TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Gleb Komissarov IAIB

Apache Kafka config manager with HTTP API

Bachelor thesis

Supervisor: Lt Cdr Kieren Niĉolas Lovell

RNorN Head of TalTech CERT

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogiate kool

Gleb Komissarov IAIB

Apache Kafka konfiguratsioonihaldur HTTP API-ga

Lõputöö

Juhendaja: Lt Cdr Kieren Niĉolas Lovell

RNorN Head of TalTech CERT

Tallinn 2022

Author's declaration of originality

I hereby certify that I am the sole author of this thesis and this thesis has not been presented for examination or submitted for defence anywhere else. All used materials, references to the literature of others have been cited.

Author:	Gleb Komissarov	....
		(signature)
Date:	23.05.2022	

Abstract

Open-source software sometimes lacks features, that are needed for operation and maintenance on a large scale. In such cases, companies that require those features have to use paid SaaS/PaaS solutions or create their own. This thesis is dedicated to a project that solves this problem for estonian company Pipedrive by creating software to manage Apache Kafka clusters components on a large scale. The development consists of 3 parts: research, proof of concept prototype development and actual software development. All 3 parts are described in this thesis.

This thesis is written in English and is 31 pages long, including 6 chapters, 16 figures and 0 tables.

List of abbreviations and terms

ACL	Access Control List
API	Application Programming Interface
JVM	Java Virtual Machine

Table of Contents

1 Related work **9**

1.1 Kafka platforms . . . 9

1.2 Official Apache documentation . . . 9

1.3 Kafka academic research papers . . . 9

2 Introduction **10**

2.1 What is Apache Kafka? . . . 10

2.2 What problem does this thesis solve? . . . 10

3 Apache Kafka overview **12**

3.1 What is event streaming? . . . 12

3.2 What can I use event streaming for? . . . 12

3.3 Apache Kafka® is an event streaming platform. What does that mean? . . 13

3.4 How does Kafka work in a nutshell? . . . 13

3.5 Main Concepts and Terminology . . . 14

3.6 Kafka APIs . . . 15

4 How to solve the problem **17**

4.1 Why 3rd party solutions cannot be used . . . 17

4.1.1 Pricing . . . 17

4.1.2 Migration . . . 18

4.1.3 Critical infrastructure depending on 3rd party . . . 18

4.2 Possible ways to solve the problem . . . 18

5 How it has to be done **20**

5.1 Requirements . . . 20

5.1.1 HTTP API endpoints for Topics . . . 20

5.1.2 HTTP API endpoints for Users . . . 22

5.1.3 HTTP API endpoints for ACLs . . . 25

5.2 Python POC . . . 27

5.3 Scala vs Python . . . 27

5.4 How it is going to be used . . . 28

6 Conclusion **29**

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis **30**

References **31**

List of figures

1 Architecture overview 11

2 This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate. 15

3 Confluent Cloud pricing 18

4 List all topics with their configs in the specified cluster 20

5 Create topics from a map of topics and their configs in the specified cluster 21

6 Delete a topic and all the ACLs related to this topic 21

7 Update topics partition count 21

8 List all users and their auth configs(used to verify the password) 22

9 Create users from a map of users and their passwords in the specified cluster 23

10 Delete a user and all the ACLs related to this user 24

11 Update users password 24

12 List all ACLs grouped by user 25

13 Give {user-name} write access to {topic-name} 26

14 Give {user-name} read access to {topic-name} 26

15 Give {user-name} All access to cluster resource 26

16 Give {user-name} Describe access to cluster resource 27

1 Related work

1.1 Kafka platforms

In the beginning of my research on how to solve the problem, I looked at platforms, that provide enterprise grade Kafka distributions with commercial features and support for Kafka components that open-source version does not have. There are some platforms on the market, such as Confluent [1], which is the most developed right now. Using it would solve the problem of not having a proper interface to manage Kafka internal components, since one of the feature it provides, is HTTP API for Kafka Admin. However, it is really expensive and time consuming process to migrate the whole Kafka infrastructure of a big company to a completely new platform. Another thing to mention is that unlike open-source version, such platforms cost money, come with a set of extra tools and features, that are not needed and make us rely on a closed 3rd party code, which is better to be avoided. Overall reading documentations made for different platforms still helped to understand how things could be done, and provided some useful information about Kafka internals.

1.2 Official Apache documentation

After it became clear, that no Kafka platform will be able to solve the problem, without creating more serious problems, I went to the Kafka Admin API documentation [2] to find ways to programmatically alter Kafka configurations.

There I found what methods Kafka Admin Java API supports, how to use those and what versions of Kafka it will support. As in many other open-source projects, this documentation lacked usage examples, so I often had to find missing information in feature proposals [3], development tickets [4] and just by reading the source code [5].

1.3 Kafka academic research papers

During the research I have also read academic papers about Kafka. However they did not provide any useful data for completing the thesis project.

[1] `https://www.confluent.io/subscription`
[2] `https://kafka.apache.org/documentation.html#adminapi`
[3] `https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Improvement+Proposals`
[4] `https://issues.apache.org/jira/projects/KAFKA/issues`
[5] `https://github.com/apache/kafka`

2 Introduction

2.1 What is Apache Kafka?

Apache Kafka (Kafka) is an open source, distributed streaming platform that enables (among other things) the development of real-time, event-driven applications. So, what does that mean?

Today, billions of data sources continuously generate streams of data records, including streams of events. An event is a digital record of an action that happened and the time that it happened. Typically, an event is an action that drives another action as part of a process. A customer placing an order, choosing a seat on a flight, or submitting a registration form are all examples of events. An event doesn't have to involve a person—for example, a connected thermostat's report of the temperature at a given time is also an event.

These streams offer opportunities for applications that respond to data or events in real-time. A streaming platform enables developers to build applications that continuously consume and process these streams at extremely high speeds, with a high level of fidelity and accuracy based on the correct order of their occurrence.

LinkedIn developed Kafka in 2011 as a high-throughput message broker for its own use, then open-sourced and donated Kafka to the Apache Software Foundation. Today, Kafka has evolved into the most widely-used streaming platform, capable of ingesting and processing trillions of records per day without any perceptible performance lag as volumes scale. Fortune 500 organizations such as Target, Microsoft, AirBnB, and Netflix rely on Kafka to deliver real-time, data-driven experiences to their customers. [1] [2] [3]

2.2 What problem does this thesis solve?

Here I would like to explain the problem, this bachelor thesis project solves and why this project is a valid solution to this problem. Despite being a state of the art streaming platform with a lot of capabilities and features, Apache Kafka releases still lack some useful features. One of them is Admin API, that is used to configure and manage Kafka key resources – topics, users and ACLs for those. Right now Apache provides Java API and shell scripts that utilize the Java API to manage those resources. In the company I work at, we cannot use the Java API directly to manage Apache Kafka and the scripts are too slow and will not work well with the infrastructure setup, we have planned for the future. The problem is the absence of a utility to manage Kafka internal components

using HTTP API in the infrastructure. The idea of the project is to create software that will translate HTTP requests to Java API methods and send API requests to different Kafka clusters. This later can be utilized by any configuration manager to manage configurations of all accessible Kafka clusters of any form and shape.

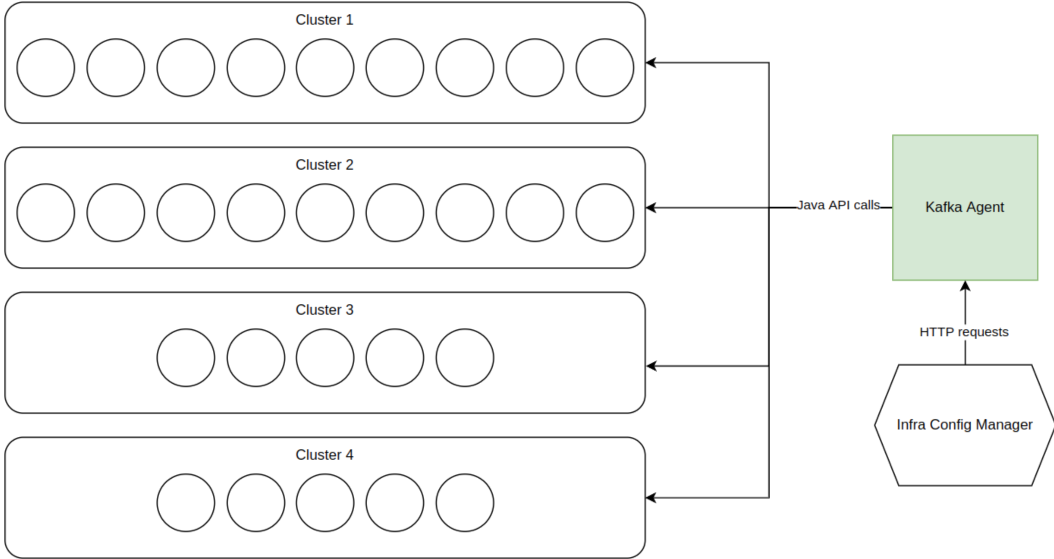


Figure 1. Architecture overview

3 Apache Kafka overview

3.1 What is event streaming?

Event streaming is the digital equivalent of the human body's central nervous system. It is the technological foundation for the 'always-on' world where businesses are increasingly software-defined and automated, and where the user of software is more software.

Technically speaking, event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

3.2 What can I use event streaming for?

Event streaming is applied to a wide variety of use cases [1] across a plethora of industries and organizations. Its many examples include:

- To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.
- To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
- To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
- To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.
- To monitor patients in hospital care and predict changes in condition to ensure timely treatment in emergencies.
- To connect, store, and make available data produced by different divisions of a company.
- To serve as the foundation for data platforms, event-driven architectures, and microservices.

[1] https://kafka.apache.org/powered-by

3.3 Apache Kafka® is an event streaming platform. What does that mean?

Kafka combines three key capabilities so you can implement your use cases for event streaming end-to-end with a single battle-tested solution:

1. To publish (write) and subscribe to (read) streams of events, including continuous import/export of your data from other systems.
2. To store streams of events durably and reliably for as long as you want.
3. To process streams of events as they occur or retrospectively.

And all this functionality is provided in a distributed, highly scalable, elastic, fault-tolerant, and secure manner. Kafka can be deployed on bare-metal hardware, virtual machines, and containers, and on-premises as well as in the cloud. You can choose between self-managing your Kafka environments and using fully managed services offered by a variety of vendors.

3.4 How does Kafka work in a nutshell?

Kafka is a distributed system consisting of servers and clients that communicate via a high-performance TCP network protocol[4]. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments.

Servers: Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect [1] to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters. To let you implement mission-critical use cases, a Kafka cluster is highly scalable and fault-tolerant: if any of its servers fails, the other servers will take over their work to ensure continuous operations without any data loss.

Clients: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures. Kafka ships with some such clients included, which are augmented by dozens of clients provided by the Kafka community: clients are available for Java and Scala including the higher-level Kafka Streams [2] library, for Go, Python, C/C++, and many other programming languages as well as

[1] https://kafka.apache.org/documentation/#connect
[2] https://kafka.apache.org/documentation/streams/

REST APIs.

3.5 Main Concepts and Terminology

An event records the fact that ”something happened” in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here’s an example event:

- Event key: ”Alice”
- Event value: ”Made a payment of $200 to Bob”
- Event timestamp: ”Jun. 25, 2020 at 2:06 p.m.”

Producers are those client applications that publish (write) events to Kafka, and consumers are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.

Events are organized and durably stored in topics. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be ”payments”. Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka’s performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.[5]

Topics are partitioned, meaning a topic is spread over a number of ”buckets” located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic’s partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer

of a given topic-partition will always read that partition's events in exactly the same order as they were written.

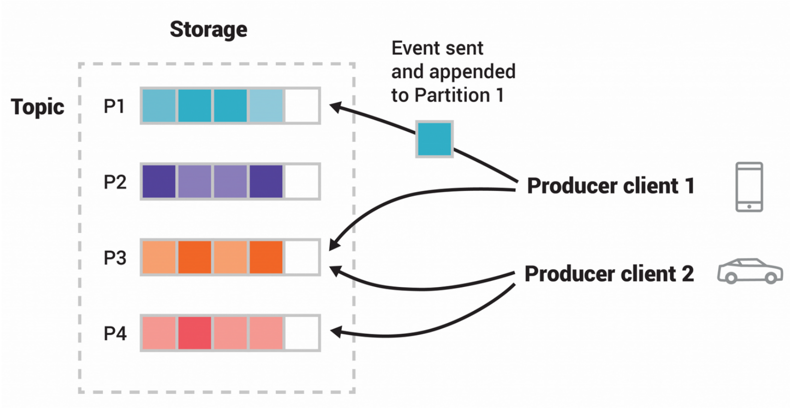


Figure 2. This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

To make your data fault-tolerant and highly-available, every topic can be replicated, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on. A common production setting is a replication factor of 3, i.e., there will always be three copies of your data. This replication is performed at the level of topic-partitions.

3.6 Kafka APIs

In addition to command line tooling for management and administration tasks, Kafka has five core APIs for Java and Scala:

- The Admin API [1] to manage and inspect topics, brokers, and other Kafka objects.
- The Producer API to publish (write) a stream of events to one or more Kafka topics.
- The Consumer API to subscribe to (read) one or more topics and to process the stream of events produced to them.
- The Kafka Streams API to implement stream processing applications and microservices. It provides higher-level functions to process event streams, including transformations, stateful operations like aggregations and joins, windowing, processing based on event-time, and more. Input is read from one or more topics in order to

[1] `https://kafka.apache.org/documentation.html#adminapi`

generate output to one or more topics, effectively transforming the input streams to output streams.

- The Kafka Connect API to build and run reusable data import/export connectors that consume (read) or produce (write) streams of events from and to external systems and applications so they can integrate with Kafka. For example, a connector to a relational database like PostgreSQL might capture every change to a set of tables. However, in practice, you typically don't need to implement your own connectors because the Kafka community already provides hundreds of ready-to-use connectors.[6]

4 How to solve the problem

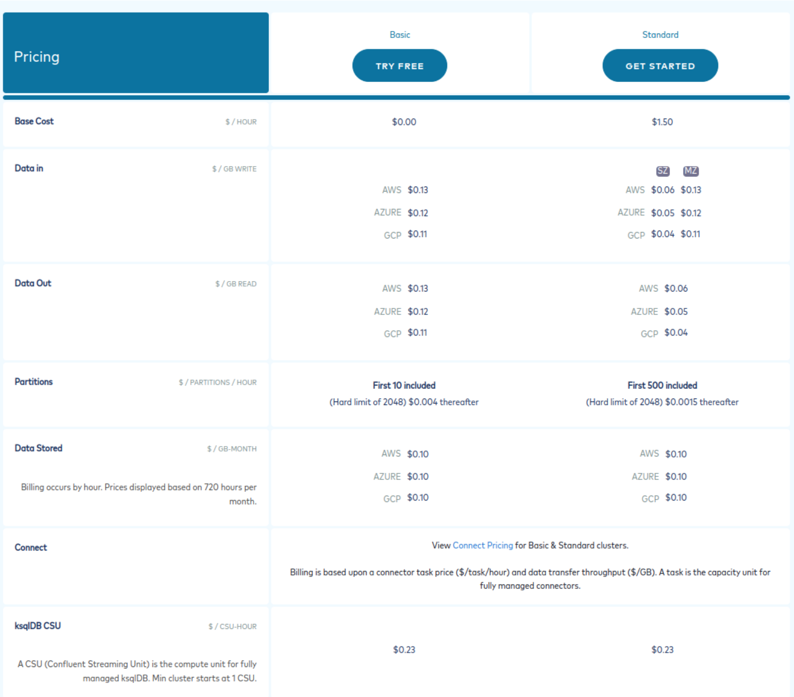
4.1 Why 3rd party solutions cannot be used

The most obvious way to solve the problem seems to be the use of 3rd party service or platform for hosting and managing Apache Kafka. Examples of those platforms would be [Confluent Cloud, conduktor.io, axual.com].

Those platforms provide a similar set of features: hosting Kafka, managing topics, users and ACLs, security features, compliance with data protection laws. They also share the same problems, if you are planning to migrate a big enterprise infrastructure onto those platforms.

4.1.1 Pricing

In the company, I am doing this project for, we have around 6 live regions. In every region there is at least 7 Kafka clusters, built for different services to share data with each other. My company has a bit more than 100,000 clients(other companies) that use those services to read/write/analyze their sales/marketing data. This results in tens of terabytes of temporary data stored, hundreds of gigabytes worth of data read/write operations every day and networking load that is not supported by any of those platforms. It is unnecessarily hard to estimate the possible price, that the providers would ask for this infrastructure. Required resources go way beyond standard and enterprise plan limits on those platforms and it would require a bigger research done for those companies, before they can tell if they are able to do it and name the price. It is also highly unlikely, that their price can be cheaper than our current setup, which is just servers on cloud platforms like AWS and Rackspace.



Pricing	Basic (TRY FREE)	Standard (GET STARTED)
Base Cost $ / HOUR	$0.00	$1.50
Data in $ / GB WRITE	AWS $0.13 AZURE $0.12 GCP $0.11	SZ / MZ AWS $0.06 $0.13 AZURE $0.05 $0.12 GCP $0.04 $0.11
Data Out $ / GB READ	AWS $0.13 AZURE $0.12 GCP $0.11	AWS $0.06 AZURE $0.05 GCP $0.04
Partitions $ / PARTITIONS / HOUR	First 10 included (Hard limit of 2048) $0.004 thereafter	First 500 included (Hard limit of 2048) $0.0015 thereafter
Data Stored $ / GB-MONTH Billing occurs by hour. Prices displayed based on 720 hours per month.	AWS $0.10 AZURE $0.10 GCP $0.10	AWS $0.10 AZURE $0.10 GCP $0.10
Connect	View Connect Pricing for Basic & Standard clusters. Billing is based upon a connector task price ($/task/hour) and data transfer throughput ($/GB). A task is the capacity unit for fully managed connectors.	
ksqlDB CSU $ / CSU-HOUR A CSU (Confluent Streaming Unit) is the compute unit for fully managed ksqlDB. Min cluster starts at 1 CSU.	$0.23	$0.23

Figure 3. Confluent Cloud pricing

4.1.2 Migration

Another big problem with switching to Kafka platform is the technical process of migrating data there and making services consume and produce data to the new infrastructure. While it is possible to make this process seamless to the customers, it is still a big risk and will require months worth of preparations and work done. This process will not make any useful impact on the infrastructure, except for the possibility to use HTTP API for managing Kafka internal components.

4.1.3 Critical infrastructure depending on 3rd party

Most of the microservices in my company rely on Kafka to deliver real time data to the customers and other services. Making it rely on a 3rd party platform is a business risk and another possible breakpoint for the application.

4.2 Possible ways to solve the problem

As mentioned earlier, Apache Kafka can be configured via Java API or shell scripts, both of which come with the software. While scripts can be executed by any configuration manager, their execution is slow, compared to running Java API directly (5-10 vs 0.01-0.5 seconds for a single request). Shell scripts are also limited, compared to Java API in terms

of bulk editing. On the other hand, there is no configuration manager that can use Java API directly. Sending HTTP requests is a common task for a configuration manager, so the only solution to the problem is to write a software that will translate HTTP requests to Java API requests.

5 How it has to be done

5.1 Requirements

Finished product must meet the following requirements:

5.1.1 HTTP API endpoints for Topics

```
GET /topics/{cluster-id}
response = {
    "{topic-name}": {
        "cleanup.policy": "{cleanup-policy}",
        "partitions": {partitions-count},
        "replication-factor": "{replication-factor}"
    },
    "{2nd-topic-name}": {
        ...
        ...
    },
    ...
    ...
}
```

Figure 4. List all topics with their configs in the specified cluster

```
POST /topics/{cluster-id}/
data = {
    "{topic-name}": {
        "cleanup.policy": "{cleanup-policy}",
        "partitions": {partitions-count},
        "replication-factor": "{replication-factor}"
    },
    "{2nd-topic-name}": {
        ...
        ...
    },
    ...
    ...
}
response =
    200 OK if creation was successful
```

Figure 5. Create topics from a map of topics and their configs in the specified cluster

```
DELETE /topics/{cluster-id}/{topic-name}
response =
    200 OK if deletion was successful
```

Figure 6. Delete a topic and all the ACLs related to this topic

```
PUT /topics/{cluster-id}/{topic-name}
data = {"partitions": {partitions-count}}
response =
    200 OK if update was successful
 400 Bad Request if updated partitions count was less than current
```

Figure 7. Update topics partition count

5.1.2 HTTP API endpoints for Users

```
GET /users/{cluster-id}
response = {
    "{user-name}": {
        "SCRAM-SHA-512": {
            "salt": "{salt}",
            "stored_key": "{stored_key}",
            "server_key": "{server_key}",
            "iterations": "{iterations}"
        },
        "SCRAM-SHA-256": {
            "salt": "{salt}",
            "stored_key": "{stored_key}",
            "server_key": "{server_key}",
            "iterations": "{iterations}"
        }
    },
    "{2nd-user-name}": {
        ...
        ...
    },
    ...
    ...
}
```

Figure 8. List all users and their auth configs(used to verify the password)

```
POST /users/{cluster-id}/
data = {
    "{user-name}": {
        "password": "{password}"
    },
    "{2nd-user-name}": {
        ...
        ...
    },
    ...
    ...
}
# Returns a list of created users and their auth configs
response = {
    "{user-name}": {
        "SCRAM-SHA-512": {
            "salt": "{salt}",
            "stored_key": "{stored_key}",
            "server_key": "{server_key}",
            "iterations": "{iterations}"
        },
        "SCRAM-SHA-256": {
            "salt": "{salt}",
            "stored_key": "{stored_key}",
            "server_key": "{server_key}",
            "iterations": "{iterations}"
        }
    },
    "{2nd-user-name}": {
        ...
}, 200 OK if creation was successful
```

Figure 9. Create users from a map of users and their passwords in the specified cluster

```
DELETE /users/{cluster-id}/{user-name}
response =
    200 OK if deletion was successful
```

Figure 10. Delete a user and all the ACLs related to this user

```
PUT /{cluster-id}/users/{user-name}
data = {"password": {password}}
response =
    200 OK if update was successful
```

Figure 11. Update users password

5.1.3 HTTP API endpoints for ACLs

```
GET /acls/{cluster-id}
response = {
    "{user-name}": {
        "group": {
            "{group-name}": "{permissions}",
            "{2nd-group-name}": "{permissions}",
            ...
            ...
        },
        "topic": {
            "{topic-name}": "{permissions}",
            "{2nd-topic-name}": "{permissions}",
            ...
            ...
        },
        "cluster": {
            "{cluster-id}": "{permissions}"
            ...
            ...
        }
    },
    "{2nd-user-name}": {
        ...
        ...
    },
    ...
    ...
}
```

Figure 12. List all ACLs grouped by user

```
POST /acls/{cluster-id}/producer/{topic-name}/;
data = {
    "user": "{user-name}"
}
response =
    200 OK if creation was successful
```

Figure 13. Give {user-name} write access to {topic-name}

```
POST /acls/{cluster-id}/consumer/{topic-name}/;
data = {
    "user": "{user-name}"
}
response =
    200 OK if creation was successful
```

Figure 14. Give {user-name} read access to {topic-name}

```
POST /acls/{cluster-id}/admin/;
data = {
    "user": "{user-name}"
}
response =
    200 OK if creation was successful
```

Figure 15. Give {user-name} All access to cluster resource

```
POST /acls/{cluster-id}/describer/;
data = {
    "user": "{user-name}"
}
response =
    200 OK if creation was successful
```

Figure 16. Give {user-name} Describe access to cluster resource

5.2 Python POC

Before starting to write the thesis, I have spent some time working on a proof of concept for this software. I made a small web application using Python language, Flask framework and Kafka admin scripts. For every endpoint described above it was running a corresponding Kafka Admin script with required arguments. It was deployed to a Kubernetes pod in a development region and given network access to all the Kafka clusters in that region. After testing it out, it was clear that the scripts really lack performance. I suspect that it happened due to the fact, that every time the script was ran, JVM had to be started up in order for script to use the Java API. After that the output had to be processed and optionally passed to another script. Some operations required multiple scripts to be ran or rerunning one script multiple times with different arguments. It resulted in endpoints such as ”GET /users/cluster-id” taking 20 and more seconds to complete the request. While this time is okay for a proof of concept, it is not suitable for production. Therefore it was decided to use the Java API directly to possibly get better performance.

5.3 Scala vs Python

After finishing tests for the Python proof of concept, I have started creating the Scala application, that uses Kafka AdminClient library to interact with Java API. The following benefits of using the it were immediately noticed:

Unlike the proof of concept Python app that parses standard output of the Kafka Admin scripts, Kafka AdminClient library provides types and exceptions for safer and more predictable interactions with Kafka Admin API. This allows to create a more robust and fault tolerant software.

It is also much faster for various reasons. AdminClient library provides tools for bulk operations on Kafka objects, which scripts do not support. Using AdminClient is also much faster, because the API response does not need to be processed to human readable text and then parsed by Python, it all happens inside one Java process on the object level. Multithreading, supported by Scala is also helpful.

Using Kafka AdminClient directly from Scala will make future Kafka version upgrades more convenient and safe, since the library is downloaded directly from Maven and changing a version will take just one change in the build file.

The only negative side of using Scala is the web framework to be used. Unlike Flask for Python which is pretty popular and well explained, with Scala I had to use Akka framework which is certainly less popular, therefore less solutions for common problems can be found on the Internet. Since the project uses only minimal functionality of the framework, it is not a big problem.

5.4 How it is going to be used

By the time the thesis is defended the first version of Kafka Agent written with Scala will probably be ready. After that it is going to be deployed the same way as Python proof of concept and tested on Kafka clusters in our development and test regions with some configuration manager. If it works well, it will be also used in live regions. If not, it will be fixed until it works well.

6 Conclusion

To conclude, the author was able to successfully research the topic and produce a fully working prototype that solves the problem of absence of a utility to manage Kafka internal components with HTTP API. Knowledge, that was acquired during the research has been proven to be very useful for completing the production grade software to solve this problem.

Main contributions of this thesis are:

- Definition of requirements for the solution
- Kafka internal components research on the code level; documentation
- Definition of possible methods to meet the requirements for the solution
- Development of prototype software to define and test final product architecture
- Final product development

This results in a solved problem, a good amount of time saved for the infrastructure department and a big amount of money saved for the company.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis

I Gleb Komissarov

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis ”Apache Kafka config manager with HTTP API”, supervised by Lt Cdr Kieren Niĉolas Lovell
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

References

[1] "Apache kafka." https://www.ibm.com/cloud/learn/apache-kafka, Feb 2020. Accessed on 2022-07-04.

[2] N. N. . R. J. Kreps J., *Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB (Vol. 11, pp. 1-7).*, ch. 1. 2011.

[3] G. N., *Apache kafka (pp. 30-31). Birmingham, UK: Packt Publishing.*, ch. 2. 2013.

[4] "Apache kafka with real-time data streaming." https://www.researchgate.net/publication/348575301_Apache_kafka_with_real-time_data_streaming. Accessed on 2022-20-04.

[5] "Streams and tables: Two sides of the same coin." https://dl.acm.org/doi/pdf/10.1145/3242153.3242155. Accessed on 2022-20-04.

[6] "Kafka 3.1 documentation." https://kafka.apache.org/documentation/. Accessed on 2022-20-04.