IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 23, Issue 1, Ser. II (Jan. – Feb. 2021), PP 33-38
www.iosrjournals.org

Understanding Microservices Best Practices for the Optimal Architecture

Pradeep Kumar
Technology Architect (Specialist), British Telecom

Abstract *–This paper deals with the best practices of Microservices and importance of CQRS (Command Query Responsibility Segregation) and Event Sourcing for optimal architecture. Also, this paper guides you through the relevant AWS services and how to implement typical patterns, such as CQRS and Event Sourcing, natively with AWS services.*
Many companies falling into the trap of using REST nowadays, due to popularity and power REST provides based on its own merits, as an "all-in-one" tool.Most microservices architectures will have at least one of those use cases but it seems it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail. Developers are also used to design applications with synchronous request/reply since APIs and Databases have trained developers to invoke a method and expect an immediate response. But Synchronous communication is the crystal meth of distributed softwarebecause it feels good at the time but in the long run will lead with many problems.It has been identified a good Microservice Architecture can't be designed without Event-driven design using Event Sourcing and CQRS.

Date of Submission: 25-02-2021 Date of Acceptance: 10-03-2021

I. Introduction

Let us talk very first how REST helps in designing the Microservices. Let's define what problems REST solves best. HTTP itself is a request/response protocol, so REST is a great fit for request/reply interactions.Since HTTP is a de facto transport standard, the transport layer of the APIs created using REST are interoperable with every programming language. And due to the wide range of security threats present on the internet, the security ecosystem for REST is robust, from firewalls to OAUTH.With API Gateway you can create an API that acts as a "front door" for applications to access data, business logic, or functionality from your backend services, such as workloads running on Amazon EC2 and Amazon ECS, code running on Lambda, or any web application. An API object defined with the API Gateway service is a group of resources and methods. A resource is a typed object within the domain of an API and may have associated a data model or relationships to other resources.Synchronous communication is the crystal meth of distributed softwarebecause it feels good at the time but in the long run will lead with many problems.It is applied primarily to the communication between microservice within the enterprise and that in some cases are at worst with the principles ofmicroservice architecture. The use of REST and synchronous patterns have negative consequences sometimes as mentioned below:
Service Blocking - While invoking a REST service the service is blocked waiting for a response. Itdegrades the application throughput because this thread could be processing other requests.
Tight Coupling - What happens when another service comes online in the future and needs the data? Youadd the new endpoint, but that leads to anunnecessary coupling. Retry logic also leads to tightly coupled to the other services which is antipattern of "single in purpose".

II. Proposed System

2.1 Event-driven architecture with microservices using Event Sourcing and CQRS

The solution to the shortcomings associated with RESTful/Synchronous [i] interactions is to combine the principles of event-driven architecture with microservices. Event-Driven Microservices (EDM) [ii] are inherently asynchronous and are notified when any event is fired. In many cases, asynchronous communications is how many of our daily events take place. Take the example of WhatsApp: It would be incredibly inefficient to navigate to each friend and check to see if they have a status update. Instead we are notified when a friend has updated their status.
Below are the various approaches to integrate an event-driven architecture with microservices using event sourcing and CQRS[iii] and best practices will be identified:
Approach 1 - Suppose, there are two microservices running in their own containers: Order and Customer.When microservices share the same database, the data model among the services can follow relationships among the

tables associated with the microservices. Each microservice is responsible of managing own data and Customer and Order has one to many relationships in tables inside the single DB. The Order service and Customer service can access the tables from the same database. It will lead to proper transactions with ACID properties, where customer data is updatedOrder data can also be updated to guarantee proper atomicity.<u>But there is a limitation to this approach because if there is a change in one data model, then other services are also impacted.</u>

Approach 2 - Let us try to improve the approach 1. As the microservices best practices each microservice should have its own database so Customer and Order microservice will have separate DB and in this situation, there will be no relationship among the tables. Isn't it great?<u>however, there is a limitation of this approach, transaction management cannot be properly handled. If customer data is deleted, the corresponding order also has to be deleted for that customer.</u>

Approach 3 - To overcome this limitation approach 2, Let us try integrating an event-driven architecture with our microservices components.As per this approach any change in the customer data will be published as an event to the messaging queue, so that the event consumer consumes the data and updates the order data for the given customer that generated the new event.Isn't it great? <u>But there is a limitation of this approach also. The atomic updates between the database and published events to the message queue is a challenge. Though these types of transactions can be handled by distributed transaction management, but this is not recommended in a microservices approach.</u>

Approach 4 - To overcome this limitation approach 3, event-sourcingcan be introduced in this microservices architecture. In this approach any event triggered will be stored in an event store. There is no update or delete operations on the data, and every event generated will be stored as a record in the database. If there is a failure in the transaction, the failure event is added as a record in the database. Each record entry will be an atomic operation. It solves the atomicity and maintains the audit records which is later helpful on data analytics. The benefits of messaging for event-driven microservices are many and varied:

I. **Non-Blocking -** It is a waste of resources to have many threads blocked and waiting for a response. With asynchronous messaging applications canprocess otherrequestsin parallel instead of waiting for a response.

II. **Loose Coupling -**Services are independent and can talk to each other by publishing and consuming the messages.Each service is notified of new events and these events can be consumed by any number of services. Loose coupling allows microservices to be ready for high adaptiveness under the enterprises with never ending changes. It easy for microservices to scale since they're decoupled and do not block. This also makes it easy to determine which service is the bottleneck.

III. **Greater Resiliency -** Messaging system offer guaranteed delivery can manage event failures and enable rapid recovery without message loss. In the case of less service failure, the use of messaging allows healthy services to continue processing since they are not blocked on the failed service. Once healed, the failed service will start processing, making the system eventually consistent.

IV. **Error Handling** – In event driven microservices code becomes much cleaner and readable as all the cumbersome retry and error handling logic is gone. In event driven microservices the messaging tier handles the retry of unacknowledged messageswhich frees the service to be small in size and single in purpose.

Isn't it great? <u>But to make the data eventually consistent, this involves asynchronous operations because the data flow integrates with messaging systems.The event store capacity has to be larger in this case also.</u>

Approach 5 - To overcome this limitation approach 4,we integrate CQRS (Command Query Responsibility Segregation) [iv] with event sourcing [v] to overcome the above limitations.In this microservices architecture design pattern which will have a separate model, service and database for write operations in the database. This acts as a command layer and separate model,service and database for query data that acts as a query layer.The command layer is used for inserting data into a data store. The query layer is used for querying data from the data store.In the Customer microservice, when used as a command model, any event change in customer data, like a customer name being added or a customer address being updated, will generate events and publish to the messaging queue. This will also log events in the database in parallel.The event published in the message queue will be consumed by the event consumer and update the data in the read storage.The Customer microservice, when used as a query model, needs to retrieve customer data that invokes a query service, which gets data from read storage.Go through the below diagram for better understandings.

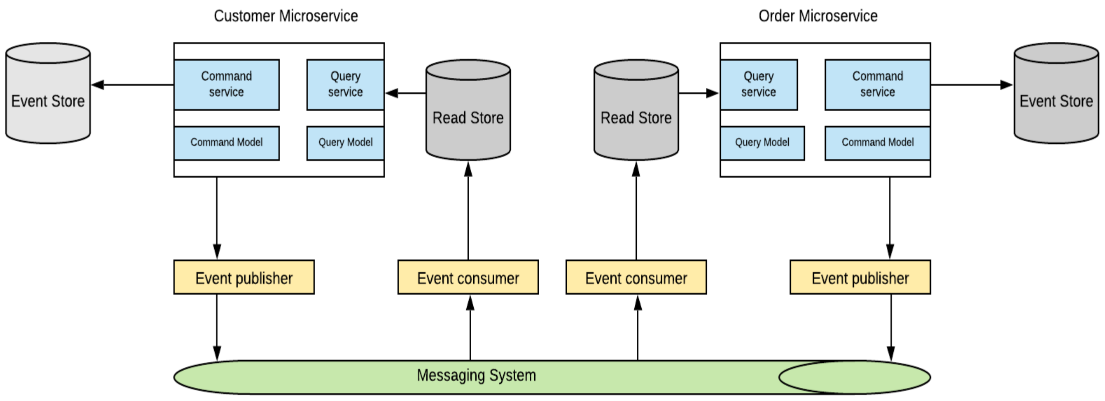

Figure1. *Microservices with CQRS and Event sourcing*

Here we can see few benefitslike CQRS having separate models and services for read and write operations, leveraging microservices for modularity with separate databases, leveraging event sourcing for handling atomic operations, Maintain historical/audit data for analytics with the implementation of event sourcing. Isn't it great? Yes, this is great with a simple limitation with additional maintenance of infrastructure, like having separate databases for command and query requests.

2.2 Microservices Architecture based on Asynchronous messaging using CQRS and Event Sourcing on AWS

CQRSis very much helpful to optimize your architecture for consistent writes in a relational database and very low latency reads, you might instead want to optimize for very high write throughput and flexible query capabilities. You can use a NoSQL datastore, such as Amazon DynamoDB [vi], to get high write scalability. Amazon Aurora can be used to provide complex, one-time query functionality with scalability on read. With this option, you can use Amazon DynamoDB streams that send data to an AWS Lambda function that makes appropriate updates to keep the data on Amazon Aurora up to date.

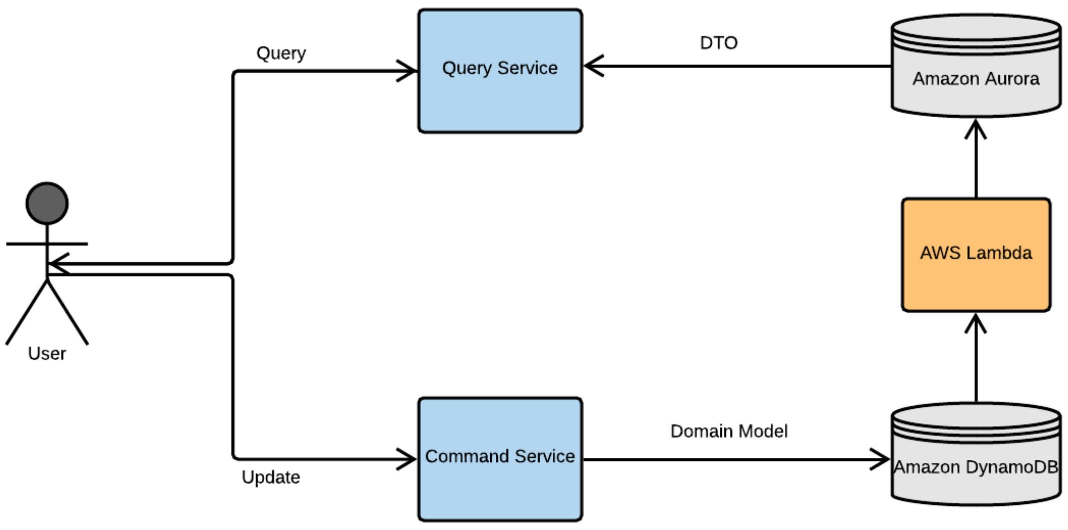

Figure 2. *CQRS architecture on AWS with DynamoDB, Lambda, and Aurora*

Now let us discuss on command part of a CQRS architecture with the event sourcing pattern. When you combine these patterns, you can rebuild the service query data model with the latest application state by replaying the update events.CQRS pattern generally results in eventual consistency between the read/write.Anadditional pattern to implement communication between microservicesis Event Sourcing. Services communicate by exchanging messages via a messaging queue. One major benefit of this communication style is that it's not necessary to have a service discovery and services are loosely couple.This is great, Isn't it? If we have synchronous systems tightly coupled which means a problem in a synchronous downstream dependency has immediate impact on the upstream callers. Retries from upstream callers can quickly fan-out and amplify problems.

Let us try to event sourcing using AWS [vii]. In the event sourcing pattern, each event that changes the system is stored first to a message queue, and then updates to the application state are made based on that event. Depending on specific requirements, like protocols, AWS offers different services which help to implement this pattern. One possible implementation uses a combination of Amazon Simple Queue Service and Amazon Simple Notification Service.Both services work closely together: Amazon SNS allows applications to send messages to multiple subscribers through a push mechanism. By using Amazon SNS[viii] and Amazon SQS[ix] together, one message can be delivered to multiple consumers.

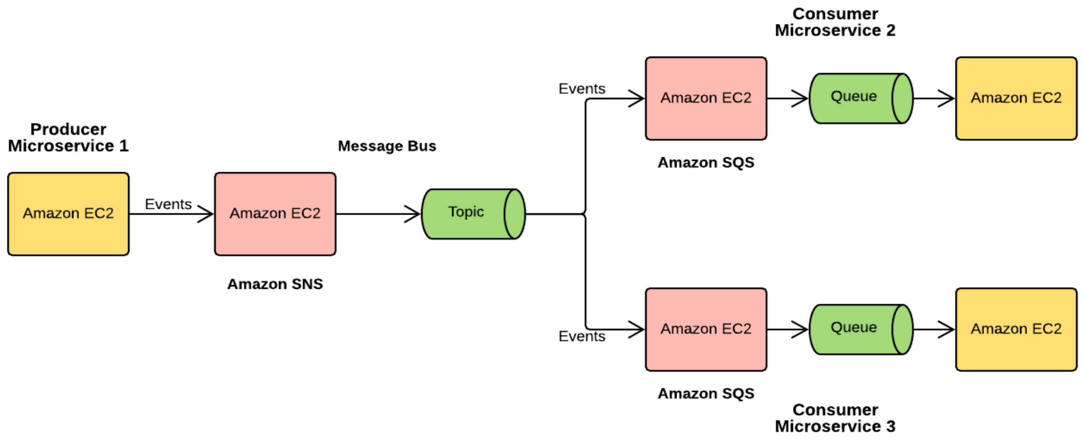

Figure 3.*Message Bus integration using Amazon SNS and Amazon SQS.*

When you subscribe an SQS queue to an SNS topic, you can publish a message to the topic and Amazon SNS sends a message to the subscribed SQS queue. The message contains subject and message published to the topic along with metadata information in JSON format.

A different implementation strategy is based onwith Amazon Kinesis [x] Data Streams, which allows multiple consumers to retrieve data from a stream. For example, an event can be written as a record in an Amazon Kinesis stream, and then a service built on AWS Lambda [xi] can retrieve the record and perform updates in its own data store.

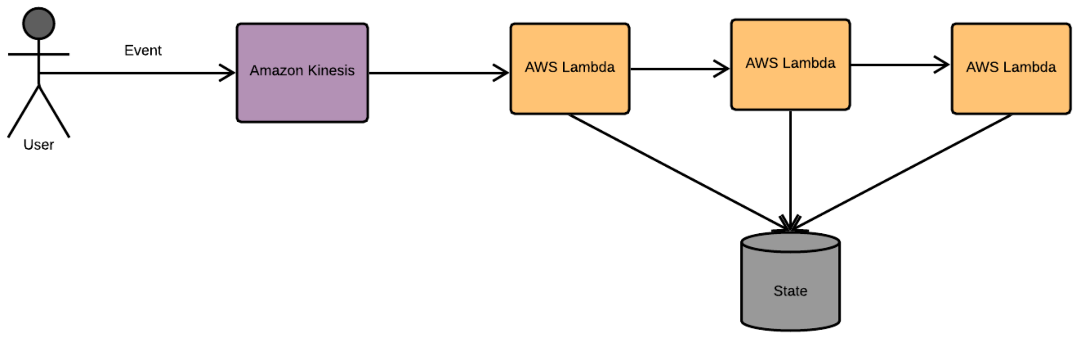

Figure 4.*Event sourcing pattern using Amazon Kinesis and AWS Lambda*

III. Few Microservice Best Practices

I. The Single Responsibility Principle- Just like with code, where a class should have only a single reason to change, microservices should be modelled in a similar fashion. Building bloated services which are subject to change for more than one business context is a bad practice.

II. Consider Using Domain-Driven Design - This is a design approach where the business domain is carefully modelled in software and evolved over time, independently of the plumbing that makes the system work. It is a type of design principle[xii] that makes use of practical rules and ideas to express an objectoriented model. In simpler terms, microservices are designed around your business domains. It is used by platforms such as Netflix who use different servers to run their content delivery and related tracking services.

III. **Using Event sourcing and CQRS-** As discussed in approach 4 and 5 above in the detailed discussion.

IV. **Right API with the Right Microservice -** If you are not sure which technology is best for your project, consider the following parameters during the decision-making process like Maintainability, Fault-tolerance, Scalability, Cost of architecture and Ease of deployment.

V. **Use asynchronous communication to achieve loose coupling -** To avoid building a mesh of tightly coupled components, consider using asynchronous communication between microservices.

VI. **Proxy your microservice requests through an API Gateway -** Instead of every microservice in the system performing the functions of API authentication, request/response logging, and throttling, having an API gateway[xiii] doing these for you upfront will add a lot of value. Clients calling your microservices will connect to the API Gateway instead of directly calling your service. This way you will avoid making all those additional calls from your microservice.

VII. **Use Automation for Independent Deployment -** It would make no sense to break down the monolithic architecture into individual microservices if they cannot be deployed independently. Also, while you are at it, make sure to implement a 'build and release' automation structure [xiv]. Not only will this help you to reduce the overall lead time, but it will also make releases quicker and enhance the deployment process.If you have a system of e.g. 50 Microservices and only one Microservice needs to be changed, then you can update only one Microservice without touching the other 49. But deploying 50 Microservices independently without Automation is a tough task. To take full advantage of this Microservice feature, one needs CI/CD and DevOps.

VIII. **Proper monitoring -** This is only possible if you have been monitoring the performance of individual components in the first place. So, if monitoring is not something that you have focused on, it is a great place to begin the cleaning process.With one Monolith, it is much simpler to monitor the application. But having many microservices run on containers, observability of the whole system became very crucial and complicated. Even Logging became complicated to aggregate logs from many containers/machines into a central place. Often one API request to a microservice leads to several cascaded calls to other microservices. To analyse the latency of a Microservice system, it is necessary to measure the latency of each individual Microservice. Zipkin/Jaeger offers excellent tracing support for Microservices.

IX. **Micro Frontends -** As most Software Architects are Backend Developers, they have little regard for Frontend and Frontend is usually neglected in the Architecture Design. Very often in Microservice projects, backends are very finely modularized with their database but there is one Monolith Frontend. In the best case, they consider one of the hottest SPA (React, Angular) to develop the Monolith Frontend. The main problem of this approach is that Frontend Monolith is as bad as Backend MonolithThere are many ways to develop SPA based Micro frontends: with iFrame, Web Components or via (Angular/React) Elements.

X. **Have dedicated infrastructure hosting your microservice -** You can have the best designed microservice meeting all the checks, but with a bad design of the hosting platform it would still behave poorly. Isolate your microservice infrastructure from other components to get fault isolation and best performance. It is also important to isolate the infrastructure of the components that your microservice depends on.

IV. Conclusion

It has been discussed about the solution to the shortcomings associated with RESTful/Synchronous interactions is to combine the principles of event-driven architecture with microservices asit has been believedsynchronous communication is the crystal meth of distributed softwarebecause it feels good at the time but in the long run will lead with many problems. RESTful/synchronous is applied primarily to the communication between microservice within the enterprise and that in some cases are at worst with the principles of microservice architecture.So,in this paper it has been focused more on event driven architecture using CQRS/Event Sourcing to optimize a microservice Architecture. It has been also discussed that a Microservice Architecture can't be designed without event-driven mechanism.This paper also guides you through the relevant AWS services and how to implement typical patterns, such as CQRS and event sourcing, natively with AWS services.

534–558, 2006.

nference on Software Maintenance and Evolution
(ICSME), 201
updaters," 9thEuropean Conference ore ntenance and
Reengineering (CSMR 2005), 21-23 March 2005, Manch-

ester, UK, Proceedings, 2005
European Conference on Software Maintenance and
Reengineering (CSMR 2005), 21-23 March 2005, Manch-
ester, UK, Proceedings, 2005
Software Maintenance and
Reengineering (CSMR 2005), 21-23 March 2005, Manch-
ester, UK, Proceedings, 2005European Conference on Software Maintenance and
Reengineering (CSMR 2005), 21-23 March 2005, Manch-
ester, UK, Proceedings, 2005
ware releases through build, test, and deployment
automation. Addison-Wesley Professional, 2010
of the Third International Workshop on Release Engineer-
ing, Firenze, 2015
e Quality, Reliability and Security (QRS), pp. 262–
of the 17th European Conference on Pattern Languages
of Programs, Irsee, 2012.
ia Seybold Group, vol. 2, 2006

References

Technology, vol. 57, pp. 21–31, 2
Technology, vol. 57, pp. 21–31, 201REFERENCES

[1]. How Synchronous REST Turns Microservices Back into Monoliths – The New Stack

[2]. B. M. Michelson, "Event-driven architecture overview", Patricia Seybold Group, vol. 2, 2006

[3]. G. Young. (2016) A Decade of DDD, CQRS,Event Sourcing –Domain-Driven Design Europ 2016. visited on 2016-10-11. [Online]. Available: https://www.youtube.com/watch?v=LDW0QWie21s

[4]. G. Young. (2010) CQRS and Event Sourcing. visited on 2016-10-11. [Online]. Available: http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/

[5]. D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability. Microsoft patterns & practices, 2013

[6]. Amazon DynamoDB | NoSQL Key-Value Database | Amazon Web Services

[7]. AWS white paper available online: Implementing Microservices on AWS - Microservices on AWS .

[8]. Amazon Simple Notification Service (SNS) | Messaging Service | AWS

[9]. Amazon SQS | Message Queuing Service | AWS

[10]. Amazon Kinesis - Process &Analyze Streaming Data - Amazon Web Services

[11]. AWS Lambda – Serverless Compute - Amazon Web Services

[12]. Domain-driven design - Wikipedia

[13]. Whitepapers online: Best Practices for Designing Amazon API Gateway Private APIs and Private Integration

[14]. G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," Information and Software Technology vol.57 pp.21-31, 2015