Mphasis
The Next Applied

Events are facilitators of Decoupled Architecture

Whitepaper by Rajasi Balan, AVP – Enterprise Application & Integration

Contents

1.
Abstract

Microservices architectural style helps in building highly scalable, loosely coupled (or highly decoupled) applications. While there are numerous benefits of this architecture, there are few challenges around inter-service communication and data integration. Services need a seamless way of exchanging data in a (near) real-time manner. Data must always be present before it is requested.

Event-Driven Microservices provide solutions for many of these problems. This white paper focuses on common design patterns which are used in event-driven microservices for service and data integration. The patterns are illustrated using an Order application, which uses Apache Kafka's inherent design for building a resilient, reliable messaging channel. It is assumed that the Order application is comprised of multiple microservices with a defined bounded context.

2.
Why Event-Driven Microservices?

Event-driven microservices allow for (near)real-time data exchange. They enable seamless communication between different application components. We can build highly scalable, decoupled services using events as a basis of communication. Let us take a quick look at how events can be used to solve the two major challenges stated above.

Inter-service Communication

Let us take an example of modernizing a traditional monolithic application. During this transformation journey, parts of the application get transformed in microservices architecture in an incremental manner. It is impossible to modernize the entire application using a big bang approach. In the transition phase, new microservices will require to communicate with this monolithic application. Using synchronous mode of communication will create tight coupling between application components, thereby compromising the flexibility of the application.

In such situations, asynchronous communication pattern helps to build a loosely coupled architecture. Publish/subscribe, fire and forget, notifications, publish/asynchronous response are some of its common sub-patterns. Each of these sub-patterns use event-driven architecture as a backbone for communication.

Data Integration

Modern-day solutions require a single source of truth to improve customer experience. Combined with high-quality data, it is possible to leverage information more effectively and efficiently. Event-driven architecture helps to build integration pipelines that can support high-volume data. Using event streams and stream processing layer, it is possible to build scalable, reliable pipelines that combine data from disparate systems to create a unified data view.

3.

Design guidelines for Event-Driven Microservices

Let us look at some design patterns which are used to build distributed applications that are scalable, decoupled, built on reliable and flexible event-based architecture.

Events as First-Class Citizens

Problem Statement

In a traditional application, the database acts as the system of record (SoR). In a microservices architecture, each bounded context would have its own database, typically known as micro-databases. Data is always exposed through services. This works fine when services require a small subset of data, not very frequently.

Consider a situation wherein a common set of data is required by services all the time. In the example of the Order tracking application, user profile data is required by order processing and shipping service.

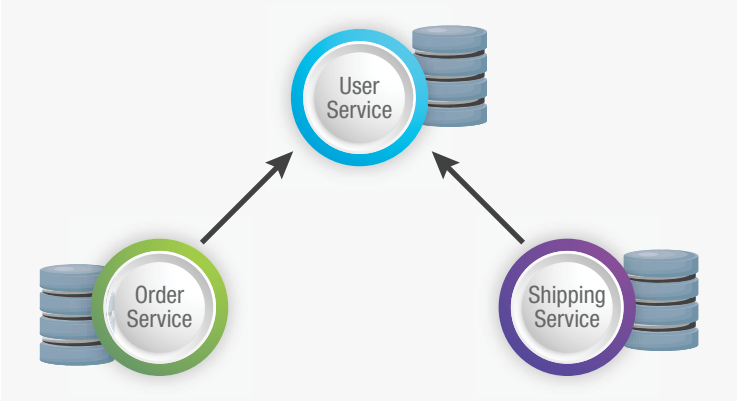

Figure 1: Traditional inter-service communication

Remote calls in such situations would prove very inefficient, adding a latency to process most of the requests. The problem multiplies as the ecosystem grows big.

As an improvement, we modified the application so that each service caches relevant data from other services in its own database. With this approach, application performance is optimized. But now we have another problem – data consistency. How do we ensure that cached data is always updated? Which caching policies to implement?

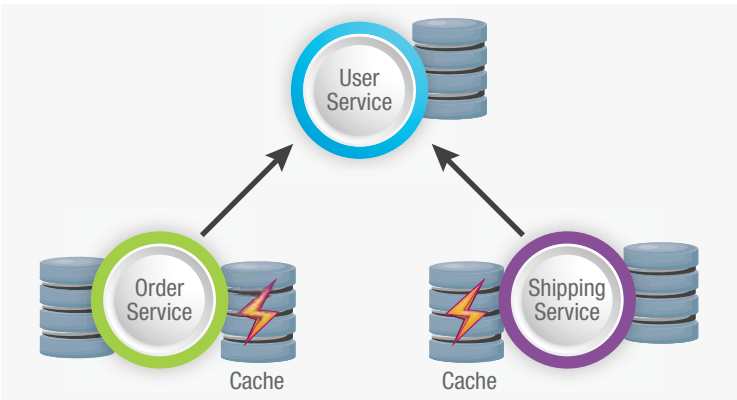

Figure 2: Data caching

Solution

Using event-driven architecture, we can build a performance-optimized, scalable solution which addresses the above problems. There are mainly two types of events – notification (notify change in state of data) or state transfer (push data from one domain to the other).

In the above scenario, the application is enhanced to use events for state transfer in addition to the caching mechanism. Since order processing and shipping service requires a subset of the user profile, they save that information in their own database. Any change in user profile information will trigger an event. Order processing and shipping service consume these events by subscribing to the relevant topics and updating user information saved in their database.

Event Collaboration

Problem Statement

In most situations, events contain all the information that is required for processing. However, there could be instances wherein these events must be enriched to create more meaningful events, what we call as converting raw events to business events.

Solution

The golden rule to remember is that events must be enriched as close to the processing layer as possible, so that every connected system can interpret events as per its own business functionality.

There are three main design patterns that can be applied for event enrichment.

1. Pure stateless implementation using data lookup

In our Order tracking application, the Order Created event is raised when an order is placed. It contains the customer id, order id, and other order details. A notification engine processes this event to generate an SMS/email notification. It first checks for the payment status by querying the Payment Service. Next, it calls customer service to look up the customer details like preferred SMS number, email id, etc. These details are added to the notification email/SMS for a complete reference.

As we can see, the event is completely stateless. It only contains the information that is captured while placing the order. One aspect to consider in this approach is that the notification service must be designed to be highly reliable. Appropriate design considerations must be given to edge situations like if customer service is down / not reachable, notifications will be queued and retried once the service is up.

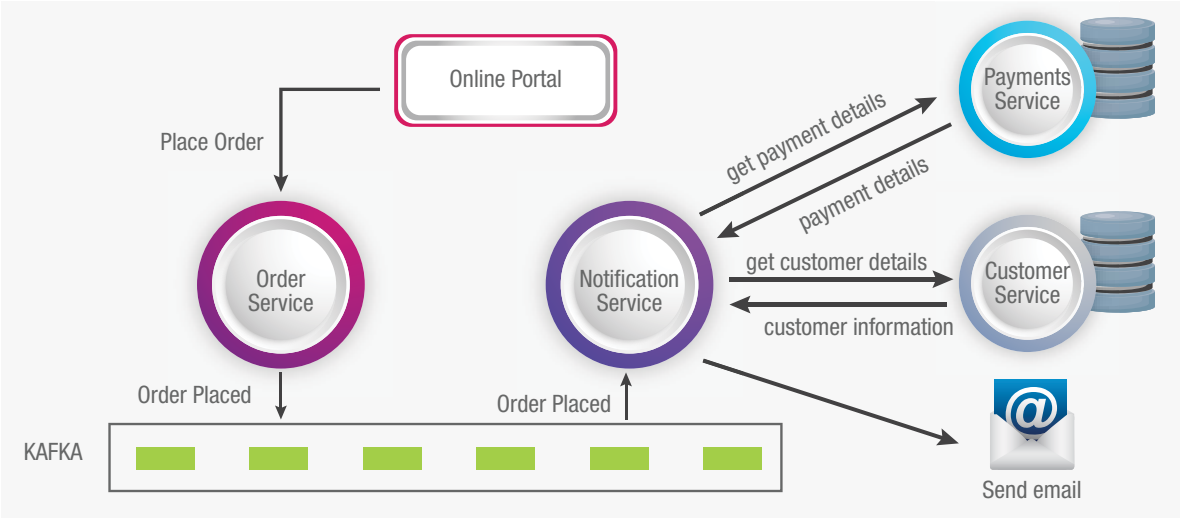

Figure 3: Pure stateless processing

2. Stateless streams using buffering

Using the same example as above, Notification Service receives events for Order Created event. In addition, it also receives a Payment Processed event. This is an additional event, which is raised by the Payment microservice, when order payment is processed. Notification service is designed to consume both these events and then act upon them as per the business rules.

However, it may happen that the order created event may reach after the payment processed event. The notification service must buffer each event in its cache for some time to handle these delays. Only when both events are received for an order, will it send a notification to the customer. As in the above pattern, it still queries the customer service to get the required information.

This pattern has reduced the number of service calls as compared to the previous option, since integration with Payment service is using events. However, there is increased complexity of buffering and handling messages that may come in out of sequence.

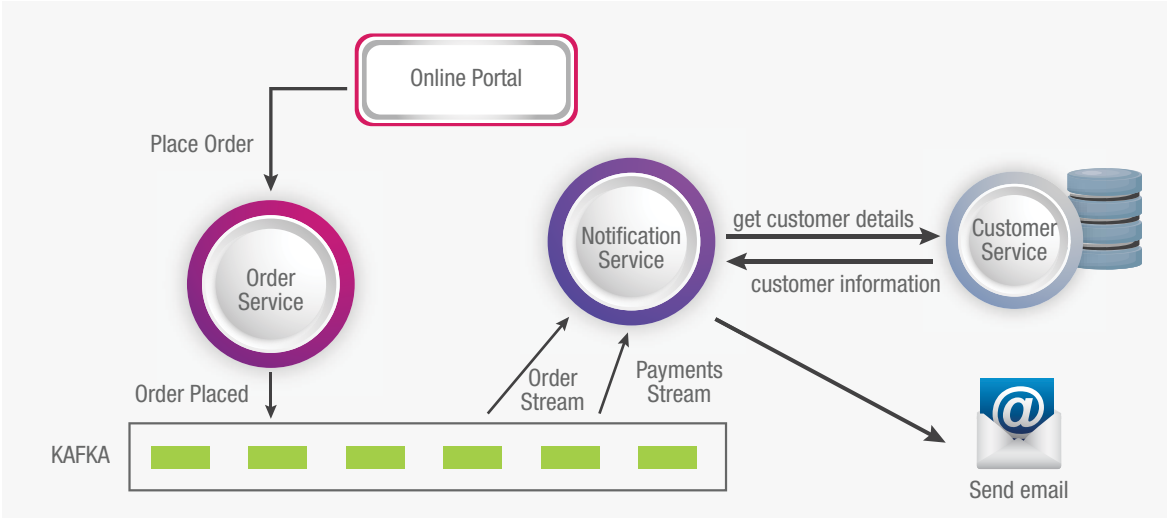

Figure 4: Stream-based processing

3. Stateful implementation

Extending the above option to use caching mechanism, it will now be possible to make notification service a stateful service. Instead of querying customer service every time, it caches the whole customer dataset in memory at startup. As we saw in the Events as First-class Citizens pattern, we will use events to update the cache.

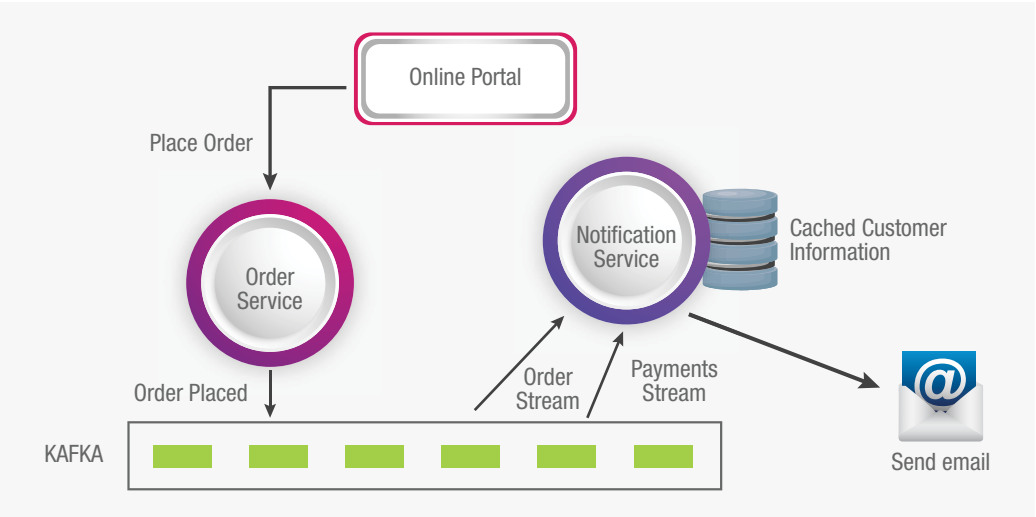

Figure 5: Stateful implementation

The biggest advantage of this approach is that the event processor is not dependent on any other service for processing a message. It also does this processing with reduced latency. However, the application would require a longer startup time since it will have to re-build the state.

Which is the best option?

The choice of option depends on the application and its use cases. Stateless processing services can be restarted faster in case of failure since there is no state to build up. Stateless processing provides horizontal scalability in a cookie-cutter style. However stateless processing may increase latency, depending on the use case complexity. In the above example, there will be a service call every time a notification needs to be sent.

A stateful implementation may provide improvement in the performance, since the number of API calls is reduced. But it requires additional processing logic. In certain use cases, a service may consume multiple event streams and co-relate them with a correlation id. We also need to consider some edge use cases like events coming out of sequence or missing events.

Event Sourcing

Problem Statement

In a typical monolithic application, the persistence layer or database is considered as the system of truth and system of record. It always stores the latest state of any dataset. If there is a need to save the transaction history, a special transaction history table is created.

Solution

Events can play a much bigger role than simply being used for asynchronous inter-service communication. When a series of events are stored in the order they were created, immutably, it can serve as an audit log. We can replay all the transactions that have happened in the system on an object. The resultant state of an object can be deduced from this event replay. This is also known as Event Sourcing.

If the Order processing application does not follow an event-driven architecture, we only have the orders table giving us the latest state of the order. Any changes to the order state might get logged in a separate transaction log, based on the application design.

Using events, you eliminate the need to maintain a separate log. The order status is derived from event logs. The order table always contains the latest state. Historical data is fetched from events and an order state can be created for any given time window.

4.

Command Query Responsibility Segregation (CQRS)

CQRS is a very common design pattern that is applied over event sourcing to create scalable architecture. In this pattern, we expose different service endpoints, using separate data models, for a command (write data) and query (reading data) operation.

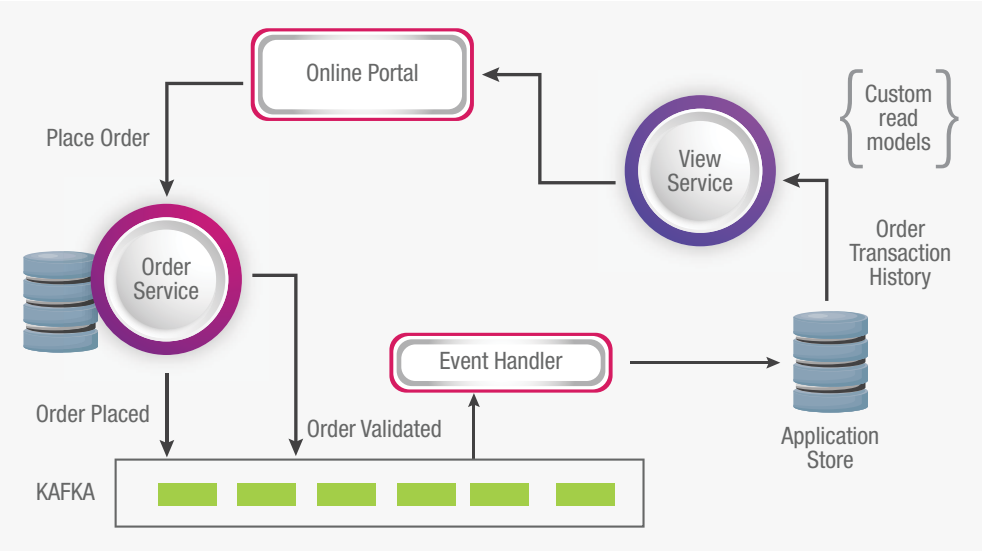

Figure 6: CQRS and event sourcing

There are multiple options to implement event handlers. Apache Kafka Streams can be used to process the events. It provides the benefits of real-time stream processing without an additional cluster overhead.

The Application store could be any relational data store or can be implemented within the Kafka Streams itself.

Information is enriched using custom data models on top of the event log like materialized and polygot views. A materialized view is a very common construct used in databases for fetching query results in a performance-optimized way. These views work on a pre-computed query, which fetches results from the underlying database tables. Using the same analogy, we can use KSQL or KTable over event topics to combine data for read services.

5.

Conclusion

Event-driven architectures have evolved from providing a simple publish-subscribe pattern to becoming complex integration platforms. Using technologies like Apache Kafka it is possible to create an architecture that facilitates building loosely coupled, highly agile application components. There is a paradigm shift from “Asking for data” towards “Having data before being asked”. That is the major design change which modern applications adhere to operate globally, break down application silos, and create a seamless user experience. We can process workloads that are much more than what a traditional architecture can handle.

6.

References

1. Inter-service Communication Patterns - https://www.enterpriseintegrationpatterns.com/patterns/conversation/BasicIntro.html
2. Understanding CQRS - https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/
3. Event Collaboration - https://martinfowler.com/eaaDev/EventCollaboration.html

Author

Rajasi Balan

AVP – Enterprise Application & Integration

Enterprise Architect with 20+ years of experience in architecture & Implementation of Enterprise Integration Platform for large scale applications.

About Mphasis

Mphasis (BSE: 526299; NSE: MPHASIS) applies next-generation technology to help enterprises transform businesses globally. Customer centricity is foundational to Mphasis and is reflected in the Mphasis' Front2Back™ Transformation approach. Front2Back™ uses the exponential power of cloud and cognitive to provide hyper-personalized ($C = X2C^{2}_{TM} = 1$) digital experience to clients and their end customers. Mphasis' Service Transformation approach helps 'shrink the core' through the application of digital technologies across legacy environments within an enterprise, enabling businesses to stay ahead in a changing world. Mphasis' core reference architectures and tools, speed and innovation with domain expertise and specialization are key to building strong relationships with marquee clients. To know more, please visit www.mphasis.com

For more information, contact: marketinginfo.m@mphasis.com

USA
460 Park Avenue South
Suite #1101
New York, NY 10016, USA
Tel.: +1 212 686 6655

UK
1 Ropemaker Street, London
EC2Y 9HT, United Kingdom
T : +44 020 7153 1327

INDIA
Bagmane World Technology Center
Marathahalli Ring Road
Doddanakundhi Village
Mahadevapura
Bangalore 560 048, India
Tel.: +91 80 3352 5000

www.mphasis.com

NR 05/01/22 US LETTER BASIL7183