FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Metadata extraction from tool Kafka
Student:	Michaela Weberová
Supervisor:	Ing. Michal Valenta, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2021/2022

Instructions

The aim of this work is to analyze, design and verify the implementation of a prototype method of metadata extraction from the Kafka tool, especially for the purpose of monitoring data flows within the Manta tool.

Follow these steps:
1. Formalize metadata extraction requirements from Kafka for monitoring purposes.
2. Describe the Kafka tool and analyze the possibilities of obtaining metadata.
3. Based on the analysis, design a system for extracting metadata from the Kafka tool.
4. Implement the module prototype in the Manta tool for metadata extraction from the Kafka tool and evaluate the effectiveness of the proposed procedure on suitable data.

Electronically approved by Ing. Michal Valenta, Ph.D. on 21 October 2020 in Prague.

Bachelor's thesis

METADATA EXTRACTION FROM TOOL KAFKA

Michaela Weberová

Faculty of Information Technology CTU in Prague
Department of Software Engineering
Supervisor: Ing. Michal Valenta, Ph.D.
May 13, 2021

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Michaela Weberová. *Metadata extraction from tool Kafka.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Contents

List of Figures

List of Tables

List of code listings

I would like to take this opportunity to express gratitude to my supervisor Ing. Michal Valenta Ph.D. for valuable advices and Ing. Petr Košvanec for his guidance in Manta and also within this work. Last but not least, I would like to thank my family and friends for their support, especially during the writing of this work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 13. May 2021 . . . .

Abstract

The aim of this thesis is to implement a module integrated into Manta software for metadata extraction from the tool Apache Kafka. The work begins with a general analysis of Kafka and its essential elements. Then it continues with the analysis of different approaches for metadata extraction. The second half of the work is focused on the design and implementation of the functional prototype using the methods examined in the analysis. The last chapter is dedicated to testing and evaluation of the used solution.

Keywords data lineage, Apache Kafka, event streaming, Manta, Schema Registry, Confluent, metadata

Abstrakt

Cílem této práce je implementovat modul sloužící pro extrakci metadat z nástroje Apache Kafka, který bude integrovaný do softwaru Manta. Práce začíná obecnou analýzou Kafky a jejími základními objekty. Poté pokračuje analýzou různých přístupů pro extrakci metadat. Druhá polovina práce se pak už věnuje designu a implementaci funkčního prototypu na základě zkoumaných metod. V poslední kapitole je otestováno a vyhodnoceno použité řešení.

Klíčová slova datová linie, streamování událostí, Apache Kafka, Manta, Schema Registry, Confluent, metadata

List of abbreviations

API	Application Programming Interface
CSV	Comma Separated Values
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
REST	Representational State Transfer
Protobuf	Protocol Buffers
TCP	Transmission Control Protocol
SVN	Apache Subversion
UI	User Interface
XML	Extensible Markup Language

Chapter 1

Introduction

Apache Kafka is software with a variety of use cases, from being a messaging system for delivering the data over tracking the applications activity and data processing. These use cases have something in common: Kafka typically has lots of integrations to other systems and is often used as a backbone in complex software infrastructures. As the infrastructures evolve over time, the number of system integration and the amount of transferred data grows in parallel.

When the developers want to make a significat or even slight change in their system or extend it, they have to face questions like "*What will be the impact of this change?*" or *"Can I guarantee that these data will be available only to authorized people?"*. And usually, the answers to these questions are not trivial.

Manta is a company that offers a solution for simplifying these decisions by providing software that connects to the various systems in the software infrastructure, extracts the metadata, and visualizes them as a graph. The resulting graph captures the data flows between the systems. Manta currently supports connections to various technologies from different reporting and ETL tools to databases and programming languages. The aim of this work is to add support for Kafka.

1.1 Thesis goals

The goals of the work correspond to individual chapters. The first chapter **Classification and terminology** aims to give a general introduction to essential terms related to metadata extraction. It also introduces the Manta software with which the resulting module will be integrated. The following chapter **Apache Kafka introduction** introduces Kafka and explains the main Kafka principles on which is based the further analysis. The **Kafka scanner requirements** formalizes the functional and non-functional requirements for metadata extraction module.

The **Kafka metadata capabilities analysis** is focused on finding different approaches for the extraction. Based on the result of the analysis chapter, the **Kafka scanner design** deals with differents aspects of the module's design. The last two chapters, **Kafka scanner implementation** and **Kafka scanner testing and effectiveness evaluation** are dedicated to the implementation of a functional prototype, testing, and evaluation of the used solution.

Chapter 2

Classification and terminology

This chapter defines essential terms used later on within the work. It starts with an explanation of data lineage. After that is introduces the Manta software and standardized terms used in Manta. At the end is a high-level overview of data serialization and data serialization formats.

2.1 Data lineage

"*Data lineage states where data is coming from, where it is going, and what transformations are applied to it as it flows through multiple processes.*" [7]

As was mentioned in the introduction 1, the data lineage can help different organizations to make changes within their systems. With the knowledge of data lineage, it is easier to predict the consequences of a particular change. Another use case can be that today's organizations need to comply with different data privacy regulations, so they must know where their data moves within the system. [7]

2.2 Manta

Manta is a data lineage platform. It works on the principle of individual connections to various technologies and metadata extraction from them. Manta then uses this extracted metadata and creates a dataflow graph based on the connections between particular technologies. Manta currently supports connections to different technologies started from databases, reporting and ETL tools, programming languages, and more. [1]

2.2.1 Data flow graph

"*A data-flow is a path for data to move from one part of the information system to another.*" [8] The data lineage can then be represented as a sequence of data flows - as a data flow graph. The data flow graph is oriented; the nodes represent individual systems and the edges the flows of data between them. In the picture 2.1 is shown an example of a data flow graph created by Manta.

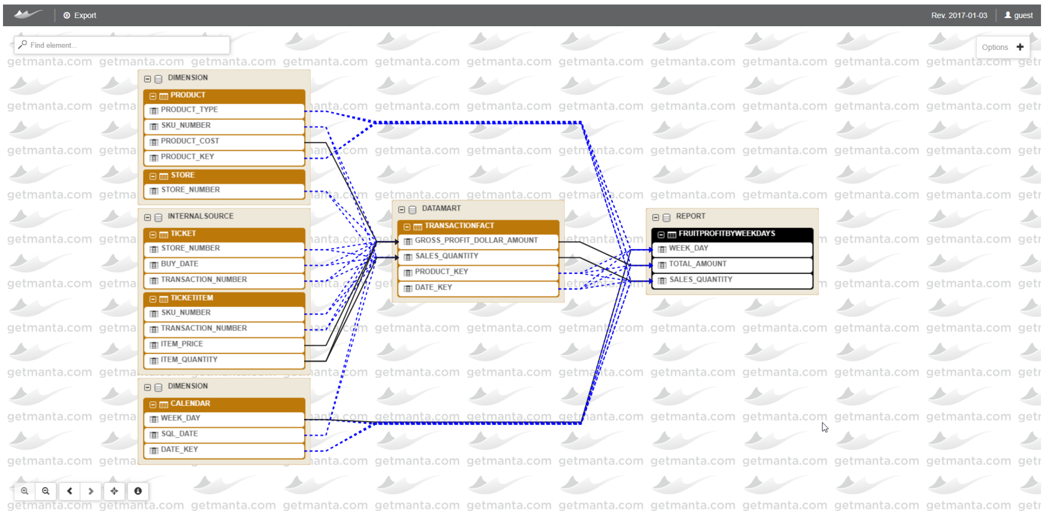


Figure 2.1 Dataflow Graph in Manta. [1]

2.2.2 Metadata

Metadata are the data that provide other information about the data. In the context of databases, MANTA extracts and analyzes technical metadata about the database structures, such as schemas, schemas, views, tables, columns, data types, etc. Important is that the metadata does not include the data itself. Again, on the database example, from Manta perspective, it is essential to obtain only the name of the columns, but not their content. [1]

2.2.3 Extraction and analysis

Manta distinguishes two main parts of the process of creation a data lineage of a particular technology – extraction and analysis. During the **extraction phase**, the metadata are extracted from the host system. After that, it comes the **analysis phase** during which are the extracted metadata merged into the final lineage. The software that connects to the source systems, obtains metadata and analyze them is called **scanner**. [1]

2.2.4 Data Dictionary

The data dictionary is a universal implementation of **metadata storage** that stores information about existing objects in the source system. Database scanners mainly use the data dictionary to store the information about the database objects and their hierarchy. The metadata in the dictionary are persisted on the disk in H2 database.

2.3 Data serialization and deserialization

In the following chapters it is often talked about data serialization, deserialization, and data serialization formats. “*Serialization is the process of translating data structures, or objects state into binary or textual form to transport the data over the network or to store on some persistent storage. Once the data is transported over the network or retrieved from the persistent storage, it needs to be deserialized again.*” [9]

2.4 Data serialization formats

Data serialization formats represent different ways of converting complex data objects to forms that are storable and transferable over the network. During this work are often discussed three of the data serialization formats – JSON, Apache Avro, and Protocol Buffers. These three data formats have in common that they partially or fully rely on **schemas**.

The **schema** defines the structure of the data, data types, and optionally meaning, which can be provided through the documentation comments. [2]

2.4.1 Protocol Buffers and Avro

Protocol Buffers and Avro are data serialization systems which allows serialization of messages into the the binary representation. The advantage of serialization into the binary data format is that the result message is compact. Both of them fully rely on schemas, in the listing 2.1 is example of Profobuf schema definition and in the listing is definition of schema for Avro 2.2.

Code listing 2.1 Protobuf schema definition example. [3]

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional uint32 age = 3;
}
```

Code listing 2.2 Avro schema definiton example. [4]

```
{
 "namespace": "example.avro",
 "type": "record",
 "name": "User",
 "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number",  "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
 ]
}
```

2.4.2 JSON schema

JSON schema is different from the previous two data formats because it does not fully relies on schemas. It is possible to use JSON without the need to define schema. JSON schema is only used for validation of schema and describtion of data format. The example of JSON schema is shown in the following chapters 7.3.

2.5 Pipeline

"*A data pipeline is a set of tools and activities for moving data from one system with its method of data storage and processing to another system in which it can be stored and managed differently. Moreover, pipelines allow for automatically getting information from many disparate sources, then transforming and consolidating it in one high-performing data storage.*" [10]

2.6 Messaging system

"*A messaging system is responsible for transferring data from one application to another so the applications can focus on data without getting bogged down on data transmission and sharing. Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging system.*" [11] The important part of the definition is that messaging systems are typically based on queue principle, meaning that the messages are deleted from the log with the first consumption.

Chapter 3

Apache Kafka introduction

The second chapter aims to give a general introduction to Apache Kafka. It begins with Kafka's classification and its main capabilities. Then it continues with the main Kafka concepts, terminology, and a brief introduction into Kafka's internal functioning. The end of the chapter is dedicated to different distributions and what they offer on top of Kafka's standard functionalities.

3.1 Apache Kafka classification

Apache Kafka is a **distributed event streaming platform** created by LinkedIn and eventually open-sourced. The need to develop a system such as Kafka was caused by the transition from the monolithic application infrastructure to the one based on microservices. LinkedIn engineers developed few pipelines whose main functionality was streaming and querying the data from individual microservices. Still, these pipelines needed to be maintained and scaled individually, which lead to the decision that LinkedIn would develop a single platform that would scale and maintain the pipelines for them. [12]

3.1.1 Event streaming platform

As indicated in the introduction of this chapter, Kafka is used for **streaming events**. What is meant by an event is a change of the state over time e. g. in the context of web activity tracking, the event can be a reaction to someone's post or that someone viewed specific content on the web. Kafka allows source systems to publish into streams of events, also heavily called logs, and sink systems to subscribe to them. Each event has a key, value, timestamp, and optionally metadata headers. [13]

The events are stored in these streams *durably* and *reliably* for a configured amount of time. Reliably means that Kafka is capable of recovering from the accident of one or more servers in the cluster; it depends on the number of servers and the log configurations. Durability refers to the fact that the data are stored in the stream even after the first consummation, which is the key difference from messaging, explained in the chapter 2.6. The last important functionality of event streaming is the ability to process the streams of events. Kafka offers APIs providing the possibility to do filtering or aggregation operations on events streams. [13]

3.1.2 Distributed system

Kafka is *distributed system*. That means it is composed of multiple components (servers), each of them on different machines, that communicate between each other to achieve the illusion of them being a single system. [14] Being a distributed system gives Kafka multiple advantages, including the already mentioned ability to recover from accidents some of the server or parallel reading from the event streams.

3.2 Kafka terminology

In the previous section 3.1.2 is Kafka classified as a distributed system. Kafka works as a cluster composed of multiple servers that store the events. In Kafka terminology, these servers are called **brokers**. The picture 3.1 shows one Kafka cluster with two brokers. [13]

At the time of writing this work, Kafka uses an external service called Zookeeper, which stores metadata information crucial for cluster operation, like the data needed for broker coordination. Zookeeper does not store any metadata information about data and their structure, so it is not helpful for dataflow analysis. For this reason, it is not going to be discussed further in the following chapters about Kafka metadata capabilities. In future Kafka versions, it is planned to remove Zookeeper, and brokers will direct all necessary information. [15]

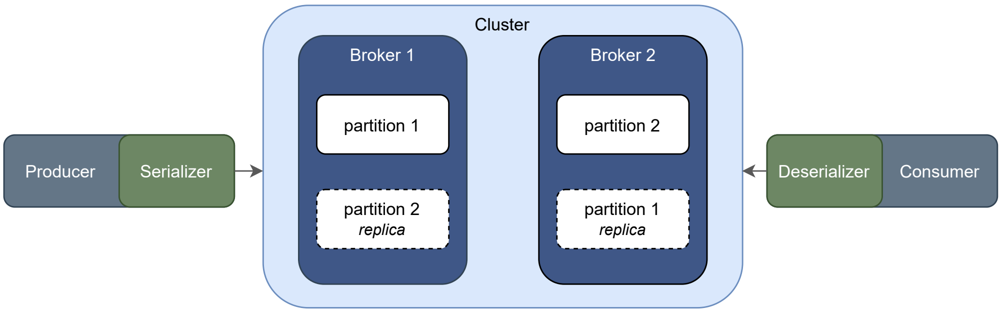


Figure 3.1 Apache Kafka cluster. Created using `draw.io`.

On the brokers are stored logs with events discussed in 3.1. The logs are called **topics**. Each topic has its own name unique within the cluster. This name usually reflects the type of data stored in it, for example, page views, users, orders, etc. Besides the name, it is necessary to configure more parameters while creating the topic. Between the most important properties belongs the broker(s) address, the number of partitions, and the replication factor.

The **address of the broker(s)** is used for discovering all of the brokers in a particular cluster. It is not guaranteed that the topic will exist on a given broker, but only in the same cluster. One topic usually exists on multiple brokers, it depends on the replication factor and the number of partitions properties.

The **number of partitions** corresponds to the number of brokers across them is the topic split. In the picture 3.1 is a cluster with a topic with a number of partitions equals two, the first of the partitions is located on broker 1 and second on broker 2. The **replication factor** then defines the number of copies for each partition. In the example, the replication is set to one, so partition number 1 is replicated on broker 2 and conversely. [13]

3.3 Kafka publish and subscribe process explanation

The chapter 3.1 mentions that Kafka works on publish and subscribe principle. The client applications or source systems that publish data into topics are called **producers** and the applications or sink systems that subscribe to the topics **consumers**. [13]

Apache Kafka offers five core APIs [13]:

- Producer API,
- Consumer API,
- Kafka Connect API,
- Admin API,
- Kafka Streams API.

The first three APIs are the ones considered interesting in terms of building consumer and producer applications. **Producer API** is intended for the applications that publish to the topics, and **Consumer API** for the ones that subscribe to the topics. Kafka **Connect API** is used for building connectors to source and sink systems. There is a lot of already built connectors, often as part of Kafka distribution to which is further elaborated upon in section 3.4. Between supported technologies belongs e. g. MangoDB, Cassandra, Salesforce or JDBC.

The producers and consumers communicate with Kafka via TCP protocol. Before the consumer sends a message, he has to configure some basic properties, among other things also broker(s) address, topic name, and the key and value serializer. The broker(s) needs to be specified for the same reason as during the topic creation 3.2 – it is used to discover all brokers in the cluster.

Kafka uses the event key for computation of hash which is then used to decide to which partition the event is sent. This way, an even distribution between the partitions and brokers is achieved, and the user has a guarantee that two messages with the same key have the same hash value and will be placed in the same partition. [13]

Before the message is sent, it needs to be serialized using the chosen serializer. The serialization is closely explained in chapter 2.3. Kafka offers some serializers for primitive data types. Still, when necessary to serialize some more complex messages, for example, in JSON data format, it is recommended to use the external library or write a custom serializer. The second option is preferred because there are many already existing solutions. On the opposite site, the consumer has to do the opposite process to serialization – deserialization.

3.4 Kafka distributions

There are several Kafka distributions. The overview of some of the Kafka distributions can be found in the table 5.2. These distributions were developed to solve some of the issues that come with using Kafka in production or add additional functionality on top of vanilla Kafka. It is possible to use Kafka itself or choose some of the distributions that should simplify Kafka's usage and offer some solution to typical problems. [2] [16]

Between the most common features provided by distributions belongs:

- UI, because Kafka itself doesn't have any user interface;
- monitoring tools for tracking Kafka cluster health metrics;

- connectors to different technologies for consuming and producing messages;
- Schema Registry for data's schema management;
- databases intended for event streaming;
- non-Java clients;
- and Kafka adapted for containers, cloud environments, etc.

From the metadata extraction perspective are interesting event streaming databases and especially Schema Registries, more about these functionalities and description of the way how it is possible to extract the metadata can be found in the chapter 5.

Chapter 4

Kafka scanner requirements

This chapter formalizes the requirements for the metadata extraction module for Kafka. They are split into two categories – functional and non-functional requirements. The second half of the chapter is dedicated to used technologies for implementation.

4.1 Functional requirements

The functional requirements define system behavior, in other words, how the system reacts to given inputs. The module's requirements were created in iterations. The items F1 - F4 in the list below were obtained before the initial analysis, F5 - F9 are based on decisions made after, and they mainly expand the F1 requirement. Not all of them will be part of a prototype, the requirements are excepted by plans to the future.

- F1: Module extracts the metadata from Kafka. The approaches to the extraction are chosen based on the analysis and the client's Kafka environment survey.
- F2: Extracted metadata are saved in the dictionary persisted in the H2 database.
- F3: Module uses the predefined processors for the generation of dataflow graph from the dictionary, or if the existing processors are not sufficient for Kafka purposes, it implements additional ones.
- F4: Dictionary structure is designed considering the common use cases for searching.
- F5: Module provides a possibility of manual extraction, which consists of reading the information about Kafka objects and their relationships from a defined file format. The part of the assignment is also to define the format structure for manual extraction.
- F6: Extractor supports the extraction from Confluent Platform Schema Registry (from now on only Schema Registry) implementation.
- F7: In the first version, the Schema Registry extractor supports the `TopicNameStrategy` and the design is made with a view to possible extension by other strategies from which is possible to derive the topic name – `TopicRecordNameStrategy` and `SimpleTopicIdStrategy`.
- F8: The prototype can process the JSON schema definition, and it is extensible by other data formats like Protobuf or Avro.
- F9: Schema Registry extractor supports HTTP basic authentication.

4.2 Non-functional requirements

- N1: Module extracts the metadata about one topic with the average count of schemas three in 3 seconds with the minimal system requirements
  - CPU: 4 cores at 2.5 GHz,
  - RAM: 12 GB,
  - OS: Windows 7 / Server 2008 or newer, Linux or Solaris, Mac (without installer),
  - HDD: 1 GB for MANTA installation + 50 GB of space for metadata; SSD, minimum of 1500 IOPS.
- N2: Module is documented and readable. It follows the Java naming conventions and use the JavaDocs for code documentation. All public and protected members has to be documented.
- N3: Module is tested for reduction of potential errors. The code is covered with unit tests and the total code coverage is at least 70 %. During the implementation it is used SonarQube for code quality and code security improvement.
- N4: The module design is extensible and allows easily add new functionality. It is possible to add new method of extraction. In Schema Registry extractor is possible to easily add support for other Schema Registry implementations, data formats, naming strategies, authentication and authorization mechanisms.
- N5: The module is implemented with technologies used in Manta - Java 8, Maven, SVN for version control – and it uses only the 3rd party libraries with compliant licenses approved by Manta.
- N6: The module uses Manta naming conventions, follows the typical scanner module structure, and leverages the elements from the Manta ecosystem when possible.
- N7: Module uses the Manta logging framework to provide information about the extraction and errors that occurred during the extraction.

4.3 Used technologies

The chapter describes the technologies used for implementation of module for Kafka metadata extraction based on the Manta assignment.

- **Java** - "*The Java programming language is a general-purpose, concurrent, strongly typed, class-based object-oriented language. It is normally compiled to the bytecode instruction set and binary format defined in the Java Virtual Machine Specification.*" [17] Thanks to its capabilities, Java can be used in various industries for building desktop, mobile, or even web-based applications.
- **Spring** - Spring is the most popular Java framework. It shields users from the need to focus on low-level programming aspects by providing solutions for common programming problems. Spring's set of extensions and third-party libraries are developed with a focus on performance and security. [18]
- **Maven** - "*Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.*" [19]

- **SVN** - Apache Subversion, also known as SVN, is a centralized version control system. It allows to make typical version control activities like check out the repository, update it, or commit the local changes.. [20]

- **SonarQube** - "*SonarQube is an automatic code review tool to detect bugs, vulnerabilities, and code smells in your code. It can integrate with your existing workflow to enable continuous code inspection across your project branches and pull requests.*" [21]

Chapter 5

Kafka metadata capabilities analysis

The chapter describes the analysis of Kafka metadata capabilities. During the research, several approaches to the extraction were examined. The chapter describes three of the strategies for metadata extraction. It starts with the extraction from Schema Registry and continues with the extraction using the manual input; these two methods will be implemented in the module in the future. As only the extraction from Schema Registry is a part of the prototype implementation. The last method is an extraction from ksqlDB, which is not going to be implemented because it contains extraction from sources that Manta clients do not use.

Table 5.1 Metadata sources overview. The table shows whether the method will be implemented in the prototype (within this work), in the final version or not.

Method	Prototype	Final version
Schema Registry	✓	✓
Manual Input	✗	✓
KsqlDB	✗	✗

5.1 Schema Registry extraction

Schema Registry is a service that works on top of Kafka. There are several implementations of this service, and some of them are listed in the table 5.2. This chapter mainly focuses on Confluent Platform because based on the decisions made in Manta, described together with requirements 4.1, the first version of the prototype will support extraction from this implementation. However, the other implementations are almost identical, whether in terms of design or metadata extraction.

5.1.1 Schema Registry purpose

Kafka users use Schema Registry as their schema definition storage. There were several reasons for the creation of such a registry. As explained in 3.3, before sending the message to Kafka, consumers have to serialize the event. Kafka does not have any metadata information

Table 5.2 Schema Registry implementation overview

Distribution	Schema Registry	REST API
Confluent Platform [2]	Confluent Schema Registry	✓
Confluent Cloud [2]	Confluent Schema Registry	✓
Cloudera Stream Messaging [22]	Hortonworks Registry	✓
Aiven for Apache Kafka [23]	Karapace	✓
IBM Event Streams [16]	from version 10.1.0 Apicurio before Event Streams Schema Registry	✓
Strimzi [24]	✗	✗

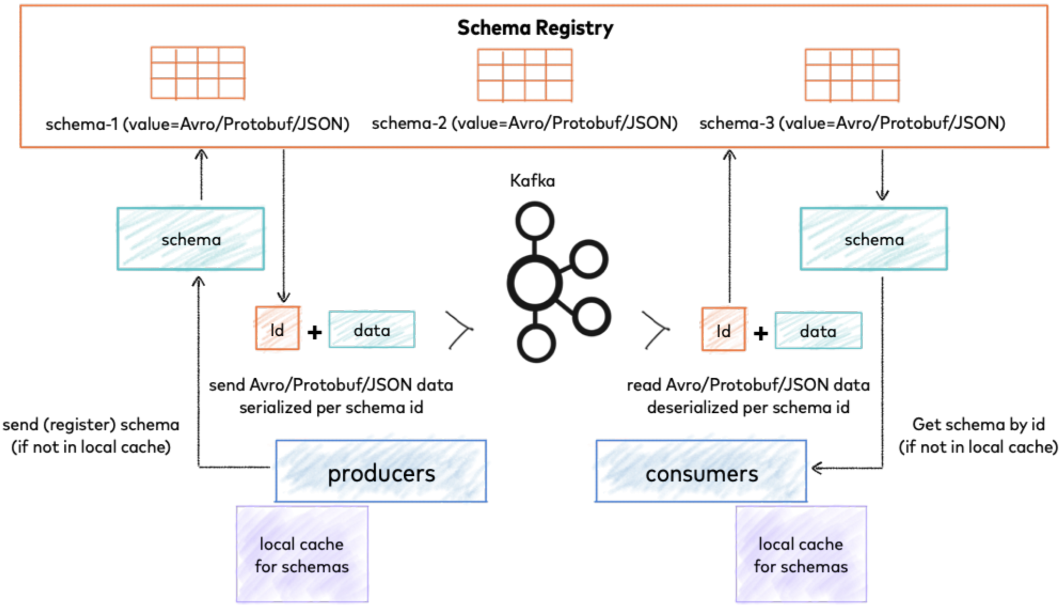

Schema Registry
schema-1 (value=Avro/Protobuf/JSON)
schema-2 (value=Avro/Protobuf/JSON)
schema-3 (value=Avro/Protobuf/JSON)
Kafka
schema
schema
Id
data
Id
data
send Avro/Protobuf/JSON data serialized per schema id
read Avro/Protobuf/JSON data deserialized per schema id
send (register) schema (if not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

Figure 5.1 Kafka with Schema Registry. [2]

about the records because it sees only the serialized message. This approach can cause issues on the consumer side because at the moment, when the structure of the transmitted data changes, problems may arise with following deserialization and interpretation of the data serializer. [2]

Schema Registry is primarily dedicated to data formats of events that define the data structure using schemas like Avro, Protobuf, and JSON. Because this schema can be stored in the registry, the message can be sent without the information about the field's names and other metadata. Consumers get the schema from the registry and use it to deserialize the message. This principle leads to a reduction of the required memory space for the message. [2]

5.1.2 Main Schema Registry concepts

In the picture 5.1 is shown how the message sending works with Schema Registry. Producers and consumers use the Avro format for their messages, but it would work similarly with any other format supported by Schema Registry. The section also neglects validity checks of the schema done before producing the record because it is not essential from the metadata extraction perspective.

One of the changes compared to workflow in 3.3 is that the producer will now use a different serializer provided by Confluent – `KafkaAvroSerializer`. During the serialization, the producer checks if a given schema is available in the registry; if it is not, he will register it. A globally unique ID is assigned to every schema. This ID is returned from the Schema Registry, and the producer includes it in the message which it wants to publish to Kafka. The schema is not sent with the record because the consumer gets it from the registry against the ID and uses it for deserialization. [25]

5.1.2.1 Subjects and naming strategies

Except for storing the schema, the registry allows users to evolve the schema over time and set validation checks, e. g. If the consumers can deserialize all messages created with the old schema version with a new version of the schema. [26]

Table 5.3 Schema Registry naming strategies

Naming strategy	Description

`TopicNameStrategy`
`TopicIdStrategy`
	`<topic_name>-value`
	`<topic_name>-key`

`RecordNameStrategy` `RecordIdStrategy`	`<schema_name>`
`TopicRecordNameStrategy` `TopicRecordIdStrategy`	`<topic_name>-<schema_name>`
`SimpleTopicIdStrategy`	`<topic_name>`

`HortonworksTopicNameStrategy`	`<topic_name>`
	`<topicname>:k`

All schema versions are stored in **subjects** in the registry. The subject is automatically created when the first version of the schema is uploaded to Schema Registry. [2] The subject name depends on configured **naming strategy**. Table 5.3 gives an overview of some naming strategies. Some of them have multiple names listed because the name can differ across different implementations, but the subject name creation is the same.

Confluent Platform uses by default `TopicNameStrategy` and also supports `RecordNameStrategy` and `TopicRecordNameStrategy`. The `TopicNameStrategy` uses the name of the topic to which the data are sent and adds a suffix "-value" or "-key" depending on whether it is the schema intended for message value or key.

The `RecordNameStrategy` does not contain information about a related topic. It is only composed of the name of the schema, e. g., in the context of Avro, each schema has a "name" field [4]. The first two strategies have in common that they support only one schema of the data per topic. This limitation solves `TopicRecordNameStrategy` which has two parts – topic name and schema name – separated by a dash. [2]

`SimpleTopicIdStrategy` is used e. g. by Apicurio Schema Registry implementation [27], and it contains only the topic name, so it does not consider the schema for a key.

The `HortonworksTopicNameStrategy` is used by default by Hortonworks Registry. The name of the subjects is created using the topic name, and in the case of the key schema, adds the suffix ":k". [28]

5.1.3 Extraction process from Schema Registry

The main goal of the Schema Registry extractor is to extract schema definitions from the Schema Registry. These schemas must be possible to connect later on with the associated topics. As indicated in 5.2, all examined registries provide REST API for interaction with the registry. The examples below are obtained from the Confluent Platform API reference.

For extracting the schema definitions, it is necessary to get the list of subjects first, which is done using the request in the listing 5.1 and the corresponding response in the listing 5.2. In the content of example response is a JSON array with all existing subjects, in this case `subject1`, and `subject2`.

Code listing 5.1 Confluent Schema Registry - GET subjects request. [2]

```
GET /subjects
```

Code listing 5.2 Confluent Schema Registry - GET subjects example response. [2]

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

["subject1", "subject2"]
```

All of the schema versions in a particular subject are possible to get with the request from the listing 5.3, corresponding response is in the listing 5.4.

Code listing 5.3 Confluent Schema Registry - GET subject versions request. [2]

```
GET /subjects/{subject}/versions
```

With the knowledge of subject and version it is possible to use request in the listing 5.5 to get the final schema, which can be seen in the listing 5.6. The final response contains several fields – `subject`, `version`, `schema`, and `id`. The first three listed fields are clear from the context, but it is worth mentioning the `id`. This identifier is globally unique in given Schema Registry which is later on used for saving the schema after the extraction.

As shown in the last listing 5.6, the response does not contain any information about the related topic because the registry does not store that kind of information. That means, the topic name has to be found out other way. For the strategies - `TopicNameStrategy`, `TopicRecordNameStrategy`, `SimpleTopicIdStrategy`, and `HortonworksTopicNameStrategy` - it is possible to use their name to obtain information about the topic. For the rest of the strategies, there is no other way than manual configuration from the client, which assigns the subject to the topic.

5.1.4 Advantages and disadvantages of Schema Registry

The biggest advantage of extraction from Schema Registry is that the extraction process is done automatically with the minimal needed configuration from the client. As is shown in the

Code listing 5.4 Confluent Schema Registry - GET subject versions example response. [2]

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

[1, 2, 3, 4]
```

Code listing 5.5 Schema Registry - GET schema request. [2]

```
GET /subjects/{subject}/versions/{version}
```

questionnaire made with Manta's clients in the appendix A, there are also clients who use the Schema Registry. The disadvantage could be the limited options for finding the associated topic to the schema and that the fully automatic process can be done only if the client used the specific naming conventions for his subjects.

5.2 Manual input extraction

Manual input extraction method does not connect to Kafka, or to any other service to retrieve information about the Kafka cluster. The principle of this extraction method is to define **common data format** for clients to give them option to deliver the required data. They provide the metadata in this predefined format and the extractor will parse this file(s) and save the information from it.

Because the manual input can go to the large scale with the cluster with dozens of brokers, and hundreds or thousands of topics, it is crucial to design the input format as friendly as possible. The manual input is designed in the next chapter 6.1 and it should at least contain information about:

- broker URIs in the cluster,
- list of topics,
- schemas for individual topics and its columns,
- and optionally information about the Schema Registry environment.

5.2.1 Advantages and disadvantages of manual input

The manual extraction method has three main advantages:

- The method gives the user full control over the extraction and the extracted object.
- The module can be used by clients that do not use the Schema Registry.
- The manual input can be used together with the Schema Registry extraction. It is convenient because the Confluent Schema Registry supports only a limited amount of data format types for messages and it does not support the heavily used data formats like CSV and XML. This way the client can add the information about the objects not covered within the Schema Registry extraction.

■ **Code listing 5.6** Schema Registry - GET schema example response. [2]

```
HTTP/1.1 200 OK
Content-Type: application/vnd.schemaregistry.v1+json

{
  "subject": "subject1",
  "id": 1,
  "version": 1,
  "schema": "{\"type\": \"string\"}"
}
```

On the other hand, the method has one main disadvantage and that is the need to fill in the predefined file. With a larger cluster size this can be challenging and the files have to be modified with each change in the cluster.

5.3 KsqlDB extraction

"*ksqlDB is a new kind of database purpose-built for stream processing apps, allowing users to build stream processing applications against data in Apache Kafka...*" [29] It is available with Confluent Platform on-premises deployments, a fully managed service in Confluent Cloud, and in Standalone mode, which only requires running the Kafka environment. [5]

KsqlDB started as KSQL, and the main functionality at the beginning was to give the Kafka users SQL engine for Kafka to provide another alternative instead of building streaming applications [30]. Later, the creators renamed it to ksqlDB because it offered more than just SQL engine, but it also could store tables of data with fault tolerance, and it had REST API for interaction with the cluster. [31]

5.3.1 KsqlDB collections

KsqlDB allows creation of **collections** from individual events in Kafka topics, which provides user with the ability to store the related events together. The collections are, similarly to partitions in Kafka, replicated across multiple servers for achieving the fault tolerance.

"*Collections are represented as a series of rows and columns that have a defined schema. Only data that conforms to the schema can be added to the collection.*" [5] The fact that all of the collections have **corresponding schema** is important because it is why the ksqlDB can be used for the extraction, as is described in the following section 5.3.2.

There are two types of collections – **streams** and **tables**. One of the key differences between these two is that streams are immutable collections and tables are mutable. Mutable objects can change after they are created, but the immutable cannot. It is only allowed to append new records to streams, which makes them useful for saving historical facts. Tables store only the last value for the specific key. In the context of streams is a new value, even for the key already existing in the collection, appended at the end, and the previous value does not change. [5]

To register stream, resp. table over Kafka topics user has to use `CREATE STREAM`, resp. `CREATE TABLE` statement. Their syntax can be found in the listings 5.7 and 5.8 and it is very similar.

The usage of these `CREATE` statements is easier to illustrate on an example. The most significant difference in usage depends on the user using Schema Registry with Kafka or not from the

Code listing 5.7 KsqlDB CREATE Stream syntax. [5]

```
CREATE [OR REPLACE] STREAM stream_name
    ( { column_name data_type [KEY] } [, ...] )
    WITH ( property_name = expression [, ...] );
```

Code listing 5.8 KsqlDB CREATE Table syntax. [5]

```
CREATE [OR REPLACE] TABLE table_name
    ( { column_name data_type [PRIMARY KEY] } [, ...] )
    WITH ( property_name = expression [, ...] );
```

Code listing 5.9 KsqlDB CREATE Stream with Schema Registry. [5]

```
CREATE STREAM pageviews WITH (
        KAFKA_TOPIC = 'keyless-pageviews-topic',
        VALUE_FORMAT = 'JSON'
    );%
```

Code listing 5.10 KsqlDB CREATE Stream without Schema Registry. [5]

```
CREATE STREAM pageviews (
        page_id BIGINT,
        viewtime BIGINT,
        user_id VARCHAR
    ) WITH (
        KAFKA_TOPIC = 'keyless-pageviews-topic',
        VALUE_FORMAT = 'JSON'
    );
```

Code listing 5.11 KsqlDB CREATE Table with Schema Registry. [5]

```
CREATE TABLE users (
        id BIGINT PRIMARY KEY
    ) WITH (
        KAFKA_TOPIC = 'my-users-topic',
        VALUE_FORMAT = 'JSON'
    );
```

Code listing 5.12 KsqlDB CREATE Table without Schema Registry. [5]

```
CREATE TABLE users (
        id BIGINT PRIMARY KEY,
        usertimestamp BIGINT,
        gender VARCHAR,
        region_id VARCHAR
    ) WITH (
        KAFKA_TOPIC = 'my-users-topic',
        VALUE_FORMAT = 'JSON'
     );
```

metadata extraction perspective. As was mentioned before, the tables and streams have to have a defined schema. This schema is specified during the creation of the collection. With Schema Registry, it is unnecessary to specify the schema manually, but it is downloaded automatically from Schema Registry. The syntax with the usage of Schema Registry is shown at the stream collection in the code listing 5.9 and at the table in 5.11.

In both cases it is enough to specify only the topic name and value format. That also means that these collections are not important for metadata extraction because their definitions can already be extracted from Schema Registry, as was described in the chapter about Schema Registry extraction 5.1.

Interesting are the ones that have to have a manually defined schema. Creation of stream is in the picture 5.10 and creation of table in 5.12. Other than the value format and the Kafka topics, the statement also contains list of columns names and their formats.

5.3.2 KsqlDB extraction process

KsqlDB extraction is quite similar to extraction from Schema Registry because ksqlDB also provides REST API for user requests. Unless otherwise configured, the default HTTP API endpoint is `http://localhost:8088/`. Two main endpoints are intended for data inspection – `ksql` and `/query`. Both of the resources expect ksqlDB SQL statement as a parameter . The difference between these two endpoints is that `/query` is used for `SELECT` statements, and `/ksql` for every other statement.

For the extraction are useful these statements:

- `LIST STREAMS`,
- `LIST TABLES`,
- `DESCRIBE (stream_name|table_name)`.

`LIST STREAMS` and `LIST TABLES` are used for getting a list of defined streams, respectively tables. `DESCRIBE` returns list of the columns in a stream or table. Because all of the statements are not `SELECT` statements, for the extraction is used `/ksql` endpoint.

In this chapter the extraction process is demonstrated on an example with streams, but it would be similar with tables. First of all, in the listing 5.13 is created a stream with the name `pageviews_original` corresponding to the Kafka topic `pageviews`. The data structure is defined manually and it says that the data in the topic uses JSON data format with columns `viewtime`, `userid`, and `pageid`.

Code listing 5.13 KsqlDB - Stream creation. [5]

```
CREATE STREAM pageviews_original
(viewtime bigint, userid varchar, pageid varchar)
WITH (kafka_topic='pageviews', value_format='JSON');
```

Code listing 5.14 KsqlDB REST API - LIST STREAMS response. [5]

```
[
    {
        "@type": "streams",
        "statementText": "LIST STREAMS;",
        "streams": [
            {
                "type": "STREAM",
                "name": "KSQL_PROCESSING_LOG",
                "topic": "default_ksql_processing_log",
                "format": "JSON"
            },
            {
                "type": "STREAM",
                "name": "PAGEVIEWS_ORIGINAL",
                "topic": "pageviews",
                "format": "JSON"
            }
        ],
        "warnings": []
    }
]
```

For obtaining the list of defined streams are used the `LIST STREAMS` statements and example response is shown in the listing 5.14. In the model situation, the response contains only two streams called `KSQL_PROCESSING_LOG` and `PAGEVIEWS_ORIGINAL`. First of them is the default stream used for storing *metadata information*, the second one is the one created in the previous statement.

After finding out which streams or tables exist in given ksqlDB cluster, follows `DESCRIBE` for individual collections. In this case `DESCRIBE pageviews_original`. The example response which is shortened for the purposes of this work is in the listing 5.15. The output includes all necessary information for metadata extraction, columns and corresponding topic, exactly like it was defined in the listing 5.13.

Compared to extraction from Schema Registry, this approach has two advantages. The first one is that the topic does not have to be derived from subject but it is included in the REST API response. The second one is that, unlike Schema Registry, the ksqlDB does not have limited data formats to operated one, because it provides an option to configure the data formats manually, so it can be used for the topics that use for example CSV data format.

Code listing 5.15 KsqlDB REST API - DESCRIBE abbreviated response. [5]

```
[
    {
        "@type": "sourceDescription",
        "statementText": "DESCRIBE PAGEVIEWS_ORIGINAL;",
        "sourceDescription": {
            "name": "PAGEVIEWS_ORIGINAL",
            "fields": [
                {
                    "name": "ROWTIME",
                    "schema": {
                        "type": "BIGINT",
                    }
                },
                {
                    "name": "ROWKEY",
                    "schema": {
                        "type": "STRING",
                    }
                },
                {
                    "name": "VIEWTIME",
                    "schema": {
                        "type": "BIGINT",
                    }
                },
                {
                    "name": "USERID",
                    "schema": {
                        "type": "STRING",
                    }
                },
                {
                    "name": "PAGEID",
                    "schema": {
                        "type": "STRING",
                    }
                }
            ],
            "type": "STREAM",
            "keyFormat": "KAFKA",
            "valueFormat": "JSON",
            "topic": "pageviews"
        },
    }
]
```

5.3.3 ksqlDB advantages and disadvantages

The main advantage of ksqlDB is that it provides the extraction method for both schemas in Schema Registry and schemas that cannot be used with Schema Registry. However, as the survey has shown A non of the clients use the ksqlDB. Based on the decisions made in Manta, this method will not be used for now.

Chapter 6

Kafka scanner design

The chapter describes Kafka scanner design. It begins with the design of the Kafka node in Manta dataflow visualization. Then, it continues with the creation of the input format necessary for manual extraction, discussed as one of the metadata extraction capabilities in 5.2. The second half of the chapter is already focused on the design of the Kafka metadata extraction module.

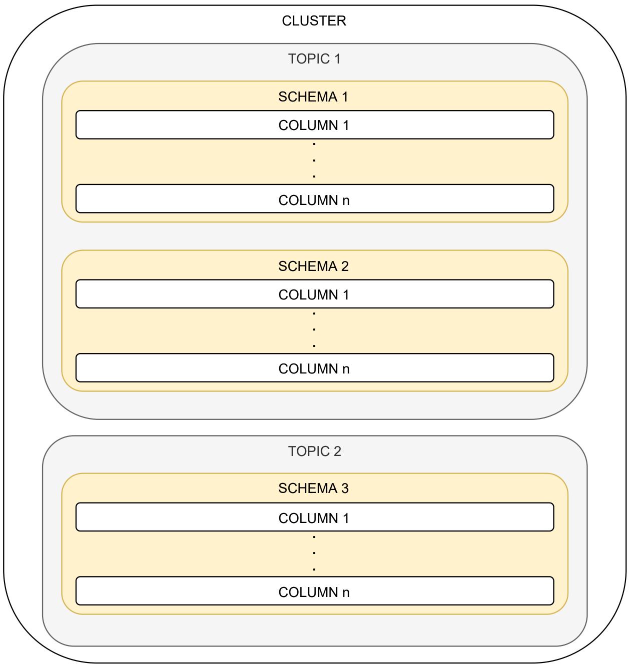


Figure 6.1 Kafka Manta node design. Created using `draw.io`.

6.1 Manta Kafka node design

At the beginning of this work is a sample picture of a dataflow graph in Manta 2.1. The graph is composed of multiple nodes where each of them represents an individual system in the way of data flows. The section describes the creation of such node for Kafka, which will capture the main Kafka elements and their hierarchy.

The final design is shown in the figure 6.1. As a root node is elected a cluster that contains the individual topics. Each topic can have multiple schemas in it and underneath the schema are its columns.

Cluster is on top of hierarchy because in Manta typically the root node represents one configured connection. During the design, it also was discussed an option where under the cluster are the brokers, but it was not chosen because the topic is often duplicated across multiple brokers which could lead to multiple problems like duplicated data in visualization or inability to decide through which broker should the data lineage go. Although it is recommended to use only one schema for data in each topic, the questionnaire A showed that most of the clients interesting in Kafka scanner use more than one. That is the reason why the visualization allows multiple schemas per topic.

6.2 Manual input format design

The manual input was introduced as one of the metadata capabilities in the chapter 5.2. To provide the manual extraction possibility is necessary to define a common format that the clients can use to fill in the information about the Kafka cluster. The format should contain the information from the final Kafka node in visualization and the other fields that appeared in the context of data dictionary and Schema Registry – subject, version, and schema id.

In the listing 6.1 is the manual format design with sample data in it. The final design uses JSON data format because it is necessary to use some complex data format for capturing the Kafka environment. JSON can hold all needed metadata, it is easily readable and unlike formats like XML light-weight. In addition, the Java and its libraries has a good support for JSON.

The format allows for each listed topic to define external and inline schemas. External schemas are saved outside in separate files with schema and internal directly in the manual import file by listing the present columns. This solution allows the client to capture both schemas – the ones for each he has a schema definition and the ones he does not. For each schema regardless its type is possible to optionally configure id, subject, and version.

6.3 Kafka scanner design

This chapter is already focused on the design of the module for metadata extraction. The design is intented for extraction method from Schema Registry because the manual extraction method is not part of the prototype. However, the design takes into account the extension of the manual extraction. The module contains three sub-modules, their dependencies are shown in the figure 6.2. Namily

- `manta-connector-kafka-dictionary-extractor` is responsible for metadata extraction and saving the extracted metadata into the dictionary,
- `manta-connector-kafka-dictionary` defines the Kafka entity types within the dictionary and their hierarchy,
- `manta-connector-kafka-model` stores needed information about Kafka objects.

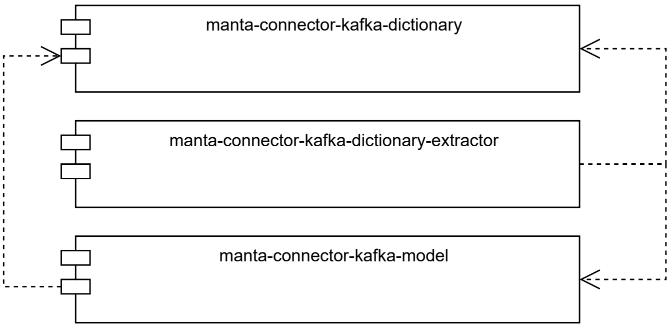

manta-connector-kafka-dictionary
manta-connector-kafka-dictionary-extractor
manta-connector-kafka-model

Figure 6.2 Kafka Scanner module diagram. Created using `draw.io`.

6.3.1 Kafka dictionary extractor design

The extractor (`manta-connector-kafka-dictionary-extractor`) is responsible for connection to the source system (Schema Registry), metadata extraction, their evaluation, and saving them for later use. In the figure 6.3 is module's class diagram. The extraction process is started from `KafkaExtractorImpl` class which is the implementation of `KafkaExtractor` interface that can be used by other modules to run the extraction.

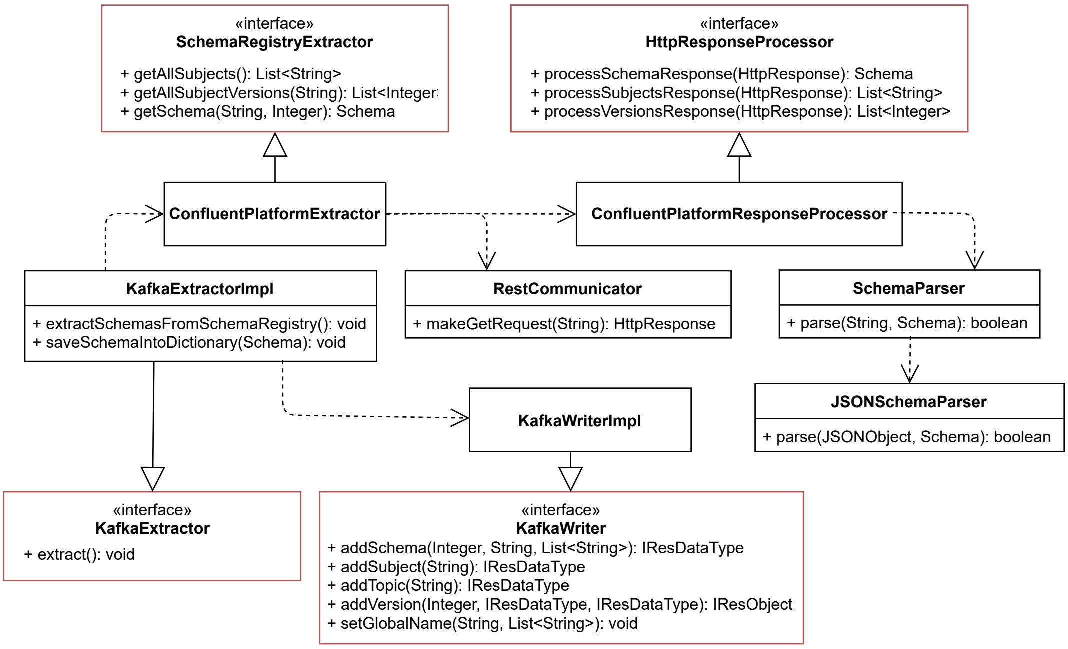

«interface»
SchemaRegistryExtractor
+ getAllSubjects(): List<String>
+ getAllSubjectVersions(String): List<Integer
+ getSchema(String, Integer): Schema
«interface»
HttpResponseProcessor
+ processSchemaResponse(HttpResponse): Schema
+ processSubjectsResponse(HttpResponse): List<String>
+ processVersionsResponse(HttpResponse): List<Integer>
ConfluentPlatformExtractor
ConfluentPlatformResponseProcessor
KafkaExtractorImpl
+ extractSchemasFromSchemaRegistry(): void
+ saveSchemaIntoDictionary(Schema): void
RestCommunicator
+ makeGetRequest(String): HttpResponse
SchemaParser
+ parse(String, Schema): boolean
JSONSchemaParser
+ parse(JSONObject, Schema): boolean
KafkaWriterImpl
«interface»
KafkaExtractor
+ extract(): void
«interface»
KafkaWriter
+ addSchema(Integer, String, List<String>): IResDataType
+ addSubject(String): IResDataType
+ addTopic(String): IResDataType
+ addVersion(Integer, IResDataType, IResDataType): IResObject
+ setGlobalName(String, List<String>): void

Figure 6.3 Kafka Dictionary Extractor Design. Created using draw.io.

For extraction from Schema Registry is prepared an interface `SchemaRegistryExtractor` that defines uniform method of extraction for Schema Registry. The interface can be used to implementing the logic for different Schema Registry implementations. Since the first version should supports the extraction from Confluent Platform, the interface is implemented by

ConfluentPlatformExtractor. ConfluentPlatformExtractor uses two main classes for the extraction – RestCommunicator and HttpResponseProcessor. First of them handles the connection to the given host system and communication over REST API. It is design to work independently of the Confluent Platform extractor so that it can be used by other services that will also use REST API. HttpResponseProcessor process the schema response and parse its content using the SchemaParser class.

After KafkaExtractor runs the Schema Registry extraction using SchemaRegistryExtractor, it saves the metadata into the dictionary. KafkaWriterImpl is a class responsible for communication with the data dictionary. The orchestration of the process is shown in the figure 6.4 and it is closely explained in the next section about Kafka dictionary design.

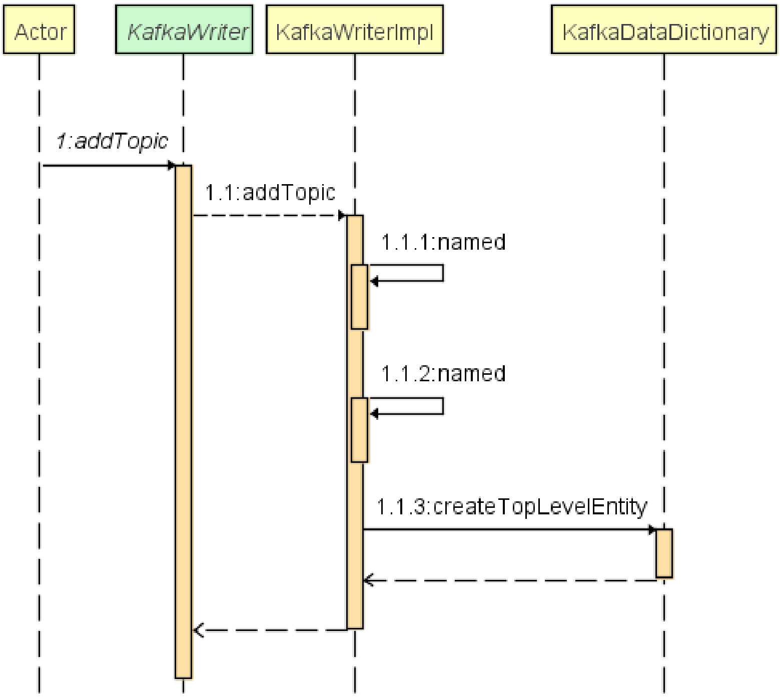


Figure 6.4 Data dictionary add topic sequence diagram. Created using IntelliJ idea sequence diagram plugin.

6.3.2 Kafka dictionary design

In the chapter 2.2.4, which introduced the basic terms and concepts used in Manta, data dictionary was mentioned as a universal implementation for storing metadata. The dictionary provides resources for the creation of objects like columns, tables, database schemas, etc. The data dictionary also allows the definition of different objects on top of existing ones. The custom types are defined in KafkaDataDictionary which extends the AbstractDataDictionary.

The data dictionary forms a directed rooted tree structure where each entity can have one parent, any number of children, attributes, and properties. Each entity has to have at least one property, because it is used for distinguishing the different objects within the dictionary; for example, when the dictionary creates an entity representing the column, it adds to the entity properties COLUMN property. The attributes are used to provide additional information about the object.

Except for the definition of custom data types, the manta-connector-kafka-dictionary module has to define the relationships between the Kafka objects. The hierarchy is defined in `KafkaDialect` class. This class determines what objects can be in parent-child relationships and what should happen if there is a break of these rules.

6.3.2.1 Kafka dictionary entities structure

Later on, another module in Manta will use the dictionary for searching the schemas in the given Kafka cluster based on the queries of other Manta scanners. Because of that fact, the data dictionary has to be designed considering the most frequent queries to optimize the exploration of the dictionary tree.

Table 6.1 Kafka Dictionary Search Use Cases. Displays possible number of found results based on given parameters for search and provided that the given parameters are correct and exists in the dictionary. *Blank fields mean that it not depends if given parameters is present or not.*

	broker URI	topic	schema ID	subject	version	number of schemas
1	✗	✗				0
2	✓	✓	✗	✗	✗	1..*
3	✓	✓	✓			1
4	✓	✓	✗	✓	✗	1
5	✓	✓	✗	✓	✓	1

Table 6.1 shows the possible parameter combinations and the number of possible schema results. It is supposed that the query will at least contain the broker URI and the name of the topic. The most common use case will be that the query includes exactly these two parameters. This query can have multiple results because If the given topic has multiple schemas in it, it is impossible to identify which of them to choose.

In the dictionary should also be possible to search based on the identifiers from the Schema Registry - schema ID, subject, and version. Worth noting is the fourth line where it returns exactly one result even if it is on the input only the subject, which can not be used to identify the schema on its own but needs the schema version. It is because, in that case, the default behavior will be returning the last version.

Considering this use cases, it follows that the dictionary must contain for each Kafka cluster

- broker URIs,
- topics,
- schema with the columns for each topic,
- and subjects, versions, schema IDs when it is enabled the extraction from Schema Registry.

As mention before, the dictionary represents on a persistence level the objects as entities. On top of the entities creates another level of abstraction for simplifying the creation of more complex types and object structures. From Kafka perspective are interesting two object types – `IResDataType` and `IResDataObject`. The only differents between these two is that `IResDataObject` can have a data type. This mechanism is created for prevention of duplicated data accross the dictionary, i. e. when there are multiple topics with the same schema it is possible to represent them as a `IResDataObject` with the data type schema.

In the picture 6.5 is the final Kafka dictionary structure. As a root is chosen the cluster and in the dictionary is represented as a global namespace. Under neath are as children schema ID,

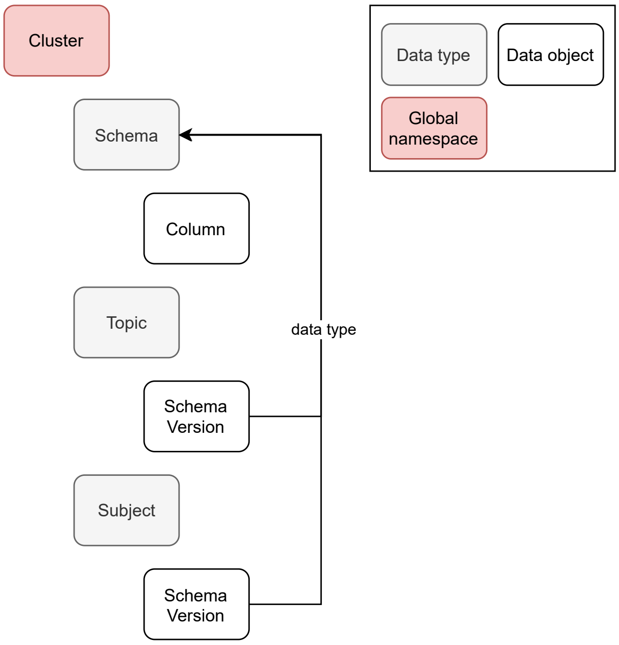


Figure 6.5 Kafka Dictionary Objects Structure. Created using draw.io.

topic, and subject. Under the schema ID are its columns. The topic has a childern version which is only used as a reference to corresponding schema using the data type relationship. Similarly is it in the case of version under the subject, but the version is not used only as a reference but also as physical extracted version from Schema Registry used for filtering.

This design provides a possibility for search in all of the use cases in the table 6.1 and it is also usable in the cases where the clients does not use the Schema Registry, but only the manual input, just by leaving out the subject nodes.

6.3.2.2 Kafka dictionary module design

`manta-connector-kafka-dictionary` contains two main classes – `KafkaDataDictionary` and `KafkaDialect`. `KafkaDataDictionary` extends the `AbstractDataDictionary` and ads on top of it one method for creating the root elements (subject, topic, schema) which wasn't covered by predefined methods. By Manta convention, the `KafkaDataDictionary` implements `ResolverEntitiesFactory` placed in `manta-connector-kafka-model`.

The writer from previous section 6.3.1 uses the `KafkaDataDictionary` for creation of objects within the dictionary. In the figure 7.4 were is shown how the process of adding the topic works. It is invoked by calling `addTopic` method on `KafkaWriter`, the implementation of the method is in `KafkaWriterImpl`. From the method is called the data dictionary method `createTopLevelEntity`.

Except of mentioned classes above, the data dictionary contains two factories that for creating the dictionary, they are in the picture 6.6. The `MemoryDictionaryFactory` creates in-memory dictionary and `JdbcDictionaryFactory` creates the dictionary saved in H2 database.

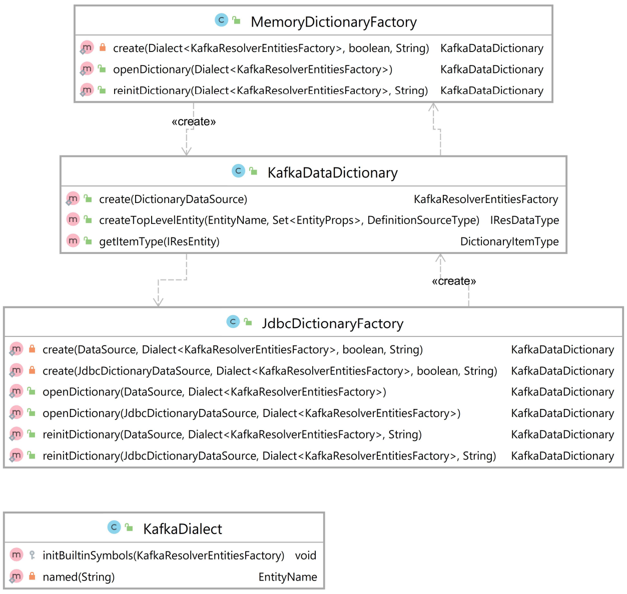

MemoryDictionaryFactory
create(Dialect<KafkaResolverEntitiesFactory>, boolean, String) KafkaDataDictionary
openDictionary(Dialect<KafkaResolverEntitiesFactory>) KafkaDataDictionary
reinitDictionary(Dialect<KafkaResolverEntitiesFactory>, String) KafkaDataDictionary
«create»
KafkaDataDictionary
create(DictionaryDataSource) KafkaResolverEntitiesFactory
createTopLevelEntity(EntityName, Set<EntityProps>, DefinitionSourceType) IResDataType
getItemType(IResEntity) DictionaryItemType
«create»
JdbcDictionaryFactory
create(DataSource, Dialect<KafkaResolverEntitiesFactory>, boolean, String) KafkaDataDictionary
create(JdbcDictionaryDataSource, Dialect<KafkaResolverEntitiesFactory>, boolean, String) KafkaDataDictionary
openDictionary(DataSource, Dialect<KafkaResolverEntitiesFactory>) KafkaDataDictionary
openDictionary(JdbcDictionaryDataSource, Dialect<KafkaResolverEntitiesFactory>) KafkaDataDictionary
reinitDictionary(DataSource, Dialect<KafkaResolverEntitiesFactory>, String) KafkaDataDictionary
reinitDictionary(JdbcDictionaryDataSource, Dialect<KafkaResolverEntitiesFactory>, String) KafkaDataDictionary
KafkaDialect
initBuiltinSymbols(KafkaResolverEntitiesFactory) void
named(String) EntityName

Figure 6.6 Kafka Dictionary class diagram. Created using IntelliJ idea.

Code listing 6.1 Manual input format design with sample data.

```
{
   "cluster":"example_cluster",
   "topics":[
      {
         "topicName":"topic1",
         "schemas":[
            {
               "id":1,
               "subject":"subject1",
               "version":1,
               "schema":"schema1.json"
            },
            {
               "id":2,
               "subject":"subject1",
               "version":2,
               "schema":"schema2.asvc"
            }
         ],
         "inlineSchemas":[
            {
               "id":4,
               "subject":"subject3",
               "version":1,
               "fields":[
                  "column1",
                  "column2"
               ]
            }
         ]
      },
      {
         "topicName":"topic2",
         "schemas":[
            {
               "id":1,
               "subject":"subject1",
               "version":1,
               "schema":"schema1.json"
            }
         ]
      }
   ]
}
```

Chapter 7

Kafka scanner prototype implementation

The implementation of a prototype can be divided into two main parts:

- **Connector** – connects to the source system, extracts metadata and saves them in the data dictionary,
- **Dataflow Generator** – analyze the extracted objects and generates a Manta graph.

7.1 Connector module

Connector module has two main parts – dictionary extractor and dictionary module. Dictionary extractor connects to the source system, extracts the metadata, and interacts with the dictionary to save the objects. The dictionary module defines the entities used within the dictionary and their relationships.

7.1.1 Connector Kafka Dictionary Extractor module

The prototype implementation of Kafka Dictionary Extractor performs four main activities:

- connects to the Schema Registry,
- extracts the metadata from Schema Registry using sequence of REST API calls,
- parses the obtained metadata (schemas),
- and saves the metadata into the dictionary.

The main class of dictionary extractor module is `KafkaExtractorImpl` which is responsible for running all of the processes listed above.

7.1.1.1 Connection to Schema Registry

`RestCommunicator` class handles the connection to the Schema Registry and communication over REST API using the Apache `HttpComponents` library. In the listing 7.1 is shown a method which is responsible for making requests. If executing of request failes, the method throws SchemaRegistryException, in the opposite case returns HttpResponse.

Code listing 7.1 RestCommunicator:makeGetRequest method.

```
public HttpResponse makeGetRequest(String request)
    throws SchemaRegistryExtractionException {
    HttpGet httpRequest = new HttpGet(request);
    LOGGER.info("Sending get request " + request + "to " + httpHost + ".");
    HttpResponse httpResponse;
    try {
        httpResponse = httpClient.execute(httpHost, httpRequest);
    } catch (IOException e) {
        LOGGER.log(Categories.httpErrors().httpRequestFailed().catching(e));
        throw new SchemaRegistryExtractionException
            ("Enable to execute HTTP request.", e);
    }
    return httpResponse;
}
```

7.1.1.2 Extraction of metadata from Schema Registry

For the extraction from Schema Registry is used SchemaRegistryExtractor interface shown in the listing 7.2. It follows the extraction process described in the chapter about metadata capabilities 5.1. This interface is implemented by ConfluentPlatformExtractor which handles the logic of extraction from Confluent Platform Schema Registry implementation.

Code listing 7.2 SchemaRegistryExtractor interface.

```
public interface SchemaRegistryExtractor {

    List<String> getAllSubjects() throws SchemaRegistryExtractionException;

    List<Integer> getAllSubjectVersions(String subject)
        throws SchemaRegistryExtractionException;

    SchemaRegistrySchema getSchema(String subject, int version)
        throws SchemaRegistryExtractionException;

    String getConnectionInfo();
}
```

The SchemaRegistryException is thrown If the extraction failes which can be cause because of some of the following reasons:

- the REST API request failes,
- the request ends up with the exist code other than 200,

- the `HttpResponse` does not contain expected fields,
- or schema parsing failes because of invalid schema structure or unsupported schema feature.

7.1.1.3 Parsing the metadata received from Schema Registry

After the schemas are extracted from Schema Registry, it is necessary to parse them to receive name of the fields and flatten these fields. That means that instead of the structure with nested objects, the schema is represented only as a set of fields with only one-level depth where the name of each is composed of the namespace derived from name of the parents and the name of the field from the deepest level. Schema in the flattened form simplifies the further implementation and in the case of a more leveled schema, it is clearer in visualization. For JSON schema from listing 7.3 the final fields would be `productId`, `productName`, `price`, `dimensions.length`, `dimensions.width`, and `dimensions.height`. The prototype implementation currently supports parsing of JSON schemas.

Code listing 7.3 JSON schema example. [6]

```
{
   "$schema":"https://json-schema.org/draft/2020-12/schema",
   "$id":"https://example.com/product.schema.json",
   "title":"Product",
   "description":"A product from Acme's catalog",
   "type":"object",
   "properties":{
      "productId":{
         "type":"integer"
      },
      "productName":{
         "type":"string"
      },
      "price":{
         "type":"number"
      },
      "dimensions":{
         "type":"object",
         "properties":{
            "length":{
               "type":"number"
            },
            "width":{
               "type":"number"
            },
            "height":{
               "type":"number"
            }
         }
      }
   }
}
```

The parsed schemas are individually stored in the scructure SchemaRegistrySchema which contains the globally unique id within the Schema Registry, set of columns and schema name.

7.1.1.4 Saving the metadata into the dictionary

After the schemas are extracted from Schema Registry, they need to be saved in the dictionary. KafkaWriterImpl class is responsible for interaction with the dictionary. It implements the KafkaWriter interface which defines method for adding the topics, subject, schemas, and other object to the dictionary. In the listing 7.4 is shown method for adding the topic.

Code listing 7.4 KafkaWriter:addTopic method.

```
public IResDataType addTopic(String topicName) {
    IResEntity existingTopic = dictionary.getGlobalNamespace()
        .getChildByName(named(topicName),
                        EnumSet.of(EntityProps.KAFKA_TOPIC), true);

    if (existingTopic != null) {
        return (IResDataType) existingTopic;
    }

    return dictionary.createTopLevelEntity(named(topicName),
    EnumSet.of(EntityProps.KAFKA_TOPIC), DEF_SRC_TYPE);
}
```

In the case of topic, the KafkaWriterImpl first checks If topic of given name exist within the dictionary. If the answer is positive, it is returned the existing topic, in the opossite case it is created and returned new one.

7.2 Connector Kafka Dictionary module

Kafka Dictionary module has two main responsibilities:

- define the Kafka objects which do not meet any existing ones in Manta,
- and define the hierarchy between the objects.

7.2.1 Implementation of Kafka entity types

KafkaDictionary defines one method for creation of Kafka top level entity. The method is in the listing 7.5 and it is used for creating the subject, topic, and schema which are the entities under the global namespace as was described in the design chapter 6.3.2. Except the name of the entity, the method also expects as a parameter the entity properties that are used for distinguishing the created data type. In case of subject, the method is called with KAFKA_SUBJECT property, in case of topic with the KAFKA_TOPIC etc. The created data type is added as a child of the global namespace.

7.2.2 Hierarchy definition

Code listing 7.5 `KafkaDataDictionary:createTopLevelEntity` method.

```
public IResDataType createTopLevelEntity(EntityName entityName,
    Set<EntityProps> properties, DefinitionSourceType source) {
    LOGGER.trace("createKafkaDataType({}, {}, {})", entityName, properties, source);
    AbstractDataType<KafkaResolverEntitiesFactory> dataType =
        new AbstractDataType<>(this, entityName, properties, source);
    registerObject(dataType);
    getGlobalNamespace().addChild(dataType);
    addUnresolvedEntity(dataType);
    fireEntityCreated(dataType);
    return dataType;
}
```

Code listing 7.6 `KafkaDialect` rule definitions.

```
dbStructure
    .parent(EntityProps.GLOBAL_NAMESPACE)
    .allows(EntityProps.KAFKA_SCHEMA,
            EntityProps.KAFKA_SUBJECT,
            EntityProps.KAFKA_TOPIC,
            EntityProps.PROP_BUILTIN,
            EntityProps.PRIMITIVE_TYPE
            )
    .policy(ChildPolicy.ALLOW_DIFFERENT_DUPLICATES);
```

`KafkaDialect` is used to define the hierarchy structure between the different dictionary entities. For creation of hierarchy rules it used the class from Manta ecosystem – `DbStructureImpl`. Each rule says what can be the child for specific parent and the policy for adding the duplicates. The listing 7.6 contains rule definition for global namespace. It says that as a child of a entity with the property `GLOBAL_NAMESPACE` can be added the entities listed in `allows` method and that there are allow duplicates in the case of that their types are different. In other words under the global namespace is possible to add subject and topic with the same name, but not two topics with the same name.

7.3 Dataflow Generator

The output of the dictionary extractor is the H2 database saved on disk. In Manta are implemented processor that process the dictionary and create a dataflow graph. The processors are used through Spring beans configurations, the sample usage is shown in the listing 7.7. It defines a processor for Kafka topic, concretely `simpleProcessor` that creates a node from the name of the dictionary entity and delegates the processing of child to child processors.

Code listing 7.7 Usage of Manta’s dataflow dictionary generator processors.

```
<bean id="topicProcessor" parent="simpleProcessor">
    <property name="supportedObjectType" value="DATABASE"/>
    <property name="targetNodeType"
    value="#{ T(eu.profinit.manta.dataflow.model.NodeType)
    .KAFKA_TOPIC.getId()}"/>
    <property name="childProcessors">
        <list>
            <ref bean="schemaVersionProcessor"/>
        </list>
    </property>
</bean>
```

Chapter 8

Kafka scanner testing

This chapter is dedicated to testing of the functional protype. It is divided into two sections. The first section describes the unit testing and the second one presents the final outputs of the protype.

8.1 Unit Tests

Based on the assignment described in the chapter 4, the code is covered with unit tests. "*A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. In most programming languages, that is a function, a subroutine, a method or property. Within the implemented prototype are heavily used two Java frameworks for testing*". [32]

- **JUnit** is framework for writing unit tests. It provides support for implementation and execution of tests and also different methods for checking the expected a actual values of the tested objects. [33]

- **Mockito** is framework used for object mocking. [34] "*Mocking is creating an object that mimics the behavior of another object. It's a strategy for isolating an object to test it and verify its behavior. The advantages of mocking is that it allows to test only the class which should be covered with unit tests and remove any dependencies on other objects by determination what the results of external methods calls will be.*" [35]

In the listing 8.1 is shown sample unit tests where are used both of the frameworks. The test is marked with an annotation `@Test`. From the jUnit documentation: "*Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded.* [33]

The method tests processing of Confluent Platform Schema Registry responses with the list of the subjects. It mocks the `HttpResponse` to return successful status code and the JSON array with the subjects that corresponds to the Schema Registry response. Similarly are in the code tested situation where it is returned unexpected status code or invalid response content.

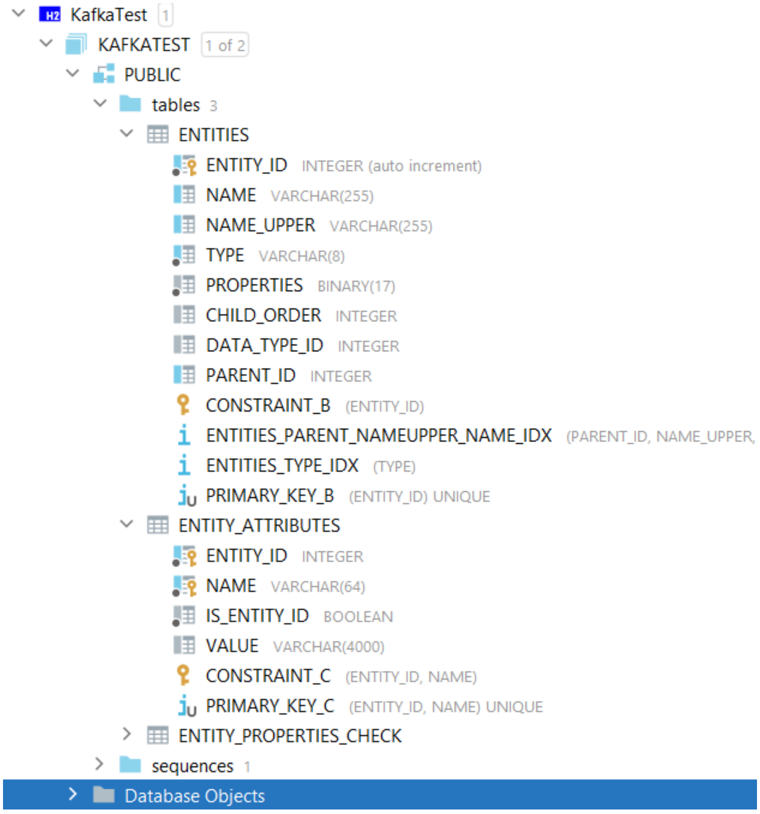
Code listing 8.1 ConfluentPlatformResponseProcessorTest unit test.

```
public HttpResponse mockHttpResponse(int statusCode, String content)
    throws IOException {
    HttpResponse response = mock(HttpResponse.class);
    HttpEntity httpEntity = mock(HttpEntity.class);
    StatusLine statusLine = mock(StatusLine.class);
    InputStream inputStream = new ByteArrayInputStream(content.getBytes());

    when(response.getEntity()).thenReturn(httpEntity);
    when(response.getStatusLine()).thenReturn(statusLine);
    when(statusLine.getStatusCode()).thenReturn(statusCode);
    when(httpEntity.getContent()).thenReturn(inputStream);
    return response;
}

@Test
public void processSubjectResponseTestWithStatusCode200() throws IOException,
    HttpResponse response = mockHttpResponse(HttpStatus.SC_OK,
                        "[subject1, subject2, subject3]");

    List<String> result = processor.processSubjectsResponse(response);
    List<String> expected = Arrays.asList("subject1", "subject2", "subject3");

    Assert.assertEquals(3, result.size());
    Assert.assertEquals(expected, result);
}
```

8.2 The metadata extractor outputs

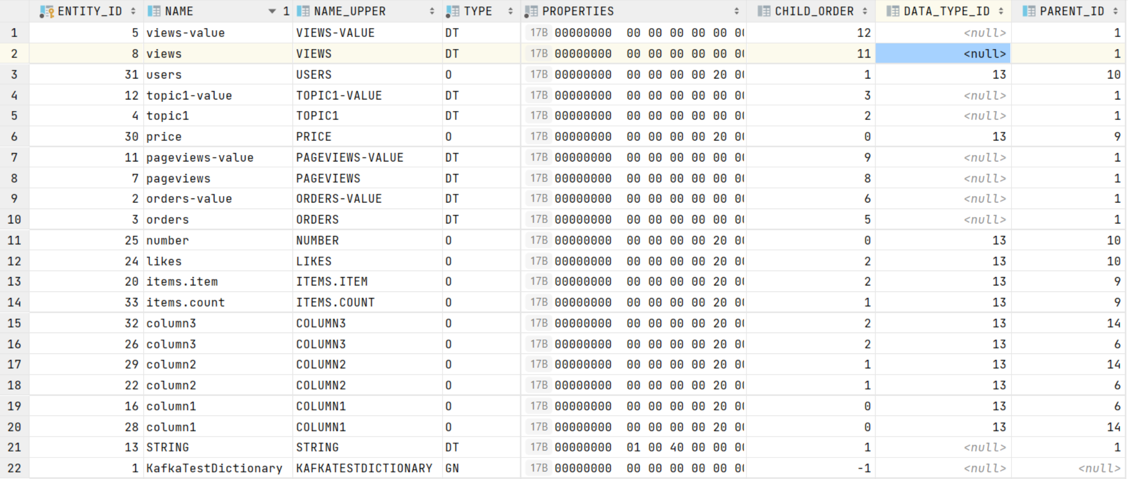
Based on the requirements 4, the module should extract the metadata from Schema Registry and save them on the disk in the dictionary persisted in the H2 database. The listing 8.1 shows the structure of the database and listing 8.2 with the table with the entities. For each entity is saved its name, type, properties, data type (if any), and parent.

The final solution of the extractor is effective from the perspective that can the extraction process fully automatize. In the future is planned to extend the prototype to support the designed manual input format. Then, the extractor will be able to meet the requirements of all clients whether they use a schema management service or not.



```
KafkaTest 1
  KAFKATEST 1 of 2
    PUBLIC
      tables 3
        ENTITIES
          ENTITY_ID INTEGER (auto increment)
          NAME VARCHAR(255)
          NAME_UPPER VARCHAR(255)
          TYPE VARCHAR(8)
          PROPERTIES BINARY(17)
          CHILD_ORDER INTEGER
          DATA_TYPE_ID INTEGER
          PARENT_ID INTEGER
          CONSTRAINT_B (ENTITY_ID)
          ENTITIES_PARENT_NAMEUPPER_NAME_IDX (PARENT_ID, NAME_UPPER,
          ENTITIES_TYPE_IDX (TYPE)
          PRIMARY_KEY_B (ENTITY_ID) UNIQUE
        ENTITY_ATTRIBUTES
          ENTITY_ID INTEGER
          NAME VARCHAR(64)
          IS_ENTITY_ID BOOLEAN
          VALUE VARCHAR(4000)
          CONSTRAINT_C (ENTITY_ID, NAME)
          PRIMARY_KEY_C (ENTITY_ID, NAME) UNIQUE
        ENTITY_PROPERTIES_CHECK
      sequences 1
    Database Objects
```

Figure 8.1 The structure of the result H2 database.



	ENTITY_ID	NAME 1	NAME_UPPER	TYPE	PROPERTIES	CHILD_ORDER	DATA_TYPE_ID	PARENT_ID
1	5	views-value	VIEWS-VALUE	DT	17B 00000000 00 00 00 00 00 0	12	<null>	1
2	8	views	VIEWS	DT	17B 00000000 00 00 00 00 00 0	11	<null>	1
3	31	users	USERS	0	17B 00000000 00 00 00 00 20 0	1	13	10
4	12	topic1-value	TOPIC1-VALUE	DT	17B 00000000 00 00 00 00 00 0	3	<null>	1
5	4	topic1	TOPIC1	DT	17B 00000000 00 00 00 00 00 0	2	<null>	1
6	30	price	PRICE	0	17B 00000000 00 00 00 00 20 0	0	13	9
7	11	pageviews-value	PAGEVIEWS-VALUE	DT	17B 00000000 00 00 00 00 00 0	9	<null>	1
8	7	pageviews	PAGEVIEWS	DT	17B 00000000 00 00 00 00 00 0	8	<null>	1
9	2	orders-value	ORDERS-VALUE	DT	17B 00000000 00 00 00 00 00 0	6	<null>	1
10	3	orders	ORDERS	DT	17B 00000000 00 00 00 00 00 0	5	<null>	1
11	25	number	NUMBER	0	17B 00000000 00 00 00 00 20 0	0	13	10
12	24	likes	LIKES	0	17B 00000000 00 00 00 00 20 0	2	13	10
13	20	items.item	ITEMS.ITEM	0	17B 00000000 00 00 00 00 20 0	2	13	9
14	33	items.count	ITEMS.COUNT	0	17B 00000000 00 00 00 00 20 0	1	13	9
15	32	column3	COLUMN3	0	17B 00000000 00 00 00 00 20 0	2	13	14
16	26	column3	COLUMN3	0	17B 00000000 00 00 00 00 20 0	2	13	6
17	29	column2	COLUMN2	0	17B 00000000 00 00 00 00 20 0	1	13	14
18	22	column2	COLUMN2	0	17B 00000000 00 00 00 00 20 0	1	13	6
19	16	column1	COLUMN1	0	17B 00000000 00 00 00 00 20 0	0	13	6
20	28	column1	COLUMN1	0	17B 00000000 00 00 00 00 20 0	0	13	14
21	13	STRING	STRING	DT	17B 00000000 01 00 40 00 00 0	1	<null>	1
22	1	KafkaTestDictionary	KAFKATESTDICTIONARY	GN	17B 00000000 00 00 00 00 00 0	-1	<null>	<null>

Figure 8.2 Extracted metadata saved in the H2 database.

Chapter 9

Conclusion

The aim of this work was to find different approaches for metadata extraction from Kafka and based on them design and implement the module that extracts the metadata. During the work, three different approaches were found and two of them were usable for Manta purposes.

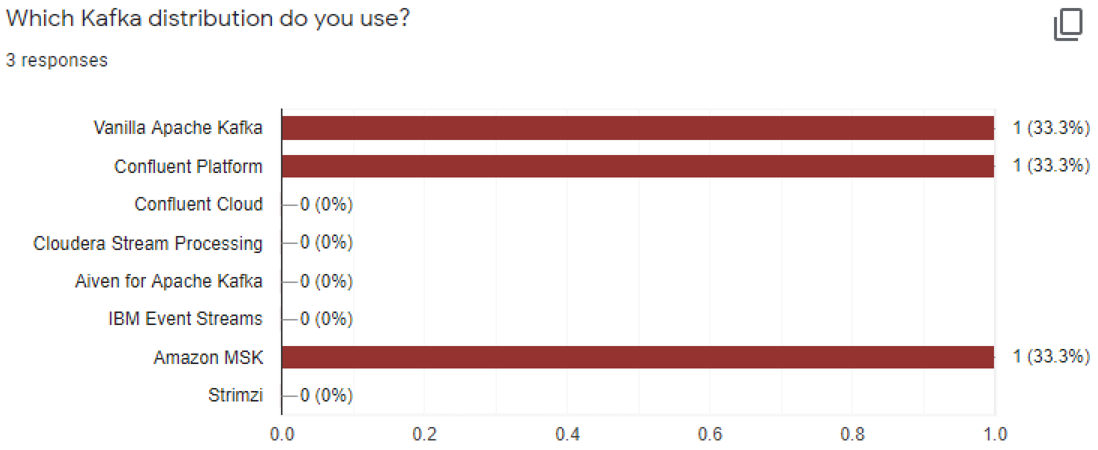
One of the extraction methods was the extraction from Schema Registry, specifically was chosen the Confluent Platform implementation. The current prototype implementation supports only the extraction for JSON schemas and topic name subject name strategy. There are plans to extend the prototype to support other naming strategies, data formats and possibly different Schema Registry implementations in the future. The implemented prototype is documented, tested and integrated with Manta software based on the collected requirements.

The second method was to use the manual input extraction. In the design chapter the manual input file structure was created and in the future it is planned to extend the prototype to provide this option as well.

Appendix A

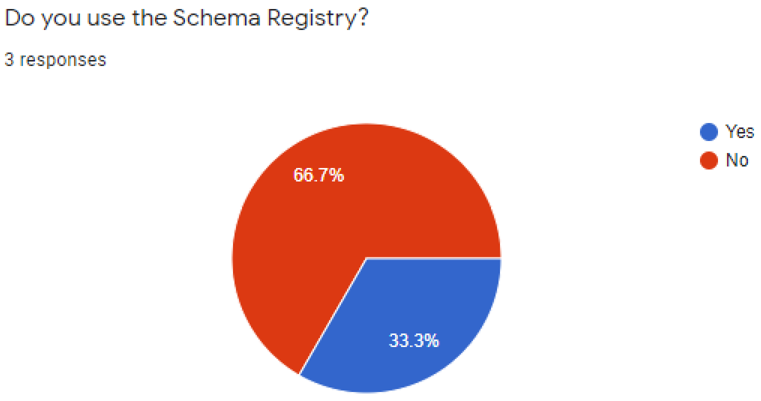
Kafka Questionnaire

This attachment contains the most important questions answered by Manta clients about Kafka and Kafka environment.



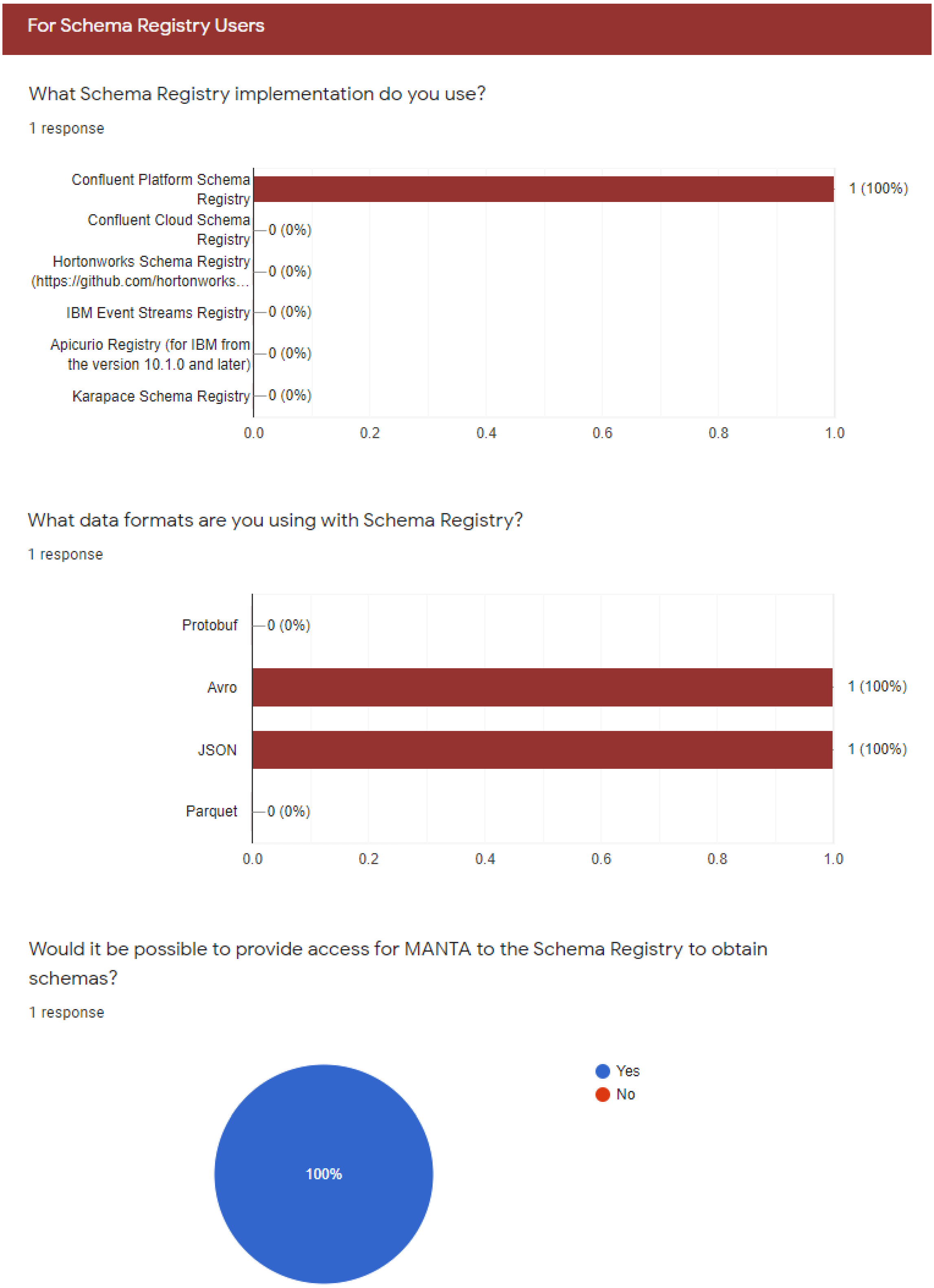
Which Kafka distribution do you use?

3 responses



Do you use the Schema Registry?

3 responses



For Schema Registry Users

What Schema Registry implementation do you use?

1 response

Confluent Platform Schema Registry
1 (100%)
Confluent Cloud Schema Registry
0 (0%)
Hortonworks Schema Registry (https://github.com/hortonworks...
0 (0%)
IBM Event Streams Registry
0 (0%)
Apicurio Registry (for IBM from the version 10.1.0 and later)
0 (0%)
Karapace Schema Registry
0 (0%)
0.0
0.2
0.4
0.6
0.8
1.0

What data formats are you using with Schema Registry?

1 response

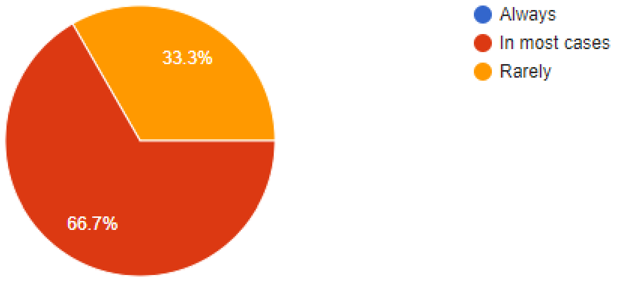
Protobuf
0 (0%)
Avro
1 (100%)
JSON
1 (100%)
Parquet
0 (0%)
0.0
0.2
0.4
0.6
0.8
1.0

Would it be possible to provide access for MANTA to the Schema Registry to obtain schemas?

1 response

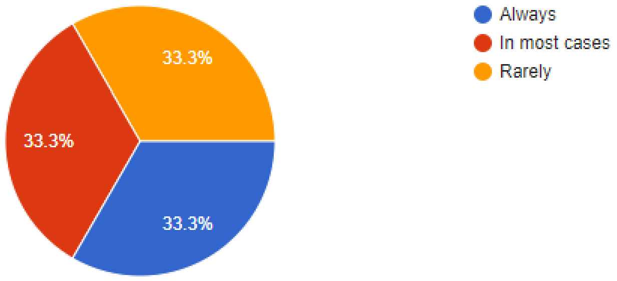
How often do you have one schema per topic?

3 responses



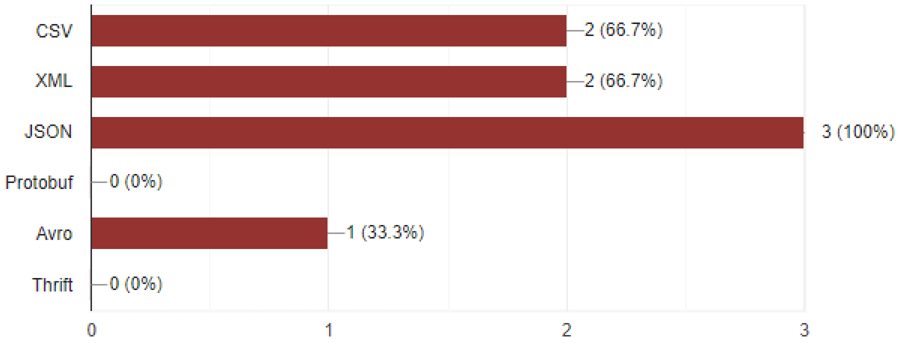
How often do you have one data format per topic?

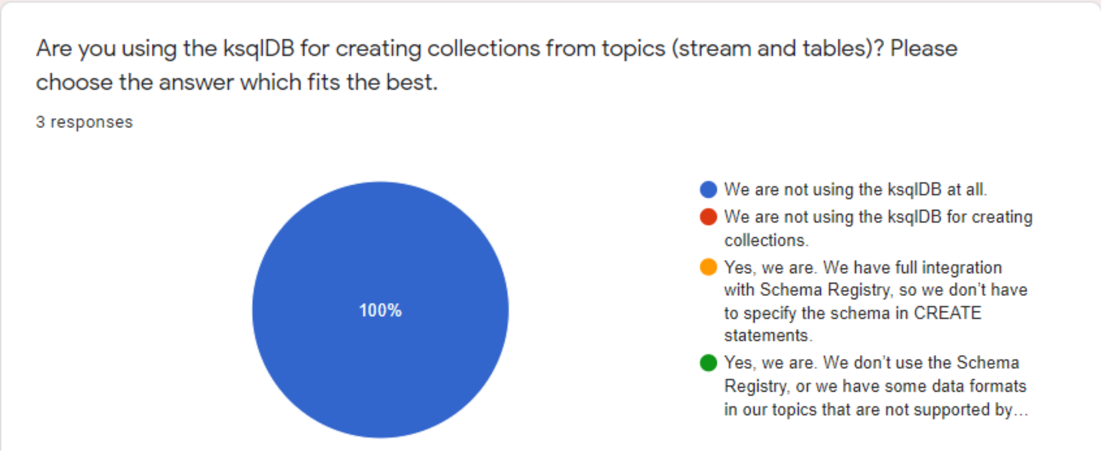
3 responses



What data formats are your producers publishing into Kafka?

3 responses





Are you using the ksqlDB for creating collections from topics (stream and tables)? Please choose the answer which fits the best.

3 responses

- 🔵 We are not using the ksqlDB at all.
- 🔴 We are not using the ksqlDB for creating collections.
- 🟠 Yes, we are. We have full integration with Schema Registry, so we don't have to specify the schema in CREATE statements.
- 🟢 Yes, we are. We don't use the Schema Registry, or we have some data formats in our topics that are not supported by…

Bibliography

[1] Unified Lineage Platform. `https://getmanta.com/`. [online] [accessed on 25-04-2021].

[2] Confluent documentation. `https://docs.confluent.io/`. [online] [accessed on 01-03-2021].

[3] František Bořánek. Apache Avro nebo Protocol Buffers. `https://blog.seznam.cz/2018/10/apache-avro-nebo-protocol-buffers/`, Nov 2018. [online] [accessed on 10-05-2021].

[4] Apache Avro 1.10.2 Specification. `https://avro.apache.org/docs/current`. [online] [accessed on 01-03-2021].

[5] ksqlDB Documentation. `https://docs.ksqldb.io/en/0.15.0-ksqldb/`. [online] [accessed on 25-04-2021].

[6] JSON Schema Specification. `https://json-schema.org/specification.html`. [online] [accessed on 01-05-2021].

[7] Mark Allen and Dalton Cervo. *Multi-domain master data management: Advanced MDM and data governance in practice.* Morgan Kaufmann, 2015.

[8] what is data flow diagram? `https://www.visual-paradigm.com/guide/data-flow-diagram/what-is-data-flow-diagram/`. [online] [accessed on 11-05-2021].

[9] AVRO - Serialization. `https://www.tutorialspoint.com/avro/avro_serialization.htm`. [online] [accessed on 10-05-2021].

[10] Editor. What is Data Pipeline: Components, Types, and Use Cases. `https://www.altexsoft.com/blog/data-pipeline-components-and-types/`, Mar 2020. [online] [accessed on 11-05-2021].

[11] Pethuru Raj and Ganesh Chandra Deka. *A Deep Dive into NoSQL Databases: The Use Cases and Applications.* Academic Press, 2018.

[12] Daniel Gutierrez. A Brief history of Kafka, LinkedIn's Messaging Platform. `https://insidebigdata.com/2016/04/28/a-brief-history-of-kafka-linkedins-messaging-platform/`, April 2016. [online] [accessed on 04-03-2021].

[13] Apache Kafka Documentation. `https://kafka.apache.org/`. [online] [accessed on 02-02-2021].

[14] Robert Gibb. What is a distributed system? `https://blog.stackpath.com/distributed-system/`, Jul 2019. [online] [accessed on 02-02-2021].

[15] Colin McCabe. Kafka Needs No Keeper - Removing Zookeeper dependency. https://www.confluent.io/blog/removing-zookeeper-dependency-in-kafka/. [online] [accessed on 22-01-2021].

[16] Event streams - overview. https://www.ibm.com/cloud/event-streams. [online] [accessed on 11-05-2021].

[17] Java™ programming language. https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html. [online] [accessed on 10-05-2021].

[18] Spring makes Java simple. https://spring.io/. [online] [accessed on 10-05-2021].

[19] Brett Porter, Jason van Zyl, and Olivier Lamy. Welcome to Apache Maven. https://maven.apache.org/. [online] [accessed on 10-05-2021].

[20] Apache Subversion. https://subversion.apache.org/. [online] [accessed on 10-05-2021].

[21] SonarQube Documentation. https://docs.sonarqube.org/latest/. [online] [accessed on 10-05-2021].

[22] Cloudera. Cloudera Streams Messaging. https://www.cloudera.com/content/www/en-us/products/cdf/streams-messaging.html. [online] [accessed on 11-05-2021].

[23] Managed Apache Kafka as a Service: Aiven. https://aiven.io/kafka. [online] [accessed on 10-05-2021].

[24] Kafka on kubernetes in a few minutes. https://strimzi.io/. [online] [accessed on 11-05-2021].

[25] Stéphane Maarek. Introduction to Schemas in Apache Kafka with the Confluent Schema Registry. medium.com/@stephane.maarek/introduction-to-schemas-in-apache-kafka-with-the-confluent-schema-registry, Oct 2019. [online] [accessed on 25-04-2021].

[26] Thiago Cordon. Schema evolution with Schema Registry. https://towardsdatascience.com/schema-evolution-with-schema-registry-8d601ee84f4b, Mar 2021. [online] [accessed on 22-03-2021].

[27] Apicurio Registry documentation. https://www.apicur.io/registry/docs/apicurio-registry/1.3.3.Final/index.html. [online] [accessed on 01-04-2021].

[28] Registry Project. https://registry-project.readthedocs.io, journal=Registry. [online] [accessed on 03-04-2021].

[29] Dani Traphagen. Kafka Streams vs. ksqlDB for Stream Processing. https://www.confluent.io/blog/kafka-streams-vs-ksqldb-compared/, Nov 2019. [online] [accessed on 15-02-2021].

[30] Neha Narkhede. Introducing KSQL: Streaming SQL for Apache Kafka. https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/, Aug 2017. [online] [accessed on 02-03-2021].

[31] Jay Kreps. Introducing ksqlDB. https://www.confluent.io/blog/intro-to-ksqldb-sql-database-streaming/, Nov 2019. [online] [accessed on 02-03-2021].

[32] What is unit testing? https://smartbear.com/learn/automated-testing/what-is-unit-testing/. [online] [accessed on 10-05-2021].

[33] Sam Brannen Stefan Bechtold. JUnit. https://junit.org/junit5/docs/current/user-guide/. [online] [accessed on 10-05-2021].

[34] Mockito Framework. https://site.mockito.org/. [online] [accessed on 10-05-2021].

[35] Erik Dietrich. What Is Mocking? - Typemock Blog. https://www.typemock.com/what-is-mocking/, Aug 2018. [online] [accessed on 10-05-2021].

Contents of enclosed media

readme.txt	the file with contents description
src	
implementation	the directory with source codes
thesis	source codes in the format LaTeX
text	thesis text
BT-Weberova-Michaela-2021	thesis text in PDF format