf Plugin Configuration

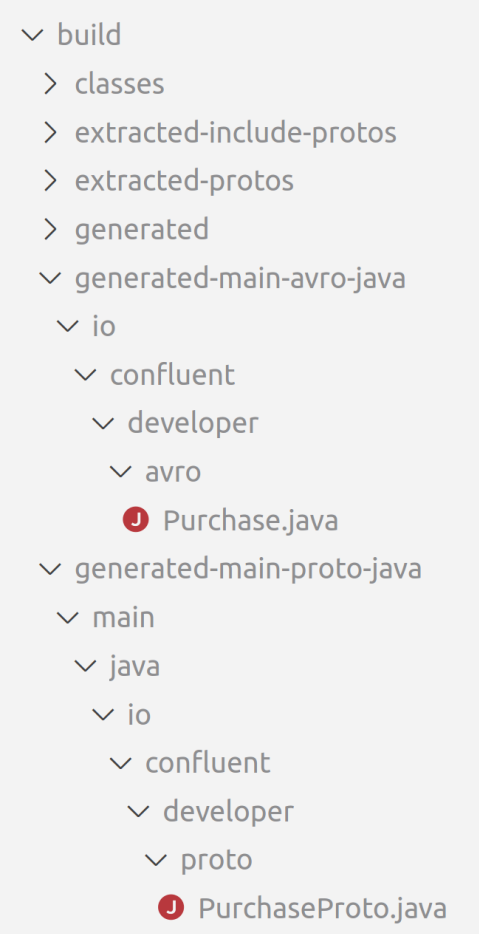
```
protobuf {

  generatedFilesBaseDir = "${project.buildDir}/generated-main-proto-java"

  protoc {
    artifact = 'com.google.protobuf:protoc:3.15.3'
  }
}
```

Schema Registry Configuration

```
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')
    }
}
```

`./gradlew clean build`

Generated Files in a Project



Hands On: Configure, Build and Register Protobuf and Avro Schemas

In this exercise you are going to complete the following tasks:

- Examine the Schema Registry settings in the gradle.build file
- Configure Protobuf and Avro schema definitions
- Generate model objects from the schema definitions using Gradle
- Register the Protobuf and Avro schemas in Confluent Cloud Schema Registry

Working with Schema Formats

1. **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;
}
```

1.1 Protobuf Collections

```
message Purchase {
  string item = 1;
  double total_cost = 2;
  string customer_id = 3;
  repeated string coupon_codes = 4;
  map<string, string> item_options = 5;
}
```

1.2 Protobuf Enumerations

```
message Purchase {
  string item = 1;
  double total_cost = 2;
  string customer_id = 3;
  repeated string coupon_codes = 4;
  map<string, double> item_options = 5;

  enum PurchaseType {
    RETAIL = 0;
    WHOLESALE = 1;
  }
  PurchaseType type = 6;
}
```

1.3 Importing Protobuf

```
import "purchase.proto";
import "page_view.proto";

message CustomerEvent {

  Purchase purchase = 1;
  PageView page_view = 2;
}
```

1.4 Alternate Values for a Field

```
import "purchase.proto";
import "page_view.proto";

message CustomerEvent {

  oneof customer_action {
    Purchase purchase = 1;
    PageView page_view = 2;
  }
}
```

1.5 Default Values

```
message Purchase {
  string item = 1;  -> default is ""
  double total_cost = 2; -> default is 0.0
  string customer_id = 3;
  repeated string coupon_codes = 4; -> default is []
  map<string, string> item_options = 5; -> default is {}
}
```

2. **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"}
  ]
}
```

2.1 Avro Collections – Arrays

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

2.2 Avro Collections – Maps

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

2.3 Avro Enumerations

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

2.4 Avro Records in a Schema

```
{
  "type": "record",
  "namespace": "io.confluent.developer.avro",
  "name": "CustomerEvent",
  "fields" : [
    {"name": "purchase", "type": "io.confluent.developer.avro.Purchase" },
    {"name": "page_view", "type": "io.confluent.developer.avro.PageView" },
    {"name": "id", "type": "string"}
  ]
}
```

2.5 Avro Unions

```
{
  "type": "record",
  "namespace": "io.confluent.developer.avro",
  "name": "CustomerEvent",
  "fields" : [
    {"name": "action", "type": [
      "io.confluent.developer.avro.Purchase",
      "io.confluent.developer.avro.PageView"
    ]},
    {"name": "id", "type": "string"}
  ]
}
```

2.6 Avro Default Values

```
"fields": [
  {"name": "item", "type":"string"},
  {"name": "total_cost", "type": "double", default = 0.0},
  {"name": "customer_id", "type":"string"},
  {"name": "item_options",
    "type": {
      "type": "map",
      "values": "string",
      "default": {}
    }
  }
]
```

Working with Generated Objects

```
Purchase.newBuilder()
   .setCustomerId("vandelay1234")
   .setTotalCost(437.83)
   .setItem("flux-capacitor")
   .build();

purchase.getTotalCost()
```

Changing State of Generated Objects

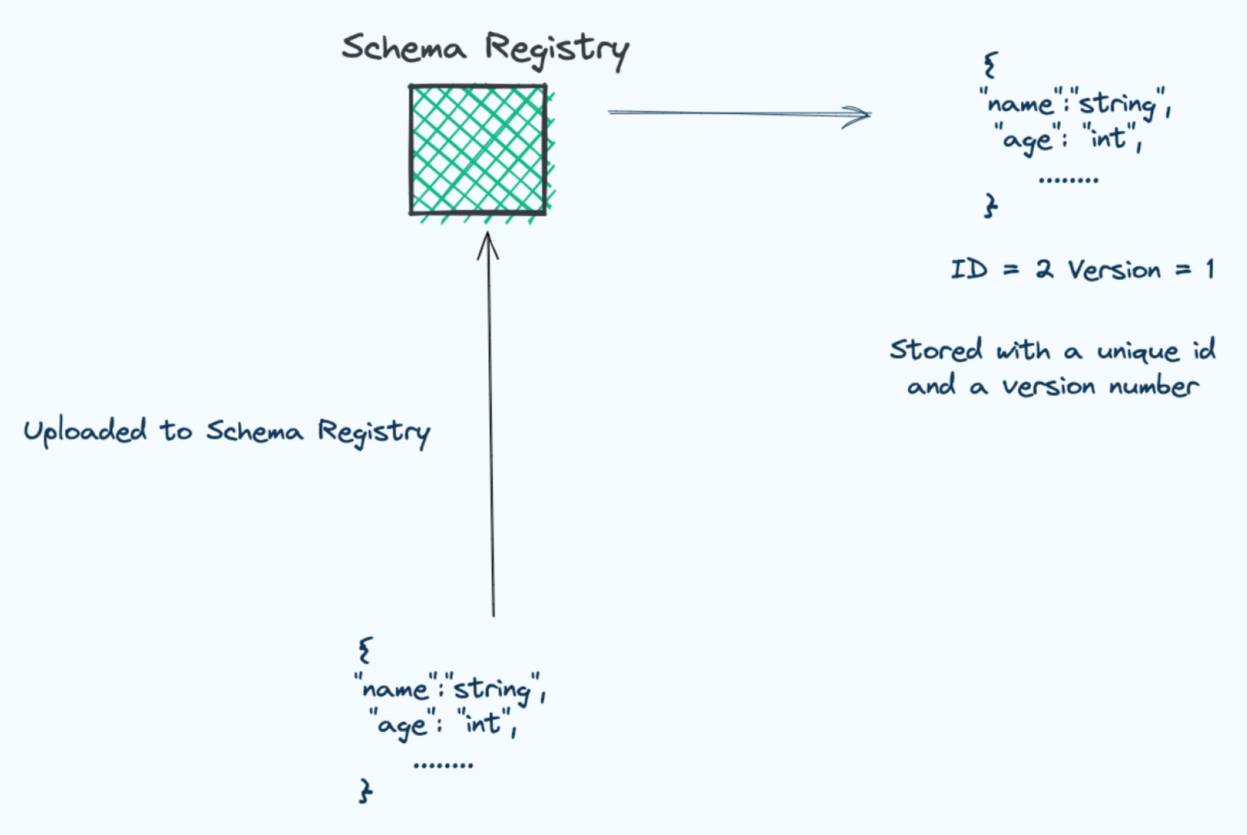
```
Avro only
   purchase.setTotalCost(100.00)

Protobuf and Avro
   Purchase.newBuilder(purchase)
           .setTotalCost(100.00)
           .build()
```

Schema Management

Effective schema management requires an understanding of the following concepts and processes:

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

Schema ID and Version



Register a Schema

There are several methods available to register a schema.

- You can use the Confluent CLI
- Schema Registry REST API
- Confluent Cloud Console / UI tool
- Maven and Gradle plugins

Register a Schema – Confluent CLI

```
confluent schema-registry schema create \
  --subject purchases-value \
  --schema src/main/proto/purchase.proto \
  --type PROTOBUF \
  --api-key YYY \
  --api-secret ZZZ
```

Register a Protobuf Schema – REST API

Protobuf schema definition passed as input to script

```
./protoJsonFmt.sh src/main/proto/purchase.proto | \
curl -u API_KEY:API_SECRET \
  -X POST https://CLUSTER_IP/subjects/purchases-value/versions \
  -H "Content-Type:application/json" \
  -d @-

PROTO_JSON=$(awk '{gsub(/\n/,"\\\n"; gsub(/"/, "\\\"");print}' $1) \
  && SCHEMA="{\"schemaType\":\"PROTOBUF\",\"schema\":\"${PROTO_JSON}\"\n}" \
  && echo ${SCHEMA}
```

JSON formatted schema definition outputted from script used as input by `curl` command

Register an Avro Schema – REST API

```
jq '. | {schema: tojson}' src/main/avro/purchase.avsc | \
curl -u API_KEY:API_SECRET \
  -X POST https://CLUSTER_IP/subjects/purchases-value/versions \
  -H "Content-Type:application/json" \
  -d @-
```

Register a Schema – Gradle

```
register {
  subject('purchases-value', 'src/main/avro/purchase.avsc', 'AVRO')
  subject('purchases-value', 'src/main/proto/purchase.proto', 'PROTOBUF')
}
```

```
./gradlew registerSchemasTask
```

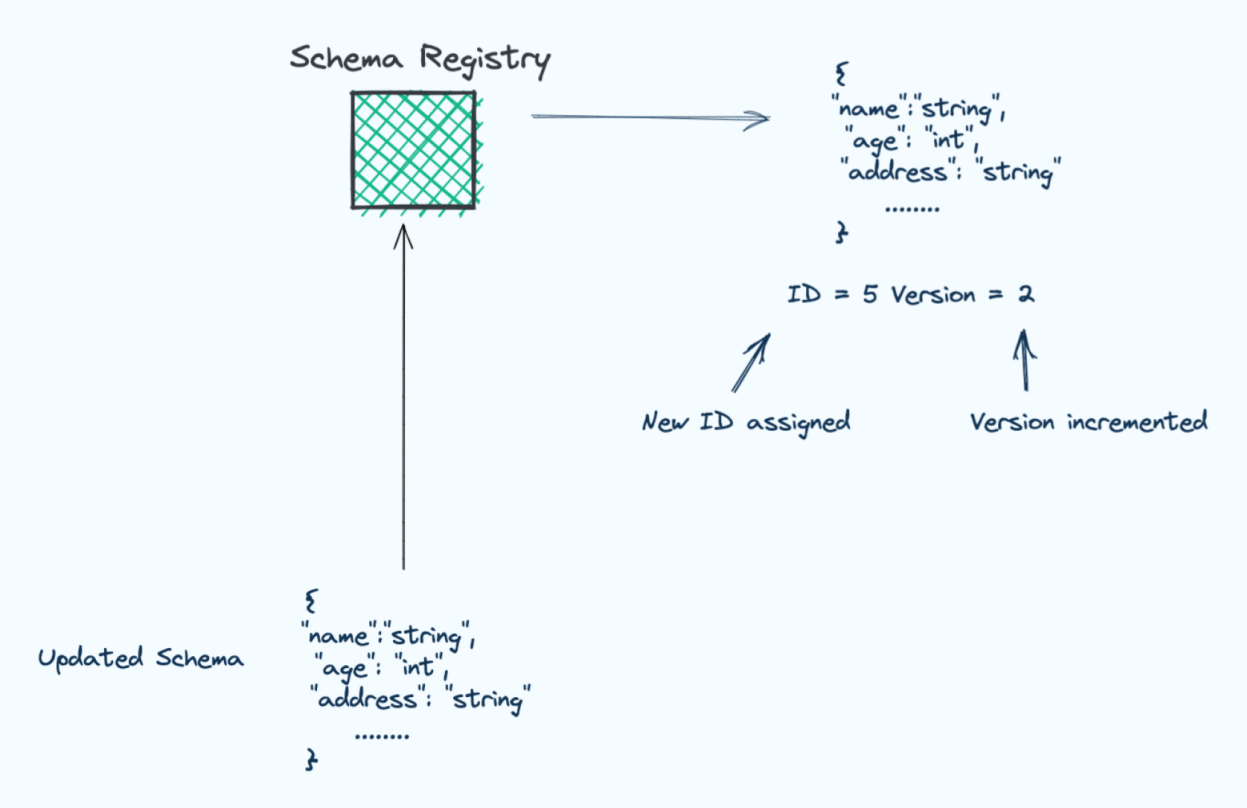
Auto-Register a Schema

Producer clients can “auto-register”

```
producerConfigs.put(“auto.register.schemas”, true)
```

Not for production!!!

Updating a Schema



View or Download a Schema

```
confluent schema-registry schema describe \
  --subject purchases-value \
  --version [version number] | "latest" \
  --api-key YYY \
  --api-secret ZZZ

curl -u API_KEY:API_SECRET \
  -s "https://CLUSTER_IP/subjects/purchases-value/versions/latest" jq '.'

curl -u API_KEY:API_SECRET \
  -s "https://CLUSTER_IP/subjects/purchases-value/versions/N" jq '.'
```

Integrating Schema Registry with Client Applications

Confluent CLI Producer

```
confluent kafka topic produce purchases \
        --value-format protobuf \
        --schema src/main/proto/purchase.proto \
        --sr-endpoint https://.... \
        --sr-api-key xxxyyyy \
        --sr-api-secret abc123 \
        --cluster lkc-45687
> {"item":"pizza", "total_cost":17.99, "customer_id":"lombardi"}
```

Confluent CLI Consumer

```
confluent kafka topic consume purchases \
        --from-beginning \
        --value-format protobuf \
        --sr-endpoint https://.... \
        --sr-api-key xxxyyyy \
        --sr-api-secret abc123 \
        --cluster lkc-45687
```

Console Producer

```
./bin/kafka-protobuf-console-producer \
   --topic purchases \
   --bootstrap-server localhost:9092 \
   --property schema.registry.url = http://... \
   --property value.schema ="$(<src/main/proto/purchase.proto)"
> {"item":"pizza", "total_cost":17.99, "customer_id":”lombardi"}
```

Console Consumer

```
./bin/kafka-protobuf-console-consumer \
   --from-beginning \
   --topic purchases \
   --bootstrap-server localhost:9092 \
   --property schema.registry.url = http://...
```

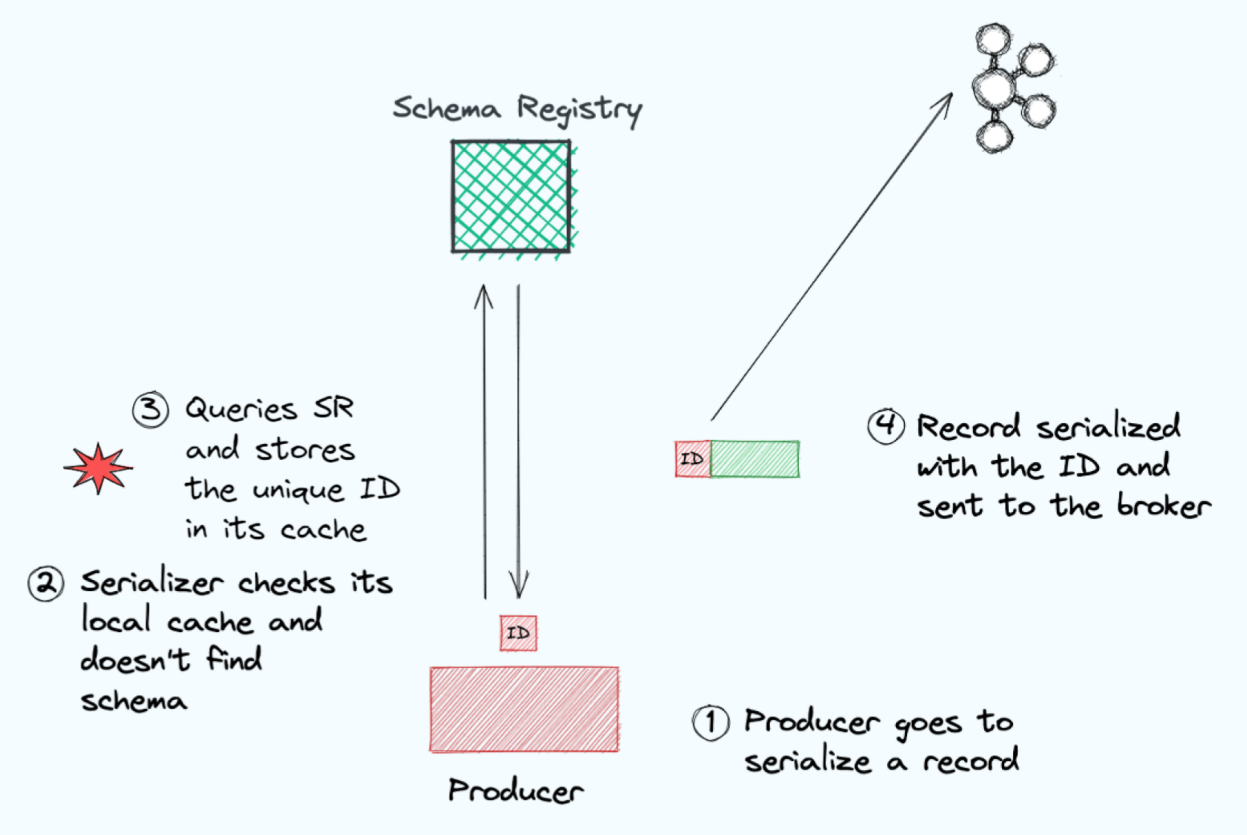
Common Client Configuration Settings

```
schema.registry.url=https://<CLUSTER>.us-east-2.aws.confluent.cloud

basic.auth.credentials.source=USER_INFO
basic.auth.user.info=API_KEY:API_SECRET
```

KafkaProducer

```
// Possible serializer classes are:
//   KafkaAvroSerializer.class
//   KafkaProtobufSerializer.class
//   KafkaJsonSchemaSerializer.class

configs.put("value.serializer", <SERIALIZER_CLASS>);

Producer<Purchase> producer = new KafkaProducer<>(configs);
```

Schema Lifecycle



1. The producer attempts to serialize a record and calls serializer.serialize.
2. This triggers the serializer to check its local cache for the schema file, but it doesn't find it.
3. The serializer then queries Schema Registry and retrieves the schema and ID and stores it in its local cache.
4. Now the serializer serializes the record and stores the ID as part of the byte payload sent to Kafka

---

KafkaConsumer

```
// Possible deserializer classes are:
//   KafkaAvroDeserializer.class
//   KafkaProtobufDeserializer.class
//   KafkaJsonSchemaDeserializer.class

configs.put("value.deserializer", <DESERIALIZER_CLASS>);

Consumer<Purchase> consumer = new KafkaConsumer<>(configs);
```

```
specific.avro.reader = true|false

specific.protobuf.value.type = proto class name

json.value.type = class name
```

`Hands On: Integrate Schema Registry with Clients`

Understanding Schema Subjects

Schema Subject – What Is It?

```
confluent schema-registry schema create --subject purchases-value \...

curl -u API_KEY:API_SECRET \
     -X POST https://CLUSTER_IP/subjects/purchases-value/versions \....

register {
       subject('purchases-value', 'src/main/avro/purchase.proto', 'PROTOBUF')
```

The schema subject name is used for:

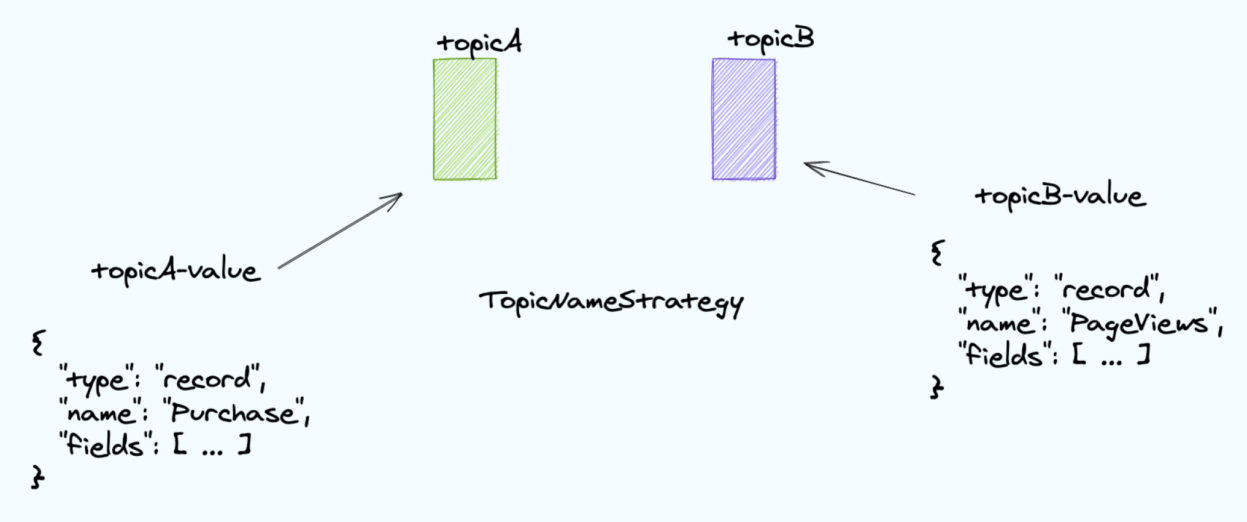
- Compatibility checks—they are done per subject
- Linking version numbers to a subject

Subject Name Strategies

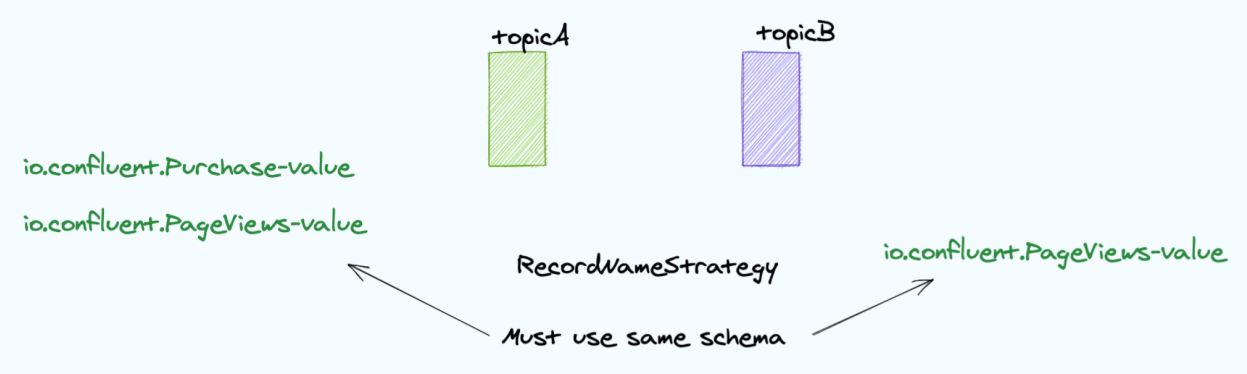
When providing a subject name, there are three strategies or ways to provide it:

- TopicNameStrategy (default setting)
- RecordNameStrategy
- TopicRecordNameStrategy

TopicNameStrategy



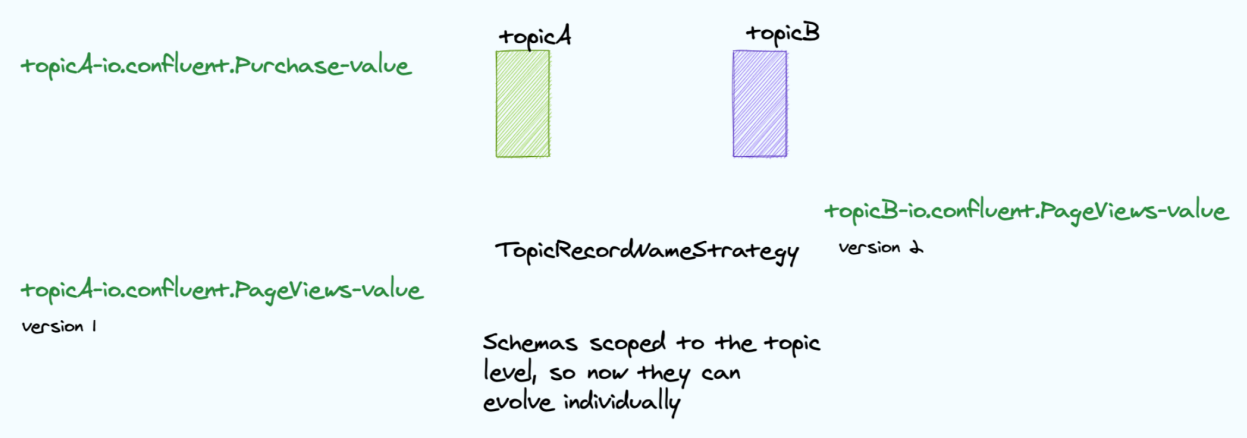
RecordNameStrategy



RecordNameStrategy allows for different schemas in a topic since the individual records only need to comply with a schema that has the subject name that matches its class.

But you have to use the same schema and version across all the topics in a cluster FOR THAT PARTICULAR RECORD TYPE, since there's no way to tell which topic the record belongs to.

TopicRecordNameStrategy



Subject names are created using `<topic name>-<fully qualified record name> appended with -key or -value.`

Since you've now scoped the schema to a particular topic you don't have to use the same version across all topics in the cluster. FOR THAT RECORD TYPE you can evolve the schemas separately.

Using Different Subject Name Strategies

Choosing and using a particular strategy involves two steps:

1. Registering the schema with the correct subject name format.
2. Setting the subject strategy on the clients.

Since TopicNameStrategy is the default, clients are already set to use it. To use a strategy other than the default,
set `key.subject.name.strategy` or `value.subject.name.strategy` on the client as needed.

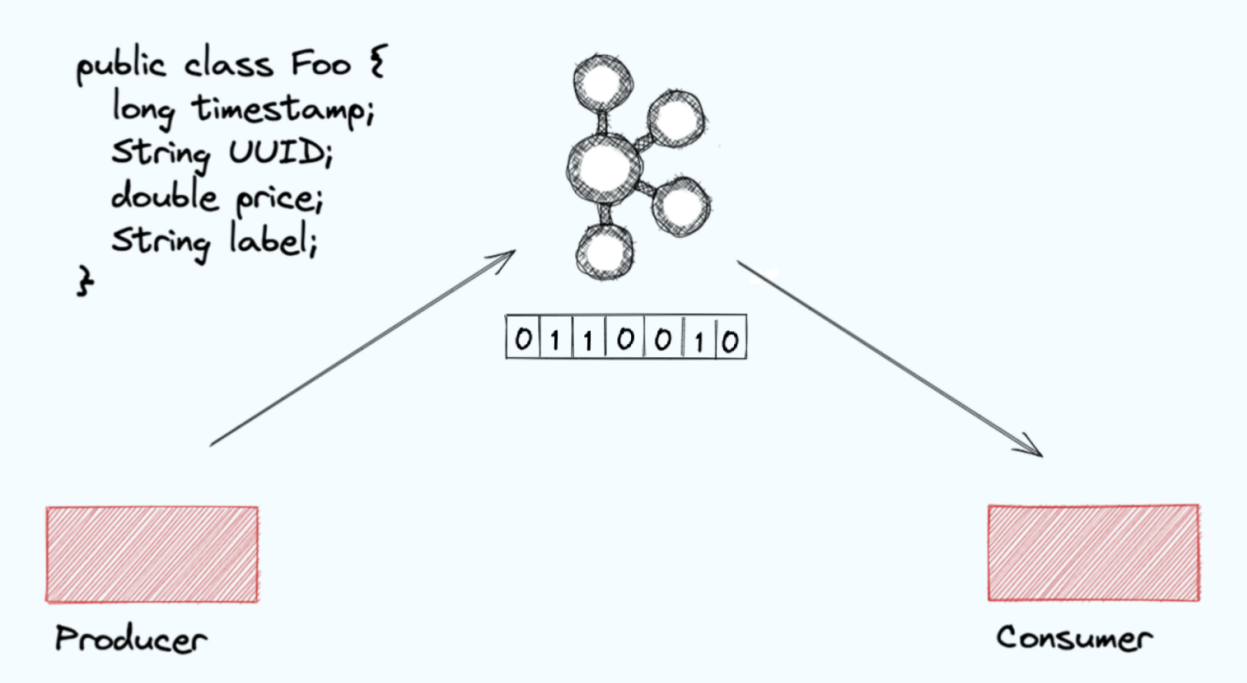
- Record Name Strategy

- `io.confluent.kafka.serializers.subject.RecordNameStrategy`

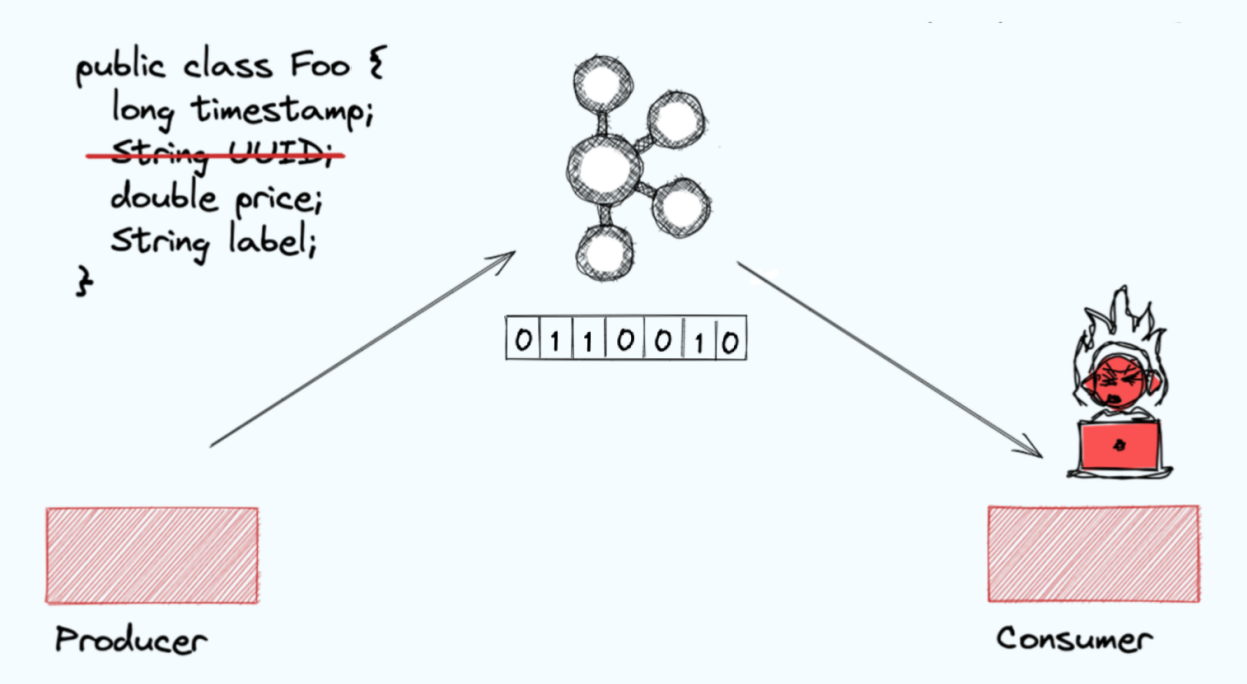
- Topic Record Name Strategy
  - `io.confluent.kafka.serializers.subject.TopicRecordNameStrategy`

Testing Schema Compatibility

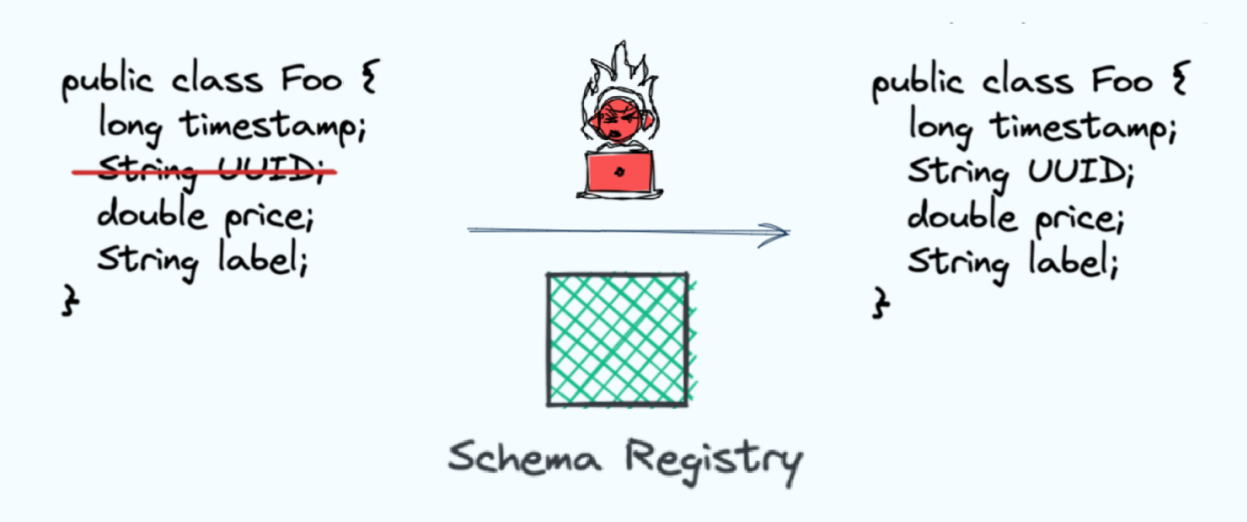
Expectations of Object Structure



Unexpected Changes May Cause Exceptions



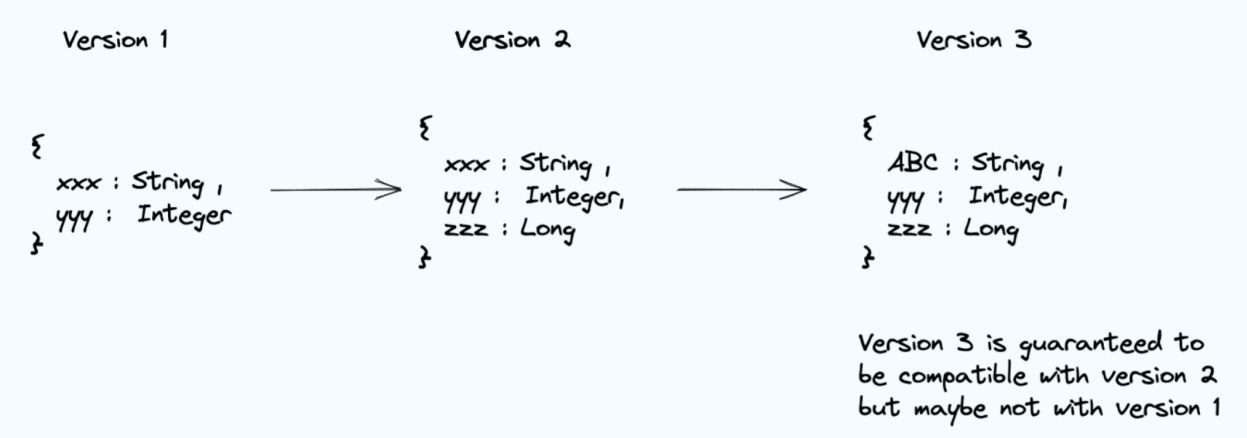
Schema Compatibility



Schema Compatibility Modes

Compatibility Type	Schema Resolution
BACKWARD	Consumers using the new schema version can read data produced with the previous version
FORWARD	Consumers using the previous schema version can read data produced with the new version
FULL	The combination of BACKWARD and FORWARD compatibility
NONE	Compatibility checking disabled

Schema Compatibility Example

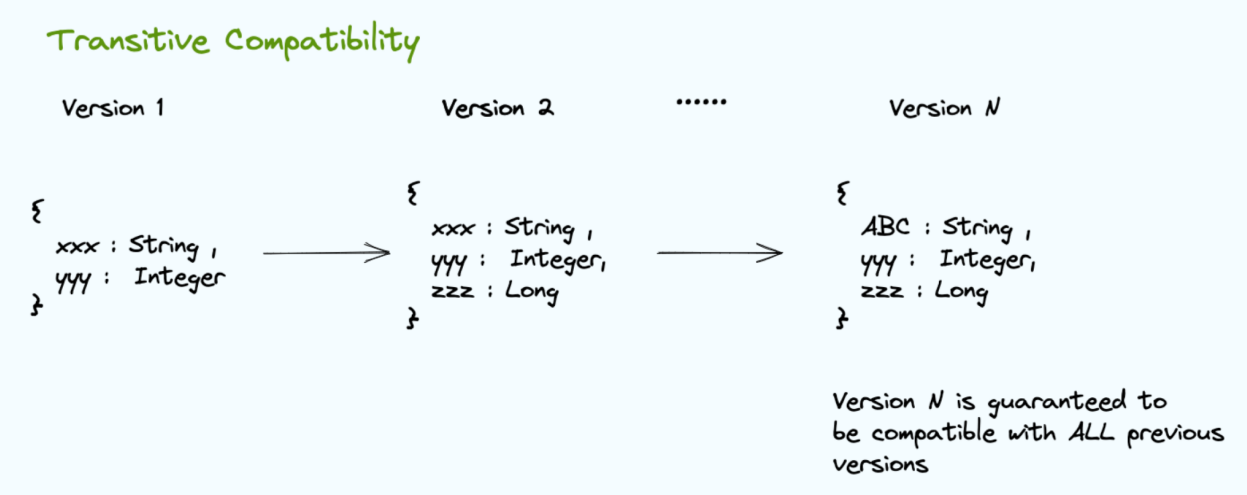


In this example, if the compatibility mode is backward, forward, or full, when version 3 is registered, Schema Registry will only verify that it is compatible with version 2. Compatibility with other previous versions is not verified.

Transitive Schema Compatibility Modes

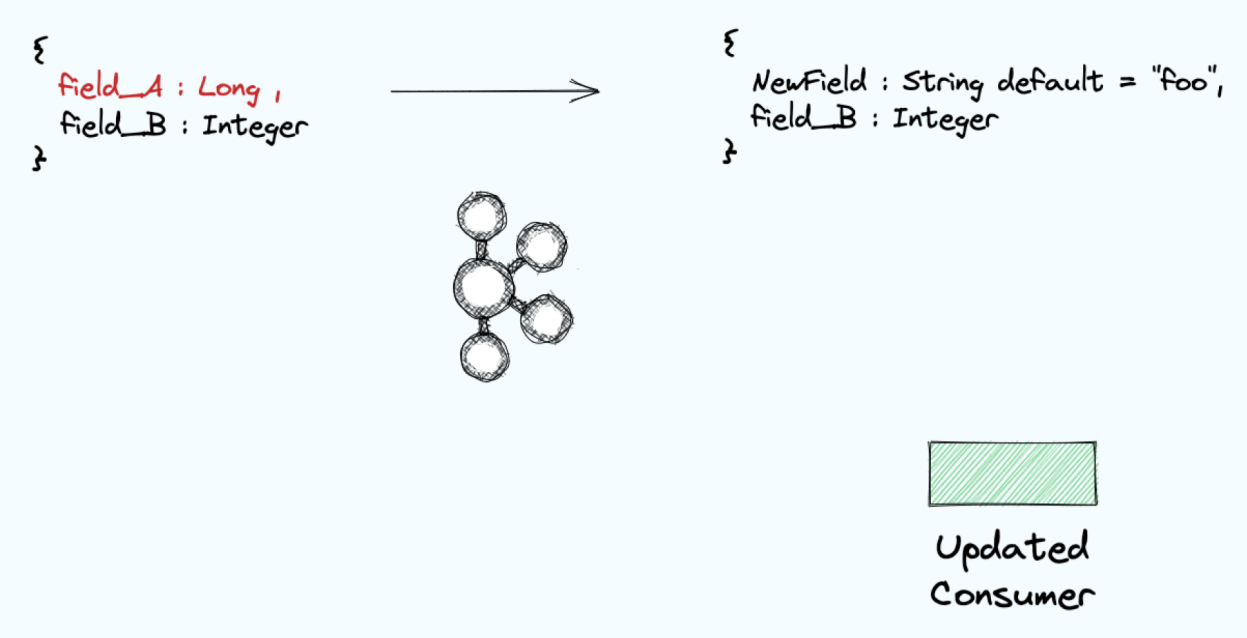
Compatibility Type	Schema Resolution
BACKWARD_TRANSITIVE	Consumers using the new schema version can read data produced with all previous versions
FORWARD_TRANSITIVE	Consumers using the previous schema version can read data produced with all later versions
FULL_TRANSITIVE	The combination of BACKWARD and FORWARD compatibility

Transitive Schema Compatibility Example



The difference with transitive compatibility is that the latest version of a schema you just updated is verified to be compatible with all previous versions, not just the immediate previous one.

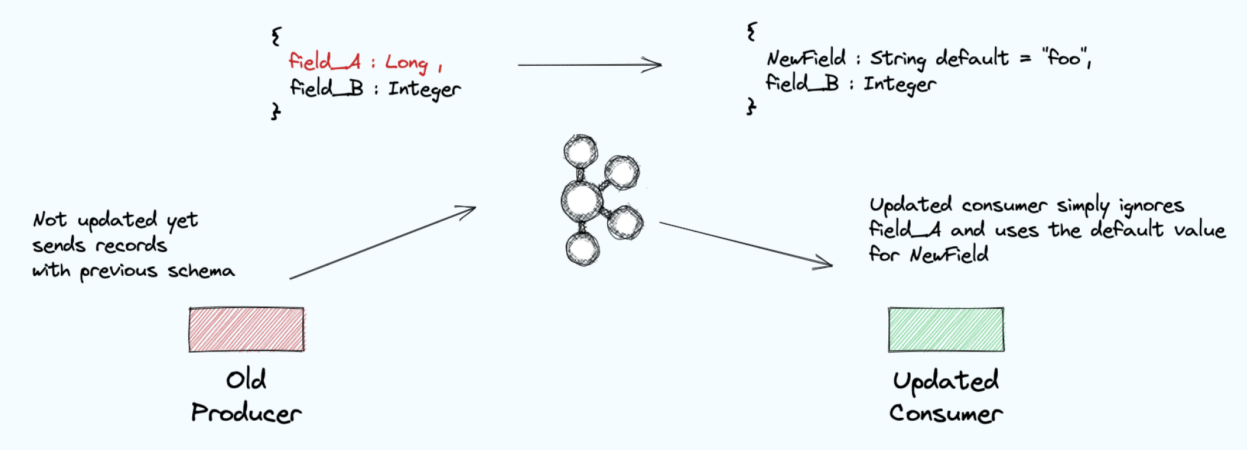
Backward Compatibility Illustrated



With backward compatibility you can delete fields from a schema and add fields as long as they have a default value.

The order of updating your clients is to update the consumer clients first.

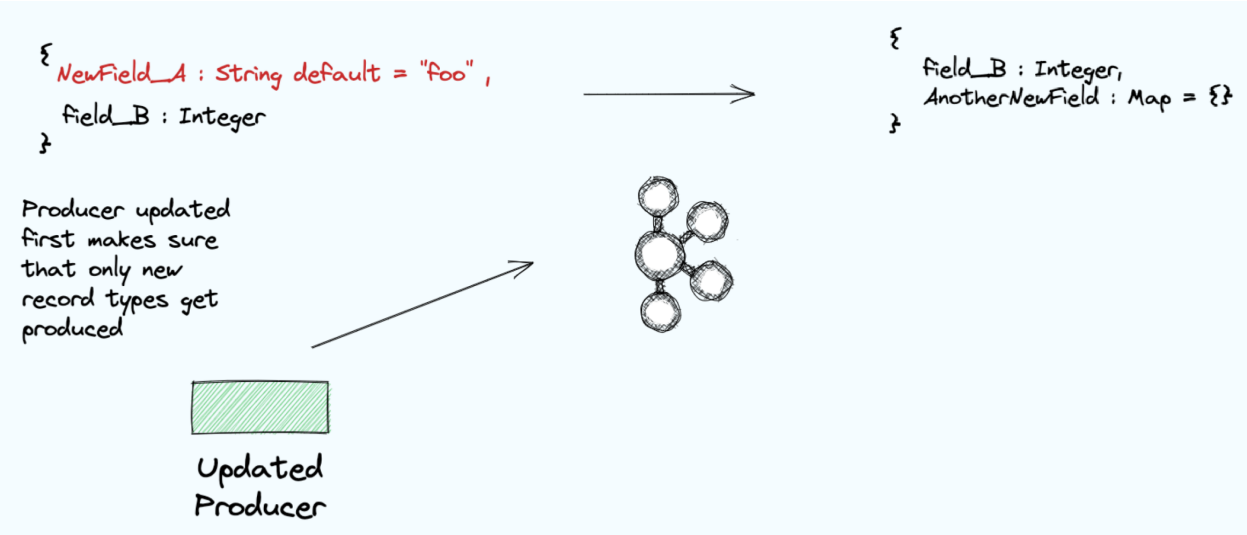
In this example the schema is evolved by removing field_A and adding NewField that has a default value associated with it.



By updating the consumer client first, if a producer using the previous schema sends records to Kafka the updated consumer client will continue to work fine.

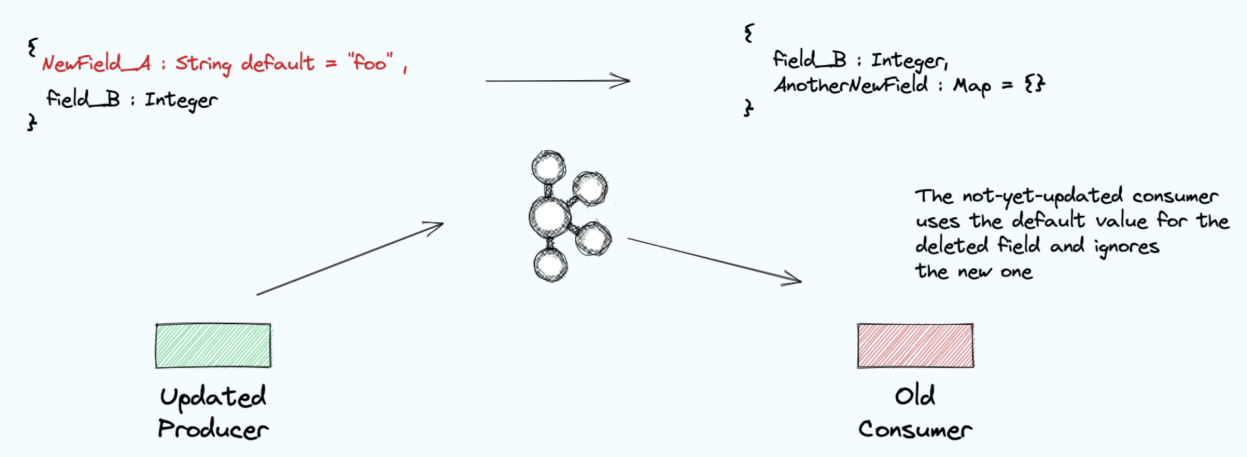
It will ignore the deleted field_A on the record and use the default value provided for NewField.

Forward Compatibility Illustrated



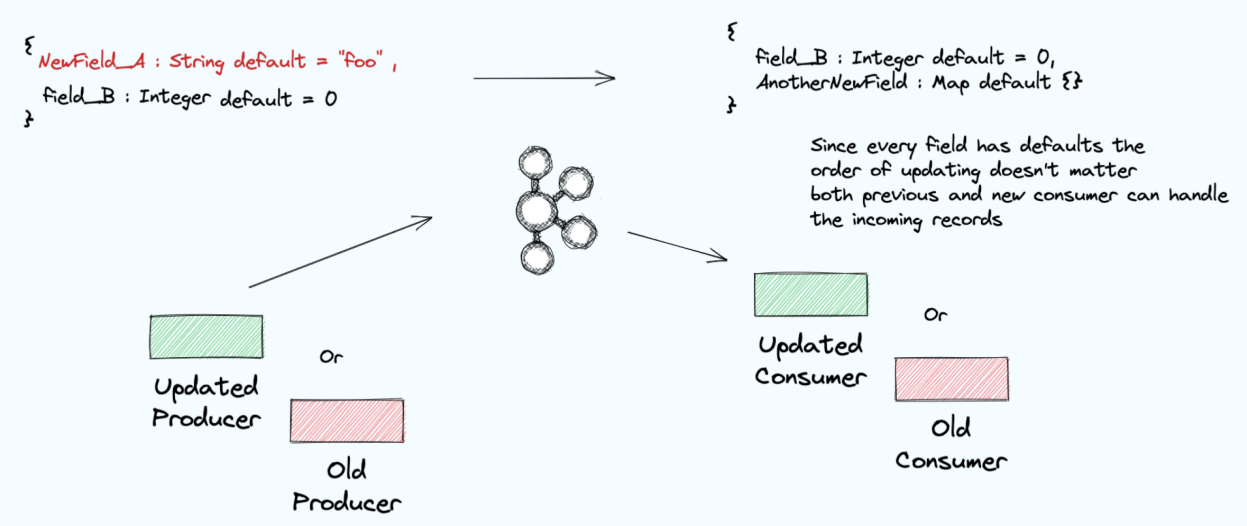
With forward compatibility you can delete fields that have a default and add new fields.

In this case, you update the producer clients first.



Now if you have a consumer you haven't updated yet, it will continue to operate and use the default value for the field that was deleted and simply ignore the newly added field.

Full Compatibility Illustrated



With full compatibility every field in the schema has a default value, so the order of updating clients doesn't matter—with either order both old and new consumers will

continue to work properly.

Schema Compatibility Considerations

Backward	Forward	Full
Delete fields Add fields with default values	Delete fields with default values Add fields	Delete fields with default values Add fields with default values
Update consumer clients first	Update producer clients first	Order of update doesn't matter

This table summarizes the compatibility matrix for Schema Registry.

- With Backward compatibility you can delete fields and add fields with default values and you update the consumer clients first.
- With Forward compatibility you can delete fields with default values and add new fields. You need tol update your producer clients first under forward compatibility.
- With Full compatibility, both deleted or added fields must have a default value. Since every change to the schema has a default value, the order in which you update your clients doesn't matter.

Setting Schema Compatibility

```
confluent schema-registry subject update <SUBJECT> \
    --compatibility = <MODE>
```

```
curl -X PUT -H "Content-Type: application/vnd.schemaregistry.v1+json" \
    --data '{"compatibility": "FULL"}' \
    http://localhost:8081/config/my-kafka-value
```

```
schemaRegistry {
    url = 'https://xxxxxx:8081'

    //Other details left out for clarity

    config {
        subject('purchases-value', ‘FORWARD’)
    }
}
```

Checking a Schema for Compatibility

```
confluent schema-registry compatibility validate \
    --schema src/main/proto/purchase.proto \
    --type PROTOBUF \
    --subject purchases-value \
    --version latest \
    --api-key YYY \
    --api-secret ZZZ
```

```
jq '. | {schema: tojson}' src/main/avro/purchase.avsc | \
curl -u API_KEY:API_SECRET \
     -X POST https://CLUSTER_IP/compatibility/subjects/purchases-value/version/latest \
     -H "Content-Type:application/json" \
     -d @-
```

```
schemaRegistry {
    url = 'https://xxxxxx:8081'
    //Other details left out for clarity

    compatibility {
        subject('purchases-value',
                src/main/proto/purchases.proto,
                'PROTOBUF')
    }
}
```

Hands On: Evolve Existing Schemas
