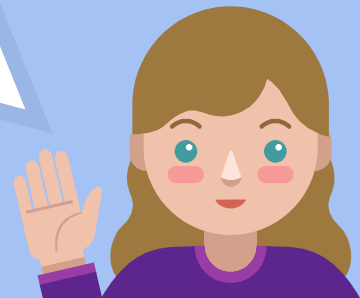
More Kafka Stack

What We Will Learn

What We Will Learn

- Avro
  - No need kafka
  - Kafka has good support
  - Avro details
  - How to use it in Kafka
- Kafka Graphical User Interface
- Kafka schema registry
  - Related with avro
- Kafka REST API
  - Topic
  - Produce & consume message



The Tools

- Apache avro
- Confluent schema registry
- Confluent rest proxy
- See ***Resource & References*** for links

Setup More Kafka Stack

Installing The Tools

- Docker compose
- Script available at lecture Resource & Reference (last section of course)
- **IMPORTANT :** set docker memory to at least 4GB

```
#> docker-compose -f [script-file] -p [project] down
```

Part	What to run (in sequence)
1 - core kafka	`#> docker-compose -f docker-compose-core.yml -p core up -d`

2 - kafka connect	`#> docker-compose -f docker-compose-core.yml -p core down`
	`#> docker-compose -f docker-compose-connect.yml -p connect up -d`
	`#> docker-compose -f docker-compose-connect-sample.yml -p connect-sample up -d`

3 - kafka user interface	`#> docker-compose -f docker-compose-connect.yml -p connect down`
	`#> docker-compose -f docker-compose-connect-sample.yml -p connect-sample down`
	`#> docker-compose -f docker-compose-ui.yml -p kafka-ui up -d`

4 - kafka full (schema registry, REST proxy, ksqldb, kafka connect, UI)	**`#> docker-compose -f docker-compose-ui.yml -p kafka-ui down`** **`#> docker-compose -f docker-compose-full.yml -p full up -d`** **`#> docker-compose -f docker-compose-full-sample.yml -p full-sample up -d`**

Running containers

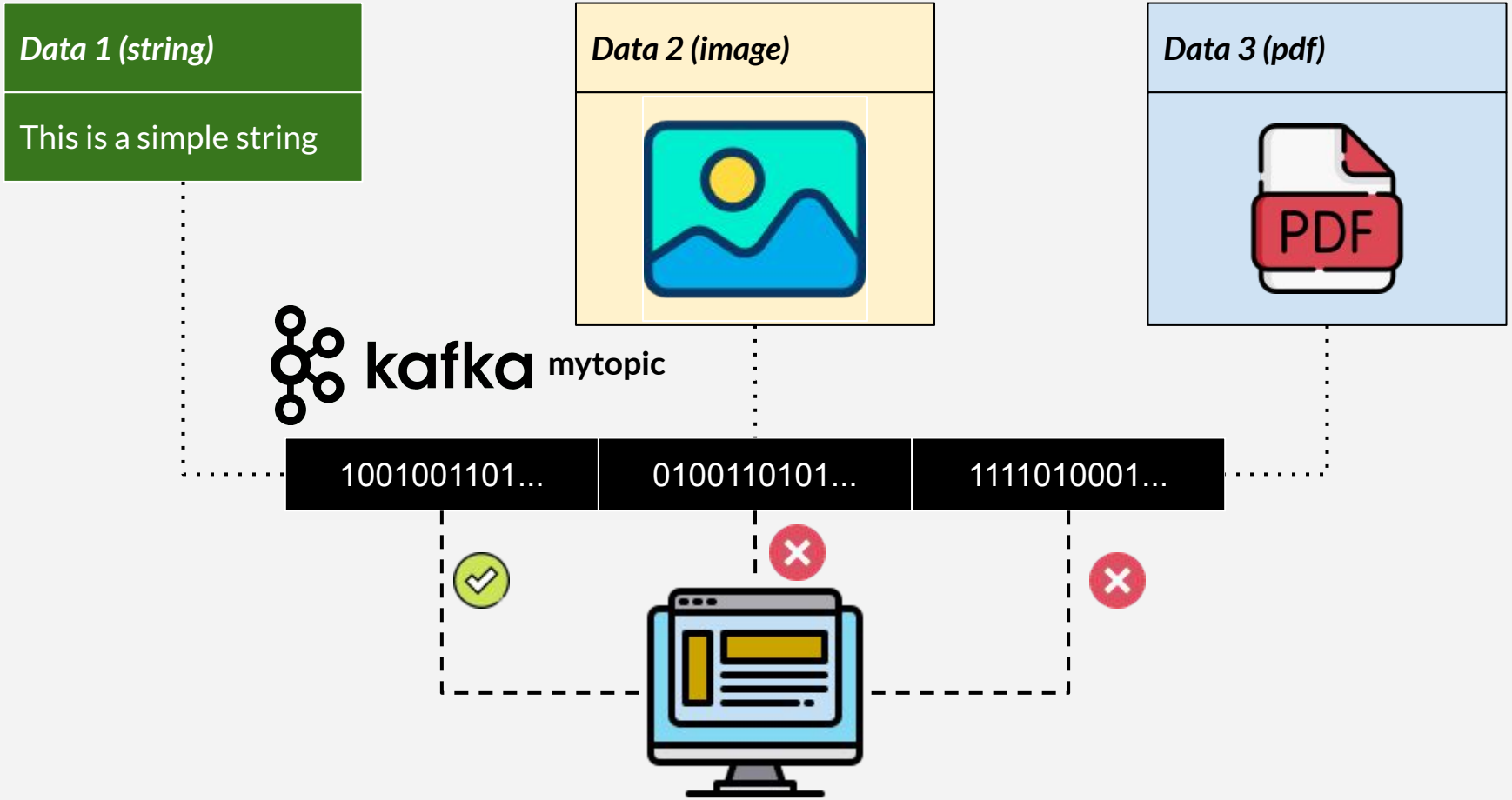
- Kafka at port 9092
- Kafka connect at port 8083
- Schema registry at port 8081
- REST Proxy at port 8082
- KsqlDB at port 8088

JSON Drawback

Kafka Topic

- Kafka topic saves data as binary (0s and 1s)
- String, JSON, image, etc : saved as binary
- What we have done:
  - Send data as string / JSON
  - Easy with drawback
  - Kafka just stores, not validates
  - Different data in topic might break consumer
  - Not just Java consumer

Different Data



JSON Message

- JSON for kafka is safe?
- JSON has structure
- Good as long as structure same & consistent
- Take binary & convert to json
- Process the json, e.g. save data

Different Data

Data 1

```
{
  "customerId":"BY9327",
  "creditScore":4
}
```

Data 2

```
{
  "customerId":"ZW0297",
  "creditScore":”GOOD”
}
```

Data 3

```
{
  "customerId":"LP2715",
  "credit_score":3
}
```

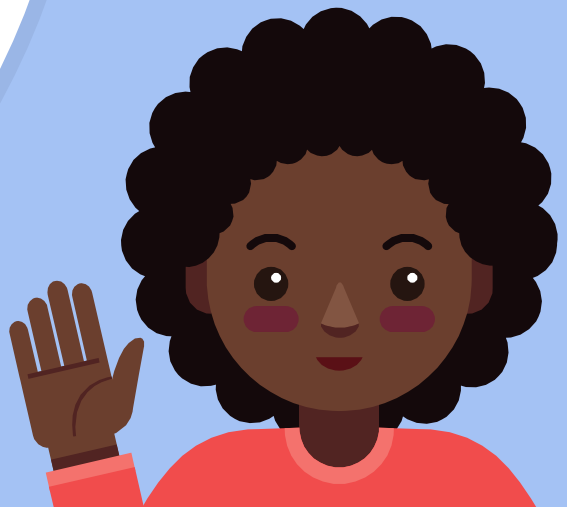
kafka myTopic

Column	Data type	Allow null?	Default value
customer_id	varchar(20)	no	none
credit_score	int	no	none

Different Data

- Producer send different data?
- Coding change on producer side
- Multiple producers
  - Multiple teams
  - Does not aware exact JSON structure
  - Mobile team vs web team

Tip : Binary Data



Sending Binary Data

- Send binary to kafka
- Send image file
- Algorithm, no step-by-step hands on
- Try it yourself

Sending Binary Data

- Send base64 encoded file
- Libraries:
  - Apache commons IO
  - Base64 encoder
- Value serializer : **`StringSerializer`**
- Value deserializer : **`StringDeserializer`**

Sending Binary Data

```
KafkaTemplate<String, String> kafkaTemplateFile;

void send(File file) {
    var fileBytes = FileUtils.readFileToByteArray(file);
    var fileBase64 = Base64.getEncoder().encodeToString(fileBytes);

    kafkaTemplate.send(topic, fileBase64);
}
```

```
kafka-topics.sh --bootstrap-server localhost:9092 --create --topic the-topic
--partitions 1 --replication-factor 1
```

Tip : Large Message

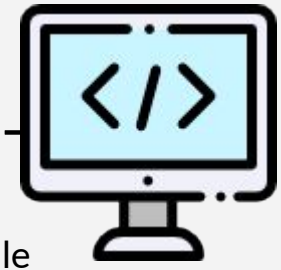
Big Message

- × Kafka is not meant for large data
- × Default message size : 1 MB max
- × Works best for huge message amount

Large file

File storage

kafka



Consumer

4. Listen & retrieve file from storage

1. Save file to storage

```
Kafka message value

{
  ...
  "fileRef" : "https://storage.com/path/to/my-file"
  ...
}
```

Producer

2. Get file reference (URL)

3. Send file reference

Large file

Producer

1. Split into smaller parts (chunks)

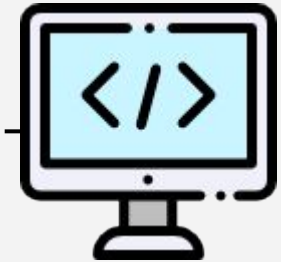
Large file Part 1	Large file Part 2	...	Large file Part n

kafka

2. Send file chunks

```
{
  ...
  "part" : 1,
  "totalParts" : n,
  "chunk" : "base64-chunks"
  ...
}
```

3. Get & cache file chunks



Consumer

4. assemble

Part 1	Part 2
Part ...	Part n

Large file

Which One?

- 1st approach (by file reference) is easier
- Partition & re-assemble needs to maintain data integrity & performance
- 2nd approach : put key on chunks
  - All chunks go to same partition
  - Same consumer got whole chunks

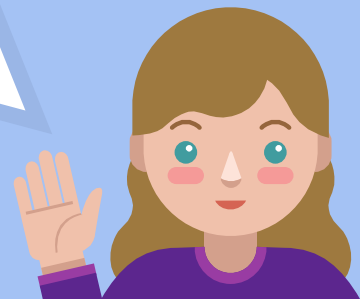
The Need for Schema

Database Case

- Enforces things
- Column name
- Data type
- Database will validates data before insert

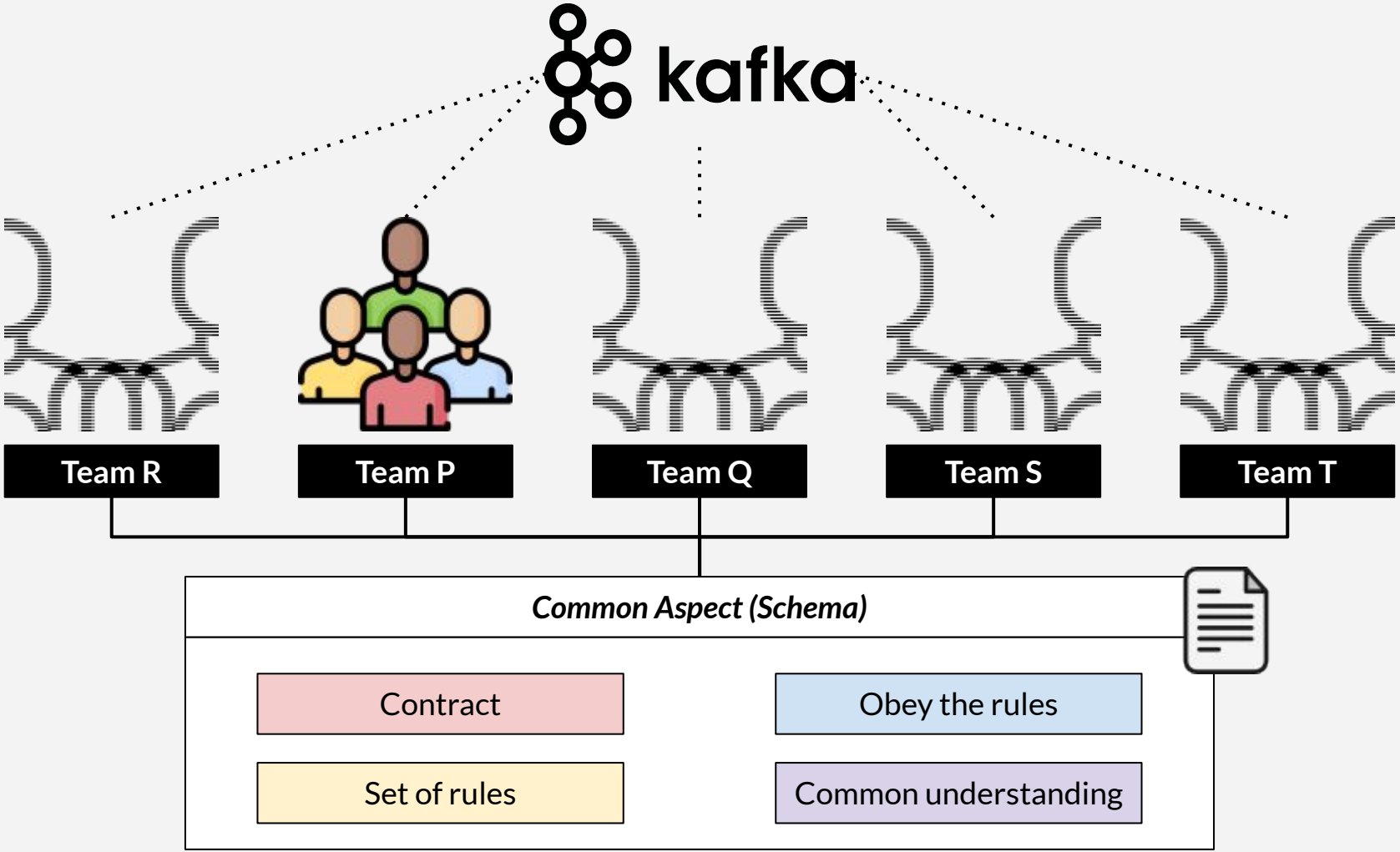
Data Validation in Kafka

- Same thing in kafka?
- Validates field name & type before insert
- Validates against rule/ schema
- Not in kafka



Data Validation in Kafka

- × No validation on kafka
- × Kafka does not process data
- × Kafka does not care about data
- × These things makes kafka fast
- × Enforcing validation is not for kafka
- × There’s still hope


kafka
Team R
Team P
Team Q
Team S
Team T
Common Aspect (Schema)
Contract
Obey the rules
Set of rules
Common understanding

Schema

Database table
Column name
Column data type (int, varchar, …)
Optional (nullable) column
Column default value
Table evolution

Message Schema
Field name
Field data type (int, string, …)
Optional (nullable) field
Field default value
Schema evolution & versioning

Schema Registry

What We Needs?

Schema

Describes data structures
- fields name
- data type
- allow null
- default value

Avro

Schema Registry

Accessible by kafka producers & consumers

Separated item from kafka broker

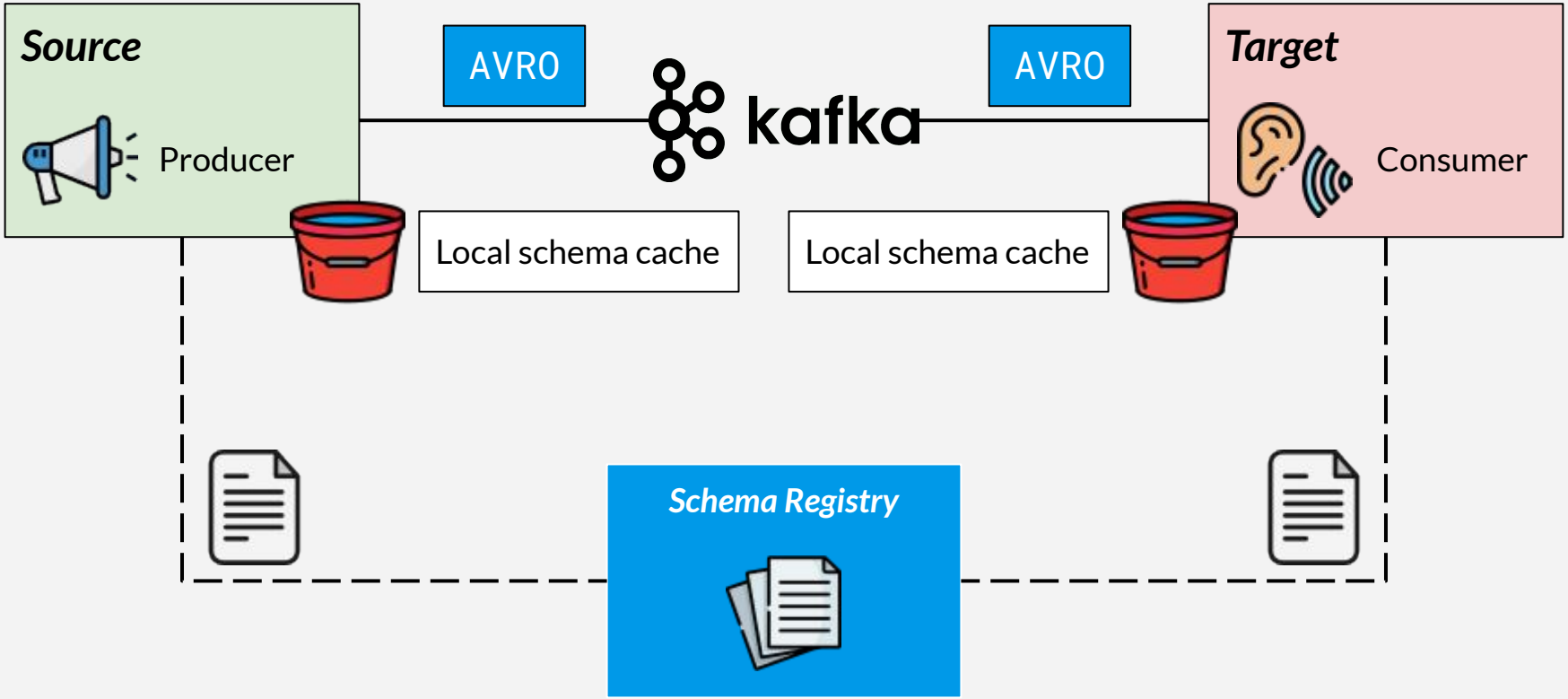
Validate & reject bad data

Confluent schema registry

Lightweight
Support evolution
Don’t reinvent the wheel

The Architecture

The Architecture (with Schema Registry)



What is Avro

Data Transfer

Timestamp
Country
Location (lat, lang)
Weather

System X

kafka

System Y

mongoDB.

timestamp	country	location	weather
Long, epoch time	*String, ISO 3166-1 country code*	*Point (lat, long)*	*String, custom weather code*
1640945914	ID	-6.3040, 106.6435	STORM
1752317434	ID	-6.3040, 106.6435	LIGHT_RAIN

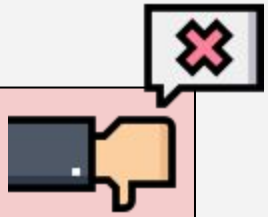
Data Transfer

- From postgre to mongodb?
- Database-to-database is not practical
- Kafka
- Data format on kafka?
- Recognized by both system

CSV

1752231025,ID,-6.3040,106.6435,STORM
1752317434,ID,-6.3040,106.6435,LIGHT_RAIN

- Good programming language support



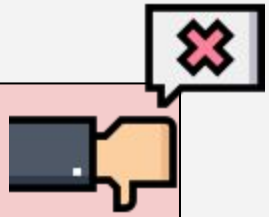
- Sensitive column order
- No data type validation : change timestamp from epoch second to string 2025-07-12T10:50:25.000Z
- No data structure (lat & long must be on separated column)

JSON

```
{
   "timestamp":1752231025,
   "country":"ID",
   "geolocation":{
      "lat":-6.3040,
      "long":106.6435
   },
   "weather":"STORM"
}
```

```
{
   "weather":"LIGHT_RAIN",
   "country":"ID",
   "timestamp":1752317434,
   "geolocation":{
      "lat":-6.3040,
      "long":106.6435
   }
}
```

- Good programming language support
- Column order is not matter
- Good data structure (JSON elements, array)



- No data type validation : change timestamp from epoch second to string 2025-07-12T10:50:25.000Z

Avro

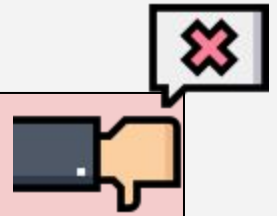
Avro Schema

```
Timestamp : long, not null
Country : string, not null
Weather : string, not null
Geolocation : geolocation
(another avro), not null
```

Avro Body (binary)

```
{
   "weather":"LIGHT_RAIN",
   "country":"ID",
   "timestamp":1752231025,
   "geolocation":{
      "lat":-6.3040,
      "long":106.6435
   }
}
```

- Column order is not matter
- Good data structure (AVRO elements, array)
- Enforce data validation with schema
- Binary (less size)
- Integrates well with hadoop (big data)
- Schema evolution

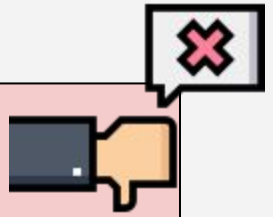


- Less wide programming language
- Read / write data is not straightforward

JSON + JSON Schema

`https://json-schema.org`

- Good programming language support
- Column order is not matter
- Good data structure (JSON elements, array)
- Use json schema for validation



- Less known & less integration compared to avro.

Why Avro?

- Current schema registry supports avro, protobuf, & JSON schema
- Why avro?
  - Big data ecosystem
  - Requests for course update on avro
- Alternative: protobuf (Protocol Buffer)
  - Similar concept with avro
  - Kafka schema registry for avro / protobuf

Avro Theory

Avro Schema Definition

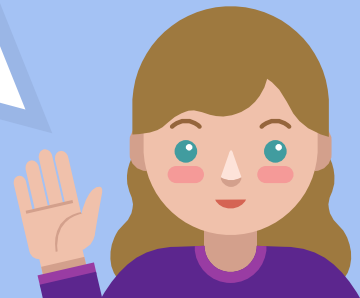
```
{
  "type": schema type as defined on avro specification,
  "namespace": like package in Java,
  "name": schema name,
  "aliases": (optional) array of name alias for this schema,
  "doc": (optional) documentation,
  "fields": array of fields specification [
    "name": field name,
    "type": field type, sometimes combined with "logicalType",
    "default": (optional) default value for field.
    "doc": (optional) field documentation,
    "order": (optional) field sort order
    "aliases": (optional) array of field alias
  ]
}
```

Avro Schema Definition

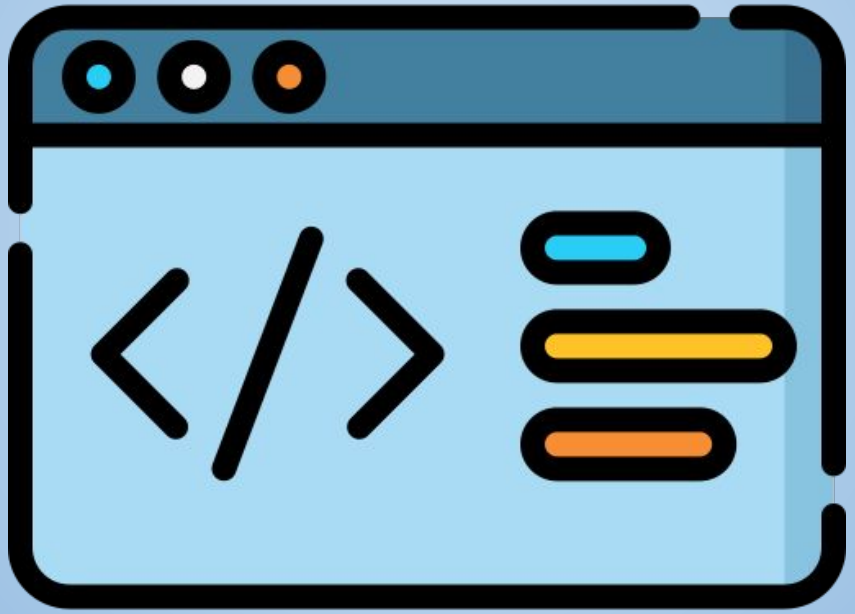
```
{
  "type":"record",
  "namespace":"com.course",
  "name":"AvroExample",
  "aliases":["AvroSample", "SampleAvro"],
  "doc":"This is just an example doc",
  "fields":[
    {"name":"name", "type":"string"},
    {"name":"birthDate", "type":"long", "logicalType":"timestamp-millis"},
    {"name":"email", "type":"string"},
    {"name":"maritalStatus", "type":"string", "default":"UNKNOWN"}
  ]
}
```

Avro Schema

- **`type`**
  - Primitive : int, string, boolean, etc
  - Complex :
    - `record` (mostly will use this)
    - `enum`
    - `array`
    - `map`
    - `union`
    - `fixed`
- **`logicalType`**
  - Give more meaning to primitive

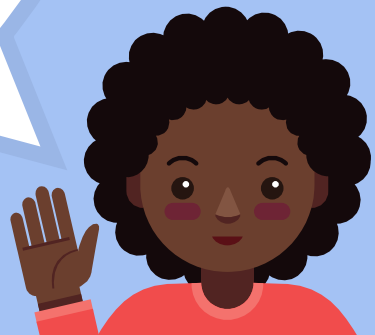


Avro Primitive Types



Primitive Types

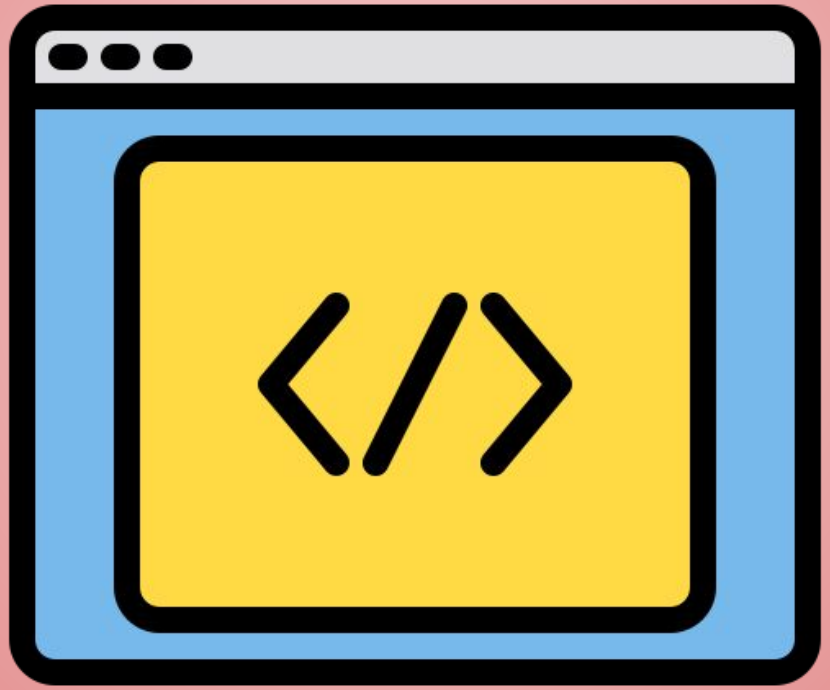
- **`null`**: no value
- **`boolean`**: a binary value (true/false)
- **`int`**: 32-bit signed integer
- **`long`**: 64-bit signed integer
- **`float`**: 32-bit floating-point number
- **`double`**: 64-bit floating-point number
- **`bytes`**: sequence of 8-bit unsigned bytes
- **`string`**: unicode character sequence



Avro Schema Definition

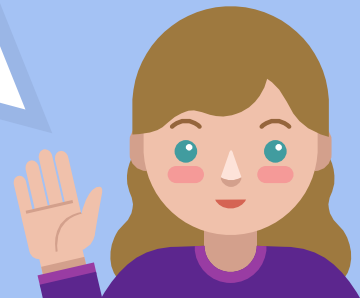
```
{
   "type":"record",
   "namespace":"com.course",
   "name":"AvroExample",
   "aliases":["AvroSample", "SampleAvro"],
   "doc":"This is just an example doc",
   "fields":[
      {"name":"name", "type":"string"},
      {"name":"email", "type":"string"},
      {"name":"customerRating", "type":"float"},
      {"name":"acceptPromotionEmail", "type":"boolean"}
   ]
}
```

Avro Logical Types



Logical Types

- More than primitive
- Data type is common
- Logical type gives more meaning to existing primitive
- Example:
  - **`decimal`** - bytes
  - **`uuid`** - string
  - **`date`** - int
  - **`time-millis`** - int
  - **`timestamp-millis`** - long
  - … (complete at avro documentation)



Enriching?

```
{
    "name":"myUuid",
    "type":"string",
    "logicalType":"uuid"
}
```

```
{
    "name":"birthDate",
    "type":"int",
    "logicalType":"date"
}
```

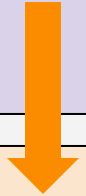
Avro Complex Types

Avro Complex Types

- `record`
- `union`
- `enum`
- `array`
- `map`

Record

```
{
   "type":"record",
   "namespace":"com.course",
   "name":"Weather",
   "fields":[
      {"name":"timestamp", "type":"long"},
      {"name":"country", "type":"string"},
      {"name":"weather", "type":"string"},
      {"name":"location", "type":"com.course.Location"}
   ]
}
```



```
{
   "type":"record",
   "namespace":"com.course",
   "name":"Location",
   "fields":[
      {"name":"latitude", "type":"double"},
      {"name":"longitude", "type":"double"}
   ]
}
```

Union

One field allow multiple data types

Usually to define optional field

First allowable type is null

```
{
    "name":"myField",
    "type":["int","string"]
}
```

```
{
    "name":"myOptionalField",
    "type":["null","float"]
}
```

```
{
    "name":"myOtherField",
    "type":["boolean","int"],
    "default":true,
}
```

Enum

```
{
   "type":"enum",
   "namespace":"com.course",
   "name":"Gender",
   "symbols":[
      "MALE",
      "FEMALE"
   ]
}
```

List of values

- Gender (male, female)
- Customer status (regular, premium, vip, etc)

Define values as array of string

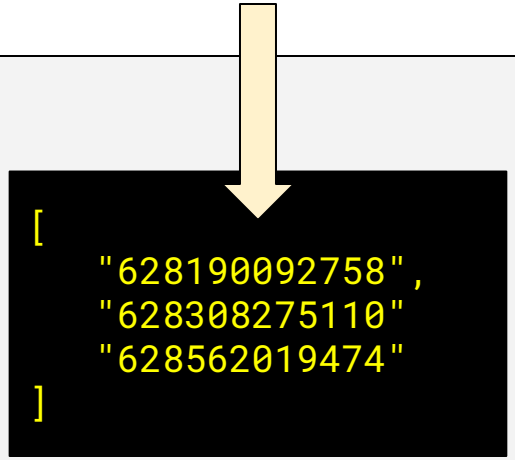
Once set, changing enum values is forbidden to maintain compatibility

Add / remove value is forbidden

Array

List elements with same type
Dynamic size
Define element type at "**items**"

```
{
   "name":"phoneNumbers",
   "type":"array",
   "items":"string"
}
```



Map

```
{
   "name":"favourites",
   "type":"map",
   "values":"string"
}
```

List of key-value pair
One key - one value
Key is string

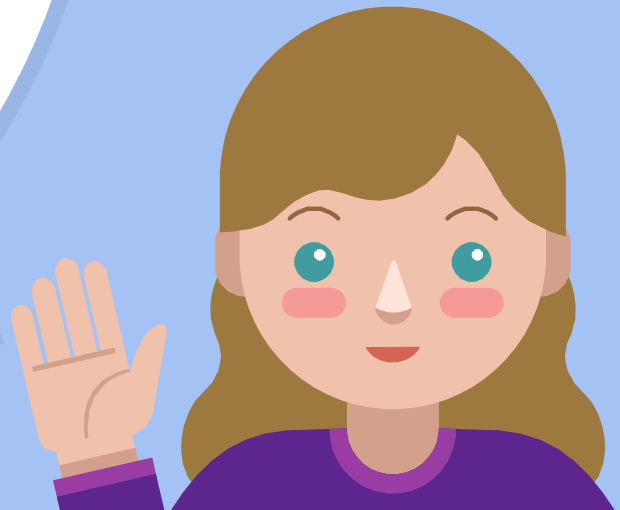
Fixed

```
{
   "name":"sha256Hash",
   "type":"fixed",
   "size":32
}
```

Binary data with fixed length (bytes)

Avro Hands-On

Hello Avro



Getting Started

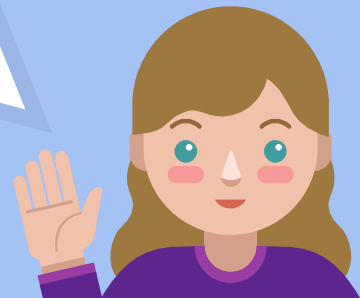
- Avro not depends on spring
- Write avro schema (avsc), then generate java class
- Video:
  - How to start (dependency, plugin)
  - Not line by line
  - Source code & avro schema available in Resource & Reference

Avro Plugin

- Maven?
- Avro plugin for Maven is available
- Outside the course scope (please google it)

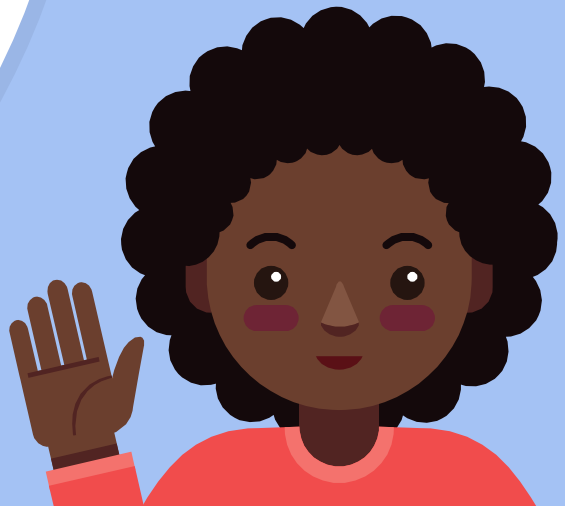
Avro Plugin

- https://github.com/bakdata/gradle-avro-dependency-plugin
- Gradle / Maven plugin is optional
- Can generate Java source code without plugin
- The course will use plugin for convenience



Avro Hands-On

Generic



Generic Avro

- Generic builder avro
- Process avro direct from schema (no java class)
- Schema : file or string
- **Not a recommended way :**
  - Set everything manually
  - Error prone (typo, wrong data type)
  - Runtime failure risk
- For knowledge only

Avro Hands-On

Specific

Specific Avro

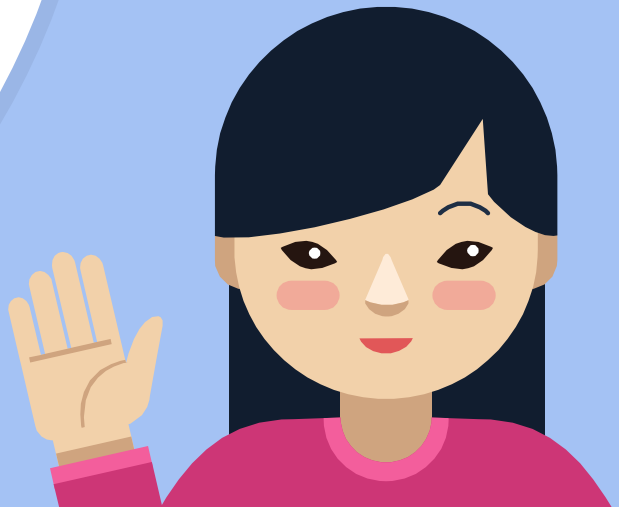
- **Recommended way!**
- Use Gradle or Maven plugin
- Schema > java class
- Process the java class
- Not just simple POJO
- Don’t mix avro class with other (e.g. JSON / database table representation)

Specific Avro

- Focus on avro structure
- More or less same algorithm
- Download source & schema from Resource & Reference
- Compile error? No worry
- Run gradle / maven task to generate avro java class

Avro Hands-On

Schema From JSON



Reflection from Java

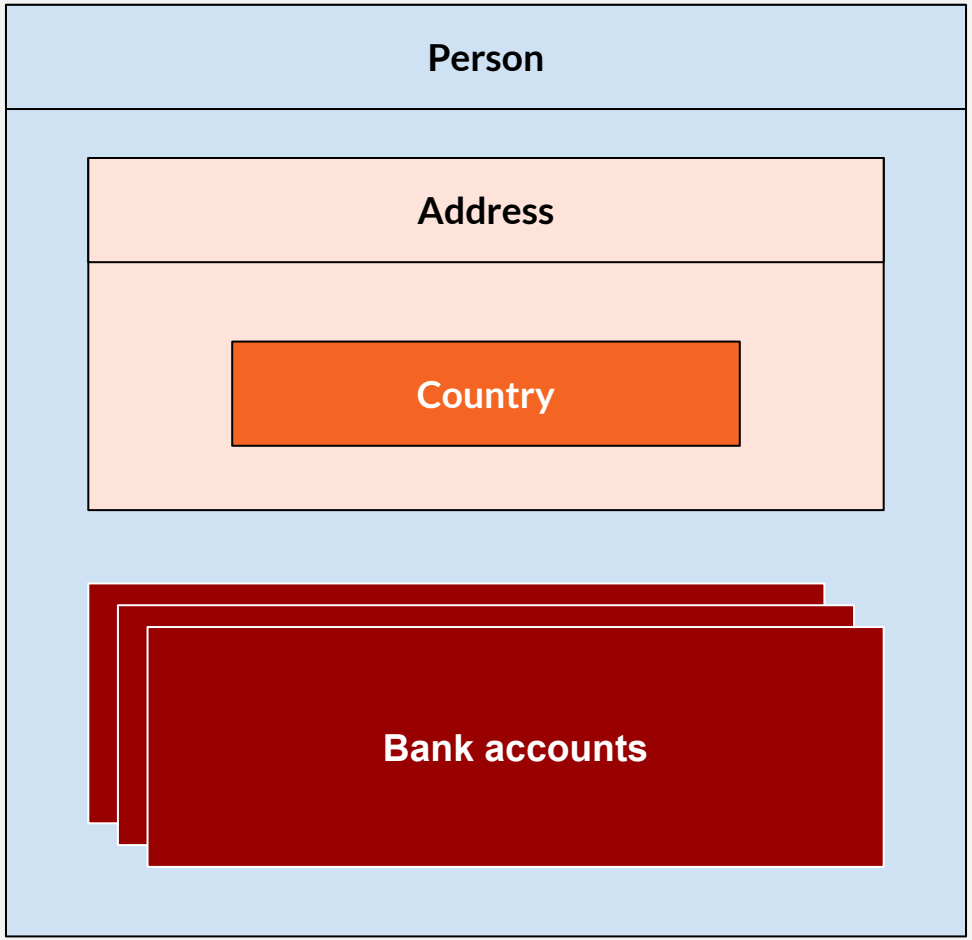
- Existing application to avro
- Alternatives:
  - Write schema from scratch
  - Use avro reflection
- From java class to avro schema
- Not perfect
- For starting point

From JSON

- × No java source
- × JSON result
- × Online generator
- × Starting point only

Avro Hands-On

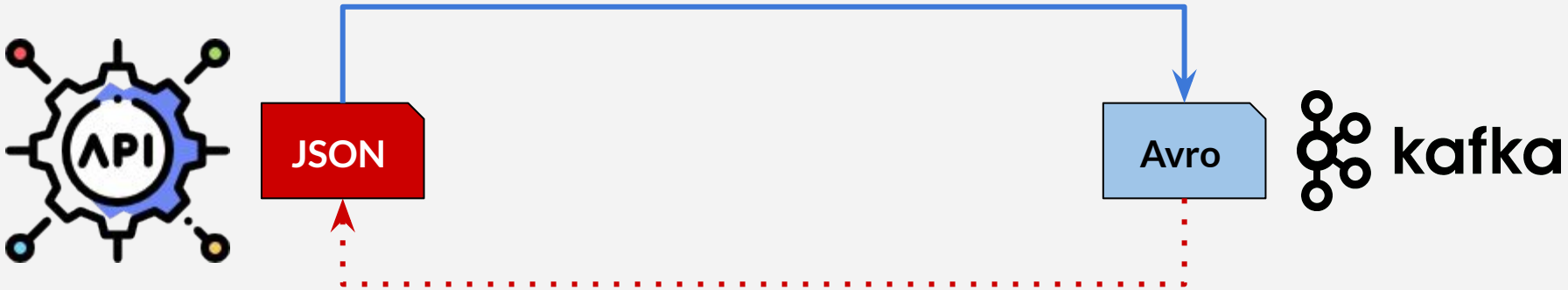
Nested Record


Person
Address
Country
Bank accounts

Avro Hands-On

From JSON to Avro Binary

From JSON to Avro (and Back Again)



Using Jackson as JSON library

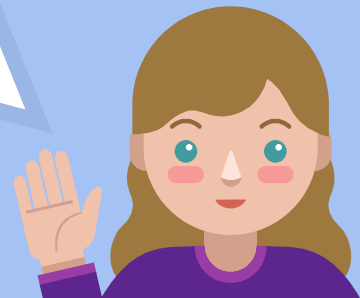
```
{
   "title":"Book title",
   "author":"Book author",
   "price":123
}
```

Avro Hands-On

Avro Tool

Avro Tools

- Avro contains unreadable binary characters
- Quick look on avro content
- Writing every reader is troublesome
- Avro tools to the rescue



Avro Hands-On

Generate Java Without Plugin

Generate Java Using Avro Tool

```
#> java -jar /path/to/avro-tools.jar compile schema <schema file> <destination>
```

Example

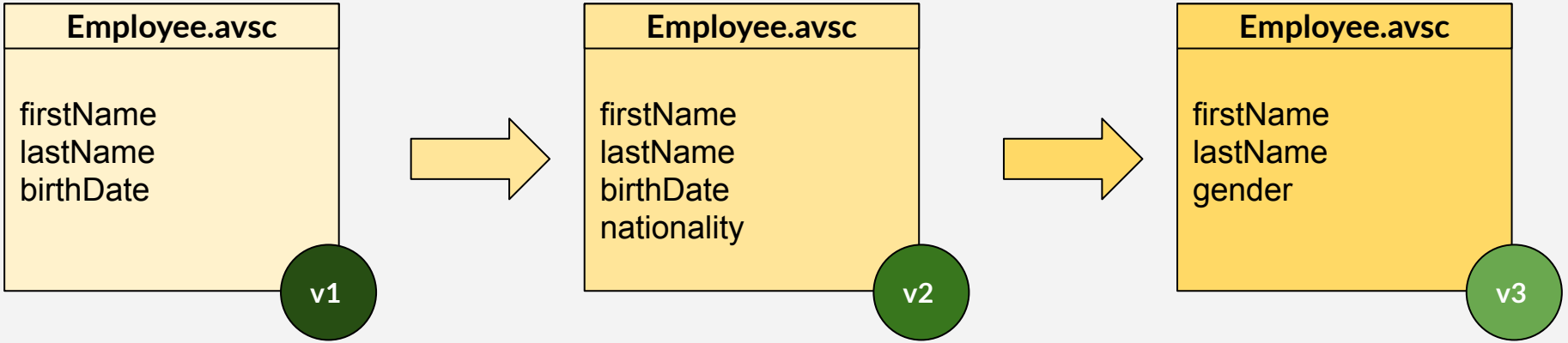
```
#> java -jar /path/to/avro-tools.jar compile schema HelloSchema.avsc /my/output
```

What is Schema Evolution?

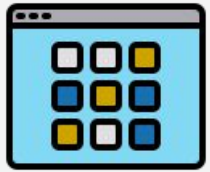
Evolution

- Application might change in future
- Change impact data (in avro format)
- Avro schema might change
- Avro enables schema changes over time
- Schema evolution
- The most important feature

Evolution


Employee.avsc
firstName
lastName
birthDate
v1
Employee.avsc
firstName
lastName
birthDate
nationality
v2
Employee.avsc
firstName
lastName
gender
v3

App 1



App 2

App 3

kafka

Avoid breaking consumers

v1

v2

App 2

App 3

v3

App 2

App 3

Schema Evolution

Type	Meaning	Check against which schema?
BACKWARD	New schema can be used to read old data (data created using previous schema)	Latest (current - 1)
BACKWARD_TRANSITIVE		All previous versions
FORWARD	Old schema can be used to read new data (data created using later schema)	Latest (current - 1)
FORWARD_TRANSITIVE		All previous versions
FULL	Both backward and forward	Latest (current - 1)
FULL_TRANSITIVE		All previous versions
NONE	No compatibility checking	-

Employee.avsc

v1

```
{
  "type": "record",
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name":
  ]
}
```

Backward compatible

We can read v1 data (old schema) using v2 schema (new schema)
Older data will have maritalStatus UNKNOWN (default value)

v2

```
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name": "lastName",  "type": "string"},
    {"name": "maritalStatus",  "type": "string", "default": "UNKNOWN"}
  ]
}
```

Employee.avsc

v1

```
{
  "type": "record",
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name":
  ]
}
```

Not backward compatible

App with v2 schema can't read v1 data, no default value on maritalStatus

v2

```
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name": "lastName",  "type": "string"},
    {"name": "maritalStatus",  "type": "string"}
  ]
}
```

Employee.avsc

v1

```
{
  "type": "record",
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name":
  ]
}
```

Forward compatible

We can read v2 data (new schema) using v1 schema (old schema)
Application with v1 schema will simply ignore email field

v2

```
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name": "lastName",  "type": "string"},
    {"name": "email",  "type": "string"}
  ]
}
```

Employee.avsc

v1

```
{
  "type": "record",
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name":
  ]
}
```

Not forward compatible

App with v1 schema can't read v2 data, no default value on lastName

v2

```
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"}
  ]
}
```

Employee.avsc

v1

```
{
  "type": "record",
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name":
  ]
}
```

Full compatible

Add field with default value
Remove field which has default value

v2

```
  "namespace": "com.course",
  "name": "Employee",
  "fields": [
    {"name": "firstName", "type": "string"},
    {"name": "location",  "type": "string", "default": “head-office”}
  ]
}
```

Not compatible

- × Change type (e.g from **string** to **long**)
- × Renaming a required field (field without default)
- × Add / remove / modify enum element

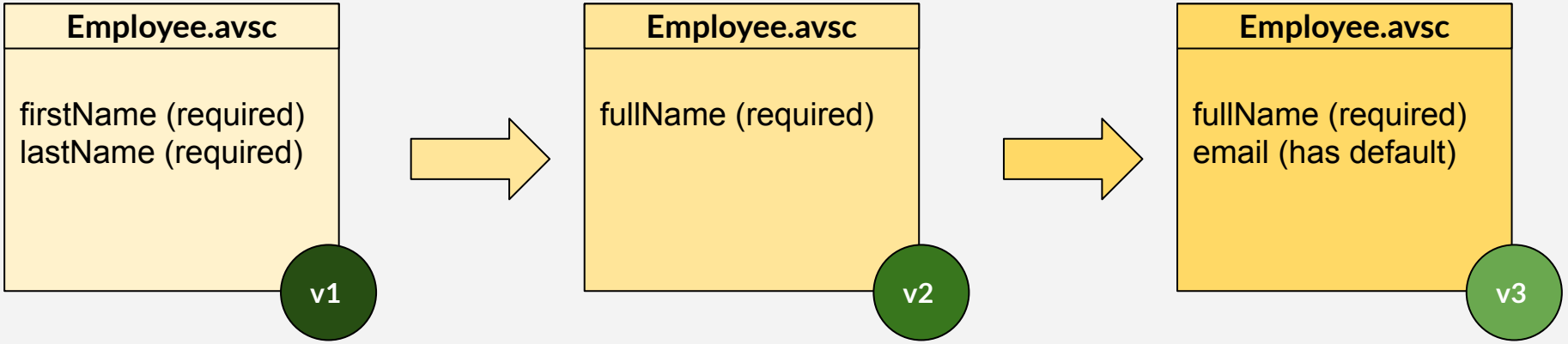
Guidance on Schema Evolution

- × Default value for fields (unless you know exactly)
- × Use alias instead renaming fields
- × Be careful on enum

Sample Use Cases

- Backward : query data stream
- Forward : change data stream without change consumers
- Full : target compatibility

Transitive



Not focus much on transitive
Set from early so implicitly match

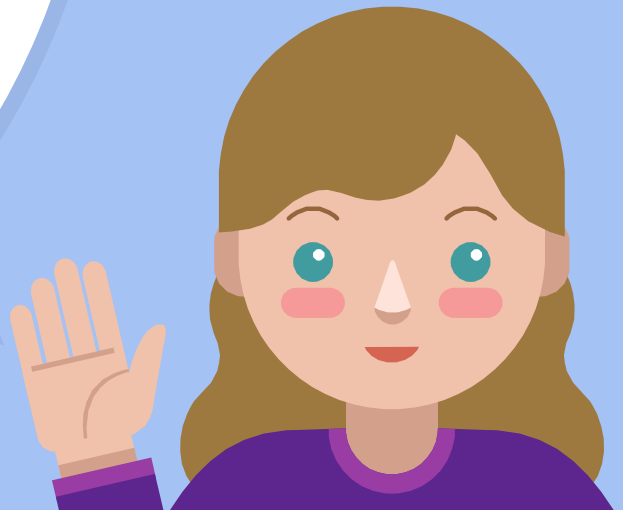
Forward
Backward
Full

Transitives

Avro Schema Evolution

Hands On

What Is Schema Registry?



What We Needs?

Schema

Describes data structures
- fields name
- data type
- allow null
- default value

Avro

Schema Registry

Accessible by kafka producers & consumers

Separated item from kafka broker

Validate & reject bad data

Confluent schema registry

Don't use git

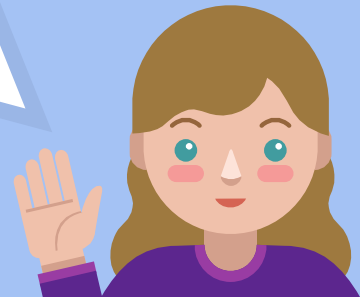
Lightweight
Support evolution
Don't reinvent the wheel

Confluent Schema Registry

- Free from confluent.io
- Save & restore schema
- Create, read, update, delete schema
- Validates vs schema on topic
- Works for key and / or value

Confluent Schema Registry

- Schema evolution support
- Set global or per-topic compatibility
- Smaller payload size
- Avro, protobuf, JSON schema
- This course focus on avro
- Protobuf course (with Go) discount code available on **Resources & References**
- Serializer / deserializer from confluent

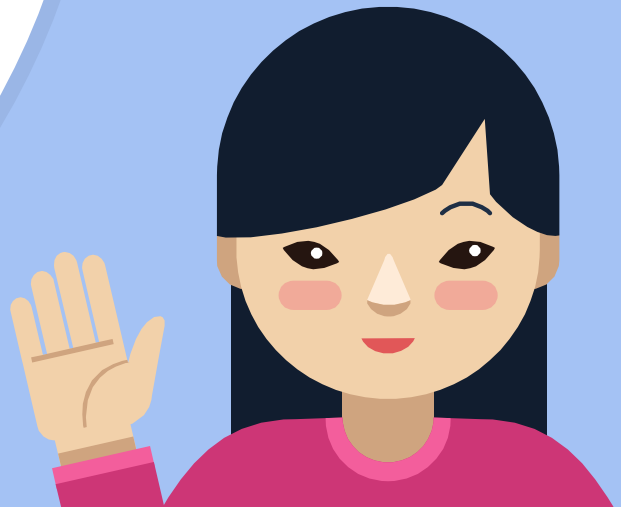


Confluent Schema Registry

- REST API (out-of-the-box)
- Sample API available at Resource & Reference
- Complete API on confluent website
- Subject
  - Schema registered for certain usage
  - mytopic-key : schema for message key
  - mytopic-value : schema for message payload

Schema Registry

In This Course



User Interface

- This course uses conduktor
- Can use built-in REST API for schema registry
- Postman collection available at **Resource & Reference**

Schema Registry

Hands On

Schema Registry Operation

- REST API / User Interface application
- This course uses Conduktor UI
- Check conduktor website for pricing & license (*conduktor.io*)

Practice

- Use conduktor.io UI
- Feel free to use others
- Topic name for lesson : `sc-...`
- Feel free to use your own topic

Avro & Spring Hands On

Getting Started

Spring Initializr

- From *start.spring.io*
- Dependency : Spring for Apache Kafka, Spring for Apache Kafka Streams (on consumer only)
- Settings:
  - Java project with Gradle
  - Spring Boot 3.x
  - Group : **`com.course.kafka`**
  - Artifact : **`kafka-avro-producer`** and **`kafka-avro-consumer`**
  - Adjust package name to **`com.course.kafka`**
  - Java version : 17 or later

Avro & Spring Hands On

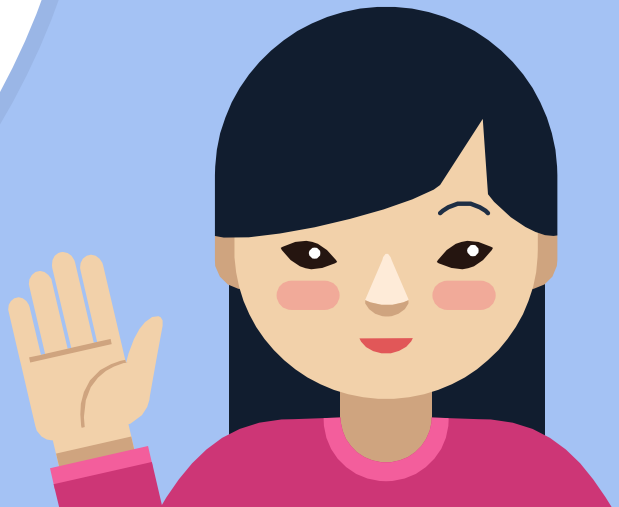
Avro Producer 1

Schema Naming

- Key : **`topicname-key`**
- Value : **`topicname-value`**
- Sending data first (schema auto generated)
- Creating schema first in kafka

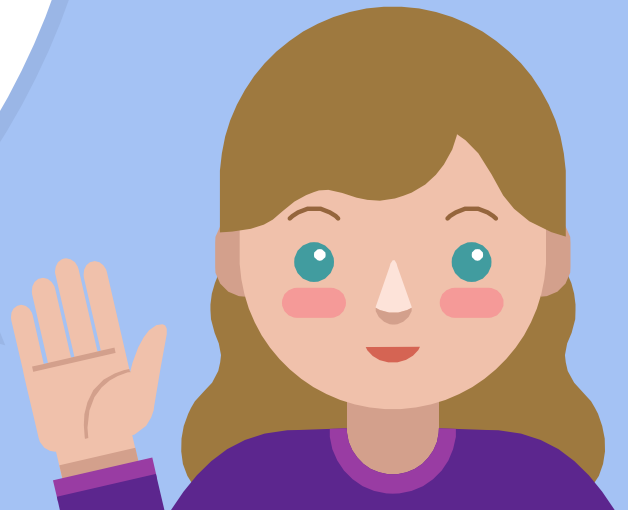
Avro & Spring Hands On

Avro Consumer 1



Avro & Spring Hands On

Avro Producer & Consumer 2

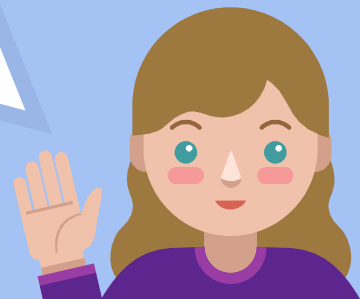


Avro Producer Summary

- No difference on Kafka code
- Steps
  a. Generate class from avro schema
  b. Use **`KafkaAvroSerializer`**
  c. **`schema.registry.url`**
  d. Use **`KafkaTemplate`**

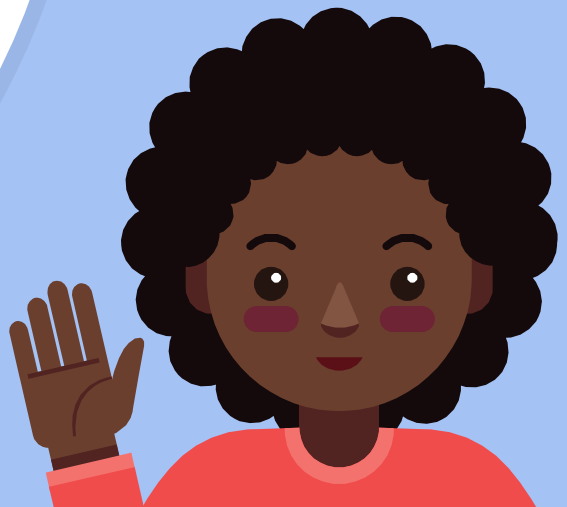
Avro Consumer Summary

- Steps
  a. Generate class from avro schema
  b. Use **`KafkaAvroDeserializer`**
  c. **`schema.registry.url`**
  d. Use **`KafkaListener`**



Avro & Spring Hands On

Kafka Stream



Avro Stream Summary

- Similar syntax for Kafka Stream (with previous lesson)
- Steps
  a. Define configuration map containing schema registry URL
  b. Use **`SpecificAvroSerde`**
  c. Use the configuration map on **`SpecificAvroSerde`**
  d. Create the stream

Avro & Spring Hands On

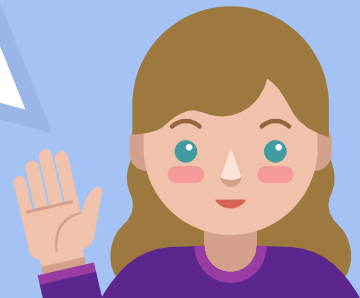
Backward Compatibility

What We Will Write

- Backward compatible demo
- Produce using V1
- Consume using V2
- Consumer updated first

How We Do It

- No **`MyAvroV1.java`** or **`MyAvroV2.java`**, just **`MyAvro.java`**
- Same namespace and name in avro schema
- Plugin will generate same namespace & name on java class
- Schema content (fields) can be different



Schema V2?

- × Not put version 2 using user interface
- × Consumer do this update
- × Can also put version 2 into registry first

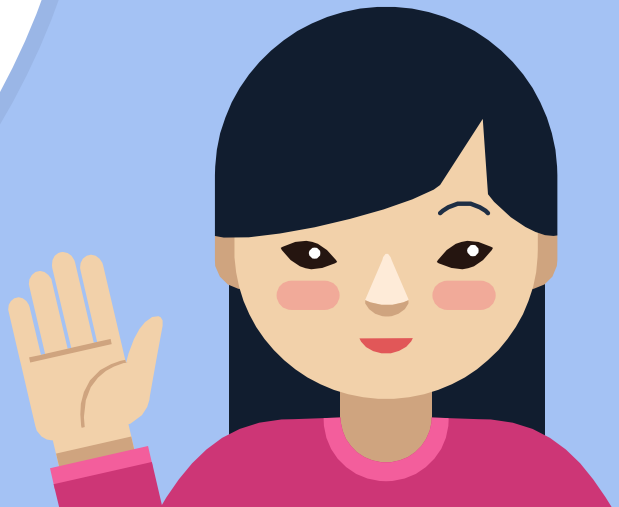
Avro & Spring Hands On

Forward Compatibility

What We Will Write

- × Forward compatible demo
- × Produce using V2
- × Consume using V1
- × Producer updated first

Full Compatibility & Tips



Full Compatible

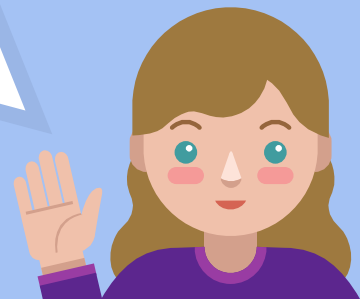
- Write using V1, read using V2
- Or the other way around
- Java code is not different
  - Don’t forget to use same namespace & name on avro schema
  - No versioning (V1 / V2 / V…) on schema name
- Set FULL compatibility on schema registry

Schema Registry

- Critical component
- Make sure it is highly available
- Aim for full compatible schema

Schema Workflow Tips

- Assume familiarity with git
- Don’t put every schema changes on schema registry
- Use git for schema change draft
- Work on git branch for update
- Merge final change on master branch
- Merge request (pull request) approval & review
- Publish final updated schema to schema registry

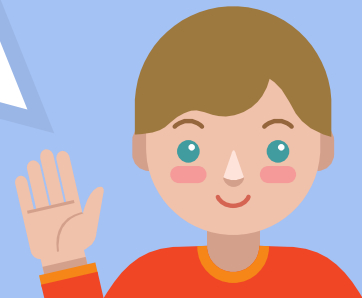


Schema Registry API

Quick Look

API Quick Look

- Brief overview on schema registry API
- Built-in Confluent API
- See official documentation for full reference
- Quick look postman collection on Resource & Reference



Kafka Connect & Schema Registry

Overview

Kafka Connect Schema

- Kafka connect use schema
- Embedded on JSON body (depends on converter)
- JSON Converter does not need schema registry
- Kafka connect avro converter available

Kafka Connect Converter

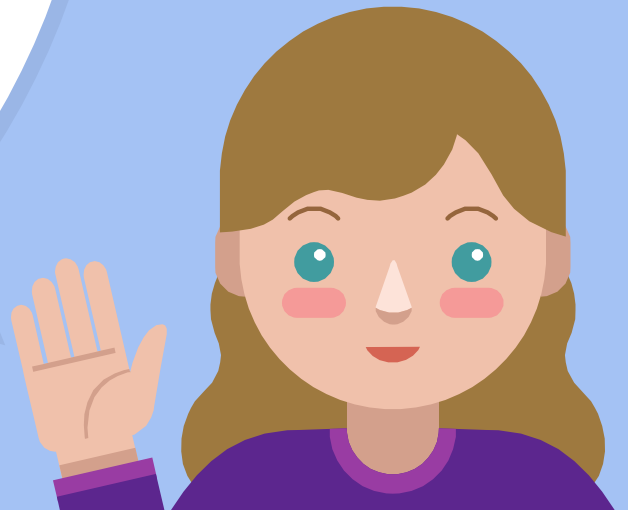
- Key & value converter can be different
- Avro converter need schema registry running
- Benefit vs embedded schema
  - Message size reduced (schema not included on message)
- Need to maintain schema registry
- Consider your choice
- Remember : can set default converter & override

Kafka Connect & Schema Registry

- Source connector automatically generates schema
- Auto-generated schema saved on schema registry
- Sink connector read existing schema from schema registry

Kafka Connect & Schema Registry

Source Connector & Consumer



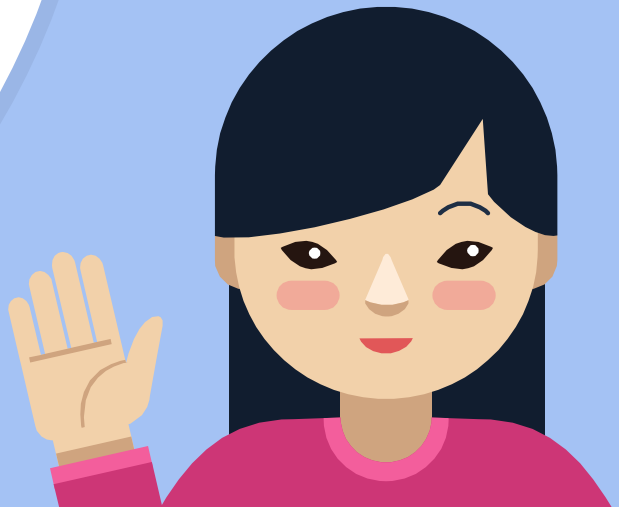
Kafka Connect & Schema Registry

Producer

Kafka Connect & Schema Registry

Sink Connector

What Is Kafka REST Proxy



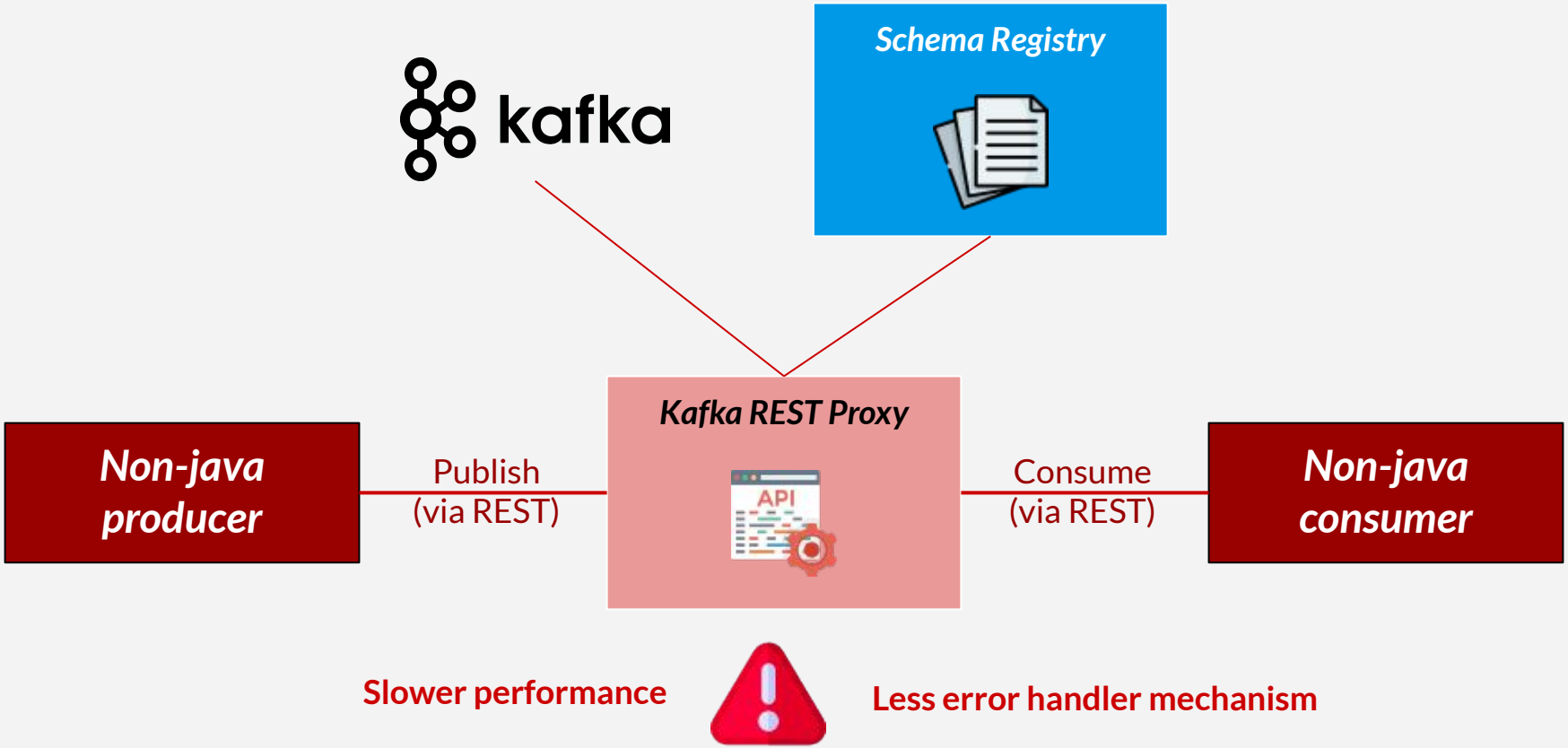
Learning Kafka

- × Good integration with Spring
- × Spring **`KafkaTemplate`** & annotations
- × Not everybody uses Spring (or even Java)
- × Learning curve on native Kafka library
- × Source might be less compared to Spring / Java
- × Shorten the learning curve by not coding kafka
- × What?

Kafka REST API

- Write codes to publish & consume using REST API
- Shorter learning curve (assuming familiarity with REST API)
- REST API is not kafka built-in feature
- Confluent provides Kafka REST Proxy (open source)

Architecture (with Kafka REST Proxy)



What We Will Learn

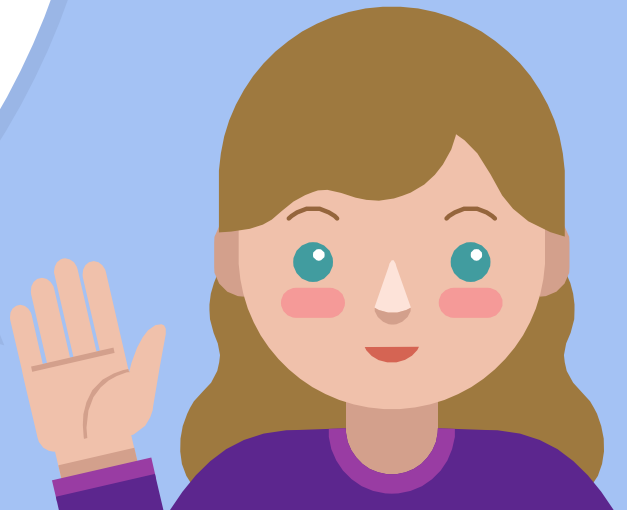
- Some operation on Kafka REST Proxy
- Complete reference refer to confluent documentation for Kafka REST Proxy
- Download postman collection from Resource & Reference
- Kafka REST Proxy on localhost:8082

API Version for Kafka REST Proxy

- When recording this video : v2 and v3 endpoints
- Generally, use v3 (newer)
- v2 migrated to v3?
- Use v2 and v3 at this video

Kafka REST Proxy

Cluster & Broker



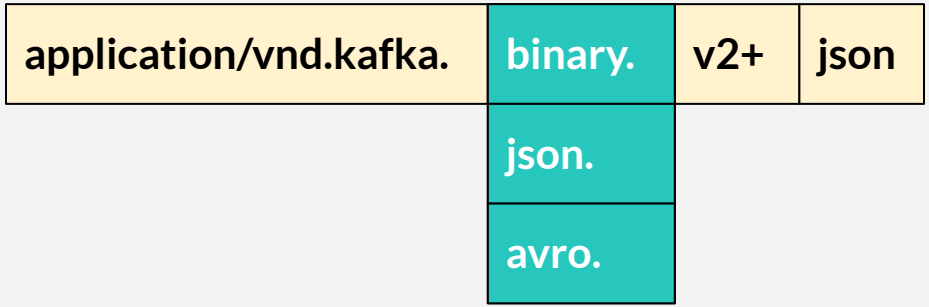
Kafka REST Proxy

Topic

Kafka REST Proxy

Produce Binary

Content-Type Request Header (v2)



application/vnd.kafka.binary.v2+json

application/vnd.kafka.json.v2+json

application/vnd.kafka.avro.v2+json

Kafka REST Proxy

Consume Binary

Consuming with API

- Steps
  - Create consumer
  - Subscribe
  - Start consume
- v2 endpoints
- **`Content-Type`** on create & subscribe
- **`Accept`** on consume

Kafka REST Proxy

Produce & Consume JSON

Kafka REST Proxy

Produce & Consume Avro