POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

The Faculty of Information Technology

Chair of Software Engineering

Software and Database Engineering

Alicja Koper
Student no. 17900

Event streams and stream processing as a response to communication and data challenges in microservice architecture.

Master Thesis written under supervision of:

Mariusz Trzaska Ph. D.

Warsaw, January 2020

Abstract

One of the troubles that can be faced in microservice-based architecture is data – how it is moved between services, shared and stored. Following thesis provides analysis of different approaches to problem of communication between microservices and investigates how using events and stream processing with Apache Kafka and Kafka Streams library could address difficulties of data sharing and exchange in decentralized environments. It presents analysis of chosen tools and approach themselves as well as provides conclusions based on created prototype.

Key words

Microservices, Communication, Stream Processing, Messaging

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Alicja Koper
Nr albumu 17900

Strumienie danych jako odpowiedź na problemy komunikacji i współdzielenia informacji w architekturze mikroserwisowej

Praca magisterska napisana
pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, styczeń 2020

Streszczenie

Jednymi z wiodących problemów, na które natrafić można projektując oraz wdrażając systemy oparte na architekturze mikroserwisowej są komunikacja między poszczególnymi komponentami oraz obsługa współdzielonych danych. Problemy te, oraz różnorodne próby radzenia sobie z nimi, stanowiły motywację dla podjęcia tematu niniejszej pracy. Praca skupia się na koncepcji wykorzystania platformy Apache Kafka oraz biblioteki Kafka Streams do przesyłania oraz procesowania strumieni danych w celu zastąpienia innych form komunikacji między mikroserwisami. Przedstawiona została analiza tematu oraz wnioski wyciągnięte na podstawie zebranej wiedzy oraz stworzonego prototypu systemu.

Słowa kluczowe

Mikroserwisy, Komunikacja, Strumienie danych, Messaging

Table of contents

1. Introduction

This chapter describes context of the thesis and author's motivation for chosen topic. It also briefly presents tools and technologies used to build a prototype as well as highlights the structure and results of work.

1.1. Work context

While designing and building applications may seem easier as the technological progress continues, it is at the same time becoming more and more complex. Interesting and sometimes even revolutionary inventions are being discovered across almost every area related to system architecture and development - from databases through frameworks to distributed systems and messaging. But every improvement brings also new obstacles and problems to solve.

This also applies to the concept of microservices. While they give developers the opportunity to deliver software in a faster, fine-grained way, they also come with their own complications. Author believes that data – how it is moved between services, shared and stored in a microservice-based architecture is one of the core problems of such concept.

One possible way of mitigating the risks that come with data in microservice-based architecture may be to choose an event-driven approach and use streaming platform – such as Apache Kafka with Kafka Streams library – to both store and process the data. This idea is what provided the main motivation for given thesis and it will be discussed and investigated within further chapters. The innovative character of those tools encouraged author to investigate how they can be used in microservice architecture and what should be considered in order to make them an appropriate solution for communication and data sharing.

1.2. Results of work

The main result of this paper is to provide the in-depth analysis of event driven approach to building microservices with the use of Apache Kafka and Kafka Streams library as well as to investigate how such approach addresses problems of data sharing and exchange in decentralized environments. It evaluates the concept of 'database inside out`[1] - described in detail in chapters 3 and 4 - based on a prototype of application built with such concept in mind, using aforementioned mechanisms. It also presents analysis of chosen tools themselves while emphasizing the crucial configurations and properties. Moreover, this work will discuss experiences and problems encountered during prototype creation and compare examined approach with most popular and widely used solutions.

1.3. Tools and technologies

The key technologies in context of this thesis are Apache Kafka and Kafka Streams library. What they are, how they work and how they can be used as a communication and storage layers is described in detail in chapters 4 and further.

Java was the programming language of choice for the presented prototype. This was mainly motivated by author's experience as a Java developer but also by this language's popularity and the fact that Kafka Streams is available only as a Java or Scala library.

Spring (including Spring for Apache Kafka 2.2.5.RELEASE) together with Spring Boot (2.1.4.RELEASE with spring-boot-starter-web) were used in the development process. Spring is currently one of the most popular and widely used Java frameworks for developing scalable and robust web applications. Including Spring and Spring Boot was motivated by the need of investigating how chosen approach would fit into typical business application environment where such solutions are in many cases already in use.

For development, Confluent Platform in version 5.1.2 together with Confluent Platform CLI were used. Confluent Platform is a powerful tool that brings together multiple community and commercial components supported by Confluent, like Apache Kafka, Zookeeper, KSQL server and many others. It was used only locally for the ease of development process - with one Kafka broker (Kafka version 2.1.1) running - and only few components of the platform were used, what is described in detail in chapter 5.

Prototype was written in IntelliJ IDEA Ultimate as an IDE of choice and run locally on one machine with MacOs High Sierra operating system and 8GB of RAM. As the prototype consist only of backend services Postman was used to interact with the exposed REST API.

1.4. Structure of work

First chapter provides an overview of the thesis, presents author's motivation and briefly describes goal of the paper and its structure as well as introduces the choice of tools used during prototype creation.

Chapter second and third are built around the concept of microservices – they provide the overview of the term and discuss the importance of data and communication in this type of architecture.

Chapter four describes in detail two core elements of investigated approach – Apache Kafka and Kafka Streams library. It illustrates the mechanism of aforementioned solutions and outlines key elements in context of given thesis.

Chapter five presents the prototype being the intermediate effect of analysis presented in this paper. It writes up the architecture and choice of tools, shows functionalities and describes particular components.

Chapter six elaborates on the concept of events, materialized views and stream processing in context of communication and data storage in microservice-based architecture. It both shows the solutions implemented in a server side application based on experience from developing the prototype as well as theoretically describes what configurations are crucial in analyzed approach.

The last chapter sums up the conclusions and final thoughts gathered in the process of writing the thesis.

2. Microservices – concepts and challenges

2.1. What does `microservice` mean?

It is challenging to provide a brief definition of what a microservice is. As Martin Fowler summarized – "*there is no precise definition of this architectural style, there are certain common characteristics (...)*"[2]. The general concept, however, was quite well captured by Sam Newman who in preface to his book titled "Building microservices" wrote that: "*Microservices are an approach to distributed systems that promote the use of finely grained services with their own lifecycles, which collaborate together. Because microservices are primarily modeled around business domains, they avoid the problems of traditional tiered architectures. Microservices also integrate new technologies and techniques that have emerged over the last decade, which helps them avoid the pitfalls of many service-oriented architecture implementations." [3]*

As the quoted definition is relatively broad and undetailed it should be expanded upon. The fact of microservices being finely grained could be understood in few different ways. First of all – it does not necessarily mean they have to be small to some defined extent. Size is less of a benchmark than the separated functionality. Like it is described in further part of this chapter – bounded context could be what defines the size and boundaries of each particular service. The very general idea of difference between monolith application and microservices is presented on Figure 2.3-1 below. As it shows – each of the monolithic application's areas was extracted to an individual independent component. This independence is what enables separate lifecycles, deployment strategies, updates and scaling. Each of those services is now managed independently of others and is focused on its own set of capabilities – hence the avoiding of *problems of traditional tiered architectures*.

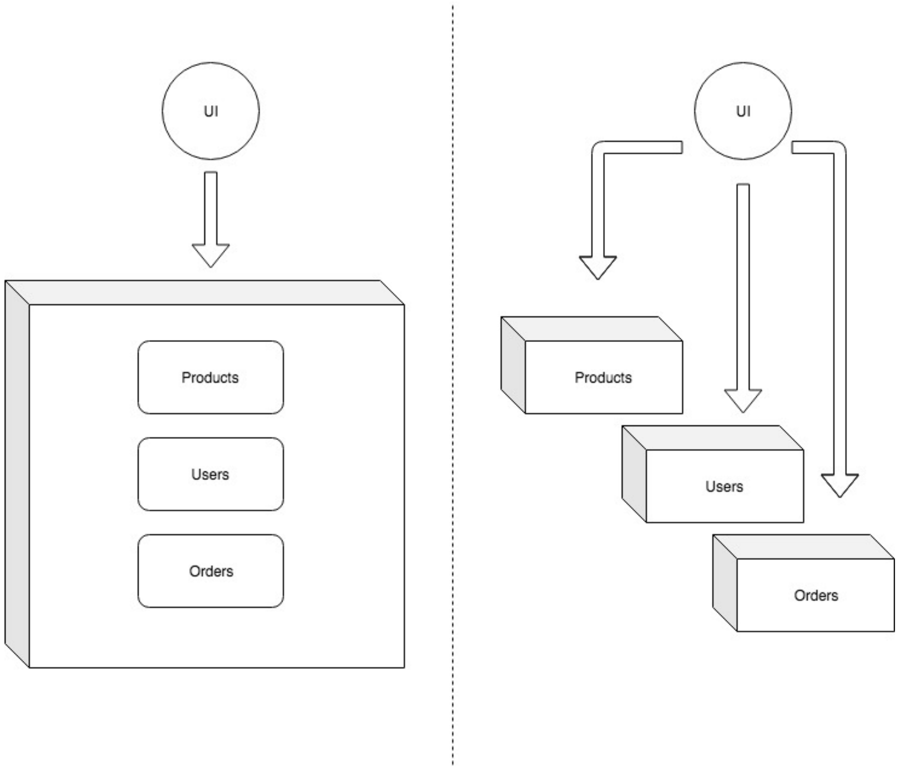


Figure 2.3-1 Simple presentation of monolithic application vs microservices;
Source: own elaboration

The possibility to implement new technologies and innovative solutions is relatively higher in microservice-based architecture due to the aforementioned independence between particular services. Each service can be built with different tools depending on what would be the best approach for solving problems of this service's domain. This technological flexibility is directly connected with another principle of microservices – decoupling. Making a change in the area of one service should be possible without any need of interference with other services.

2.2. Key concepts

The topic of microservices is broad and the attempt of understanding it should begin with describing core definitions. Two key concepts – bounded contexts and loose coupling – will be described in more details in this subsection.

2.2.1 Bounded Context

To describe the term of bounded context it is important to first at least briefly mention what Domain Driven Design (also referenced to as *DDD*) is. Domain Driven Design is an approach to software development that was introduced in 2003 by Eric Evans in a book "*Domain-Driven Design: Tackling Complexity in the Heart of Software.*"[4]. It represents set of rules and concepts that are made to design and implement complex business models. It emphasizes way of thinking about projects in terms of domains, where the word *domain* can be defined as an area of business (or how Eric Evans named it – *sphere of knowledge[4]*), and models – that exist to solve domain problems.

Bounded context is a limited area in which model is applicable. It draws both logical boundaries as well as boundaries for separate teams to work within – because maintenance of a bounded context should not be split between teams. It could be compared to different divisions of one company. Each division has its own responsibilities, handles its own tasks and possibly has its own set of acronyms but still all those divisions are parts of a bigger whole. The role of bounded contexts is not connected with physical boundaries. It is rather about the problem-specific responsibilities within certain area of business.

Identifying bounded context is usually a troublesome process – not only because the term itself is difficult to thoroughly understand but also because often there are concepts that exist in multiple spheres of knowledge and are problematic to be captured. As an important tool for separating bounded context Eric Evans mentions the language. Within one model everyone should have no difficulties with recognizing meanings of any word. Both developers and domain experts should use one vocabulary in their discussions and all used terms should be clear and understandable and have precisely one exact meaning. In other words – certain ubiquitous language must be present within each bounded context.

Each microservice should exist within one bounded context. As Sam Newman sums up "*if our service boundaries align to the bounded contexts in our domain, and our microservices represent those bounded contexts, we are off to an excellent start in ensuring that our microservices are loosely coupled and strongly cohesive*"[3] That observation leads to another two key concepts – loose coupling and high cohesion.

2.2.2 Loose Coupling and High Cohesion

Loose coupling is a term which - in context of systems design - states that services should not be connected in a way that enforces high level of dependency between them. Any dependencies between services should be as small as it is possible and therefore interference made within one service should in most cases not have any implications in other services.

The extent to which services are coupled together is strongly related with communication between them. Event-based communication is by definition highly decoupled. The service emitting particular event does not need to know what other services will react to it and how exactly would they do that. As it is shown on Figure 2.3-2 new subscribers can be added and the publisher does not have the need to know about that happening. The less direct knowledge one service needs to have about other service's inner workings – the looser they are coupled.

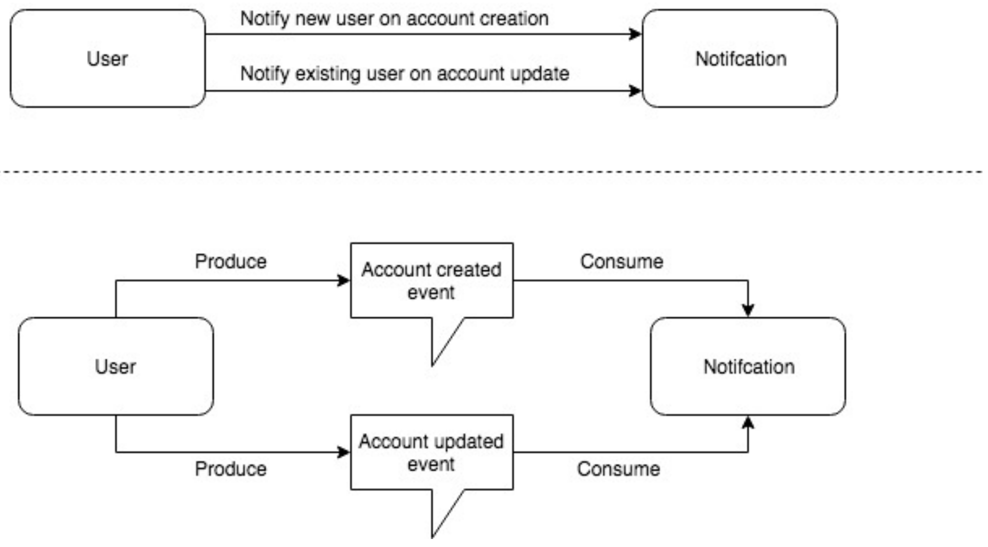


Figure 2.3-2 Handling user-related notifications via request/response calls (up) and events (down); Source: own elaboration

While concept of loose coupling refers to the whole ecosystem of microservices and relations between them, high cohesion on the other hand is what should define each service internally. Code responsible for one feature should be kept together and have a precisely defined purpose. Building a microservice within boundaries defined by previously identified bounded context makes maintaining a strong cohesion within it easier as the boundaries themselves are cohesive.

High cohesion is in some way imposed by loose coupling. When services must know as little as possible about other services' internals they need to embed all the information regarding their domain so that there is no demand for excessive communication and complicated adjustments of service-to-service collaboration.

2.3. Advantages, disadvantages and challenges of microservices

Although microservices come with many benefits and tend to become a concept widely used across different business areas they come – as every solution – with their own advantages and disadvantages and not in every case they will provide the best results. They also bring new challenges and require considering obstacles that are not relevant in case of monolithic applications.

2.3.1 Advantages

Some of the conveniences and improvements that microservice-based architecture brings were already mentioned in previous subsections but there is more that is worth listing. Advantages of microservices, similarly to their definition, differ from source to source but there are some common points that seem universal:

- Scalability
    - There is no need to scale whole system when only one service experiences performance issues
    - Horizontal scaling (adding new servers with additional instances of particular service deployed) can be used instead of vertical scaling (adding more power to already used machines)
    - Costs are reduced as scaling each service separately enables better use of available machines.
- Ease of deployment
    - Each microservice can have its deployment scheduled and performed separately
    - Small change does not implicate the need to re-deploy whole system, only the service that is influenced by given change
- Resilience and failure isolation
    - Availability of the application is a sum of available microservices, one service failing does not implicate whole system unavailability
    - There are tools designed for deferring service failure from cascading to other services (for example health-checks and circuit-breakers)
- Technology and innovation heterogeneity
    - Each service can be built with use of different technologies and frameworks
    - It is easier to implement new, even experimental techniques, as they affect only a small separated part of whole system
- Maintainability and organizational alignment

  - One team works on one service what makes the responsibility for particular component clearly defined and results in easier process of maintenance

2.3.2 Downsides and challenges

Besides the topic of data and communication – which will be described in detail in the later chapter – there are multiple other challenges that come with microservice architecture. Even though the popularity of this way of building systems may create an impression of a perfect solution, microservices come with their own difficulties.

First of all it is important to identify whether the microservice based architecture is the right choice for particular business case. Defining well divided bounded contexts is a process that requires great understanding of a domain and in situations when architects do not have a sufficient knowledge from this particular area it may be better to start with monolithic system and – as the expertise progresses – consider splitting it into microservices only later on.[3]

Running multiple services instead of one, centralized application, requires particular care when it comes to all of the operations regarding the system, that includes (but is not limited to) :

- Monitoring
- Testing
- Log tracing
- Versioning

Another thing worth mentioning is the effect of scale. Microservices work well because of aforementioned advantages that make them stand out in complex environments. However, when all that is needed within the system can be captured in relatively small monolithic application it may be the better choice. In general microservices architecture is complex and it requires careful planning and multiple operations well prepared in the background – like automation tools, security etc. In this architecture it is not enough for each separate service to work properly – they all need to coexist and cooperate in effective and efficient way. Handling it well requires time and knowledge and is an investment that not always will return.

3. Communication and data sharing between microservices

Previous chapter presented introduction to the topic of microservices and drew overview of what they are and what key concepts stand behind microservice-based architecture. As it was presented on Figure 2.3-1 the logic of an application as a whole is separated to independent components what brings new problems that needs to be handled. Among them there are two resulting directly from that separation - communication between services and sharing data. Following chapter will discuss those two concepts.

3.1. Synchronous communication

There is no single answer which method of communication is the best suited for microservice architecture as it depends on the specific business aspects. It is possible to choose several different mechanisms of communication depending on each service nature. However, choosing the communication style must begin with distinguishing two types of possible information exchange: synchronous and asynchronous. Following subsection focuses on synchronous communication and its most common implementations whereas the next one will elaborate on asynchronous way of exchanging information.

To show the overview it is possible to say that synchronous communication simply means that one service communicates with another by sending a request and awaiting for the response. This means that process of communication is blocking – calling service requires confirmation from the server within specified period of time, that the requested action or command was received and processed. Until that response comes the calling thread becomes blocked.

Synchronous communication is in majority of cases the most intuitive one. Something is requested and it is either successfully completed or failed with the known result. At the same time it has its disadvantages, one of which can be strong coupling. Requesting side must be aware of the presence of the service it communicates with, it must know the appropriate URI to call for resources and it cannot complete its action without obtaining the response or reaching the timeout.

3.1.1 REST

The most common approach to implement synchronous communication is by using HTTP protocol, usually via REST – Representational State Transfer – architectural style proposed in 2000 by Roy Thomas Fielding [5]. REST defines a set of rules and guidelines that should be followed while designing web services. It specifies six architectural constraints (five required and one optional) that must be obeyed in order for a web service to be called RESTful. Those constraints could be briefly described as follows:

1. Client-server
    - User interface interests should be separated from servers concerns.
    - Client and server can be developed independently of each other and the only thing that must stay consistent is the interface between them.
2. Statelessness
    - Communication between client and server must remain stateless.
    - Each request should be independent of previous requests and contain all of the information needed for its processing.
3. Cache
    - Responses from server should be labeled as cacheable or not-cacheable to allow clients for using cache when it is possible and avoid it when it could lead to getting invalid data.
    - Caching can be an improvement in performance by eliminating redundant interactions. On the other hand it requires verification that cached data is still representing the same information that would be obtained directly from a server.
4. Uniform interface
    - Interface between components should be uniform.
    - That translates to complying with another four constraints: "*identification of resources, manipulation of resources through representations, self-descriptive messages and hypermedia as the engine of application state"[5]*
5. Layered system
    - System can consist of multiple layers from which each layer has knowledge only about the ones that it directly interacts with.
    - Gateway and proxy components can be added and it remains transparent to the client whether it is connected with intermediary server or just the outer layer.
6. Code on demand (optional)
    - Implemented functionality can be extended by clients by allowing them for downloading executable code (such as scripts or applets) and executing it locally.

While REST is just a set of rules and a paradigm without specific implementation, creating a REST API is about following aforementioned standards. In practice often only some of the constraints are followed and REST can be realized throughout different styles. One example of distinguishing those different styles and techniques was described by Leonard Richardson in his Restful Maturity Model [6] where he characterized four levels of REST implementations maturity. Starting from level zero, where a single URI is accessed by only one HTTP method, through level one – where many URIs are accessible but still with one HTTP method – and level two –where many URIs support different HTTP methods – and finally ending and level three where HATEOAS (Hypertext As The Engine Of Application State) is introduced.

3.1.2 Google Remote Procedure Calls

Even though synchronous communication is usually identified with REST there are other ways to implement it. A good example of an alternative is gRPC - Google Remote Procedure Calls – an open source system enabling client applications to call remotely methods on the server [7]. It requires the application server to run gRPC server and on client side it uses client stubs providing the same methods that server provides.

It allows for clients and servers to run in different environments and to be written in different languages (among those that are supported) and by default it uses Google's *protocol buffers* (language and platform neutral mechanism for structured data serialization). There are four types of methods that can be defined within gRPC:

1. Unary RPC
2. Server streaming RPC
3. Client streaming RPC
4. Bidirectional streaming RPC

As the naming suggests those methods differ depending on whether single request and response are included (hence unary RPC) or either request, response or both are send as a sequence of messages. Client can send a stream of requests followed by single response from server and vice versa – server can produce a stream of messages as a response to single client request. In case of bidirectional methods those approaches combine allowing for both client and server to exchange sequences of requests and responses.

3.2. Asynchronous communication

Contrary to the synchronous communication the asynchronous one does not involve awaiting for the response. It does however require some kind of a message broker between the services that acts as an intermediary responsible for making the access to one service's message available for other services. There are two approaches to asynchronous communication that vary depending on the way this access is enabled: one-to-one service communication or publish-subscribe pattern. Figure 3.2-1 presents simplified concept of both those approaches.

One-to-one communication (also referred to as point-to-point) could be compared to sending a letter – one service addresses the message directly to the recipient. It does not await for the response, it just produces the message and the message broker delivers it to its destination. Publish-subscribe pattern is more like posting an advertisement. Producer publishes to information and does not specify to whom it is destined. Then whichever service is interested in this type of message can consume it from the message broker and react to it in its own way.

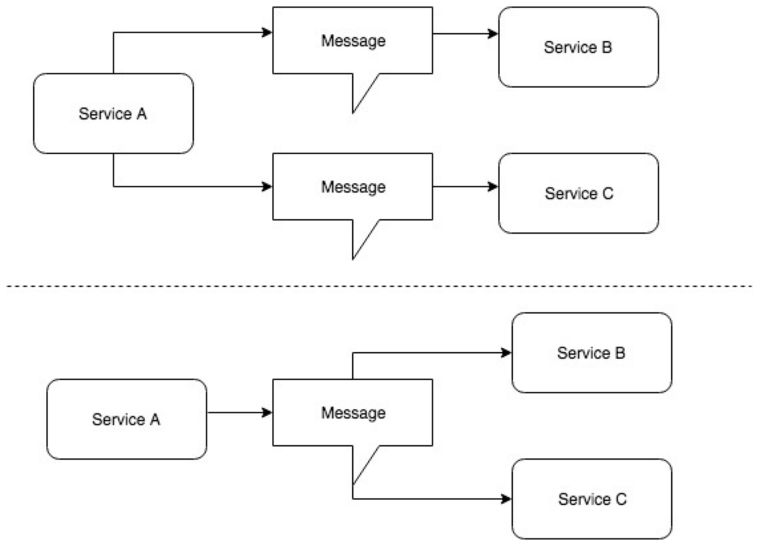


Figure 3.2-1 Simple representation of one-to-one (up) and publish-subscribe (down) communication
Source: own elaboration

In case of asynchronous communication there is no disadvantage of strong coupling – services do not need to have knowledge about each other as long as they can connect to the broker. They also do not need to be running at the particular moment the message is sent as they can process it once they become available.

3.2.1 Apache ActiveMQ

When it comes to asynchronous communication choosing an appropriate message broker is a significant step. There are multiple options to choose from but Apache Kafka[8], ActiveMQ[9] and RabbitMQ[10] are nowadays the most popular choices. As Apache Kafka plays crucial role in this paper, its detailed description is presented in next chapter. Therefore, ActiveMQ and RabbitMQ will be described briefly in this and following subsection.

Apache ActiveMQ is an open source message broker written in Java in 2004, what makes this one of first open source message brokers. It uses the JMS (Java Message Service) specification and consists of broker (centralized and responsible for message distributions) and clients (exchanging messages using the broker as intermediary). Clients consists of the application code, JMS API providing interfaces to interact with broker and the system's client library with API implementation to communicate with broker through wire protocol.[11]

Producing a message in ActiveMQ is done by using `MessageProducer` interface from JMS API. Client passes created message as an argument to the interface's *send* method, together with the name of destination queue. Message, after being marshalled by client library, gets sent to the broker which unmarshalls it and – if the default setting of persistent delivery mode was not changed – passes the object to persistence adapter which writes it to storage. Once this process is successfully finished acknowledgment is sent from persistence adapter to the broker and then by the broker to the client.[11] It is worth noticing that persistent messages which are sent to the broker outside of a transaction are by default synchronous, which means that the thread calling `send` method remains blocked until it receives acknowledgement from the broker or a `JMSException` is thrown. This could be changed to achieve better performance, however the eventual loss of data may occur. In all other cases asynchronous mode is the default.[9]

There are two ways to consume messages in ActiveMQ – either by calling explicitly the `receive` method on `MessageConsumer` or by setting up the `MessageListener` to consume messages in the moment they arrive. Broker is responsible for dispatching messages to the consumers and keeping track of which messages were already passed and to which consumers they went to. After reaching the consumer message goes to the *prefetch buffer* – area in the memory that plays the role of internal queue to which broker can pass multiple messages and from which they are processed by application logic one after another. After the processing is finished client acknowledges consuming the message and broker removes the consumed message.[11]

In case of more than one consumer being subscribed to particular queue the broker dispatches messages among those clients that have available space within their prefetch buffers following FIFO (first in, first out) order. Unless specified otherwise all customers are included within the dispatching, this can be however altered either by specifying the exclusive consumer for particular queue or grouping messages in message groups where each group is consumed by single consumer.[9]

3.2.2 RabbitMQ

RabbitMQ is a message broker written in Erlang and developed by Pivotal. As a default messaging protocol it uses AMQP 0-9-1 (Advanced Message Queuing Protocol) which specifies the way messages should be dispatched, delivered and queued in a dependable way. Initially it was created as an implementation of aforementioned protocol, it does however provide support for multiple other protocols, including STOMP (Simple Text Oriented Messaging Protocol), MQTT (MQ Telemetry

Transport) and AMQP 1.0. (which is a significantly different protocol than the default AMQP 0-9-1)[12].

Terminology in RabbitMQ, similarly to the AMQP itself, is built mostly around *exchanges*, *queues* and *bindings*. Exchanges are server endpoints identified by unique key, meant to distribute the messages. Those messages are distributed across queues (also identified by unique key) accordingly to the bindings – rules defining in what way the exchanges are supposed to perform that distribution between queues. Following the very accurate analogy from the RabbitMQ website: queue can be compared to destination in specified city, exchange – to an airport - and bindings - to routes leading from that airport to the destination (wherein there can be zero or many such routes)[13].

There are four exchange types provided:

1. Direct exchange
    - Messages are delivered to queues according to the provided routing key.
    - When a message with routing key "A" arrives at the exchange of type direct, then the exchange is responsible for routing the message to the appropriate queue – that is to the queue which has bound to this exchange with a key equal to "A".
    - This type of exchanges is especially useful for distributing messages among multiple instances of the same service.
2. Fanout exchange
    - Unlike the direct exchanges the fanout ones do not use routing key.
    - In this case upcoming message's copy is routed to each queue that has been bound with this exchange which makes it suitable for broadcast-like use cases.
3. Topic exchange
    - This exchange is in a way an extended version of direct exchange.
    - In this case queue is bound to the exchange using a pattern of dot-separated words and possibly wildcards like asterisk or hash. Routing key of the incoming message is compared to those patterns and routed to the queues with patterns matching the routing key.
4. Headers exchange
    - Works similarly to the direct exchange but headers are used instead of routing keys.
    - Depending on the headers of the incoming message it is routed by this exchange to those queues whose binding value was equal to the header's value.

Similarly to ActiveMQ, RabbitMQ is also designed to use smart brokers and dumb consumers. It is the broker that keeps information about current consumer state and “forgets” about the message once it is consumed.

3.3. Data in distributed systems

Having mentioned variety of options available to implement in terms of communication in microservice architecture it is possible to say that the choice itself is one of the challenges. After all, microservices are all about communication – instead of one service handling all the information it needs internally, we have are a group that needs to work together efficiently and this cannot be achieved with miscommunication issues. On the other hand - one of the advantages of microservices is the ability for separate teams to work independently on their own components and this has its reflection in architecture itself – we want each service to be as autonomous as possible. Combining the need for transparency in communication and ownership in data is what makes handling information in distributed systems one of its biggest hassles.

3.3.1 Shared database

Adjusting the communication type to use case is a challenge not only because of diverse choice but also due to the fact that this choice is tightly connected with underlying data and the way it is stored and shared. Since the communication between services is in most cases the result of one service needing access to information stored within the boundaries of a different service it could seem that sharing the database between the microservices would resolve the problem – at least partially. This concept of shared database (also referred to as *Integration Database* [14]) is a solution that – even though by some may be treated as a reasonable choice – is mostly perceived as an antipattern and can lead to multiple difficulties covering the benefits of microservice architecture. Also the fact of being bound to use one database itself is a difficulty, since not necessarily the same database will be best suited for all the involved services.

As it was discussed in chapter 2, the concept of bounded context and loose coupling are what makes the convenience of easier development, deployment and scalability possible. Allowing services to manage their own data is a significant part of keeping them loosely coupled. There will be however, no matter how well separated the bounded contexts are, situations when one service will require information from outside its boundaries. It is not hard to imagine that some sort of order service may require product details to show order summary. Should it be able to access product service’s data by itself or should it request that data from product service? Even though in many cases it may seem easier to allow all the services access the whole database, such approach comes with many drawbacks. One example could be product service being unable to access the data from its own context due to the transaction lock held by order service accessing that data at the moment. In general sharing the same database between services makes it hard to predict whether one service's behavior is going to affect in any way some other services.

There are ways to mitigate the risks of shared database, like assigning database tables to services in a way that each service has exclusive rights to particular tables. As long as those rights include both read and write privileges it may be considered a simplified approach to keeping database per service. In this approach each service would still need to ask for data that is not within its boundaries - just like in case with whole databases being separate – it may however, have some disadvantages resulting from the fact that we still have one database. All services still would be constrained to using the same database. Also, it is worth mentioning that some tools used within application development may require they own tables in database and in this approach such tables would often have to be shared between services. One example case be Hibernate Envers[15] creating REVINFO table for storing revision ids and timestamps, which under such circumstance would be shared across all services. This is just an example and probably majority of such cases could be configured. It is however, worth calculating the overheads, whether keeping all data in one database, even though services still have direct access only to their own data, is still more convenient than separating their databases.

3.3.2 Database per service

Contrary to the shared database there is a pattern of database per service (also referred to as *Application Database* [16]) which aligns well with the concept of loose coupling. Data management is decentralized allowing each service to be the only owner of its data. In such approach there is no requirement to use the same database in all cases and one service's database can be chosen specifically with its needs in mind. Schema for this kind of database would usually be simpler that in case of shared one due to the fact that it does not need to combine data for use in different contexts.

Database per service implies that exchange of data between services is available only via their APIs. This leads to complexity and requires a significant rise in the amount of requests or messages between services in comparison to the shared database approach. In practice it may lead to some service being just a proxy between other services and some particular database. There also is an issue of data intensive applications. In the aforementioned order and product example order service would have to ask product service for product details each time the order summary would be requested. Alternatively it could keep a local copy of products embedded into order summary but applying this approach in all possibly problematic cases would lead to data duplication and, similarly to the concept of shared database, interfere with services' boundaries. It could also "prepare" the data by caching it but that on the other hand brings the problem of keeping the cache up to date.

3.3.3 Streams of events as a shared data source

Having in mind the fact that sharing data brings convenience while on the other hand keeping data sources separate ensures loose coupling it may appear that when it comes to microservices and data we are dealing with special kind of *data dichotomy* [17]. Understanding the importance of shared data an focusing on its role within microservice ecosystem is crucial. Data that services share is the *data on the outside*[18] – something that is not held within any service directly, something that is immutable and can be treated as a point of reference. Concept of messaging enables to manipulate the data accordingly to the needs of the services that receive the message as the receiver can freely work with data it consumed. As events stay immutable and purely represent facts there are no such downsides of sharing them between services like there were in case of shared database approach.

As it was already mentioned in chapter two – event based communication is highly decoupled by design and this fact can be leveraged to the way we treat data in microservice architecture. Implementing event-driven approach not only impacts a way we must perceive the communication but also, how all data within our application should be treated. Events themselves could become the shared dataset.

This leads to the notion of *Event Sourcing [19]* which emphasizes that events, not database records, should be treated as a source of truth. According to that pattern every change of state has its reflection in an event that is immutable and stored in accordance with the order in which it appeared. It allows for keeping the whole history of actions that took place and replaying the events from the beginning to recreate the application state whenever it is needed. Figure 3.3-1 presents a simplified visualization of this concept – events with information about quantity updates are stored sequentially within a log that can be used to calculate the current state. As they are immutable and communicate about facts in case of any failure in processing the events they can be replayed and the resulting state will be recreated.

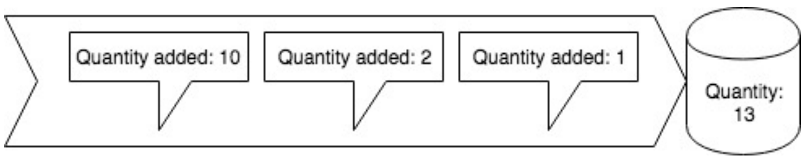


Figure 3.3-1 Simple representation of event stream and calculated state; Source: own elaboration

To share the immutable events and keep the state locally, there is a need for a messaging system that will move the data but also for a storage layer that will keep history of events. With those elements combined the stream of events can become a centralized dataset. That brings up the concept of already briefly mentioned `database inside-out`.

3.3.4 Database inside out

The concept of database inside-out was coined by Martin Kleppmann[1] and it was the main inspiration for the following thesis. He noticed that databases in today's architecture often play role of a *`gigantic, global, shared, mutable state`[1]* and elaborated on the idea of removing the databases as we know them from systems. The whole approach – and hence the name – was based on an observation that individual components of databases could be used separately instead of being wrapped inside a single place. Four aspects of database were included.

The first aspect of database that was discussed is *replication*. Kleppman emphasized that in order to keep all replicas consistent one of the mechanisms used behind the curtains within a database is maintaining a *logical log.* For every query the leader acknowledges (where leader means one node that writes are sent to) it notes this query's effects into this logical log for the other replicas to apply. Those effects are facts and unlike the query itself that is not idempotent, facts appended to logical log by the leader are simply descriptions of what took place and when did it happen. This element, presented on Figure 3.3-2, is crucial to the whole concept, what will be described in further part of this chapter.

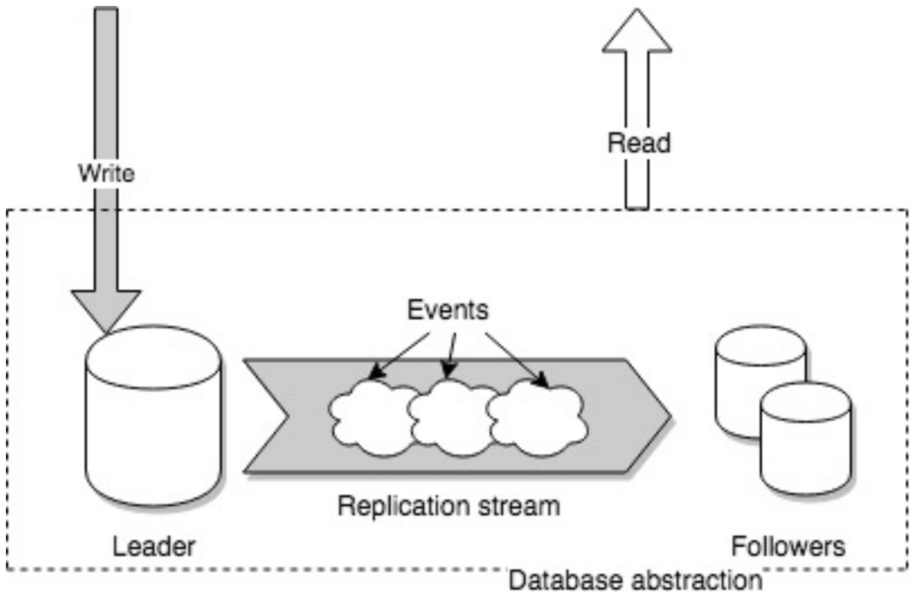


Figure 3.3-2 Simple representation of replication log; Source: own elaboration based on [1]

Another element is *secondary indexing*. Indexes are basically just the representations of data that is already stored in our database, only structured in a way that makes certain queries faster. They do not modify the underlying data and neither does deleting the index delete the data from database. Every time we tell the database to create an index it goes through all of the records and finally creates a new structure which references the initial data ordered in a way that enables efficient search by the particular field. Such structure is updated automatically whenever underlying data changes and therefore is easy to keep accurate.

Third thing mentioned is *caching* which, unlike the previously mentioned indexes, is difficult to manage and maintain. To avoid requesting the data directly from the database each time, caching layer is introduced to act as a first line of information. Only when requested data it is not available in cache then the direct call to the database is needed. There are numerous problems that can arise within that field, like knowing which cached data should be updated after some records in underlying database change or handling the state after server restart, when all cache is empty and each request is suddenly directly calling the database. Kleppmann noticed however that in a way cache and indexing have some common ground - they are both some sort of transformation of data from the database that results in a new structure which can be entirely recreated basing on the initial data from the underlying data source. Main difference lays in updating mechanisms which in case of indexes is handled directly by the database and for cache – must be taken care of by application layer.

Fourth and the last element are *materialized views*. Standard views in relational databases are just a layer of abstraction over the initial data, created by predefined query. The query that was used to create a view is executed each time the view itself gets queried. There is no possibility to write data to the view as it does not create a new table, only shows the original data in a "pre-queried" way. Materialized view, while still cannot be directly written into, presents a significantly different approach. Query passed for materialized view creation is executed once instead of being used each time the view is queried. Dataset being a result of such query execution is copied from underlying tables and written to disk. Later on database takes care of updating this table-like dataset and takes responsibility for it being accurate. This is in a way similar to the beforementioned caching but it is the database, not the application, that has to handle its invalidation and maintenance.

Kleppmann's general thought was that while replication and secondary indexing are mature and well-working mechanisms, cache is troublesome to work with and materialized views are good as a concept but could be improved when it comes to the implementation. What he proposed was to change the way we perceive the replication stream – an element that is internal to a database and is usually considered just an implementation detail – and to treat it as a core of data architecture. Having that stream of events as a first-class citizen we could then write directly to it and build materialized views from those writes. In this way materialized views would work similarly to the secondary index since data they hold would be structured in a way that is optimized for fast reads. They would also in a way play a role of continuously updated cache that holds some part of initial data and can be rebuilt from the underlying event stream. Figure 3.3-3 presents the concept of this decomposition of particular database aspects. As Kleppman says *'You can think of this as "unbundling" the database. All the stuff that was previously packed into a single monolithic software package is being broken out into modular components that can be composed in flexible ways.'*[1]

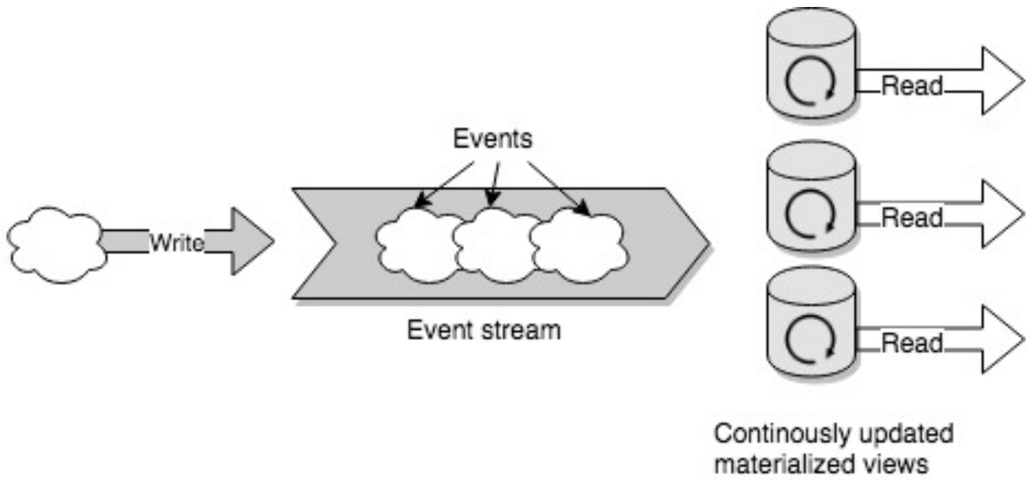


Figure 3.3-3 Concept of database inside out; Source: own elaboration based on [1]

Materialized views in presented concept could be embedded databases that would accept read requests. Any writes however would not be handled by those views but instead – they would be appended to the stream of events and then transformed into materialized views. Such approach gives many advantages over the traditional application-maintained cache. Created materialized views contain all the data that may be requested – there is no need to query the underlying source of data like it would happen in case of a cache miss. There is also an easy way to extend the set of materialized views – at all times there is an option to reprocess all events from the log from the beginning and transform them into new view.

Among many advantages of implementing such pattern Kleppmann lists better quality of data, meaning:

- complete history is maintained;
- separated responsibility of reads and writes;
- ease of creating new views;
- possibility to quickly recover after eventual errors, since the source data is never overridden;

In his example author proposes Apache Kafka as an implementation of the log. Since the original speech presenting this concept took place in 2014, Kafka Streams - chosen within this paper as to handle the part of materialized views - was not yet available, he suggested Apache Samza as a framework used to implement materialized views. However, Apache Kafka together with Kafka Streams library have been used already with this approach in mind, for example by Ben Stopford [17]. How exactly those two tools can be applied to realize described convention will be shown in next chapter.

4. Apache Kafka and Kafka Streams library

In third chapter two popular choices for message brokers – Apache ActiveMQ and RabbitMQ were shortly described. This chapter provides a detailed overview on third, widely chosen solution – Apache Kafka and discusses what are the main differences between Apache Kafka and previously mentioned message brokers. It also contains an introduction to Kafka Streams – a library used for stream processing – and presents how those two tools together can be used to implement the concept of database inside-out.

4.1. Apache Kafka

Apache Kafka was originally developed at LinkedIn and it has been open-sourced in 2011[20] . It is written in Java and Scala and, unlike previously described solutions, is not referred to as a message broker but rather a *distributed streaming platform*[8]. While it can be used as "traditional" messaging broker, it was designed with purpose of overcoming the constraints of already available message brokers.

Terminology in Apache Kafka is based mainly around four components described in further subsections: topics (including partitions), brokers, consumers (including consumer groups) and producers.

4.1.1 Topics

Topics, in case of Kafka, are simply streams of data. Each topic is identified by unique name and can be divided into one or more partitions. Within each partition ordering of records is maintained and unique id (called offset) is assigned to each message. Figure 4.1-1 shows how messages are published to topics. It is worth mentioning that since the ordering is kept only within a partition (not between partitions) to uniquely identify a message we must provide topic name and both – number of partition and offset. Data that was once published to the topic cannot be changed, new records can only be appended.

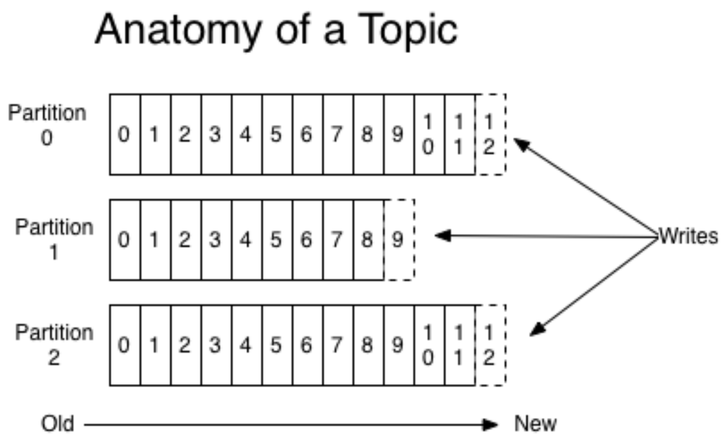


Figure 4.1-1 Anatomy of Kafka topic; Source: [8]

Since the order is maintained only within each partition there is a special care needed to assure that in cases when processing in particular order is required messages would be assigned to the same partition. Messages in Kafka by default are divided between partitions according to their keys – all messages with the same key end up in the same partition and therefore are guaranteed to be processed in the same order they were created. Selecting number of partitions for a topic plays important role in context of scalability, what will be described in section about consumers.

There is no distinction between publish-subscribe and one-to-one communication – topics are combining those concepts. Each topic can have zero, one or multiple subscribers and by using consumer groups (described in further subsection) data from one topic can be handled by multiple instances of the same service. This is what makes Kafka topics a hybrid between those two destination types.

What is also significantly different in Kafka, comparing for instance to ActiveMQ or RabbitMQ, is the retention. While typically broker marks consumed messages for deletion, in case of Kafka all the data within topics is stored – no matter if the messages were consumed or not. Period for which this data is retained can be configured and only messages older than the configured period are deleted. Alternatively size limit can be set – that data is kept until the topic size in bytes exceeds specified amount. There is also a possibility to specify that a particular topic should be compacted which means that for each key only the most recent message will be kept.

4.1.2 Brokers and replication

Each Kafka cluster contains one or more brokers and each one of that brokers holds some partitions of existing Kafka topics. Different partitions of a topic are automatically assigned across different brokers so the data is distributed. In order to ensure fault-tolerance those partitions – that are already distributed across different brokers - can be replicated so that more than one broker has the information about particular partition. Number of required replicas can be configured for each topic individually by defining topic's replication factor. Figure 4.1-2 shows an example of how topic with three partitions and replication factor of two could be distributed across three brokers.

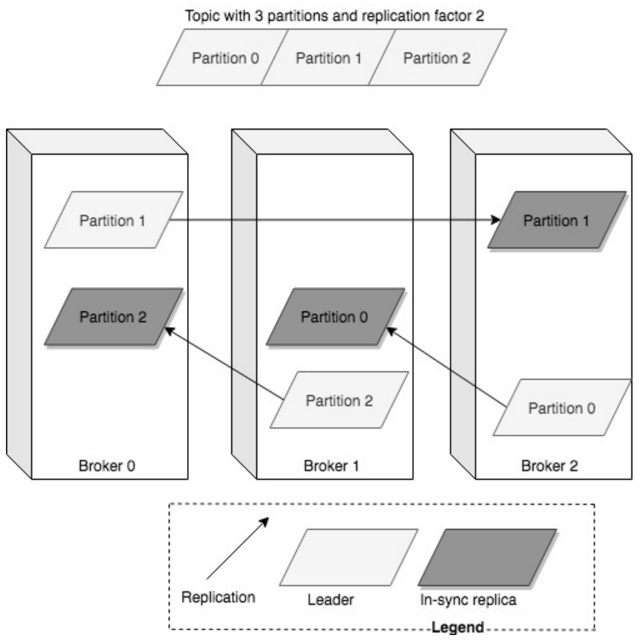


Figure 4.1-2 Distribution of partitions across brokers; Source: own elaboration

As it is also visible on Figure 4.1-2 there is a difference between replicas of each partition – one of them is a so-called leader whereas all other are followers, referred to as *in-sync replicas*. At one point of time only one broker can be selected a leader for certain partition of a topic. Similarly to the concept already described in chapter 3, only the leader is responsible for receiving the data and making it accessible to consumers. Remaining replicas are synchronizing the data with leader (hence the name – in-sync replicas). In case of a current leader being unavailable one of the in-sync replicas is selected for a new leader. Responsibility for selecting leaders lays on Apache Zookeeper (which is required for Kafka to work). Zookeeper itself is used not only for leader election but also for configuration of topics and managing Kafka brokers in general.

4.1.3 Producer and consumer

Producers in Kafka are responsible for sending data to topics. If the destination partition is not specified directly within a message, they select appropriate partition for each message – by default if there is no key messages are assigned to partitions in round-robin fashion and otherwise each message with the same key goes to the same partition. This behavior could be adjusted if needed by implementing the `Partitioner` interface of Kafka's Producer API.

Consumers, what is significantly different in comparison to traditional message brokers, not only consume the messages from topics but also keep track of their position by controlling the offset. That is not the broker's responsibility to know which messages were consumed by what consumers (in older versions of Kafka it was, however, within Zookeeper's tasks). Instead consumers store information about what messages they have already consumed in Kafka's internal topic called "*__consumer_offsets*". Each consumer can be subscribed to any number of topics and for each of them there is an information about current offset per partition stored within aforementioned internal topic.

Consumers do not handle their progress of consumption individually. Instead, they are divided into groups called *consumer groups* inside which they consume messages together, meaning that a given message on a topic is processed only once within a certain consumer group. Every consumer group can have arbitrary number of consumers. In microservice architecture consumer groups are a perfect fit for multiple instances of the same service – each service is a separate consumer but it cooperates with other instances being a part of the same consumer group. Each consumer has one or more partitions of a topic assigned to it, which means that having more consumers within a group than partitions within a topic will make extra consumers idle. In case of any instance (hence – consumer) being down, its partition is automatically assigned to another, available consumer from the group. Figure 4.1-3 shows an example of one topic with three partitions being consumed by two consumer groups. In consumer group "B" one of the consumers remains idle but it would take over in case of any other consumers in its group goes down. In case of consumer group where number of consumers is lower than number of partitions of consumed topic one of the consumers would handle more than one partition.

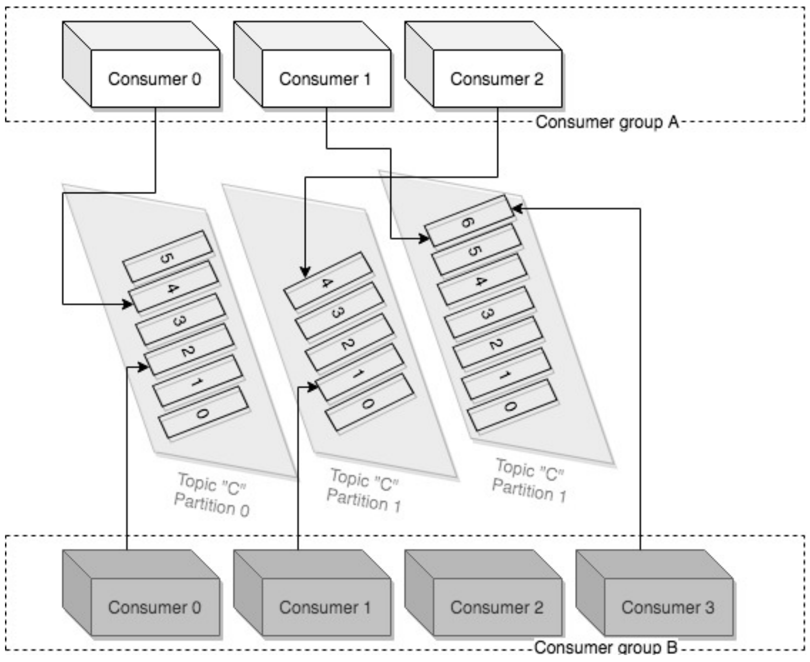


Figure 4.1-3 Consumer groups and topic partitions in Kafka; Source: own elaboration

4.2. Kafka Streams

Kafka Streams is a library created for processing streams of data from Kafka and it was released in 2016. It allows for processing each record from a particular Kafka topic, transforming one message at the time and sending changed data to another Kafka topic. It supports both stateless and stateful operations due to the concept of *state stores* that "*can be used by stream processing applications to store and query data"[21]*

Under the hood Kafka Stream library uses described previously consumers and producers but offers a lot more when it comes to functionalities, including the ease of enabling *exactly once* processing. Exactly once means that each message is processed by our application one and only one time, even if any failure occurs in the middle of the process. In contrary to *at least once* and *at most once* guarantees in which message that was failed to deliver can, accordingly, be reprocessed or not processed at all. In case of Kafka Streams the exactly once guarantee is offered simply by setting property called `processing.guarantee` for the application to `exactly_once`[22].

Two of the most important things within Kafka Streams are streams and tables and the abstractions provided by the library over those two concepts – KStream and KTable. KStreams are simply continuous data flows, infinite sequences of immutable records whereas KTables, similarly to database tables, represent the current state for given key. As the documentation for Kafka Streams emphasizes – there is a duality between those two concepts and *this duality means that a stream can be viewed as a table, and a table can be viewed as a stream[21].* To put it in a simple way – messages from a stream can be formed into a table, where every new message with already existing key is treated as update or delete (depending on whether it has new value or the value is *null*) and message with a new key is interpreted as an insert. In a way stream can be interpreted as a changelog for a table, similarly to the replication log described in chapter 3.3.4 as it is presented on Figure 4.2-1. This means that such table is always containing accurate data since it gets continuously updated as new events get appended to the stream.

Both KStreams and KTables can be joined with each other, with join-types being similar to the "traditional" joins in databases – inner join, left join and outer join. Since KStreams are continuously updating, joins between them are always windowed (captured within specified period of time). Semantics of available joins are described in detail in Kafka Streams documentation and some examples will be presented in further parts of this paper based on created prototype.

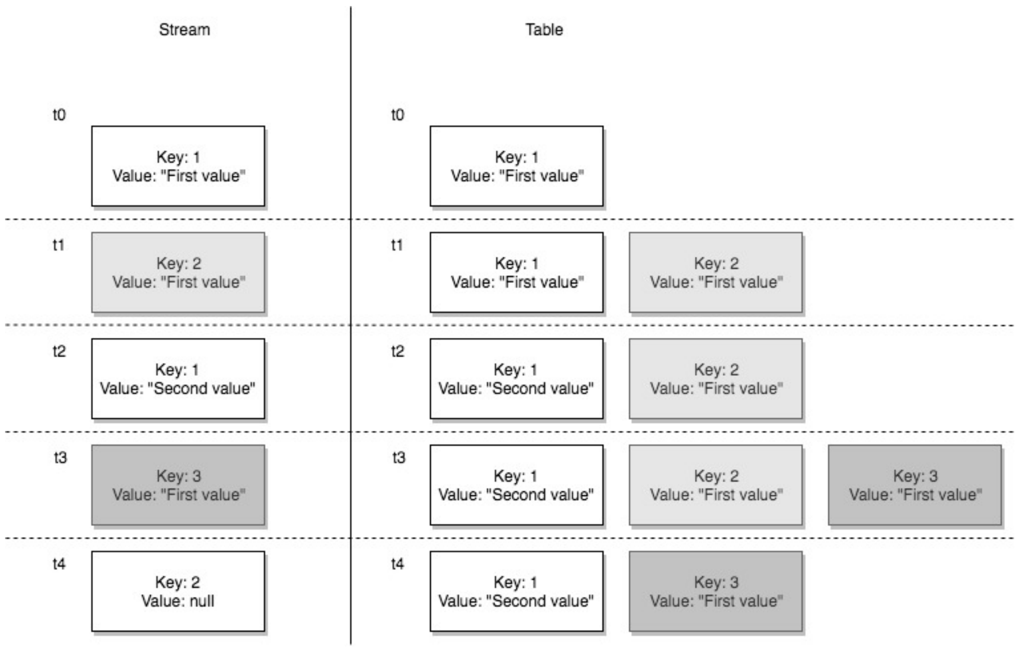


Figure 4.2-1 Duality of streams and tables in Kafka Streams; Source: own elaboration

Another concept, especially important in context of following paper, is the ability to query the state stores. Each instance of streaming application contains its own state that is a subset of whole application's state (due to the fact that data is separated into partitions consumed in parallel by different consumers). By default Kafka Streams uses RocksDB – highly performant and embeddable persistent key-value store - to store application's persistent state stores. Data stored within those state stores can be queried allowing for read-only access to that data via, for example, REST endpoints of an application. How this querying works in practice will be described based on the prototype presented in further chapters

4.3. Conclusion on selected tools

Both Apache Kafka and Kafka Streams library are very complex. Mechanisms described in this chapter, even though they cover only a part of available functionalities, should be sufficient to justify why they appear to be a good fit to implement the pattern of database inside-out introduced in chapter three. Kafka's unique ability to store data on a long-term basis in a fault-tolerant way, combined with Kafka Stream's processing and continuously updated local state stores sum up to all required components of Kleppmann's concept. Such implementation with Kafka playing the role of log and Kafka Stream's keeping materialized views available locally and optimized for read operations will be investigated in further chapters.

5. Introduction to created prototype

To examine the approach described in previous chapters and investigate how it could be implemented and what possible obstacles may appear in system following such concept, prototype application was created. Chosen domain for the application was an online shop, which was justified by the fact that – in author's opinion – online shops are usually associated with traditional databases and their concept and core functionality is well known and easy to understand. Prototype was developed under working title "NDSOS" – shortcut for Non-Database Streaming Online Shop.

5.1. Architecture overview

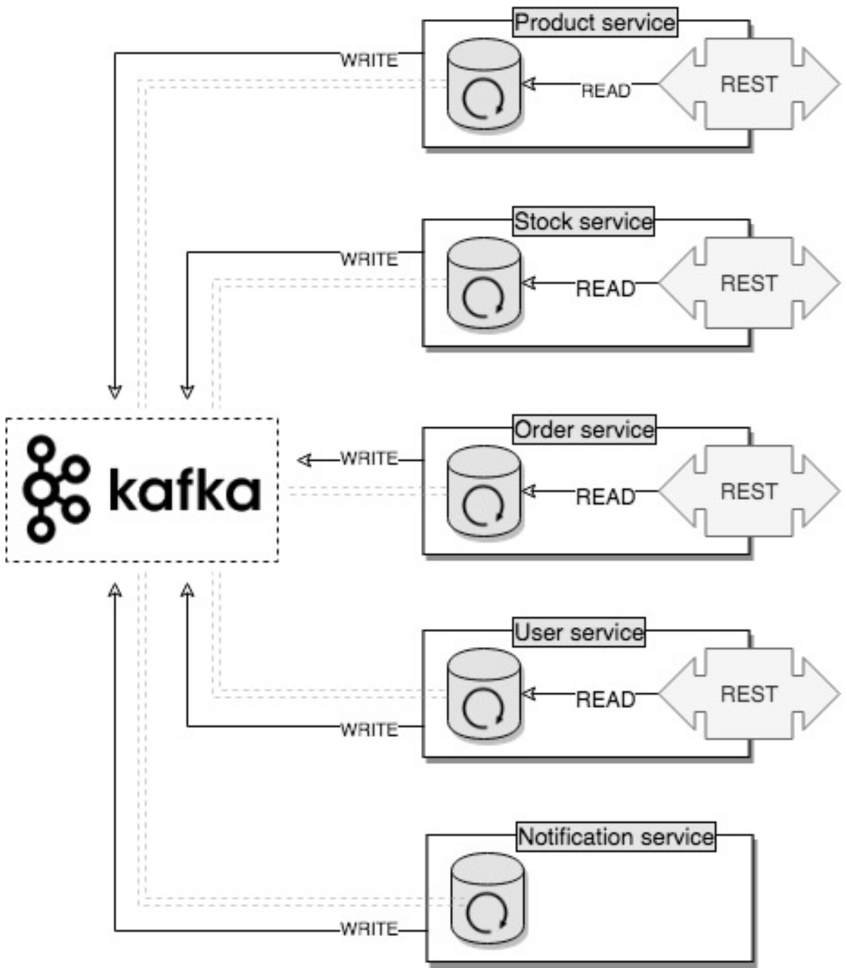


Figure 5.1-1 Simple representation of prototype's architecture; Source: own elaboration

Figure 5.1-1 shows a visualization of prototype's core elements. It is worth noticing similarity between this visualization and Figure 3.3-3 described in chapter three. System consists of five services – product, stock, user, order and notification – four of which expose REST endpoints. All read actions are handled by embedded Kafka Streams state stores and all write operations are directed to Kafka topics. There is also a sixth service, not visible on the visualization – demonstration service. It was omitted on the drawing since its only purpose is to demonstrate flow of messages in the system by reading messages from all Kafka topics and writing them to console.

Key components, beside already mentioned Apache Kafka and Kafka Streams library, are:

- Spring Boot (including the web starter)
- Spring Kafka
- Apache Avro together with Confluent Schema Registry
- Apache Maven

Each service is a separate Spring Boot application with Apache Maven used as a dependency management and build automation tool. For the ease of development all services are created as modules of shared parent project to manage dependency versions in one place. In terms of deployment they can be however independent.

5.1.1 Spring Boot and Spring Kafka

Spring Framework is an open source framework providing support for developing Java applications. It implements the concept of Dependency Injection and is divided into multiple modules that can be chosen depending on the requirements of particular application. Even though the term Spring Framework originally referred to project under the same name, today it is widely used to describe whole family of projects that were developed basing on the initial Spring Framework one.[23]

Spring Boot on the other hand is not a framework but a project aiming at making development of Spring applications easier. It allows for creating stand-alone Spring applications by providing grouped sets of dependency descriptors called starters (such as spring-boot-starter-web used within discussed prototype), simplified configuration (by bringing auto-configuration and possibility to externalize the configuration), embedded web server and other tools reducing the time and complexity required to develop Spring-based applications. In developed prototype Spring Boot in version 2.1.4.RELEASE was used together with a starter dedicated to building web applications: spring-boot-starter-web.

Spring for Apache Kafka (referred to as Spring Kafka) is one of the projects from beforementioned family of projects built on top of Spring Framework. It introduces a set of abstractions over Kafka's concepts and provides support for developing Kafka-based applications in Spring environment. Starting from version 1.1.4 this projects offers also support for Kafka Streams library. Version 2.2.5.RELEASE of Spring-Kafka was used during prototype development. As it was briefly mentioned in introduction, choice of Spring solutions was justified by the fact that they are widely used within enterprise applications.

5.1.2 Apache Avro and Confluent Schema Registry

Kafka supports variety of data types. The one being especially popular and at the same time suggested by Confluent [24] (company created by people responsible for building Apache Kafka and currently offering enterprise support for it) is Avro. Apache Avro, which was chosen for sending data from and to Kafka in developed prototype, is a data serialization system relying on schemas. It allows for defining a schema in JSON that contains definition of all fields for particular object and later on it supports evolution of such schema.

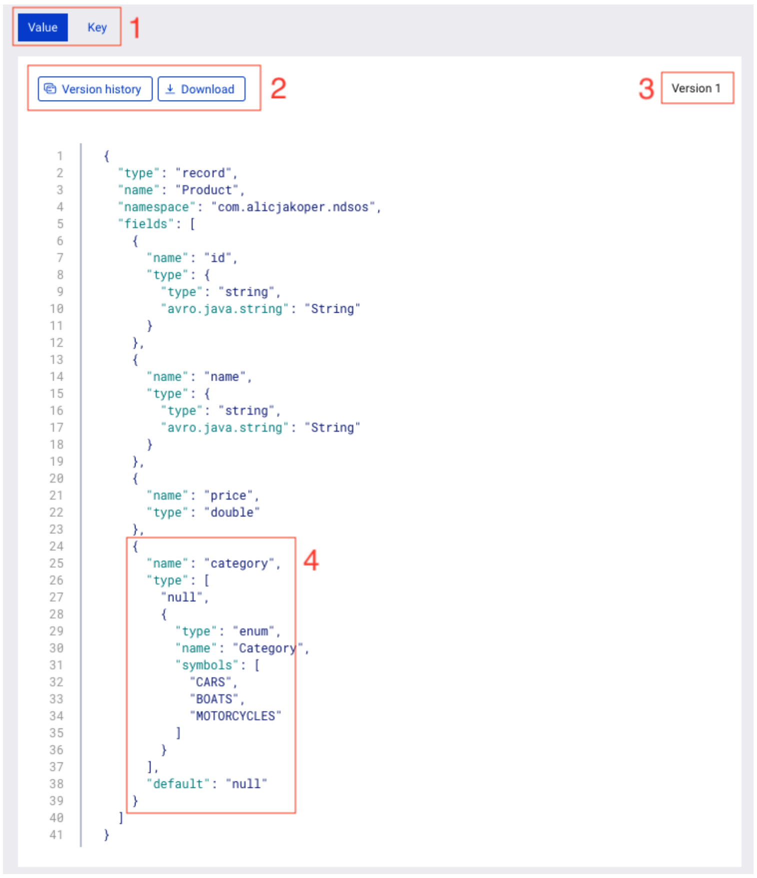
Among many advantages of Avro its speed and compact format are two that could be considered some of the most important ones when it comes to streaming systems.[24] Avro schemas can contain both primitive types as well as complex ones including for example records, enums, arrays and maps.

Listing 5.1-1 Avro schemas including complex types

```
[
{"namespace": "com.alicjakoper.ndsos",
 "type": "enum",
 "name": "Category",
 "symbols" : ["CARS", "BOATS", "MOTORCYCLES"]
},

{"namespace": "com.alicjakoper.ndsos",
 "type": "record",
 "name": "Product",
 "fields": [
     {"name": "id",     "type": "string"},
     {"name": "name",   "type": "string"},
     {"name": "price",  "type": "double"},
     {"name":"category","type":["null", "Category"], "default":"null"}
 ]
}
]
```

Listing 5.1-1 shows two of the schemas used within NDSOS. As it is visible on that listing, one definition can be used within the other – like in case of `Category` of type `enum`. Avro schemas are stored in files with `.avsc` extension and one file can either store one schema or a whole array of JSON objects, each representing separate schema (as shown on the beforementioned listing). Null values are allowed if specified, like in case of `Category` within a `Product`. Code generation for schemas specified in project was performed using dedicated Apache Maven plugin that generates Java code corresponding to each `.avsc` file available in specified directory.

Defined schemas were then stored in Confluent Schema Registry – a tool for schema management provided by Confluent. Confluent Schema Registry keeps information about schemas associated with particular Kafka topics and allocates unique identifiers for each schema that gets registered. Both Kafka consumers and producers communicate with Schema Registry to either register the schema (in case of producers) or obtain schema (in case of consumers) for particular topic. In situation when producer would try to change a schema in a way, that its new version is not compatible with previous one, the exception with message “Schema being registered is incompatible with an earlier schema” is thrown and message is not published to Kafka topic.



```
{
  "type": "record",
  "name": "Product",
  "namespace": "com.alicjakoper.ndsos",
  "fields": [
    {
      "name": "id",
      "type": {
        "type": "string",
        "avro.java.string": "String"
      }
    },
    {
      "name": "name",
      "type": {
        "type": "string",
        "avro.java.string": "String"
      }
    },
    {
      "name": "price",
      "type": "double"
    },
    {
      "name": "category",
      "type": [
        "null",
        {
          "type": "enum",
          "name": "Category",
          "symbols": [
            "CARS",
            "BOATS",
            "MOTORCYCLES"
          ]
        }
      ],
      "default": "null"
    }
  ]
}
```

Figure 5.1-2 Screenshot of Avro schema registered in Confluent Schema Registry; Source: own elaboration based on Confluent Control Center UI

Like it was mentioned in first chapter, during development free version of Confluent Platform was used. It provides multiple components used within development of streaming applications available in one place. It comes with both free as well as commercial features that are available for free as long as they are used on only one single Kafka broker. Only few components shipped with Confluent Platform were used during development process and one of them was Control Center – commercial feature providing graphical user interface for monitoring and management of Kafka-based applications. Figure 5.1-2 shows Avro schemas showed on Listing 5.1-1 that were assigned to topic `products` and registered in Confluent Schema Registry. Provided screenshot comes from Control Center panel. There are four important elements highlighted:

1. Both key and value of a message can have schema assigned

2. Older versions of particular topics are available to view and download
3. Current version of schema is displayed
4. Complex types can have schemas defined separately within .avsc files and are automatically embedded within other schemas they were used in

5.1.3 Apache Maven

Apache Maven is an open-source *"software project management and comprehension tool"* [25] that was chosen as a build automation tool for NDSOS. Maven is built based on concept of Project Object Model (POM) – xml file that stores configuration required to build a project. POMs support inheritance allowing multiple projects to be build based on the same specifications included in one parent POM. This concept was used during prototype creation since all services inherit dependency descriptors from one POM.

Previously mentioned `avro-maven-plugin` was also used allowing Maven to automatically handle code generation based on provided Avro schemas. Listing 5.1-2 shows fragment of POM that was included for each service to configure the plugin. It specifies in which phase of project's build sources should be generated, in which directory .avsc files are stored and where should the generated sources be kept. Additionally it contains specification for `stringType` set to Java's `String` since by default Avro's type string is translated to `CharSequence`.

Listing 5.1-2 Configuration of avro-maven-plugin

```
<plugins>
    <plugin>
        <groupId>org.apache.avro</groupId>
        <artifactId>avro-maven-plugin</artifactId>
        <version>${avro.version}</version>
        <executions>
            <execution>
                <phase>generate-sources</phase>
                <goals>
                    <goal>schema</goal>
                </goals>
                <configuration>

<sourceDirectory>src/main/resources/avro</sourceDirectory>

<outputDirectory>${project.build.directory}/generated-
sources</outputDirectory>
                    <stringType>String</stringType>
                </configuration>
            </execution>
        </executions>
    </plugin>
</plugins>
```

5.2. Implementation details

Previous subsection provided an overview of an architecture and choice of tools. In this subsection introduction to the implementation process is presented. Figure 5.2-1 shows what topics were used within NDSOS and which services were writing to them. Even though all services have access to data from each of the shown topics, only one service is responsible for producing messages to particular topic. Such approach was chosen to assure that there is one clearly identified service responsible for its topics, what includes maintaining schemas and handling the key choice appropriate for ordering. Within prototype such approach is not enforced with any particular security restrictions but there is a possibility to define which applications have permission to read or write certain topics.

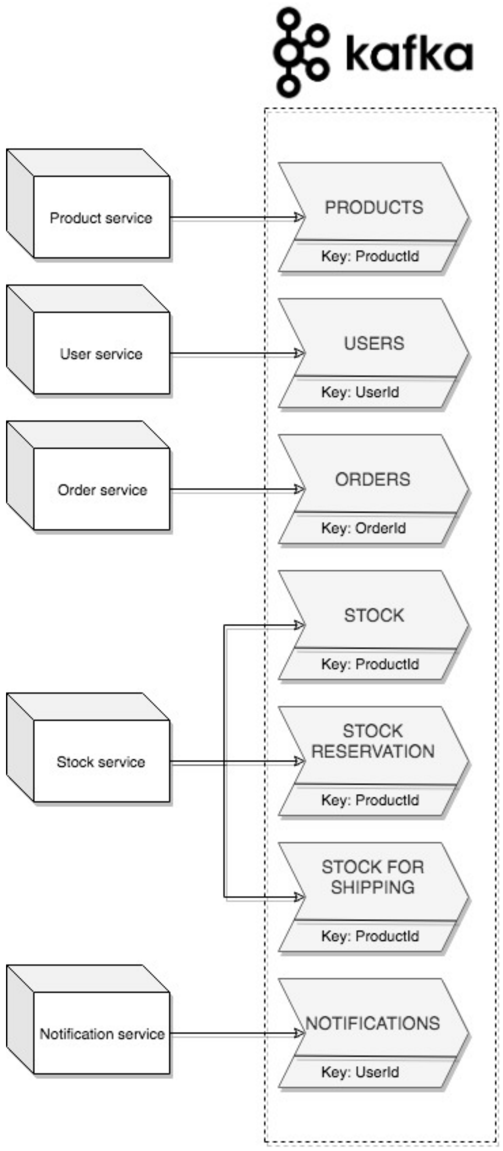


Figure 5.2-1 Kafka topics and services writing to them in prototype; Source: own elaboration

Stock service was the only one to write to more than one Kafka topic which is related with the fact that scenario of placing an order and this order's validation regarding available stock was the main use-case in presented prototype as it will be shown in next chapter. In general, there are however no restrictions on how many topics can be assigned to each service.

During development all topics were created with two partitions and replication factor of one. Number of replicas was dedicated by the fact that only one, locally running, Kafka broker was available and therefore – as it was presented on Figure 4.1-2– this enforced setting replication factor to one. Two partitions however allowed for running each service in two instances and investigating the behavior for consumer groups with multiple consumers each.

Next chapter presents details of process flows implemented within the NDSOS prototype. It is worth emphasizing that the described prototype was created with purpose of investigating what are the possible advantages, disadvantages and obstacles during development of systems following concepts that were introduced in previous chapters. Therefore, it is not designed to be used as a production system as it would require significant adjustments and improvements including for example security, gateway or load balancing. It does however present and overview on working with systems built around streaming platform and allows for conclusions regarding such approach.

6. Events and stream processing in microservice architecture

NDSOS prototype was built around simple online shop use cases in order to compare the chosen approach with traditional architecture. This chapter presents what elements were included in developed system and how they were implemented with event-driven concept in mind. It also emphasizes configurations that should be considered during development of such system.

6.1. CRUD operations

Majority of business applications, whether they are divided into microservices or not, offer among their functionalities some create, read, update and delete actions. In case of architecture where services communicate with a database, those actions are translated to SQL statements and allow users to manage records stored in the database.

6.1.1 Available endpoints

In case of presented prototype CRUD operations are all translated to events, similarly to what would have happened in database's replication log. Table 6-1 presents all exposed REST endpoints that allow for performing those operations.

Table 6-1 Description of prototype's endpoints

Method	Path	Description
POST	`/product`	Publishes new product
GET	`/product`	Retrieves all products
PUT	`/product/{productId}`	Updates product with provided `productId`
DELETE	`/product/{productId}`	Deletes product with provided `productId`
GET	`/product/{productId}`	Retrieves product with provided `productId`
POST	`/stock`	Publishes stock
GET	`/stock/{productId}`	Retrieves current stock for product with provided `productId`
POST	`/user`	Publishes new user
GET	`/user/{userId}`	Retrieves user with provided `userId`

PUT	`/user/{userId}`	Updates user with provided `userId`
DELETE	`/user/{userId}`	Deletes user with provided `userId`
POST	`/order`	Publishes new order
GET	`/order/by_user/{userId}`	Retrieves all orders for user with provided `userId`

Events representing creation and update are messages with given key and value representing – accordingly - created or updated entity. Each "updating event" looks the same as "creating event" only the message key in case of an update is allowing for identification of an entity that should be updated. As it was presented on Figure 4.2-1 Kafka Streams' KTables allow for aggregation of messages with the same key what makes the result of aggregation, available via state store, represent entity's state with all changes applied. Deletes are not in fact deleting the entity in a way that it might be expected, based on database-related experience. All delete actions are simply another events with a key of entity that should be deleted and a value of null. Such, so called tombstone records, are translated into deletes within KTables.

6.1.2 Create, update and delete operations

Spring Kafka offers `KafkaTemplate` as an intermediary for producing messages to Kafka topics. After configuring the Kafka producer it allows for specifying the default topic, to which this producer will send its messages, as well as for blocking the sending thread until the message was successfully sent. By default Kafka producers publish messages asynchronously but in some cases it may be required to know whether event publication succeeded. All of the KafkaTemplate's sending methods return an instance of `ListenableFuture` what allows for either calling `get()` and awaiting for completion or leaving the processing asynchronous by registering a callback. Considering the fact that producing messages to Kafka was triggered by synchronous REST endpoints, first approach was chosen for presented prototype.

Listing 6.1-1 presents a code fragment responsible for publishing products to Kafka. First method – `publishProductEvent` – gets called on both create and update actions. After the corresponding request gets converted into instance of `Product` class it is passed to this method and published to Kafka. Timeout is specified in configuration and can be adjusted depending on the requirements. When sending the message is succesfull published record is returned from the method, converted to instance of `ProductResponse` class and returned in response via REST API. In case of deletion only the productid is required and second method – `publishProductDeletionEvent` – is called. Similar approach is implemented in endpoints exposing CRUD operations for users, orders and stock.

Listing 6.1-1 Publishing product events to Kafka in NDSOS

```
public Product publishProductEvent(Product product) {
    log.info("Publishing product event {}", product);
    return publish(product, product.getId());
}

public void publishProductDeletionEvent(String productId) {
    log.info("Publishing product deletion event {}", productId);
    publish(null, productId);
}

private Product publish(Product product, String key) {
    try {
        return kafkaTemplate
                .sendDefault(key, product)
                .get(timeout, TimeUnit.MILLISECONDS)
                .getProducerRecord()
                .value();
    } catch (ExecutionException e) {
        String message = "Sending failed due to ExecutionException";
        log.error(message);
        throw new ProductNotPublishedException(message, e.getCause());
    } catch (TimeoutException | InterruptedException e) {
        String message = "Sending failed due to thread interruption or 
timeout of " + timeout + " ms was exceeded";
        log.error(message);
        throw new ProductNotPublishedException(message, e);
    }
}
```

As it was already mentioned, properly selected key allows for sending specified subset of events to the same partition of selected topic and therefore guarantees that they will be all processed in the order of creation. In this case, `productId` was selected as a key, what means that every new event for the same product will be processed after previous ones.

It is however worth mentioning that, unlike in case of databases, assignment of unique identifier to entities lays within application responsibilities. In presented prototype, UUID (universally unique identifier) generated by `java.util.UUID.randomUUID()` was chosen. Even though each message published to Kafka can be identified by the combination of topic name, partition number and offset, it may not be sufficient in case of assigning identifier for an entity, since the unique ID is needed before the first message gets published. Therefore other mechanism must be used, like – for example - the proposed UUID.

6.1.3 Read operations

To describe read operations, first the distinction between two concepts provided by KafkaStreams – that is KTables and GlobalKTables - must be introduced. KTables, as it was described in previous chapters, provide the *abstraction of a changelog stream, where each data record represents an update*[26]. The exact same description could be used to describe GlobalKTables. Difference, however, is in the behavior of those two types of tables in case of more than one instance of a service running.

Just like the data from different partitions of a topic is consumed by different consumers in a group, data in KTables is also divided among them. This means that each instance has only a part of the whole state. GlobalKTables, on the other hand, provide a full copy of underlying data available on each instance. Figure 6.1-1 illustrates how KTables and GlobalKTables are populated with data from the same topic. Each shape on this figure – that is circle, triangle, diamond and a star - represents an event with some particular key. In case of a consumer group "B", reading the presented topic as KTable, only one instance has the information about particular shape. For consumer group "C", which is reading the data as a GlobalKTable, full copy is available on every instance.

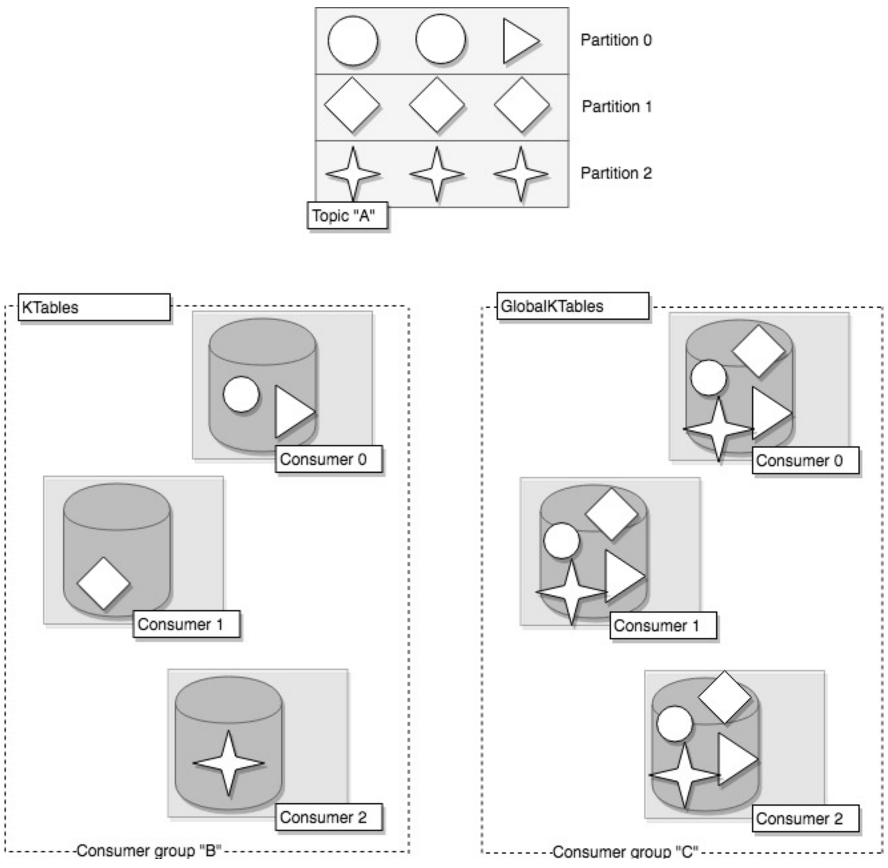


Figure 6.1-1 Illustration of difference between KTable and GlobalKTable; Source: own elaboration

Both approaches have their advantages and disadvantages and they should be taken into consideration on the stage of designing the application. GlobalKTables, as they provide information about data from all partitions, require more space and intensified data movement as events from all partitions must be read by all instances. On the other hand, KTables require implementing custom layer of communication between instances in order to obtain information about an entity that is not stored within local state store of currently queried instance.

Listing 6.1-2 Reading data from topic as a GlobalKTable

```
@Bean
public GlobalKTable<String, Product>
allProductsGlobalKTable(StreamsBuilder streamsBuilder) {
    return streamsBuilder
            .globalTable(TOPIC_PRODUCTS,
                    Consumed.with(null, productSerde),
                    Materialized.as(STORE_ALL_PRODUCTS));
}
```

Listing 6.1-2 presents fragment of code belonging to product service, in which data from topic “products” is read as GlobalKTable. This allows for querying products directly via arbitrary instance as they all contain information about current state of products. It is however worth mentioning that such approach might not be the most optimal solution for large datasets of products, considering the difference in required storage space in comparison to the distributed dataset.

On the other hand, the fact that complete state of products is kept locally by every instance, makes listing all products very straightforward. Each instance can easily return whole dataset of products as it has that state accessible locally. This would be difficult to implement in case of KTables – each service would have been asked for its part of data so that the queried instance could put all the pieces together and return complete set of products. Such complication is an example of tradeoff between having all data available in one place and having it distributed across multiple places. During development of system that is built around a streaming platform it should be considered whether particular application really requires quick access to complete state or maybe the fact of this state being distributed across all instances brings more advantages.

Listing 6.1-3 shows part of product service’s code, responsible for reading products from underlying state store. `StreamsBuilderFactoryBean` is used by Spring for managing the lifecycle of `KafkaStreams` and exposing a singleton instance of `StreamsBuilder` used for interacting with `KafkaStreams` via high-level DSL (domain specific language). Method `getProductStateStore`() returns a `ReadOnlyKeyValueStore` for provided store name (in this case – containing products). As the underlying state store contains whole state considering products, the `ReadOnlyKeyValueStore` can be both queried for product with particular key as well as for all existing products.

Listing 6.1-3 Reading products from a state store

```
public Product getProduct(String productId) {
    final Product product = getProductStateStore().get(productId);
    if(product == null) {
        throw new ProductNotFoundException();
    }
    return product;
}

private ReadOnlyKeyValueStore<String, Product> getProductStateStore()
{
    return streamsBuilderFactoryBean
            .getKafkaStreams()
            .store(STORE_ALL_PRODUCTS,
QueryableStoreTypes.keyValueStore());
}

public List<Product> getAllProducts(){
    List<Product> products = new ArrayList<>();

    getProductStateStore()
            .all()
            .forEachRemaining((product) ->
products.add(product.value));

    return products;
}
```

Since the state stores are built based on data from topics (and it is not the topic that gets queried directly) is possible that requested product, even though was already published to certain Kafka topic, will not be yet accessible. This leads to the concept of eventual consistency – data in state stores will be consistent, it may however require some processing time. Since Kafka is designed to process big volumes of data in real time, the probability of such case could be considered relatively low. It may however appear, especially in systems where high throughput of events takes place. Depending on the use case, this could be either just taken into the consideration and accepted or mitigated, for example by implementing long-polled get (blocking the request until the item is available)[17]. Also the state store itself may become unavailable, for example during the application start of rebalancing. Chapter 6.3 brings more insight into possible ways for managing such case.

Reading data from topic to a KTable is very similar to what was presented on Listing 6.1-2, only the method `table`() instead of `globalTable`() must be called on `StreamsBuilder` instance. Querying data from underlying state store is however, as it was already mentioned, more demanding. There would be no difficulty in situation when only one instance of particular service is running. In that scenario such instance would read events from all partitions and therefore whole state would be accessible in a `KTable` just like it would be available via `GlobalKtable`. However, once more instances are running and partitions are assigned between them, each `KTable` would provide only a part of information. This is unusual, in comparison with traditional databases, where state is kept externally and can be accessed through each instance in the same way.

Kafka Streams does not provide a direct mechanism to obtain data from other instance's state store. It does however come with methods and configurations that could be used in conjunction with individually implemented remote call layer. Spring for Apache Kafka also does not come with any solution for communicating between instances running Kafka Streams. It is, however, worth noticing that it is partially supported via `InteractiveQueryService` included in another Spring project – Spring Cloud. Own implementation of similar service was written during development of NDOS.

Listing 6.1-4 Fragments of utility class used for requesting other instances' local state

```
public <K> HostInfo getHostInfoForStoreAndKey(String store, K key,
Serializer<K> serializer) {
    StreamsMetadata streamsMetadata =
streamsBuilderFactoryBean.getKafkaStreams().metadataForKey(store, key,
serializer);
    HostInfo hostInfo = streamsMetadata != null ?
streamsMetadata.hostInfo() : null;
    if (hostInfo != null && !hostInfo.host().equals("unavailable")) {
        log.info("Host info for store {} and key {} is {}", store, key,
hostInfo);
    } else {
        log.info("Host info unavailable or unkown");
        return null;
    }
    return hostInfo;
}

public <T> T getFromDifferentHost(String mapping, HostInfo hostInfo,
Class<T> requested) {
    log.info("Fetching from different host {} : {}: ", hostInfo.host(),
hostInfo.port());
    RestTemplate restTemplate = new RestTemplate();
    return restTemplate.getForObject(
            String.format("http://%s:%d/%s", hostInfo.host(),
                    hostInfo.port(), mapping), requested);
}
```

Listing 6.1-4 shows two most important methods from this utility class - `RemoteStoreHelper`. Within first of presented methods, `geHostInfoForStoreAndKey`() we can see that Kafka Streams provides the mechanism to identify host that holds the part of state store containing information about record with particular key. After calling `metadataForKey`() and providing state store name and requested key, metadata containing information about the host is returned. Then, having known the host, specified object can be obtained by using the second method – `getFromDifferentHost()` – which calls another instance of an application using `RestTemplate`.

This necessity of introducing some kind of communication layer directly between instances of the same service may be considered a significant difficulty, especially in comparison to the traditional database-centered architecture where such solutions are not needed. It is however worth noting that queries – in a really event-driven architecture – should not be treated as first class citizens. In author's opinion, synchronous request-response communication, even though often obligatory, should not be a main indicator of how well the concept of database inside out suits the microservice architecture. In fact, Kleppman in his previously described speech suggests that another step in the direction of changing database architecture would be for clients to also subscribe directly to the streams of events. However, he also objectively sums up that this would require *a big rethink of the way we write applications* [1].

6.2. Joining and processing streams

Reading data from a topic to KTable is not the only way of creating such table. While treating new event as an update can be suitable in many situations, it usually happens that other stateful operations are required to be performed. Also, like it was mentioned in previous chapters, one service might need to enhance data from its context with information coming from other services. Another common scenario would be for multiple services to react to each other's events in a way creates a chains of actions. All those cases are described in following subsection.

6.2.1 Enhancing the information

Stock service, unlike the product service, publishes events that should not be treated directly as updates of previous events with the same key. Each published event indicates change of current stock for particular product. It contains two fields: `productId` and `quantity`, where quantity stands for units that were added or subtracted from stock, not total stockpile. With this convention, replacing events by key, would not provide all expected information, as sum of current stock should be calculated.

Kafka Streams provides convenient methods for grouping and aggregating data. Listing 6.2-1 shows the code fragment responsible for aggregating stock updates into current stock total. First, data is read from topic as a stream, meaning that each record is processed as it arrives. Then, quantity is extracted from `Stock` object and aggregation is performed. Each incoming event is added to aggregation and result is materialized into local state store. For each stateless operation changelog topic is created so in case of system restart or rebalancing performed operations will not be reprocessed but state will be “recovered” based on underlying changelog topic.

Listing 6.2-1 Aggregation of stock updates in stock service

```
@Bean
public KTable<String, Long> currentStockTotalKTable(StreamsBuilder
streamsBuilder) {
    return streamsBuilder
            .stream(TOPIC_STOCK,
                    Consumed.with(Serdes.String(), stockSerde))
            .mapValues((id, stock) -> stock.getQuantity())
            .groupByKey()
            .aggregate(
                    () -> 0L,
                    (id, value, agg) -> agg + value,
                    Materialized.with(null, Serdes.Long()));
}
```

Stock service however, when asked for current stock of particular product, returns more information than just the result of described aggregation. It also shows current information regarding particular product as well as quantity that is at the time reserved within some order. In situation when product service would have its own database, stock service in order to obtain such information would need to request missing data from product service. However, in ecosystem build around Kafka it has the possibility to read pure product-related events and adjust them to its own needs. Exemplary response that stock service provides is presented on Listing 6.2-2. Whole `product` section of returned json contains data published to Kafka by product service. `CurrentStock` shows aggregation of stock updates and field `reserved` indicates how many items of that product are currently reserved.

Listing 6.2-2 Example of a response returned by stock service

```
{
    "product": {
        "id": "c10b138f-783e-4f5f-a96f-f05ac552c96f",
        "name": "Black car",
        "price": 500000,
        "category": "CARS"
    },
    "currentStock": 10,
    "reserved": 0
}
```

To obtain such result topic products must have been joined with beforementioned KTable containing total of stock as well as with KTable that holds aggregated amount of currently reserved quantity. Kafka Streams offers multiple types of joins. Those joins defer, depending on whether KStreams, KTables or GlobalkTables are included. Also, similarly to the joins known from databases, the relation of records between streams/tables depends on joining type.

Listing 6.2-3 presents fragment of stream processing performed to achieve joining of product, stock and reservation information within stock service. At first, to each record in previously defined KTable with products (created by reading topic "products" as a KTable) corresponding record from KTable with stock (creation of which is presented on previously discussed Listing 6.2-1) is joined. Performed join is between two KTables what means that it can be only performed by the key, since the underlying data is divided accordingly to topic's partitions. In presented scenario events in both products and stock topics have `productId` as their key. In case when a join on other fields is required then data must be regrouped at first so that during join both tables have the same field as a key. Example of such operation will be presented in further section.

Listing 6.2-3 Enhancing stock information in stock service

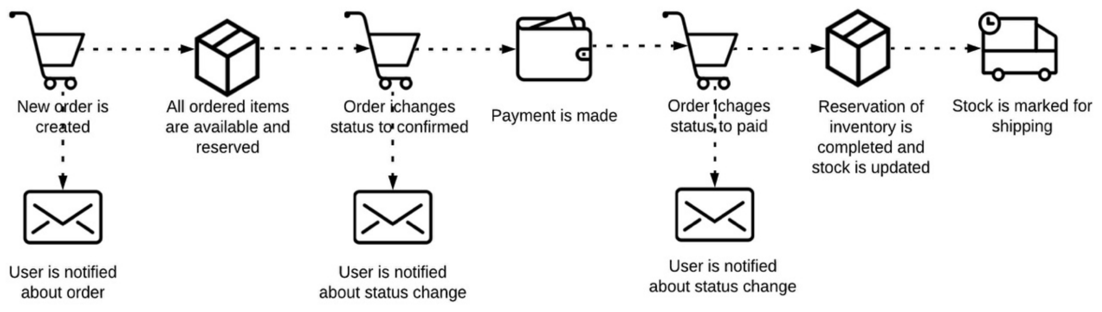
```
@Bean
public KTable<String, ProductStock> productWithStockKTable
(StreamsBuilder streamsBuilder) {

    return productKTable(streamsBuilder)
            //join current stock info
            .leftJoin(currentStockTotalKTable(streamsBuilder),
                    (product, stockQuantity) ->
createProductStock(product, stockQuantity),
                    Materialized.<String, ProductStock,
KeyValueStore<Bytes, byte[]>>as(STORE_CURRENT_STOCK)
                            .withValueSerde(productStockSerde))
            //join reservation info
            .leftJoin(reservedStockKTable(streamsBuilder),
                    (productStock, reserved) ->
createProductStock(productStock, reserved),
                    Materialized.<String, ProductStock,
KeyValueStore<Bytes, byte[]>>as(STORE_PRODUCTS_WITH_STOCK)
                            .withValueSerde(productStockSerde));
}
```

Left join was chosen over the inner join to provide a record for each product even if no stock event with corresponding id was found. Within method `createProductStock()` value for `currentStock` is set to 0 if there is no record from `currentStockKTable` available. After that, the aggregation of reserved quantity was joined, also using left join and also resulting in value for reserved set to 0 in case of no matching record found.

Working with joins between KTables is similar to performing joins within a database, with the main difference being the fact that key is the only field joining can be performed on. There is also a requirement of co-partitioning, which means that to perform a join between two KTables, both underlying topics for those tables need to have the same number of partitions. Both those restrictions are however directly correlated with beforementioned anatomy of Kafka's topics and KTables themselves.

6.2.2 Chain of actions

The most complex logic within developed prototype was implemented around the action of placing an order. This subsection provides an overview on how stream processing was used to create a chain of subsequent reactions around three of the services – order, stock and notification service.



New order is created
All ordered items are available and reserved
Order changes status to confirmed
Payment is made
Order chages status to paid
Reservation of inventory is completed and stock is updated
Stock is marked for shipping
User is notified about order
User is notified about status change
User is notified about status change

Figure 6.2-1 Process flow for successful order realization in NDSOS; Source: own elaboration

Figure 6.2-1 shows the simplified flow of the order realization process in NDSOS. Short description of the successfully realized order is following:

1. Whole process starts with order service receiving new order request, which is then complemented with additional information about ordered products.
2. New order is created and appropriate event gets published to Kafka topic.
3. Stock service evaluates whether all ordered products are available in requested quantity and reserves them.
4. Order service knowing that all items are reserved, updates the status of an order
5. After order is paid, stock service updates the reservation and reduces available stock and marks specified quantity as ready for shipping.
6. During each status change notification is sent

Even though such flow appears intuitive it may be difficult to implement, especially considering the importance of particular elements being performed in strictly defined order. In case of having database-per-service within a microservice ecosystem, coordinating stock reservation and order statuses would most probably require multiple synchronous calls. This subsection shows how the described process was handled within NDSOS prototype. Both, successful order realization as well as order failure were taken into consideration and will be presented.

First feature of stream-based approach is already visible in the implemented flow right at the beginning, even before order event gets published to appropriate Kafka topic. Since the order request that is sent to order service does not contain information on prices of ordered products, order service enhances the data it receives with data about products kept in local state store. After reading topic products as a GlobalKTable it has the possibility to assign current prices and calculate order's total without communicating directly with product service. Since the GlobalKTable is created directly from the same topic that product information is published to, the risk of local information being not up to date would be almost completely insignificant. Each new event that product service publishes is consumed in real time by order service and acts as an automatic update of local state store.

Listing 6.2-4 Fragment of code responsible for checking stock availability

```
@Bean
public KStream<String, KeyValue<StockReservation, Long>>[]
orderReservationKStream(StreamsBuilder streamsBuilder) {

    KTable<String, Long> availableQuantity =
currentlyAvailableStock(streamsBuilder);

    KStream<String, KeyValue<StockReservation, Long>>[] branch =
orderKStream(streamsBuilder)
            //only new orders are checked
            .filter((orderId, order) -> order.getStatus().equals(NEW))
            //split event into separate events for each product
            .flatMap((orderId, order) ->
createStockReservations(orderId, order))
            //join info about available quantity
            .join(availableQuantity, KeyValue::new,
Joined.with(Serdes.String(), stockReservationSerde, Serdes.Long()))
            //branch to two streams based on whether requested quantity
is available or not
            .branch((k, v) -> isProductAvailable(v),
                    (k, v) -> !isProductAvailable(v));

    //for available - reserve
    branch[0]
            .mapValues(reservationRequest ->
StockReservation.newBuilder(reservationRequest.key).setStatus(RESERVED)
.build())
            .peek((k, v) -> log.info("Reserving: {}", v))
            .to(TOPIC_STOCK_RESERVED);

    //for unavailable - set status unavailable
    branch[1]
            .mapValues(reservationRequest ->
StockReservation.newBuilder(reservationRequest.key).setStatus(UNAVAILAB
LE).build())
            .peek((k, v) -> log.info("Unavailable: {}", v))
            .to(TOPIC_STOCK_RESERVED);

    return branch;
}
```

Majority of stream processing is performed by stock service, since it is responsible for checking available quantity, reserving and un-reserving it and marking it for shipping. Listing 6.2-4 presents one of the first actions that it needs to perform – validating availability of products requested within newly created order.

After new event on topic orders appears, it is filtered based on its status so that only for orders with status NEW available quantity will be checked. Then, by using `flatMap` operation, orders are divided into separate events – one for each product included in order. This is done within `createStockReservations`() method, where not only information about products from order invent gets converted into individual `stockReservation` objects but also, the key is changed from initial `orderId` to `productId`. This is especially important considering the join with KTable holding information about currently available stock by `productId`. As it was previously mentioned, changing the key is required to perform such join. According to Kafka Streams' documentation: *setting a new key might result in an internal data redistribution if a key based operator (like an aggregation or join) is applied to the result KStream[27]*. This means, that under the hood data will be repartitioned by the newly selected key and therefore, if two orders with different keys will contain the same product within them, they still will be processed one after another.

It is worth noting that KTable `availableQuantity` that is joined in discussed part, already contains the information about reserved items. Each value in that KTable represents number of products with particular id that are on stock and were not already reserved. Once this information is joined, the resulting stream can be divided based on provided predicate. All reservations requests are converted into `StockReservation` objects. Status `RESERVED` is assigned to events for which demanded item can be reserved (hence, available quantity is equal to or bigger than ordered amount). To the remaining cases status `UNAVAILABLE` is assigned. Both results are sent back to topic stock-reservation, from which order service can consume them and react accordingly. `StockReservation` object can contain one of the following statuses: `RESERVED`, `UNAVAILABLE`, `REJECTED` and `COMPLETED`. First two, as discussed, are used to indicate whether the reservation was sucessfuly made, the last two – if it was sucessfuly completed.

Joins between KStreams and KTables are quite similar to previously discussed KTable-KTable joins. There is no windowing included and incoming events (either within KStream of KTable) trigger lookups in other side of join to fetch the required information. Order service's reaction to availability of ordered products was implemented as a KStream-KStream join to present the differences but it could be as well done using KTables. Just like in architecture involving databases, here as well similar result can be achieved in variety of different ways. In case of lookups that are just a stage in some further processing KStream-KStream join can be however more optimal. In this situation – when order service reacts to stock service's availability evaluation – we can assume that expected evaluation will arrive in some short period after new order was created. Such assumption allows for selecting an appropriate time period for window in which join between the order and beforementioned availability evaluation should be performed.

Joins between two KStreams are in general significantly different than joins known from databases and for that reason, in author's opinion, are harder to understand than any of previously described joins. Having known the concept of KTables and KStreams, justification for windowing being required in joining streams is understandable. Since KStreams are by definition infinite streams of incoming records, joining them together without specified window would make the underlying state store (used to keep information about the join and to avoid re-reading whole stream during new event's arrival) would become bigger and bigger indefinitely.

To validate the order based on stock availability order service performs following operations:

1. For each new order, number of products that were included within it (and therefore require confirmation of availability) is extracted as a separate KStream
2. `StockReservation` events with status `RESERVED` are re-keyed and counted by `orderId`.
3. Result of the count is compared to the value described in first point.
4. Once all of the required reservation confirmations are obtained, order status is changed to `CONFIRMED`
5. If at least one `StockReservation` event with status `UNAVAILABLE` is consumed then order status is changed to `FAILED`
6. Order with updated status is published back to orders topic

As it was already mentioned, similar result could be achieved by keeping required number of reservations and already obtained reservations as KTables and comparing their values for particular productId on each update. By choosing to join KStreams together there should be a careful recognition made regarding the joining window. Within what period of time expected even should appear to be considered useful? What – if anything - should be done in cases when that event does not appear at all? Or appear later that expected? In some scenarios it might be important to implement a strategy for dealing with such unusual behavior. This strategy would depend on the particular business case but it should be noted than joining KStreams requires more attention and – in author's opinion - is more error-prone in terms of development, than joins between tables.

Once order service updates the status for an order, stock service handles the reservations. In case of a failing order, reservations that were successfully made must be marked as not relevant. This way, if order contained several products and only some of them appeared unavailable, the rest will not remain reserved. Such approach allows order service to decide how orders, for which requested products are available only partially, should be handled. Only if order service publishes an event saying that particular order was failed, then reaction of stock service would be triggered.

Listing 6.2-5 presents a code fragment responsible for rejecting reservations that were made for failing orders. Shown solution is - similarly to the beforementioned processing done in order service - based on assumption that information about order failure would be available in some specified amount of time since the reservation has been made.

Listing 6.2-5 Rejecting reservations for failed orders

```
@Bean
public KStream<String, StockReservation>
rejectedReservations(StreamsBuilder streamsBuilder) {
    //only failed orders are checked
    KStream<String, Order> failedOrders = orderKStream(streamsBuilder)
            .filter((id, order) -> order.getStatus() == FAILED);

    KStream<String, StockReservation> rejectedReservationStream =
reservationKStream(streamsBuilder)
            //only reserved items are chekced
            .filter((id, reservation) -> RESERVED ==
reservation.getStatus())
            //changing key from productId to orderId
            .selectKey((id, reservation) -> reservation.getOrderId())
            //if there is a failed order - reject reservation
            .join(failedOrders,
                    (reservation, order) -> reservation,
                    JoinWindows.of(FAILED_ORDERS_WINDOW),
                    Joined.with(Serdes.String(), stockReservationSerde,
orderSerde))
            //change status to REJECTED and key back to productId
            .map((key, reservation) ->
KeyValue.pair(reservation.getProductId(),
StockReservation.newBuilder(reservation).setStatus(REJECTED).build()));

    rejectedReservationStream
            .peek((k, v) -> log.info("Rejecting reservation : {}", v))
            .to(TOPIC_STOCK_RESERVED);

    return rejectedReservationStream;
}
```

At first, both included streams – reservations and orders – are filtered in a way that only failed orders and `StockReservations` with status `RESERVED` are left. Then, for each reservation, if a failing order with matching `orderId` appears within specified time range, the reservation is marked as rejected and published again to stock-reservations topic. In this way, there is an event available for each action. Having knowledge about all possible statuses of `StockReservations`, stock-service (and any other service if there would be such need) can calculate currently reserved amount like it was described in previous subsection.

Completing reservation is – at the beginning – performed in a similar way. Orders and reservations are filtered but this time orders with status PAID are left. For each reservation that has a corresponding order event joined, the status is changed to `COMPLETED` and updated `StockReservation` is published back to stock-reservations topic. Additionally, event with opposite quantity to the one that was reserved is published to topic "stock" to indicate decreased quantity. Finally, new `StockForShipping` object is created and published to "stock-for-shipping" topic.

Since there is no payment service included in NDSOS prototype, method simulating payments was implemented within order service. For each confirmed order that has even number of products status is changed to `PAID`. In a complete business application such action could be triggered by an appropriate payment event being published by a service responsible for handling finances. All orders assigned to particular user can be accessed by calling the appropriate endpoint, as listed in Table 6-1.

At the same time that order is processed, notifications regarding the changes of state are prepared. Notification service is subscribed to two topics – user and orders. Whenever new order event arrives, key of that event is changed to `userId` and current user data is obtained from locally created KTable. Created notification is logged (to simulate sending an email) and notification event is published to Kafka. Beside order-related notifications, there are also ones resulting from account changes. Each time new user is added or existing user is updated – appropriate notification is generated. Since there is no status in user event that would indicate whether a received event represents creation of a new account or update of an existing one, the content of notification is created using aggregation. If incoming event is the first event for given key, welcoming message is sent. Otherwise, a message regarding account update is generated.

Table 6-2 shows an example of all messages that are published to Kafka after creation of new order. In presented example all ordered products were available and order was paid. For clarity, UUIDs were replaced with easier to read form and array of products was omitted in consequent order events. All those events can be observed by running NDSOS's demonstration service which reads data from all Kafka topics and prints them in console in different color for each topic. It is worth mentioning that order of consuming messages between different topics is independent, hence it may appear that – for example – multiple status changes for order will be visible before first order notification. However, since notifications are partitioned by `userId`, it is guaranteed that they would be sent in required order.

Table 6-2 Example of messages published to Kafka within successful order process

No.	Topic	Key	Value
1	`orders`	1	`{ "orderId":"1", "userId":"2", "products":[ { "product":{ "id":"3", "name":"Black car", "price":500000.0, "category":"CARS" }, "quantity":2 }, { "product":{ "id":"4", "name":"Green car", "price":300000.0, "category":"CARS" }, "quantity":2 } ], "total":1600000.0, "status":"NEW" }`
2	`notifications`	2	`{ "receiverId":"2", "receiverMail":"maria@nowak.pl", "message":"New order was created. Order id: 1; Order total: 1600000.000000; Items: Black car (2) -- price 500000.000000; Green car (2) -- price 300000.000000" }`

3	stock-reservation	4	`{ "productId":"4", "quantity":2, "orderId":"1", "status":"RESERVED" }`
4	stock-reservation	3	`{ "productId":"3", "quantity":2, "orderId":"1", "status":"RESERVED" }`
5	orders	1	`{ "orderId":"1", "userId":"2", "products":[ //omitted for clarity ], "total":1600000.0, "status":"CONFIRMED" }`
6	notifications	2	`{ "receiverId":"2", "receiverMail":"maria@nowak.pl", "message":"Status of your order with id 1 has been updated to CONFIRMED" }`
7	orders	1	`{ "orderId":"1", "userId":"2", "products":[ //omitted for clarity ], "total":1600000.0, "status":"PAID" }`
8	notifications	2	`{ "receiverId":"2", "receiverMail":"maria@nowak.pl", "message":"Status of your order with id 1 has been updated to PAID" }`

9	stock-reservation	3	{ "productId":"4", "quantity":2, "orderId":"1", "status":"COMPLETED" }
10	stock-reservation	4	{ "productId":"3", "quantity":2, "orderId":"1", "status":"COMPLETED" }
11	stock-for-shipping	3	{ "productId":"3", "quantity":2, "orderId":"1", "status":"READY_FOR_SHIPPING" }
12	stock-for-shipping	4	{ "productId":"4", "quantity":2, "orderId":"1", "status":"READY_FOR_SHIPPING" }
13	stock	3	{ "productId":"3", "quantity":-2 }
14	stock	4	{ "productId":"4", "quantity":-2 }

6.3. Additional configuration options

In a system where streaming platform plays crucial role, proper configuration of both this platform itself and particular streaming applications is, without a doubt, an important task. Some configurations, like choosing proper number of partitions and replication factor for topic, enabling exactly once processing or choosing data format were already discussed. There is however much more to choose from when it comes to adjusting the environment.

Ironically, the amount of available configuration options might be considered a downside, especially at the beginning of designing streaming applications. Kafka Streams provides mechanisms to adjust almost every aspect of streaming applications – starting from configuration of underlying Kafka consumers and producers, through adjusting RocksDB used for persistent storage, up to defining custom handlers in case of state restores or deserialization exceptions.

There are also issues that are partially configuration-related and are hard to predict unless they appear within a particular use case. One example could be the need to run on the same host two instances of some service that uses GlobalKTable. To be able to run such instances, property `state.dir` - indicating the directory in which state stores are located - must differ between each instance. Fortunately, both Kafka documentation as well as information available on Confluent's page gives guidance on how to adjust application to desired use case.

Some configurations are required and transparent in development process. One of them is `bootstrap.servers` which provides information about the broker. Just like in case of regular Kafka consumers and producers it is required to connect with Kafka cluster. It is worth mentioning that even if only one broker's address is provided, after initializing a connection, all brokers within a cluster will be accessible. Another required configuration – in this case specific for KafkaStreams – is `application.id.` This property must be unique across different streaming application but the same for all instances of the same service. Instances with the same `application.id` are treated as one consumer group.

Another important thing to consider is configuration of serdes – classes used by Kafka Streams for serializing and deserializng messages. Not only such serdes must be configured but also provided with every operation that results in sending data to underlying Kafka topic. As it was shown on listings in previous subsection, each join must have been provided with particular serde that should be used within performed operation. Default serdes can be provided in application's configuration by setting values for `default.key.serde` and `default.value.serde` properties which, depending on the chosen datatype, may not require providing serdes explicitly in beforementioned operations. However with Avro, and with setting default serde to `SpecificAvroSerde` still, serde created for particular avro-generated class is required. Listing 6.3-1 shows an example of registering Serde for Product as a Spring Bean and configuring schema registry for it. Similar beans for each avro-generated class written to or consumed from Kafka topics was created.

Listing 6.3-1 Configuration of serde

```
@Bean
public Serde<Product> productSerde() {
    final Serde<Product> serde = new SpecificAvroSerde<>();
    configureSchemaRegistry(serde);
    return serde;
}

private void configureSchemaRegistry(Serde serde) {
    final Map<String, String> config = Collections.singletonMap(

AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
SCHEMA_REGISTRY_URL);
    serde.configure(config, false);
}
```

In case when either schema registry is not used or there is a risk of consumer getting data that it will not able to deserialize, `default.deserialization.exception.handler` can be set. Two exception handlers are available out of the box. – one to log the exception and continue processing next records and another to stop processing on arrival of corrupted data. Custom handler can be used by implementing `org.apache.kafka.streams.errors.DeserializationExceptionHandler` interface, for example to send invalid data to separate topic for further analysis. There is also a possibility to provide custom implementation of few other exception handlers such as `DefaultProductionExceptionHandler` or `UncaughtExceptionHandler`.

Next aspect in terms of robustness could be to adjust internal topics created by Kafka Streams in a similar way that regular Kafka topics would be configured. For example, the default `replication.factor` for internal topics is set to 1, which means than partitions are not copied among different brokers. As it is suggested in documentation, together with `replication.factor`, changing two other default configurations could be considered to improve topic's availability - `acks` and `min.insync.replicas`. Acks parameter specifies how many acknowledgements are required by leader from in-sync replicas to consider record successfully processed. As it was shown in chapter 4.1.2 this setting should be taken into consideration only in situations when replication factor is higher than one. The same goes for `min.insync.replicas` which indicates how many of the existing in-sync replicas must be available at the same time to allow for producers to publish messages to certain partition.

Availability of state stores could also be considered. Like it was previously mentioned, state in Kafka Streams is not only persisted on local storage but also is backed-up by a changelog topic. This serves multiple purposes, one of them being the ability to quickly restore one instance's state within another instance. In case of one instance being down, Kafka triggers rebalancing and newly selected instance restores the unavailable instance's state basing on the changelog topic. Until that restore is complete, state store may be unavailable. In many cases implementing some awaiting mechanisms that would inform about temporary unavailability should be enough. KafkaStreams gives the ability to monitor state changes and react to them by using the `StateRestoreListener` interface. In scenarios when such reaction is not sufficient and temporary unavailability is not acceptable a configuration called `num.standby.replicas` – which by default is set to 0 – could be set to some positive value. In general, standby replicas are idle copies of one instance's local state store. When a fully replicated copy of a state store already exists, there is no need to reprocess changelog topic in case of one instance's being down. Therefore, the amount of time required to access a certain state store is significantly smaller.

In terms of topics, besides the configurations that were already mentioned, there are at least two more that should be carefully considered, that is cleanup policy and retention. Cleanup policy instructs Kafka how messages within particular topic should be handled in longer term. By default this configuration is set to `cleanup.policy = delete` which simply means that messages older than specified time or exceeding the provided size limit will be deleted. In combination with this cleanup policy the second of beforementioned configurations is used – that is retention. Retention can either by defined by specifying `retention.ms` or `retention.bytes` configuration, depending on whether time limit or size limit should be applied. By default only the time retention is specified. Alternative cleanup policy is `cleanup.policy = compact`. After specifying that particular topic should be compacted, it is not time or size that define what data is kept, but the message key. On compacted topics there will always be at least one record kept for each key that was published. Compaction is also a setting that by default is used in changelog topics, backing up state stores created by Kafka Streams.

7. Summary and conclusions

Following chapter summarizes author's conclusions on discussed approach to designing systems. It evaluates main advantages and disadvantages noticed during research and process of creation of prototype. At the beginning it also provides insight into possible extensions and tools that might be useful for implementing the truly event-driven ecosystem.

7.1. Possible extensions

Like it was already mentioned, the purpose of created NDSOS prototype was to give an overview on process of implementing applications built with Kafka and Kafka Streams as central pieces of architecture. This subsection provides examples of other tools and mechanisms that could be useful in microservice eco-system designed with discussed approach in mind.

First tool leveraging possibilities of streaming-based architecture is KSQL - *streaming SQL engine for Apache Kafka[28]*. It allows for processing streaming data from Kafka by using a query language similar to well-known SQL. It requires a separate KSQL server and comes with both command line as well as graphical interfaces available. Considering the demonstrated NDSOS prototype it is easy to imagine that some ad-hoc lookups of data might be required - either in development or, for example, to perform some analytics or generate reports. One possible way would be to create separate service handling beforementioned tasks, but often implementation of a specific application not only might be too complex and time consuming but also would not serve the purpose as well as direct querying of the data. KSQL, by allowing to use a language with syntax very similar to SQL, brings processing of streams to broader audience than just application developers. It provides possibility to create streams and tables, capture data within specified windows of time, and perform operations such as joins or grouping.

Another solution, also developed under Confluent's support, is Kafka Connect. It allows for both transferring data from some other external data source into Kafka as well as to for exporting data from Kafka to specified destination. Main concept in Kafka Connect are the sink and source connectors, depending on whether data should be transferred to or from Kafka. There are multiple different connectors available for variety of data sources and if needed custom connectors can also be implemented. Setting up a connector comes with different configurations to choose from. Even though Kafka Connect was not integrated within NDSOS, it was briefly explored as a possible valuable extension.

Listing 7.1-1 shows example of how sink connector for topic products could be set up. As it shows, there is a possibility to specify what topics should be considered, how updates of data should be handled or which fields should be considered a primary key. Certain fields can be also black (or white) listed and schema registry can be used together with Kafka Connect to give the connector an information about registered schemas for particular topic. It is however worth noticing that Kafka Connect, just like other solutions designed with purpose to be integrated into Kafka-centered environment, is still improved and there are areas that might not yet work as expected. One example, that author encountered during exploration of Kafka Connect is that – at the moment of NDOS creation – there was no support in `JdbcSinkConnector` for deleting records. This means that after setting up the connector configured as it is presented on Listing 7.1-1, each record published with null value (to indicate deletion for KTables) was not interpreted by underlying database. Such behavior made storing the results of stream processing done within NDSOS in relational database impossible unless custom connector would be developed. It is possible that by the time this thesis is published, beforementioned issue was already resolved but just the fact it existed shows that even for such intuitive behaviors, provided solutions might not yet be mature.

Listing 7.1-1 Possible configuration of sink connector for products in Kafka Connect

```
name=postgres-products-sink
connector.class=io.confluent.connect.jdbc.JdbcSinkConnector
connection.user=user
connection.password=password
tasks.max=1
topics=products
value.converter.schema.registry.url=http://localhost:8081
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=io.confluent.connect.avro.AvroConverter
auto.evolve=true
auto.create=true
connection.url=jdbc:postgresql://postgres:5432/postgres
insert.mode=upsert
fields.whitelist=name,price,category
pk.mode=record_key
pk.fields=id
```

Third thing worth mentioning is Processor API offered by Kafka Streams. All operations presented within NDSOS were performed using the Kafka Strems' DSL. There could be however a situation, especially in complex and mature business applications, where some more custom operations would have to be performed. For such cases Kafka Streams provides Processor API – a way to apply custom, more flexible operations to processed events. Processors, among other possibilities, give access to event's metadata, allow for setting punctuation to trigger actions periodically and may interact with defined state stores. While, in comparison to DSL, using Processor API is more advanced and more difficult to use as it operates on lower level, it may be in beneficial in cases when functionalities provided by DSL are not sufficient.

7.2. Conclusions

In author's opinion, there are two ways to look at advantages and disadvantages resulting from a discussed approach – in micro and macro scale. Macro scale meaning the benefits or drawbacks observed for the microservice ecosystem as a whole and micro scale – advantages and troubles seen in process of development from application developer's perspective (with certain stream processing and communication tools and whole event-oriented way of development in use).

7.2.1 Macro scale

First, undeniable advantage of choosing event-driven approach and relying on stream processing within a microservice architecture is the accessibility of data. Chapter 3 presented in detail how hard to manage are situations where one service needs to aggregate data from different service's boundaries. Sharing database between microservices is known as an antipattern and a separate database for each service, on the other hand, can lead to all sorts of different problems (for example it can create a situation when service's API is simply a proxy to database that stands behind it). Therefore, data aggregation in both these scenarios is hard to manage. On the contrary having immutable log as a shared source of truth and keeping operations on data locally within each service makes it possible to easily connect data from different bounded contexts without the need to manage cache or handle changes in underlying databases owned by other services. As it was presented, order service can keep information about products locally, even though they belong to the different bounded context. During order creation, current product price and details can be fetched and appended to order event directly by order service. Even better example is shown within stock service, which adjusts content of stock-reservation topic so that only filtered information is available and kept for lookups. In both cases, the KTable with required information is created exactly in a way that it contains only data needed by particular service. Therefore its content is optimized for performed reads, just like it would be when using materialized views. Only in this case, views are available locally and are "owned" by one particular service, what makes access to data fast and ensures loose coupling especially in case when this service decides to change structure of this data.

Besides the fact, that locally kept state is adjusted and customized to each service's needs there is also and advantage of instant reaction to new data being possible. There is no need to poll data from database to check if there is something new, service reaction can be always triggered instantly by new data on some topic it's subscribed to. It does not matter whether the underlying topic is "owned" by this service or a completely different one because shared access to Kafka does not implicate issues that sharing a database does. Implementing the order flow, presented in previous chapter, would have been significantly more complex in "traditional" microservice architecture, where synchronization between services must have been handled. Not only order service can rely on obtaining reservation confirmation once the stock service is available, it also has a full control over handling partially available orders. Chains of actions, like the one presented on the example of orders, can easily be considered the best use cases for discussed approach.

However, for business area of online shopping, chosen for the presented prototype, the fully stream-oriented approach might – in author's opinion – not necessarily be the best choice. That is unless also the layer of communication with the client would be taken a step further into the streaming way. This conclusion is most probably dictated by the biggest and most visible disadvantage that author encountered during following thesis – a requirement of mindset change. Simple online shops are usually associated with CRUD operations as first class citizens. That is exactly why this business area was chosen for described prototype – to evaluate if such scenario could be implemented in microservice architecture without a database behind each service. Even though Kafka Streams offers the possibility to query the state just like the database would be queried, this querying feels more like an subsidiary action than something that would be the core of a service. Where listing products and creating orders are basic operations, the mindset change required to stop treating all actions as commands and to start perceiving them in terms of events might be an overhead that is not proportional to the introduced benefits. In an environment where databases are considered essential and synchronous communication between clients and servers is deeply ingrained in how applications – especially web applications – are perceived, changing this mindset can be hard and time consuming. And while taking that time and effort might come with many advantages it is not necessarily an investment worth making in applications that are not struggling with sharing data or are not big enough to really benefit from implemented changes. In author's opinion, systems where REST API is just exposed by one service, for example with purpose of some administrative actions or analytics, while other systems are concerned only by processing the data and sending it back to Kafka topics, would be a better fit for trying out the described approach. In other cases, streaming data directly to the clients, resulting in automatically updated dashboards should be considered as a way to fully adopt the event-driven approach.

It is possible that majority of the identified troubles were related to this requirement of mindset change, which was most probably reflected in prepared prototype. Even though the significant amount of time was spent on designing how the presented functionalities should work and be divided between services, a perception of expected outcome was still based on what is familiar to the author and therefore biased from the beginning. Similar difficulty is highly likely to happen in companies aiming to create their applications in an event-driven way. It should be expected that in teams made of developers who are used to certain approach, changes as drastic as switching to the streaming platform-centered ecosystem will proceed gradually and – at least at the beginning – will not stand up to the full potential of the concept. The lack of knowledge itself could be considered a disadvantage. Throughout the years of database popularity, there were many experienced developers who could easily identify what are the best ways to work with certain database, how to optimize its usage and benefit from its characteristics. When it comes to event-driven approach – not to mention Kafka and Kafka Streams themselves – group of specialists and therefore access to knowledge is much more limited. There is a steep learning curve – both on a design and implementation level. As event driven systems using streaming platforms as a source of truth are a concept relatively new (especially some aspects, like Kafka Streams library and Interactive Queries that it provides) it is harder to find specialist on a market that can design and implement production ready system using this approach. Having that in mind a well-designed and optimized microservice eco-system build solely around a streaming platform could nowadays most probably be created only in companies gathering experts from this narrow field.

Nevertheless, in situations where such obstacles do not apply, advantages of the discussed approach would be proportionally big to the size and complexity of an eco-system. One reason for that, besides the ones that were already mentioned, would be also the expansion simplicity and better integration of heterogeneous data systems. Using the streaming platform in the middle gives the possibility to add new elements to the architecture without any need to interfere with the existing ones. New service can just subscribe to certain topics and – if needed – process all data right from the beginning. This is also the reason why this approach means better scalability. Therefore, since microservices built in accordance with this concept may be harder to create they are easier to maintain and use in ecosystems that are large and distributed.

Another advantage of chosen approach is the small number of external components that need to be managed. Since Kafka Streams is just an library and does not require any additional servers or complicated set ups, there is only one external element – that is Kafka. Even in situation when number of microservices would grow relatively big, this could remain unchanged by simply scaling up Kafka brokers. On the other hand, managing Kafka cluster might turn out to be - in author’s opinion - more demanding than managing databases. Especially in scenario where whole eco-system, each service and each action, depends on Kafka. NDSOS system was developed only locally and run using one Kafka broker. In production-ready applications however there would be a need to properly configure available brokers, to carefully choose number of partitions per topic, replication factors and other configurations that would allow for taking advantage of Kafka’s guarantees.

Kafka itself seems to be well suited for long-term storage due to its reliability and flexibility. Either by using it as an event store for pure event sourcing and keeping each event in topic or by setting the cleanup policy to compact and keeping the most recent event for each particular key. Both approaches can be combined by using Kafka Streams. It is resilient, well documented and available to handle millions of events per second. In general, whole family of Kafka-related tools, like Kafka Streams or Kafka Connect, is well cooperated and quickly gains popularity. It is, however, still relatively new and rapidly improving. While in case of mature and well known databases, like MySQL or Oracle Database, version upgrades can usually be assumed to not introduce any drastic changes, in case of Kafka, difference between current version and the one that was available – for example – a year ago would be much more significant.

Taking everything into consideration, in author's opinion presented approach would be beneficial in distributed and complicated systems where coming through the entry barrier – even though costly - would be an investment paying off in further communication ease and improvement. It is however important to mention that since NDSOS was developed only locally and by one person, it did not allow for verifying how the discussed event-driven approach would work in really distributed environment. That is also the reason it provided more insight into micro-scale advantages and disadvantages than into the macro-scale ones.

7.2.2 Micro scale

The beforementioned progressive improvement of chosen tools is also one of the obstacles in terms of application development. It could be said that despite the fact that it was firstly launched around three years ago, Kafka Streams remains a novelty comparing to older frameworks dedicated to stream processing like Apache Samza or Apache Spark that has been around for quite a long time. A library dedicated to integration with Kafka and leveraging its features of consumers and producers is taking stream processing to the next level by giving the ability to simply work with data from Kafka without the need of setting up another piece of architecture.

Documentation for Kafka Streams emphasizes that using the library, especially the DSL it provides, is intuitive and easy. However, in author's opinion, there are many elements that make the learning curve very steep, especially for beginners. Firstly, to understand how to use the library there is a requirement of at least basic knowledge of Kafka itself. Concepts of topics, consumer groups, partitions and multiple other Kafka-specific elements and configurations make it a very complex tool. Therefore, understanding the inner workings – even only on beginners level – can be relatively demanding and time consuming.

Spring's support for Apache Kafka and Kafka Streams, while relatively well documented and beneficial, at the same time introduces additional layers of abstractions and in a way makes learning curve even more steep. It is however enabling easier configuration and handles lifecycle management of crucial components like `StreamsBuilder`. It is worth specifying in this place that Spring for Apache Kafka was chosen over Spring Cloud for NDSOS development due to the lower level of abstraction. Spring Cloud, even though also widely used in microservice architecture, aims at hiding implementation details of message brokers and therefore adds even more layers on top of Kafka and

Kafka Streams but it uses the chosen Spring for Apache Kafka underneath. While for developers with little experience, like author of following thesis, understanding inner workings of Kafka and Kafka Streams might be more difficult with Spring mechanisms included, it is at the same time a great way to incorporate those tools into already existing and mature business applications that were built basing on Spring's framework.

When it comes to working with Kafka Streams library, even though methods available via Kafka Streams DSL are quite straightforward and well documented, there are various obstacles that come with using them. One of them is the necessity of providing serde with each operation that writes to Kafka topic. It is safe to assume that leading choices for data format in Kafka-centered architecture would either be Avro – due to its compact format and possibility of using schema registry – or Json – because of its already established popularity and ease of use. In first case, like it was presented on example of NDSOS, specific serde must be provided for each Avro-generated object that will be written to or read from Kafka topic. Such serdes in presented prototype were created as `SpecificAvroSerde` of certain type, registered as Spring Beans and injected into stream processing classes. Process would be similar in case of Json - custom serde would have been provided as well for each operation directly interacting with Kafka topics. However, in case of primitive and basic types – including String used for keys in NDSOS - Kafka Streams provides out of the box serdes. Even though providing serdes might not appear as a complicated process, it is application's developer responsibility to define them and properly include in each action that requires it. Such necessity does not apply to communication with databases, with variety of Object-Relational Mapping libraries like Hibernate being available and widely used.

Another, data-related difficulty, results from a fact that within discussed approach each event that is produced to Kafka topic should be available to each service that could potentially be interested in this data in future. This, on one hand brings great improvement in system scalability, since each new component can just join the ecosystem by consuming messages from Kafka and processing them in a way that is applicable to its business case. On the other hand however, such approach requires much more caution when designing data structure. Not only current consumers should be able to process the data produced to Kafka but also, any potential future components. Changes of schemas in databases are not simple as well, but the main difference is that once they are completed, the complexity ends. Current schema dictates binding data structure. In case of events being the shared data-source, each event produced to Kafka – no matter if it was published before or after changes in data structure – should be possible to be processed by consumers. This means that changes that are not backwards compatible might require creating a new topic. Choosing Avro as data format and using schema registry might be significantly helpful in controlling how data changes and avoiding publication of events that consumers might not be able to process. Still, complexity of working with data is much bigger than in database-centric systems. At the beginning of NDSOS development orders consisted of only one product. Changing that to orders being made of multiple products was done by creating a new topic and adjusting logic of both stock and order services. Since this was just a development of a prototype there was no risk of losing any data but in case of production ready systems requiring such a significant change a procedure would have been very troublesome.

Changed approach to data in general is what could be considered an obstacle in developing Kafka-centered streaming applications. It is challenging to work with only events, not some precomputed state, being a shared source of data. While working with databases, there is always a possibility of looking up current values. Database tables can be viewed and particular records can be extracted and analyzed during development. When working with Kafka, there is no possibility to quickly look up specific value. Even with previously mentioned KSQL or console consumers in place there is still some work required to be done in order to show and investigate available data. While there is a variety of user interfaces available to present databases' content, there is no such mechanism available for Kafka. Even though, knowing that stored events are representing each individual operation instead of aggregated state, such unavailability of interface presenting all data is understandable, this is still a major difficulty during development. The same applies for invalid data. While removing a single record from a database table does not bring any significant consequences and does not require complex operations, there is much more to removing a record from Kafka topic itself or the created state stores. Sending tombstone "delete" events, with value set to null, removes record with given key from state stores. However, once a new streaming application joins the ecosystem it would reprocess both – the initial events as well as the deleting one. There is a possibility to remove a record with specified offset from given partition but that brings its own problems – first of all such action in a way undermines whole concept of sharing an immutable log. Secondly, it may lead to the situation when consumers that had their state stores precomputed before the deletion would end up with different state (since the changelog topic, containing only the current information for any given key, already contains some precomputed state) than new ones. In general manually removing events from Kafka topics should not take place in discussed approach since the shared data, like in database replication-log, should always contain immutable information. To help with handling invalid data appearing in development process, Kafka Streams provides Application Reset Tool that allows for reprocessing data and cleanup method for rebuilding local state stores. However, outside development environment any data manipulation would be very risky, complicated to perform and possibly hard to recover from if anything goes not as expected. This, combined with inability to easily lookup on demand certain subset of events or current state is what makes working with streams especially difficult.

One more obstacle, that author identified during development of the prototype, was Kafka Streams' requirement for all instances of a service – sharing the same application.id – to contain the same stream processing logic. While it may not sound like an obstacle, it might in fact appear troublesome during update process in environments where always at least one instance should be available. In such cases, during the update, there is no possibility for one instance to be running with updated code, while another instance still contains the old logic.

There are also problematic areas in writing the stream processing code itself. It is possible that majority of identified obstacles were related to the beforementioned mindset change requirement. Also, working with streams, especially with Kafka Streams' DSL, might appear more intuitive to programmers with background in functional programming languages. In general, starting to write applications with Kafka Streams is – like documentation emphasizes – fairly easy. It is on the attempt of gaining deeper understanding where learning curve becomes really steep. All of the available options of optimization, different configurations, small - but unexpected, based on previous experience - issues is what, in author's opinion, altogether makes mastering discussed approach to building microservices complicated. Difficulties with data serialization or even the necessity to provide unique identifiers, combined with novelty of chosen tools and little expertise on the market, might be crucial in making a decision whether implementing discussed concept is a way to go. Separate thesis could be written on how to make streaming applications robust and how to optimize them. Beforementioned requirement of properly choosing number of partitions, replication factor, identifying whether standby replicas should be introduced are only a few examples of decisions that are very important for Kafka-centered environment to be properly working in long term. At the same time there is no place where one can read to what values this properties should be set. It all depends on particular business reason, available machines, expected throughput and many others. On micro-level choice between GlobalKTable and KTable can be a similar obstacle. Like it was discussed, on one hand GlobalKTable provides local copy of all data on each partition that is easy to access and enables listing all elements from a state store without additional calls to other instances. With KTable on the other hand application state is divided between instances and additional layer of communication must be introduced to access every record via one instance. This makes even simple operations like listing all products, much more complicated than in database-centered architecture. Suddenly they require a lot of consideration about how data should be partitioned to be stored in a resilient way and at the same time to be available via every instance. Another, similar, difficulty comes with using Kafka Streams DSL. Even though it is fairly intuitive and it covers majority of use cases, at the same time under the hood it creates many internal Kafka topics. Those topics, and data movement between them, may appear burdensome to Kafka cluster, depending on the machines it runs on. Monitoring internal topics, adjusting their configurations or even resigning from DSL in favor of low level processor API for better control over what is written back to Kafka, is another complicated aspect of working with Kafka Streams library.

7.3. Summary

This thesis provided an insight into building microservices around streaming platform, according to database inside-out concept. It discussed difficulties related to data sharing and communication in microservice architecture, drew an overview on different approaches to handling such obstacles and, basing on developed prototype, examined how stream processing and event driven architecture can be incorporated into microservice eco-system.

Since the discussed concept, as well as chosen tools – Apache Kafka and Kafka Streams – are very complex, the following thesis still leaves a lot of room to investigate the approach and improve presented prototype. It did however bring a broad perspective on both architectural and development level and resulted in variety of conclusions. Even though the discussed theory is convincing and in author’s opinion should be the direction to follow while designing complex distributed systems, it is at the same significantly more complicated than typical database-centric architecture. Choosing this approach requires significant changes in mindset and how interactions with services are perceived. While it provides an interesting alternative to storing and using data by microservices, at the same time it comes with its own difficulties. In author’s opinion, it should be considered in systems that consist of large amount of services and in which use cases are based on flows of actions. In smaller systems and the ones that are built mainly around querying the data and synchronous actions, the benefits coming from the presented approach might not be a sufficient return of investment that would need to be made to properly implement it.

Like every major architectural change, building microservices around events stored and shared in Kafka and keeping all state and operations on data locally to each service, requires time to really verify what benefits and flaws it may bring. Based on performed development and analysis, author believes that in a few years it might be a solution mature enough to define a new way of how application architects and developers will perceive building distributed systems.

8. Bibliography

[1] M. Kleppmann, ‘Turning the database inside-out with Apache Samza’, *Confluent*, 01-Mar-2015. [Online]. Available: https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/. [Accessed: 19-May-2019].

[2] M. Fowler, ‘Microservices’, *martinfowler.com*. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 17-Jun-2019].

[3] S. Newman, *Building microservices: designing fine-grained systems*, First Edition. Beijing Sebastopol, CA: O’Reilly Media, 2015.

[4] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Boston: Addison-Wesley, 2004.

[5] ‘Architectural Styles and the Design of Network-based Software Architectures’. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. [Accessed: 10-Aug-2019].

[6] ‘Richardson Maturity Model’, *martinfowler.com*. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html. [Accessed: 12-Aug-2019].

[7] ‘gRPC documentation’. [Online]. Available: https://grpc.io/docs. [Accessed: 11-Aug-2019].

[8] ‘Apache Kafka documentation’, *Apache Kafka*. [Online]. Available: https://kafka.apache.org/documentation/. [Accessed: 14-Aug-2019].

[9] ‘ActiveMQ documentation’. [Online]. Available: https://activemq.apache.org/features. [Accessed: 12-Aug-2019].

[10] ‘Documentation: Table of Contents — RabbitMQ’. [Online]. Available: https://www.rabbitmq.com/documentation.html. [Accessed: 01-Sep-2019].

[11] J. Korab, *Understanding Message Brokers*. 2017.

[12] ‘Which protocols does RabbitMQ support? — RabbitMQ’. [Online]. Available: https://www.rabbitmq.com/protocols.html. [Accessed: 12-Aug-2019].

[13] ‘AMQP 0-9-1 Model Explained — RabbitMQ’. [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts.html. [Accessed: 12-Aug-2019].

[14] M. Fowler, ‘Integration Database’, *martinfowler.com*. [Online]. Available: https://martinfowler.com/bliki/IntegrationDatabase.html. [Accessed: 17-Jun-2019].

[15] ‘Envers - Hibernate ORM’. [Online]. Available: https://hibernate.org/orm/envers/. [Accessed: 01-Sep-2019].

[16] ‘ApplicationDatabase’, *martinfowler.com*. [Online]. Available: https://martinfowler.com/bliki/ApplicationDatabase.html. [Accessed: 13-Aug-2019].

[17] B. Stopford, *Designing Event-Driven Systems*. 2018.

[18] P. Helland, ‘Data on the Outside versus Data on the Inside’. [Online]. Available: http://cidrdb.org/cidr2005/papers/P12.pdf. [Accessed: 13-Aug-2018].

[19] ‘Event Sourcing’, *martinfowler.com*. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html. [Accessed: 13-Aug-2019].

[20] ‘Open-sourcing Kafka, LinkedIn’s distributed message queue’. [Online]. Available: http://blog.linkedin.com/2011/01/11/open-source-linkedin-kafka/. [Accessed: 14-Aug-2019].

[21] ‘Apache Kafka - Kafka Streams documentation’, *Apache Kafka*. [Online]. Available: https://kafka.apache.org/documentation/streams/core-concepts. [Accessed: 08-Sep-2019].

[22] G. Wang, ‘Enabling Exactly Once in Kafka Streams’, *Confluent*, 13-Dec-2017. [Online]. Available: https://www.confluent.io/blog/enabling-exactly-once-kafka-streams/. [Accessed: 15-Aug-2019].

[23] ‘Spring Framework Overview’. [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html#overview. [Accessed: 17-Aug-2019].

[24] J. Kreps, ‘Why Avro for Kafka Data?’, *Confluent*, 25-Feb-2015. [Online]. Available: https://www.confluent.io/blog/avro-kafka-data/. [Accessed: 17-Aug-2019].

[25] ‘Maven – Welcome to Apache Maven’. [Online]. Available: https://maven.apache.org/. [Accessed: 18-Aug-2019].

[26] ‘Streams Concepts — Confluent Platform’. [Online]. Available: https://docs.confluent.io/current/streams/concepts.html. [Accessed: 21-Aug-2019].

[27] ‘KTable (kafka 2.3.0 API)’. [Online]. Available: https://kafka.apache.org/23/javadoc/org/apache/kafka/streams/kstream/KTable.html. [Accessed: 08-Sep-2019].

[28] ‘KSQL — Confluent Platform’. [Online]. Available: https://docs.confluent.io/current/ksql/docs/index.html. [Accessed: 26-Aug-2019].

Attachment A – List of figures, tables and listings

Figures

Tables

Listings