2022 Third International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE) | 978-1-6654-5664-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSTCEE56972.2022.10099533

# A Survey of Saga Frameworks for Distributed Transactions in Event-driven Microservices

Krishna Mohan Koyya
School of Computing and Information Technology
Reva University
Bengaluru, India
0000-0002-8265-6315

Dr. B Muthukumar
School of Computing and Information Technology
Reva University
Bengaluru, India
muthukumar.b@reva.edu.in

***Abstract*—Event-driven microservices use events for inter-service communication. Such an asynchronous communication model preempts distributed monoliths and allows microservices to scale well and evolve autonomously. Typically, a message broker is employed to carry the events across the participating microservices. However, it is impractical for the event-driven microservices to accomplish the distributed transactions using traditional 2-phase commits. Hence, the microservice architecture advises implementing a distributed transaction as a saga of local transactions. Though a saga does not offer the automaticity and consistency of a traditional transaction, it indeed offers eventual consistency. The saga takes the form of either orchestration or choreography depending on the number of microservices that intend to collaborate. Sagas are used across various business domains such as e-commerce, banking, social media, logistics, etc. However, saga-based transactions require careful design of the individual microservices to take care of their interactions and error handling. It is even more tedious to implement Sagas in event-driven microservices because of the asynchronous nature of the interactions. Thus, it is always suggested to use a well-tested framework for implementing the sagas instead of building them from scratch. This paper surveys some of the available frameworks for saga implementation across three popular platforms namely Java, Python, and NodeJS. This survey finds the need for a vendor-agnostic abstraction for separating the transaction layer from the business layer.**

***Keywords—Event-driven Microservices, Distributed transactions, Eventual consistency, Saga, Orchestration***

## I. Introduction

Ever since the software made a transition from being a product to a service in the early 2000s, the demand for scalability has been growing relentlessly. As a consequence, the rigid monolith systems gave way for sets of collaborating microservices.

A microservice is a relatively smaller service that directly maps to a bounded context in the domain-driven design. Most modern organizations move from monolith architecture to microservice architecture as it enables them to address the TTM (Time To Market) requirements. Organizations often deploy microservices as containers on physical or virtual machines. Since it is easy to build and deploy container images, organizations usually offer their services as a mesh of tens and hundreds of collaborating microservices. It is also very common for organizations to deploy several instances of a given microservice for web scaling, load balancing, data replication and etc., Teams often revise, remove or add microservices as an ongoing process in response to market demands.

As the application intelligence is spread across multiple microservices, most of the transactions tend to spread across several services. Implementing such distributed transactions synchronously using the REST interface defeats the very purpose of microservice architecture as synchronous transactions lead to long-term locks and thereby the infamous distributed monolith. Hence a protocol like 2PC is not a choice in the context of microservices.

Garcia-Molina and Salem [1] initially proposed the Saga pattern to solve the problem of distributed transactions in microservices. A saga is a series of several transactions that are executed locally in the participating services. This enables the services to hold only short-term local locks.

As long as the services interact with each other using a non-blocking event-based communication model, the performance and scalability are not impacted by the saga unlike in the case of 2PC. Also, it takes less effort to reconfigure a saga by introducing or removing services without impacting others. Such microservices that are asynchronous in nature are referred to as Event-driven Microservices.

Building event-driven microservices is a tedious task as it involves a careful design of the communication protocol among the services and mechanisms to handle the errors as part of Sagas. Normally teams look for well-tested and well-documented frameworks to address such common needs.

The aim of the current work is to conduct a qualitative analysis of the available frameworks for implementing asynchronous distributed transactions using the Saga pattern on three popular microservice platforms namely Java, Python, and NodeJS.

The rest of the paper is organized into four sections. In Section II we describe the study methodology and derive the specific questions for which we want to obtain answers. In section III we present a systematic literature survey. In section IV we offer a reference architecture for event-driven microservices with the help of a use case. In section V we present the findings of the survey along with a note on future work.

## II. Methodology

In this section, we describe the methodology we followed for studying the Saga frameworks for event-driven microservices and enumerate the specific questions for which we wanted to find answers by the end of the study.

### *A. Study Methodology*

Our study methodology comprises three distinct stages. In the first stage, we conducted a systematic literature survey to get deeper insights into the concepts and issues associated with the subject matter. We referred to various papers and publications to understand the nature of distributed transactions in the microservices and the application of the Saga pattern.

In the second stage, we developed a reference problem statement. For this, we gathered the case studies and the problems reported by the industry practitioners, primarily from online resources such as StackOverflow and newsletters such as InfoQ. We derived a use case that represents the common concerns of event-driven microservice architecture as a reference problem statement. We implemented this architecture on all three platforms by using Apache Kafka with the necessary configuration. We built the service ourselves instead of using any Saga framework. This gave us an opportunity to set the expectations for the frameworks.

In the third and last stages, we conducted a survey of the frameworks in the context of the expectations derived in the second stage. As part of the survey, we chose popular frameworks on each of the platforms that help in implementing the reference problem statement. We analyzed the merits and demerits of each of them. In order for this analysis, we referred to the available user-facing documentation of each of the frameworks, such as release notes, user guides, API documentation, case studies, white papers, etc.

### B. *Specific Questions*

The goal of this survey is to get answers to the following specific questions:

1) *What are the Saga frameworks available on each of the platforms?* We chose Java, Python, and NodeJS as the platforms since they are consistently popular in enterprise computing where microservices are widely practiced. We limited the scope of the survey only to event-driven frameworks.
2) *What are the merits and demerits of each of the frameworks?* We analyzed the frameworks from a holistic perspective instead of just looking from the window of Sagas. Since microservices evolve autonomously and tend to be polyglots, the chosen framework also must be conducive to such evolution.
3) *How easy is it to migrate from one framework to another framework?* As services use a plethora of tools and technologies and they get revised frequently in the case of microservices, we want to see to what degree the design and code are coupled to the Saga framework.

We present the answers that are obtained for the above questions in section V.

## III. Systematic Literature Survey

Guogen Zhang et al. [2] proposed the GRIT protocol to address the limitations of 2PC. The protocol demands traditional relational database management across all the participating microservices and enforces ACID transactions across all of them. However, expecting RDBMS across all the services is impractical since independent microservices choose different database management technologies that may not offer ACID transactions, even locally.

PanFan et al. [3] proposed 2PC* protocol for overcoming the limitations of 2PC by adopting asynchronous collaboration. The protocol aims for BASE eventual consistency and works for polyglot microservices. However, the protocol limits itself to thread-based RPC. The microservices architecture that follows the characteristics of the classic 12-factor app chooses to model the services as independent processes, instead of threads. Also, the RPC model deters the independent evolution of microservices.

Larrucea, Xabier, et al. [4] addressed distributed transactions in microservices at a general level while explaining several strategies for migrating from monoliths to microservices architecture. They suggested the Saga pattern to achieve eventual consistency in the microservices.

Michael Müller [5] compared three approaches namely shared databases, orchestration and choreography. The paper discouraged the use of shared databases for consistency as it compromises the autonomy of the microservices.

Oliveira Rocha, Hugo Filipe [6] discussed the reasons for avoiding strong consistency and transactions based on two-phase commit in distributed event-driven systems. The work explained the orchestration and choreography as alternatives to ACID transactions at length. The authors preferred choreography to orchestration as the default approach.

Malyuga, Konstantin, et al. [7] explained orchestration and choreography and proposed optimization opportunities in the case of orchestrating idempotent services. An idempotent service is a stateless service that does not result in any inconsistency even if an update is attempted multiple times. The work demonstrated the implementation in the environment of Java and Axon.

Christudas, Binildas [8][9][10] explained sagas and demonstrated distributed transactions using Axon on the SpringBoot platform. The work includes several case studies in the area of e-commerce and enterprise applications.

Dürr et al. [11] and Martinˇ Stefanko et al. [12] conducted surveys of available frameworks to implement sagas on the Java platform. The work covered Netflix Conductor, Eventuate, and Axon frameworks.

Limón, Xavier, et al. [13] proposed the SagaMAS framework for a multi-agent system to implement distributed transactions as an abstraction layer. The framework correctly abstracted the transactions to a separate layer but did not consider eventual consistency.

Gatev, Radoslav [14] proposed an event-driven runtime that addresses distributed transactions for cloud deployment by implementing the Kubernetes sidecar pattern. Though Kubernetes is a popular deployment platform for microservices, the proposed runtime is not useful in non-Kubernetes deployments.

Rudrabhatla, Chaitanya K [15] presented a quantitative analysis of orchestration and choreography. The paper recommended using choreography when the number of participating microservices is less for better performance whereas using orchestration in the other cases.

Alulema, Darwin, et al. [16] proposed a DSL along with a graphic editor to create software tools for integrating IoT systems with SOA-based web services. The generated software tools use an asynchronous communication mechanism to connect the IoT endpoints with the REST API.

Indrasiri, Kasun, and Prabath Siriwardena [17] discussed various data-management approaches in distributed microservices including Event Sourcing, Sagas and etc., This work modeled a Saga as a Directed Acyclic Graph and used Saga Execution Coordinator as the basic constructs. This is in line with the orchestration pattern of Sagas. The work also compared various frameworks like Dropwizard, Vert.x, Spring Boot, Apache Camel and etc., in implementing the microservice integration.

Nair, Vijay [18] mapped Domain-Driven design to Microservices and demonstrated the Axon framework in Saga implementation besides the other frameworks like Spring, Eclipse MicroProfile, and Jakarta EE. Other data-management techniques like CQRS and Event Sourcing are also covered in implementing the Sagas.

Xue, Gang, et al. [19] proposed a Saga-based coordination mechanism for the decentralized composite microservices for reaching consensus at runtime. This work explained mechanisms to implement compensatory transactions within a Saga, namely forward recovery and backward recovery. However, the actual implementation of these mechanisms is left to individual applications.

Ramírez, Francisco, et al. [20] extracted and discussed issues related to microservices development from StackOverFlow. The absence of configuration parameters for cache and inadequate patterns for long-running transactions emerge as trending concerns in service invocation.

Rajasekharaiah, Chandra [21] explained different ways to handle errors while executing Sagas namely (i) ignoring them, (ii) compensating them inline, and (iii) compensating them offline.

Megargel, Alan, Christopher M. Poskitt, and Venky Shankararaman [22] discussed the merits and demerits of choreography and orchestration in terms of coupling, chattiness, visibility, and design. They developed a framework to decide between either of the patterns in implementing the Sagas. They also suggested a hybrid of both patterns, when the decision framework fails to choose one of them.

From the literature survey, it is evident that the event-driven Sagas are critical to the eventual consistency in the microservice architecture and there are several frameworks available to address the needs of Sagas. This paper aims to find the merits and demerits of the popular frameworks.

For this purpose, we developed a reference architecture based on the learnings from the above literature survey which is explained in the next section.

## IV. Reference Architecture

We chose to architect an order processing service that is found in a typical e-commerce system. The usual workflow of the system can be summarized as follows:

- Customer searches for the products. The system offers the list of products along with the necessary attributes like price, ratings, specifications and etc.,
- The customer places an order by choosing a product and quantity along with other details like delivery address and etc.
- The order management system reserves the product for the customer and processes the payment. In some systems, it involves third-party payment gateways. It is also common for the e-commerce system to offer prepaid wallets.
- Once the payment is successful, the system gives confirmation to the customer along with an order ID and follows the rest of the fulfillment process.
- In case the payment fails, the system releases the reserved products to the available inventory and informs the customer about the failed order.

The inventory data, customer data, wallet information, etc are maintained in one database. And the service takes the responsibility for inventory management, order management, and the payment process besides other responsibilities.

Organizations find it difficult to maintain such monoliths and follow the domain-driven approach to decompose them into a bunch of microservices.

The traditional synchronous microservices offer REST endpoints and register themselves with a central Registry in the infrastructure. This enables the other services in the same infrastructure to discover and consume the services via HTTPS protocol for collaboration. Overall, these services are exposed to the customers via a well-configured Gateway that handles security, routing, throttling, protocol handling, etc. This setup gives a consistent view to the end users by hiding the internal complexities from them.

We decomposed the monolith e-commerce system into multiple microservices of which Order Service, Inventory Service, and Wallet Service are of importance for this discussion. Each of these services is developed autonomously with its own data management and scope of work as follows:

- The Order Service offers REST API for creating a new order and finding the details of an existing order.
- The Inventory Service offers REST API for reserving and releasing the products against an order.
- The Wallet Service offers REST API for crediting and debiting against a customer ID.

The customer only interacts with the Order Services whereas the Order Service collaborates with the Inventory Service and Wallet Service within the infrastructure.

We find that the order creation scenario is a perfect case for implementing the Saga pattern as it involves all three services in the transaction. For this purpose, we introduced an Orchestrator service as depicted in Figure 1. A Saga orchestrator commands the other services to do local transactions and maintain the state of the Saga in its own database. It also commands the services to execute compensatory transactions when required.

However, the limitation of this architecture is that the interactions among the services are purely synchronous because of the nature of HTTP REST APIs.

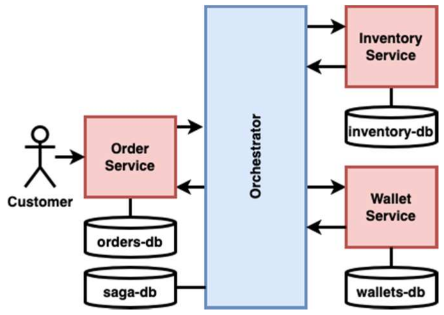

Fig. 1. Orchestrated Microservices

We refactored the architecture along the lines of event-driven microservices to overcome this limitation. The internal collaboration that was based on HTTP REST is replaced with Events. This paves the way for the services to collaborate asynchronously in a non-blocking way. Apache Kafka handles the reliable message passing between the orchestrator and the individual microservices. The messages carry both commands as well as events. The queries are left to the REST API to handle.

The updated reference architecture based on the events is depicted in Figure 2 and the collaborations are summarized as follows:

- The customer places an order with the Order Service through an HTTP POST.
- The Order Service immediately creates the order in the PENDING state, saves the order in the orders-db, and returns the Order ID to the customer. Also, it sends a command via saga.commands topic for further processing.
- The Orchestrator listens to the saga.commands topic and picks up the commands. As part of event handling, it writes a command on the topic inventory.commands to reserve the inventory.
- The Inventory Service listens to the inventory.commands and picks up the commands. As part of the event handling it either reserves or releases the products from the inventory, depending on the command and updates the database. It also writes the results into the topic inventory.results.
- The Orchestrator listens to the inventory.results topic for the events. If the event indicates a successful reservation, it writes a command to the wallet.commands for debiting the wallet. Otherwise, if the event indicates an unsuccessful reservation or a successful release, it writes an event in the saga.results indicating unsuccessful order processing.
- The Wallet Service listens to the wallet.commands topic for the incoming commands and either debit or credit the wallet accordingly. It also updates the database locally and writes the results of the operation into the wallet.results topic.
- The Orchestrator listens to the wallet.results topic for the events. If the event indicates successful debit, it writes an event in the saga.results indicating successful order processing. Otherwise, it writes a command on the inventory.commands topic to release the reserved products.
- The Order Service listens to the saga.results topic for the events. If the event indicates successful processing of the order, it updates the order status from PENDING to CONFIRMED in the database. Otherwise, it updates the order status from PENDING to FAILED.

When the customer queries for order status via REST API, the Order Service responds based on the current status of the order. This ensures eventual consistency.

We implemented this architecture on all three platforms by using Apache Kafka with the necessary configuration. We built the Orchestrator service instead of using any Saga framework. This gave us an opportunity to understand the demands of the implementation and set the following expectations on the frameworks.

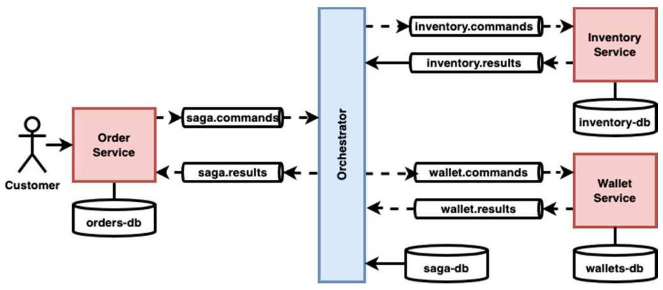

Fig. 2. Event-driven Microservices with Orchestrator

- The framework should support microservices developed on any of the three chosen platforms.
- The framework should support Saga in the form of orchestration and via events.
- The framework should support Saga in the form of choreography via events.
- The framework should support a pluggable message broker including but not limited to Apache Kafka.
- The framework should not intervene in the domain model i.e. the Saga must be abstracted out from the domain layer.
- The framework should support the declarative definition of the Saga apart from imperative definitions.
- The framework should be able to integrate with the third-party service endpoints even if they are not event-driven.

We surveyed various frameworks available for implementing Sagas in the microservice environment to meet the above expectations. However, we are aware of the fact that no framework may meet all the expectations.

## IV. RESULTS

The general observation is that a good number of frameworks are available on the Java platform. Since Java has been a leader in enterprise computing for over two decades, it's no wonder that it boasts a rich Eco-system of frameworks and libraries for microservices as well. There are a handful of Saga frameworks available though they differ in the degree to which they meet our expectations. While Python and NodeJS are also popular in microservices development, we found that there are not many frameworks available for implementing the Sagas.

Here we present the list of frameworks surveyed along with their merits and demerits, on each of the platforms.

### *A. Axon*

The Axon [23] is a Java-based event-driven framework for implementing both orchestrated and choreographed sagas. However, it is suitable only for applications that are architected along the lines of the Command Query Responsibility Segregation (CQRS) pattern. This characteristic is its strength as well as weakness. While Axon is good for applications whose domain layer is modeled as commands and query aggregates, it becomes an overhead for the applications that are not following CQRS.

Axon offers implementations for aggregates, repositories, and concurrent event handling. The framework uses the Event Sourcing mechanism as well. In other words, plugging other message brokers that are not suitable for event sourcing is not

| S. No. | Framework/ Feature | Platform | Orchestration | Choreography | Design Model | Event Bus | Persistance | DSL Support | EIP Support |
|---|---|---|---|---|---|---|---|---|---|
| 1 | **Axon** | Java | Event-driven | Event-driven | CQRS | In-built | In-built | NO | NO |
| 2 | **Eventuate Tram** | Java | Event-driven | Event-driven | CQRS | Apache Kafka | MySQL | NO | NO |
| 3 | **Narayana LRA** | Java | REST | NO | NO | NO | JPA | NO | NO |
| 4 | **Netflix Conductor** | Java | Event-driven | Event-driven | NO | dyno-queues | dynamite | JSON | NO |
| 5 | **Spring Boot** | Java | NO | NO | NO | NO | NO | NO | NO |
| 6 | **Vert.x** | Java | NO | NO | NO | NO | NO | NO | NO |
| 7 | **Apache Camel** | Java | REST | NO | NO | NA | NO | XML | YES |
| 8 | **Seata** | Java | Event-driven | NO | NO | In-built | NO | NO | NO |
| 9 | **saga-framework** | Python | Event-driven | NO | NO | Celery | Sqlalchemy | NO | NO |
| 10 | **node-saga** | Nodejs | REST | NO | NO | NA | NO | NO | NO |

Table 1. Qualitative comparison of Saga frameworks

recommended, though Kafka integration is supported to some extent.

From the development perspective, Axon offers annotations for marking the start and end of the Saga and writing the event handlers. This helps in decoupling the business domain from Saga plumbing to some extent. On the downside, Axon is of little help in terms of building the life cycle of the Sagas.

### B. *Eventuate Tram*

Eventuate Tram [24] is also a Java-based event-driven framework that is well-adopted in the industry for distributed data management. Like Axon, Eventuate Tram is also good if the whole system is architected based on CQRS and Event Sourcing patterns. The Tram uses Apache Kafka for messaging and MySQL for implementing transactional outboxing to ensure that the state is updated and the event is published atomically. If the system is already using Apache Kafka for messaging needs, Eventuate just fits into the architecture, unlike Axon which comes with its own eventing bus.

Eventuate is more developer friendly because of its builder model to define the Saga with the actions, compensatory actions, and reply definitions. The downside of the Tram is that developers are expected to write a good number of classes to represent each of the entities, commands, and events and it often leads to duplicate code or forced to use a shared library.

### C. *Narayana LRA*

This is a Java-based library compliant with Eclipse Microprofile LRA specification [25]. The library offers multiple modes to execute distributed transactions including Saga besides JTA, etc. However, Narayana LRA implements the Saga orchestration only using REST-based synchronous communication and does not have any support for event-based Sagas.

### D. *Netflix Conductor*

Being one of the first movers and influencers in the world of microservices, Netflix has open sourced its orchestration engine namely Conductor [26] a few years ago. The Netflix Conductor is developed on Java. However, it offers services to Python clients as well. The framework uses Dynamite for persistence and dyno-queues for event queuing. The Conductor offers a JSON-based DSL for defining orchestration along with a UI to visualize the state of the execution. The framework is a natural choice for Spring Boot applications.

### E. *Spring Boot*

Possibly the most popular framework for Java developers, Spring Boot [27] is a general-purpose framework that offers fundamental and critical features like dependency injection and aspect weaving. Its integration with a host of tools and technologies like databases, message brokers, and monitoring tools forms an ecosystem for enterprise applications. It offers both imperative as well as reactive programming models. Though Spring Boot does not have any inbuilt support for Sagas, the frameworks like Axon, Tram, and Conductor that are built on top of Spring Boot helps the developers in building event-driven microservices.

### F. *Vert.x*

Based on the Reactor pattern, Vert.x [28] offers an ingenious reactive programming model to develop event-driven microservices on the Java platform. However, there is no direct support for Saga orchestration.

### G. *Apache Camel*

In the space of Enterprise Integration, Apache Camel [29] has been a proven framework on the Java platform, for over a decade. It offers both a programming model as well as DSL for defining the integration flows based on EIP. Camel supports Saga as an integration pattern and uses Eclipse Microprofile LRA libraries underneath. Consequently, it supports only REST-based orchestration.

### H. *Seata*

Widely used at Alibaba, the Seata framework [30] is a Java framework for handling distributed transactions in various models including Saga. It is implemented based on a state machine engine and uses an inbuilt mechanism for eventing.

### I. *saga-framework*

The only framework on the Python platform that supports Sagas is the saga-framework [31]. It was released in 2021 and did not find much adoption at the time of reporting this. However, the framework offers a nice environment for developing event-driven Saga orchestration for Python developers along the lines of Eventuate Tram. The framework uses Celery for event queues.

### *J. node-sagas*

The support for distributed transactions in the form of Saga is almost absent in the Node environment till the node-saga [32] was released in 2020. However, the adaptation of the framework is still slow as it is lacking most of the features required by an event-driven architecture.

Though there are quite a few other frameworks available, primarily on the Java platform, they are not as widely used as the frameworks which we explained in this section. A summary of the survey is depicted in Table 1.

We found that almost all the frameworks influence the domain model of the microservices which is not a very good sign as it leads to vendor locking.

## V. Conclusion and Future Scope

Event-driven microservices offer scale and at the same time, they tend to be complex in terms of distributed transaction management. Though the Saga as a pattern is well documented, the need for a framework to implement the Saga pattern is more profound in event-driven microservices.

This survey reports a good number of frameworks on the Java platform. Python and NodeJS do not have many promising frameworks.

Even in the case of the Java platform, it is found that different frameworks force different architectural models on the participating microservices. Because of this, it is almost impossible to move from one framework to another.

The microservices architecture encourages polyglot databases and platforms. The system design should be driven by the domain rather than by the frameworks. Such a design proves to be stable and extensible.

From this survey, we found that the current scenario is not catering to this need in implementing asynchronous distributed transactions because of incompatible vendor-specific frameworks. We believe that there is a need for framework-agnostic, vendor-agnostic, and platform-agnostic ways of expressing the Sagas.

As part of our future work, we intend to research the possibility of building a lightweight declarative Saga abstraction for event-driven microservices. The expected abstraction should enable implementations across various platforms without forcing architectural constraints on the domain model.

## References

1. Garcia-Molina, Hector, and Kenneth Salem. "Sagas." ACM Sigmod Record 16.3 (1987): 249-259.
2. Guogen Zhang, Kun Ren, Jung-Sang Ahn, Sami Ben-Romdhane, “GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases”, IEEE, ISBN: 978-1-5386- 7474-1, 2019
3. Pan Fan, Jing Liu1, Wei Yin, Hui Wang, Xiaohong Chen and Haiying Sun,”2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform”, Journal of Cloud Computing: Advances, Systems and Applications,2020
4. Larrucea, Xabier, et al. "Microservices." IEEE Software35.3 (2018): 96-100.
5. Michael Müller, Consistency and Autonomy in the Microservice Architecture,(IJACSA) International Journal of Advanced Computer Science and applications, Vol. 9, No. 8, 2020
6. Oliveira Rocha, Hugo Filipe. "Structural Patterns and ChainingProcesses." Practical Event-Driven Microservices Architecture(2022): Apress with Springer, 133-186.
7. Malyuga, Konstantin, et al. "Fault-tolerant central saga orchestrator in RESTful architecture." 2020 26th Conference of Open Innovations Association (FRUCT). IEEE, 2020.
8. Christudas, Binildas. "Transactions Optimized for Microservices." Practical Microservices Architectural Patterns. Apress, Berkeley, CA, 2019. 543-587.
9. Christudas, Binildas. Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud. Apress, 2019.
10. Christudas, Binildas. "Axon Microservices and BASETransactions." Practical Microservices Architectural Patterns. Apress, Berkeley, CA, 2019. 779-812.
11. Dürr, Karolin, Robin Lichtenthäler, and Guido Wirtz, "AnEvaluation of Saga Pattern Implementation Technologies." ZEUS.2021.
12. Martinˇ Stefanko, Ondˇrej Chaloupka and Bruno Rossi, "The Saga pattern in a reactive microservices environment." Proc. 14th International Conference. Software. Technologies (ICSOFT 2019). Prague, Czech Republic:SciTePress, 2019.
13. Limón, Xavier, et al. "SagaMAS: a software framework for distributed transactions in the microservice architecture." 20186th International Conference in Software Engineering Research and Innovation (CONISOFT). IEEE, 2018.
14. Gatev, Radoslav. "Introduction to Microservices."Introducing Distributed Application Runtime (Dapr). Apress, Berkeley, CA, 2021. 3-23.
15. Rudrabhatla, Chaitanya K. "Comparison of event choreography and orchestration techniques in microservice architecture."International Journal of Advanced Computer Science and Applications 9.8 (2018): 18-22.
16. Alulema, Darwin, et al. "A model-driven engineering approach for the service integration of IoT systems." Cluster Computing 23.3 (2020): 1937-1954.
17. Indrasiri, Kasun, and Prabath Siriwardena. "Microservices for the Enterprise." Apress, Berkeley (2018). pp 143-148 and pp167-217.
18. Nair, Vijay. "Cargo Tracker: Axon Framework." PracticalDomain-Driven Design in Enterprise Java (2019): 277-372. Apress, Berkeley, CA
19. Xue, Gang, et al. "Reaching consensus in decentralized coordination of distributed microservices." Computer Networks187 (2021): 107786.
20. Ramírez,Francisco, et al. "An Empirical Study on Microservice software development." 2021 IEEE/ACM Joint 9th International Workshop on Software Engineering for Systems-of-Systems and 15th Workshop on Distributed Software Development, Software Ecosystems and Systems-of- Systems (SESoS/WDES). IEEE, 2021.
21. Rajasekharaiah, Chandra. "Microservices: What, Why, and How?." Cloud-Based Microservices, Springer (2021): 62-65
22. Megargel, Alan, Christopher M. Poskitt, and Venky Shankararaman. "MicroservicesOrchestration vs. Choreography: A Decision Framework." 2021 IEEE25th International Enterprise Distributed Object Computing Conference(EDOC). IEEE, 2021.
23. Axon Reference Guide. [Online] URL: https://docs.axoniq.io/reference-guide/
24. Eventuate Tram Manual. [Online] URL: https://eventuate.io/docs/manual/eventuate-tram/latest
25. Narayana LRA Documentation. [Online] URL: https://www.narayana.io//docs/project/index.html
26. Netflix Conductor Documentation. [Online] URL: https://conductor.netflix.com/architecture/overview.html
27. Spring Boot User Guides. [Online] URL: https://spring.io/guides
28. Eclipse Vert.x Documentation. [Online] URL: https://vertx.io/docs/
29. Apache Camel Saga. [Online] URL: https://camel.apache.org/components/3.18.x/eips/saga-eip.html
30. Seata Documentation. [Online] URL: https://seata.io/en-us/docs/overview/what-is-seata.html
31. Saga-framework [Online] URL: https://github.com/absent1706/saga-framework
32. Node-sagas [Online] URL: https://www.npmjs.com/package/node-sagas