International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 05 | May 2020 www.irjet.net p-ISSN: 2395-0072

Microservices, Saga Pattern and Event Sourcing: A Survey

Repana Reddy Sekhar[1], Prof. Veena Gadad[2]

[1]*Student, Dept. of CSE, R V College of Engineering, Bangalore*
[2]*Assistant Professor, Dept. of CSE, R V College of Engineering, Bangalore*

Abstract - *Microservices are new architectural model in which we can arrange an application into independent services, where each microservice consists of small, autonomous services, implementing a single business capability. Most of the applications can be seen as set of business or logical rules that can be executed when triggered when an event internal to the system happens or user inputs. Sagas are sequence of local transactions. In order to decode what is discussed in the microservice and saga pattern we found that 25.68% of microservice technical posts on saga pattern discuss a single technology: Redhat and microprofile. Event sourcing lets us to design our applications in such a way that we can store the past states of business entities , enabling the system to replay old events at will. This paper deals with most important aspects of today's modern world of IT that is MICROSERIVES, SAGA PATTERN AND EVENT SOURCING. We provide an overview of the concepts and historical trends.*

***Key Words*: Microservices, Saga pattern, Event Bus, Event sourcing, Docker, Transaction, Kafka.**

1. INTRODUCTION

In the traditional monolithic architecture, a software application is built as a single unit that combines several services in order to provide business functionalities. Although monolithic applications are simple to be developed, they have many limitations, such as difficulty in evolving, maintaining and scaling computational resources, which may cause an over (under)provisioning of those resources. With the advance of cloud computing and container technologies, Microservices have recently emerged as a new architectural style to break up distributed applications into small independently deployable services, each running in its own process and communicating via lightweight mechanisms [1]. These services are built around separate business capabilities and can be written using different programming languages and different data storage technologies. They are usually supported by a fully automated deployment and orchestration machinery, e.g., in the cloud, enabling each service to be deployed often and at arbitrary schedules, with a bare minimum of centralized management [2].

As a result with microservices architecture, there are number of challenges that comes with. Each independent microservice has its own database and whenever a business transaction involves so many services, so we need to have mechanism to ensure data consistency across services. Implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database in its service and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions [3].

For debugging and storing all the events in the system, event sourcing lets us to design our applications in such a way that we can store the past states of business entities, enabling the system to replay old events at will. Enabling us to track all the transactions as a sequence of events which are easy debug and take necessary actions.

This paper discusses the concepts and challenges of microservices, saga pattern and event sourcing. The rest of the paper is organized as follows. The next section gives overview of microservices, saga pattern and event sourcing. Section III presents challenges in implementing these and analysis of these implementation. Finally, Section IV offers some conclusions.

2. MICROSERVICES, SAGA PATTERN AND EVENT SOURCING: Concepts, Benefits and Drawbacks

A.*Microservices*

Microservices are new architectural model in which we can arrange an application into independent services, where each microservice consists of small, autonomous services, implementing a single business capability, which allows developers make it easier to understand, develop and test the application. Each small and independent service has its own database. Decentralizing the application into smaller individual service enables us the leverage to select any suitable technology and frameworks and deploy individually into the server.

"The characteristics of microservices architecture are:

1) Flexibility: A system is flexible to support all the required features and has flexibility in choosing the programming language .
2) Modularity: Each service is independent and can be developed independently which can be arranged with other service to achieve the overall behaviour of the system.

The key benefits of microservices are:

a) Scalability: We can scale and deploy each service independently leaving the rest of the services unaffected unlike in monolithic application.

b) Fault isolation: Failure in one service does not allow to collapse the whole system. If one service fails others are still available.

c) Decentralized data management: Each service has its own database and effective data management can be easily carried out using microservice.

d) Reusability: Reusability of functionality is the key benefit of the microservices.

Historical trends of using microservices:

Netflix is one of the earliest adopters of microservices, and one of the most discussed. Netflix started to adopt microservices architecture in 2009, when this approach wasn't known at all. They used Amazon Web Services platform to set up their microservices. They progressed in steps moving movie encoding and other non-customer facing applications. Then they decoupled all other services, which took almost 2 years to split their monolithic application to microservices. Now they have about 500+ microservices and API gateways that handles more than 2 billion requests everyday.

Spotify was looking for a solution that could scale to millions of users, support multiple platforms and handle complex business rules. Currently Spotify has over 800+ services and became less susceptible to large failures.

Uber, when it was just entering the market built their solution for a single offering in a single city. But as the company expanded, uber's system faced problems with scalability and integration as its system was based on monolith architecture. Then Uber decided to move their system into microservices architecture.

eBay, In 2011, when the company had 97 million active users and 62 billion gross merchandise volume, was moving to microservices. Everyday, the eBay systems had to deal with heavy traffic, like 70+ billion database calls and 250 billion search queries. So they thought dividing the system and introducing microservices will address all the challenges they faced. Along with other microservices leaders, they released open-source solutions for the developer community.

Zalando, with their Magento-based eCom-merce system could not handle high load. The company was desperate need of new infrastructure. Their decoupling of system started in 2015, where they switched to microservices creating a whole new working culture and increased their productivity and added so many innovations.

B. Saga pattern

A saga is a sequence of local transactions where each transaction updates data within a single service. The first transaction is initiated by an external request corresponding to the system operation, and then each subsequent step is triggered by the completion of the previous one.

Transaction ACID properties:

Atomicity: Transaction either completes fully or fails completely.

Consistency: Maintain the logical consistency, no matter of its final outcome.

Isolation: Multiple transactions can occur concurrently without causing inconcistency of database state.

Durability: The changes of a successful transaction occurs even if the system failure occurs.

Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.[Wikipedia]

Saga has ACD characteristics, distributed transactions ACID characteristics the real challenge is to deal with the lack of isolation in an elegant and effective way. A transaction in a microservice architecture should be eventually consistent. Compensations are the actions to apply when a failure happens to leave the system in an inconsistent state. Compensations actions must be idempotent; they might be called more than once.

Types of saga pattern

The two ways to perform sagas:

Choreographic sagas: In this type, the domain events act as triggers. The first transaction is initiated by an external request or user's input, then each local transaction publishes domain events into event bus that will trigger local transactions in other services.

Benefits :

- This is a natural and easy way of implementing saga pattern.
- This is easy to understand and will not require too much effort to develop.
- All the services are loosely coupled so it does not violate the principle of microservices.

Drawbacks :

- As the number of services increases or local transactions increase, this method become more complex and cumbersome because it can have cyclic dependency between the stages of saga.
- Testing would become complex as all the participants should be running.

Orchestration sagas: an orchestrator or centralized controller tells the participants or services what local transactions to execute.

Benefits:

- We can avoid cyclic dependencies between participants.
- Complexity can be reduced as participants only have to execute and reply for the commands.

Drawbacks:

- Too much load on the orchestrator or centralized controller because we will be concentrating on the orchestrator logic.
- Increase in the infrastructure complexity because we need to add extra service.

C. Event sourcing

Event sourcing is an approach where we catch all the state changes of an application or business entity as a sequence of events. It is an excellent way to update the state of business entity and publish the events automically. The traditional way to persist the business entities is by saving the current state of the business entity. Event sourcing approach is quite different, event-centric approach to persist the state of application or business entity. Whenever an entity's state changes, a new event change is added to the existing sequence of events. Since adding event to list is atomic we serve the purpose. The business entity's state can be constructed by replaying all the events associated with it.

MongoDB as event store:

MongoDB can act as event store (database of events). Because of its NoSQL properties, we can query the mongo collections, replay the events associated with it and reconstruct the business entity's state.

Benefits:

- The details of how a business entity or object reached its current state answers so many questions like how the flow of system takes place, what are stages the key stages in the transactions, stages that need more focus, and 100% audit logging.
- Event store logs can be used for debugging and testing the system, so that we can identify issues while production and help us to understand how an entity reaches bad state so that we can avoid them.

D. Apache Kafka as an event bus

Event-driven architecture is main element of developing and designing microservices. An Eventbus is a method of communicating different services with each other without knowing about one another.

Event bus is the backbone of microservices. There are so many messaging queues available like RabbitMQ, ApacheKakfa, Amazon SQS, TIBCO EMS etc. Among all these ApacheKakfa is better because it can handle high requests per second, throughput is high and less complex to include in our system.

Apache Kafka is a unified distributed platform for handling all the real time data feeds of an organization. A distributed platform in Kafka consists of:

- Message queueing
- Message processing
- Message storage

Thus ApacheKakfa acts as event bus also.

3. Problems faced in implementation of these concepts

a) The Complexity of microservice system: Each service is now an independent service that should handle the calls involving several other services. Complexity might be because of latency on remote calls, fault tolerance etc.
b) Managing and monitoring microservice: If the number of services increase in microservices system, managing them becomes more complicated. If the services are not managed properly, things can get quite difficult leading to compromise in quality of system. So, we need to even monitor the system in a quite efficient way for spotting out the problem if system fails for some reason.
c) Difficulties on Deployment: Microservi-ces have many services which need to deployed individually and there can be complexity in deployment of services because the services may need to be deployed on different server.
d) Testing: Testing in microservices is more complex compared to monolithic applications, because we need to make sure the functionality of each service is working properly before integration of all the services. And the indistinct behaviours from microservices can be very hard to predict and point out.

Problems with saga pattern

- Dirty reads: a transaction read data from a row that is currently modified by another running transaction
- Lost updates: two different transactions trying to update the same "data".
- Non-repeatable reads: re-reads of the same record (during an inflight transaction) don't produce the same results

Saga should take actions to minimize the impact of lack of isolation. How you implement this set of countermeasures (against isolation anomalies) determine how good a microservice is. Several techniques available: semantic lock, design commutative operations.

Saga Pattern: semantic lock

PENDING states, saved into the local microservice store, indicate that a Saga instance is in progress and it is manipulating some data needing an isolation level (for example a customer's account)
If another Saga instance starts, it must evaluate the existing PENDING states and pay attention on them. Some strategies when detecting PENDING states:

- The Saga instance will fail
- The Saga instance will block until the lock is released coordination and rollout plan

Problems in implementing event sourcing and event bus.

1. Handling entities with long lifespan and complex business code needs to be handled very carefully because the number of events that needs to be

replayed to construct the business entity's state is more,

2. When and how the entity's current state should be constructed is very important question one needs to know in implementing event sourcing.
3. Implementing our own event bus is very complex, so we have to rely on some third party events busses and messaging queues. There will be compromise of our needs if we go through available approaches.

4. Conclusion

Microservices architecture is changing the way our business applications are being developed nowadays. Because it handles the issue of complexity by breaking down application into independent set of services which will much faster to build, test and deploy by which we can increase the availability of services, by which we can overcome the challenges of monolithic applications.

Transactions are an essential part of applications. Without them, it would be impossible to maintain data consistency. Maintaining data consistency in business application is very important. Particularly when you are working with microservices, things get more complicated. Each service is a system apart with its own database, and it is very hard to maintain data consistency. Saga pattern can help us maintain the data consistency among the microservices architecture efficiently.

Event Sourcing ensures that all changes to business entity's state are stored as a sequence of events. We can query an entity's state to find out the current state of the application, and this answers many questions. However there are times when we don't just want to see where we are, we also want to know how we got there.

This paper summarizes the concepts, benefits and drawbacks of microservices, saga pattern and event sourcing. This paper also presents the importance of event bus. The main challenges that developers face while implementing these concepts are also discussed. This paper makes best practices and literature to other practitioners for microservices, saga pattern and event sourcing.

ACKNOWLEDGEMENT

Prof. Veena Gadad, Assistant Professor, Department of Computer Science and Engineering, R V College of Engineering, Bangalore.

REFERENCES

[1] J. Lewis and M. Fowler, "Microservices," https://martinfowler.com/articles/microservices.html, 2014, [Online;accessed 18-January-2017].

[2] S. Newman, Building Microservices. O'Reilly Media, 2015.

[3] Chris Richardson, https://microservices.io/patterns/data/saga.html

[4] M. Fowler. (2005) Event sourcing. Visited on 2016-10-11. [Online]. Available: http://martinfowler.com/eaaDev/EventSourcing.html

[5] B. Golden. (2016). 3 Reasons Why You Should Always Run Microservices Apps in Containers. Accessed: Nov. 11, 2017.

BIOGRAPHIES

Repana Reddy Sekhar is a Final year Computer Science and Engineering Student at R V College of Engineering