Open in app ↗ Sign up Sign In

Data Consistency in Microservices Architecture

Dilfuruz Kizilpinar · Follow
7 min read · Apr 27, 2021

Listen Share

In this article, I'd like to share my knowledge and experience in **Garanti BBVA**, about moving **from monolithic to microservices architectures**, especially regarding **data consistency**.

Data consistency is hardest part of the microservices architecture. Because in a traditional monolith application, a shared relational database handles data consistency. In a microservices architecture, each microservice has its own data store if you are using **database per service** pattern. So databases are **distributed** among the applications. Each application may use different technologies to manage their data like non-sql databases. Although this kind of distributed architecture has many benefits such as scalability, high availability, agility etc., in terms of data management, there are some critical points regarding data such as transaction management and data consistency/integrity.

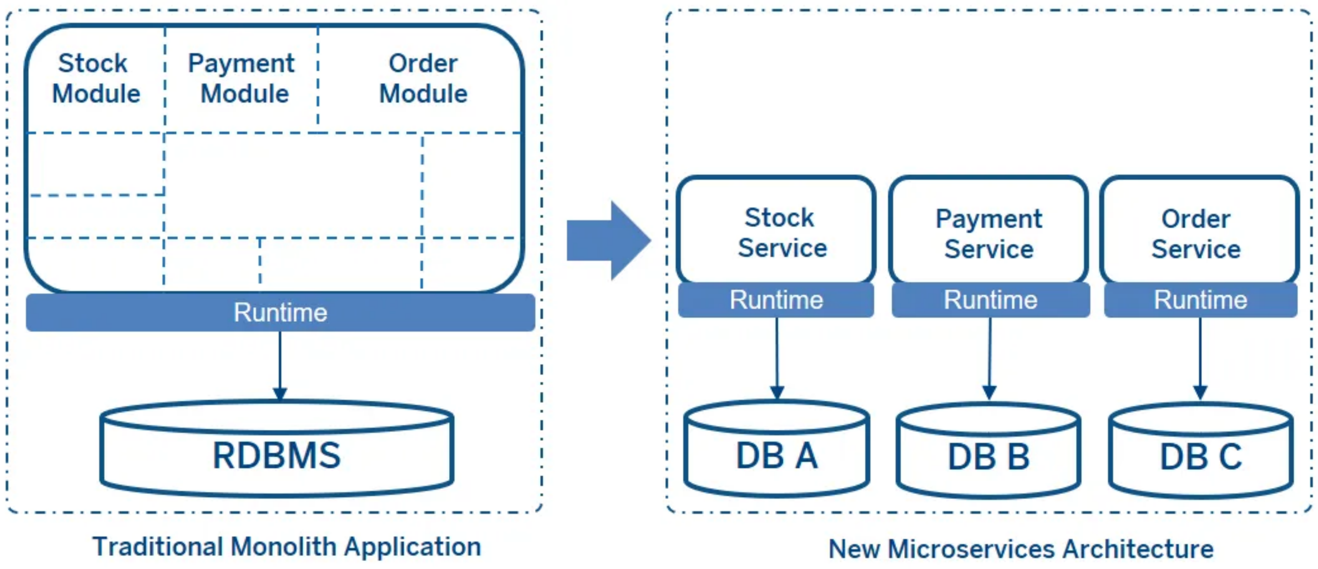


Figure 1. Sample Overall Transition Diagram

In the distributed architecture, data is highly available and scalable because each microservice has its own runtime and data store.

The Problem: Data Consistency in Distributed Systems

For monolith applications, a shared relational database handles and guarantee data consistency by ACID transactions. The acronym ACID means:

- Atomicity: all the steps of a transaction is succeeded or failed together, no partial state, all or nothing.
- Consistency: all data in the database is consistent at the end of transaction.
- Isolation: only one transaction can touch the data in the same time, other transactions wait until completion of the working transaction.
- Durability: data is persisted in the database at the end of the transaction.

In order to maintain strong data consistency, relational database management systems support **ACID** properties.

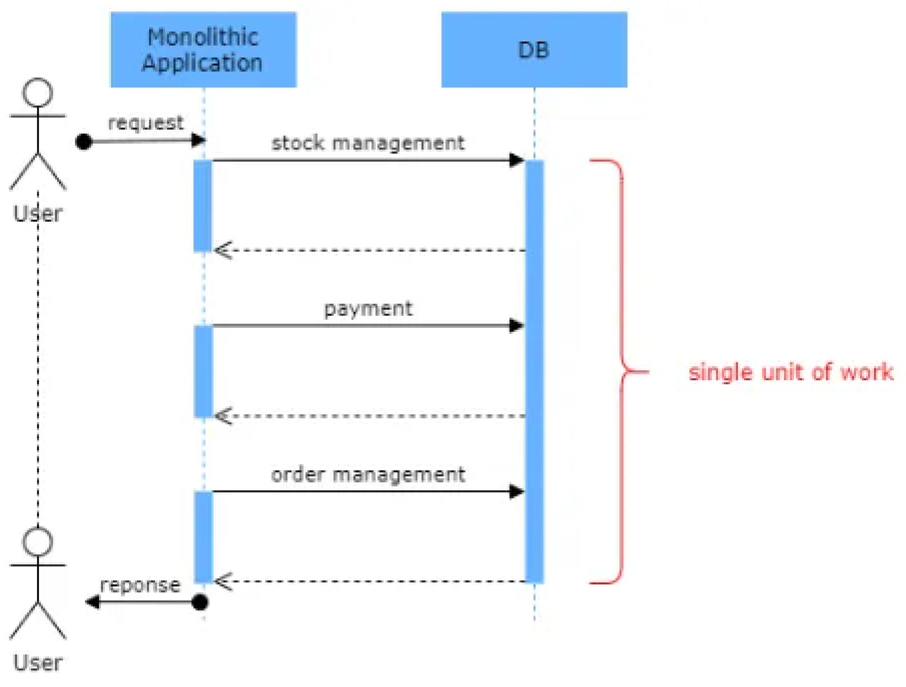


Figure 2. Sample Sequence Diagram of Monolithic Application

But in a microservices architecture, each microservice has its own data store which has different technologies. So, there is **no central database**, **no single unit of work**. Business logic is spanned to the multiple local transactions. This means that you can't use single transaction unit of work among databases in a microservices architecture. But you still need the ACID properties in your application.

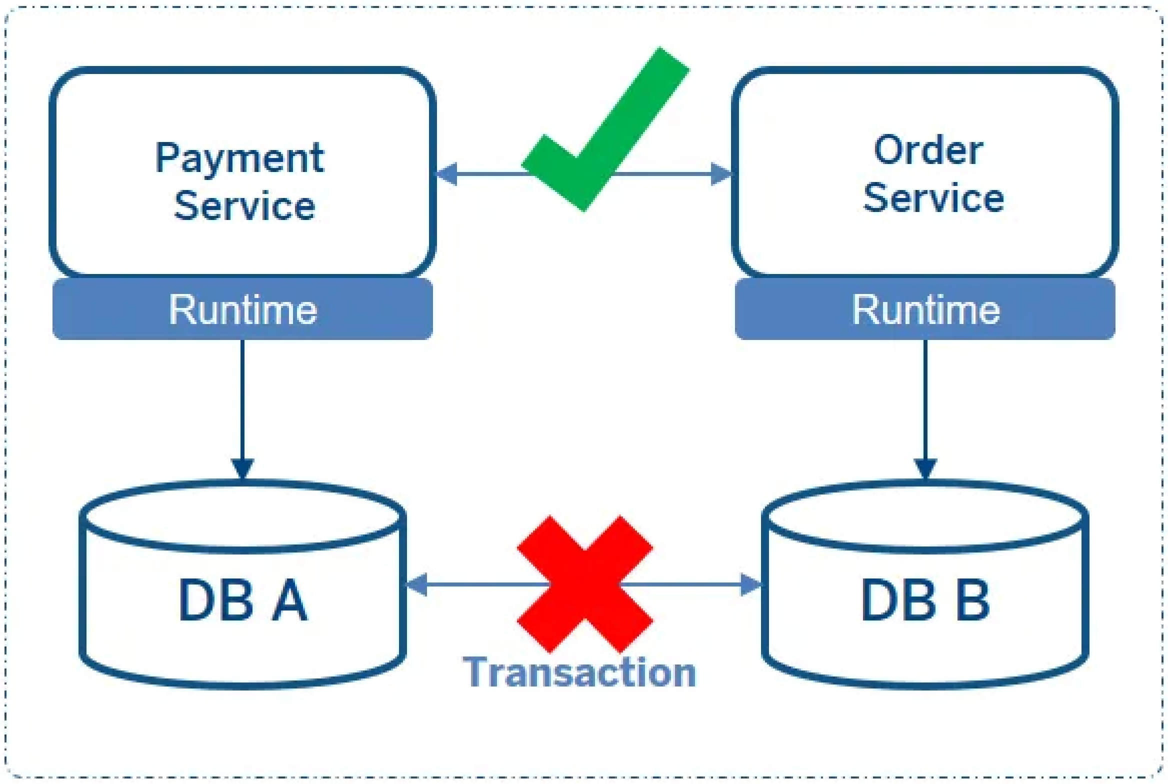


Figure 3. Sample Microservices Interaction Diagram

Let's explain with a simple **sample scenario**. In an order management system, there might be services such as stock management, payment and order management. Let's assume that these services are designed in accordance with microservice architecture and database per service pattern is applied. In order to complete the order process, the order service first calls the Stock Management service for stock control and reservation, and the relevant products in the order are reserved for not selling to another customer. The second step is the payment step. Payment service is responsible for the payment business. Order Service calls the Payment Service and completes the payment from the customer's credit card. Since each one is a separate service, updates over the separated DBs are committed within the service scope. The last step is creating order record. In this step, let's say a technical error has occurred and the order record could not be created, the order number could not be sent to the customer, but the payment was received from the customer. **Data consistency**

problem occurred here. I will talk about what can be done after this point in the Possible Solutions section in the rest of the article.

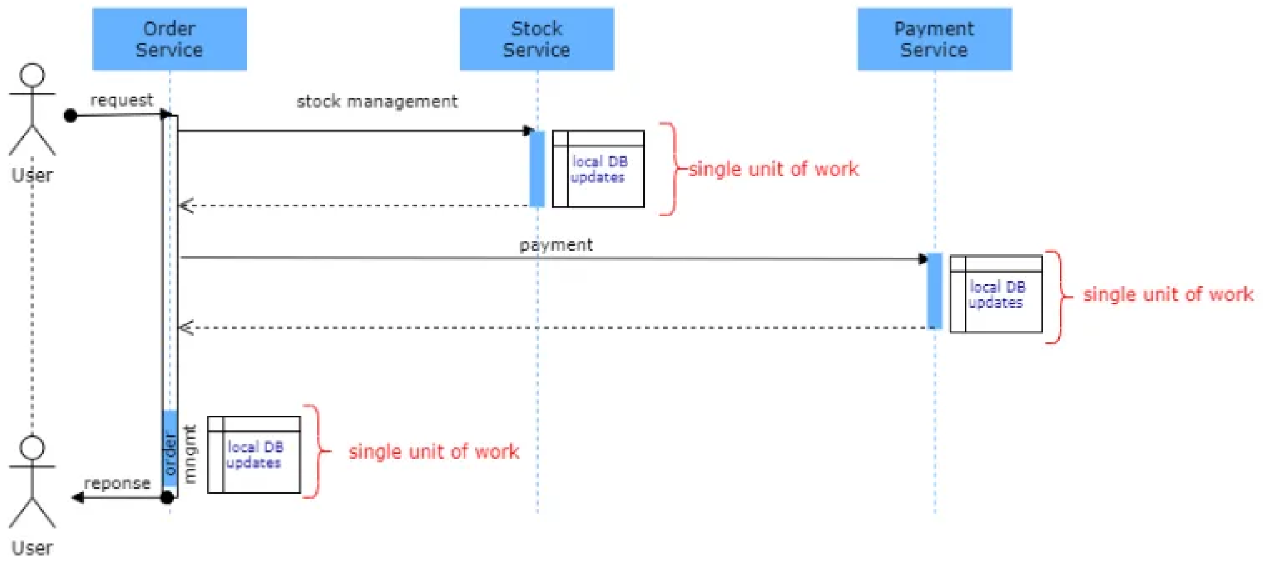


Figure 4. Sample Sequence Diagram of Microservices

Possible Solutions

First of all, there is no single solution which works well for each case. Different solutions can be applied depending on the use-case.

There are two main approaches to solve the problem:

- **Distributed transactions**
- **Eventual consistency**

Distributed Transactions

In a distributed transaction, transactions are executed on two or more resources (e.g. databases, message queues). Data integrity is guaranteed across multiple databases by distributed transaction manager or coordinator.

A distributed transaction is a very complex process since multiple resources are involved in the process.

Two-phase commit(2PC) is a blocking protocol used to guarantee that all the transactions are succeeded or failed together in a distributed transaction.

The **XA standard** is a specification for the 2PC distributed transactions. JTA includes standard API for XA. JTA-compliant application servers support XA out-of-the-box.

But all the resources have to be deployed to a single JTA platform to run 2PC. For a microservice architecture, it is not suitable.

Benefits of Distributed Transactions

- **Strong** data consistency
- Support **ACID** features

Drawbacks of Distributed Transactions

- Very complex process to maintain
- **High latency** & **low throughput** since it is a blocking process (not suitable for high load scenarios)
- Possible **deadlocks** between transactions
- Transaction coordinator is a **single point of failure**

Eventual Consistency

Eventual consistency is a model used in distributed systems to achieve high availability. In an eventual consistent system, inconsistencies are allowed for a short time until solving the problem of distributed data.

This model doesn't apply to distributed ACID transactions across microservices. Eventual consistency uses the **BASE** database model.

While ACID model is providing a consistent system, BASE model provides high availability.

The acronym **BASE** means;

- **B**asically Available: ensures availability of data by replicating it across the nodes of the database cluster
- **S**oft-state: due to lock of the strong consistency data may change over the time. Consistency responsibility is delegated to the developers.
- **E**ventual consistency: immediate consistency may not possible with BASE but consistency will be provided eventually (in a short time).

> *SAGA is a common pattern that operates the eventual consistency model.*

The Saga pattern is an asynchronous model, based on a series of services. In a Saga pattern, the distributed transaction is performed by asynchronous local transactions on all related microservices. Each services updates their own data in a local transaction. Saga manages the execution of the sequence of services.

Two most common implementations of Saga transactions are:

Choreography-based SAGA: No central coordinator exists in this case. Each service produces an event after completion of its task and each service listens to events to take an action. This pattern requires an mature event-driven architecture.

- Event Sourcing is an approach to store the state of event changes by using an Event Store. Event Store is a message broker acting as an event database. States are reconstructed by replaying the events from the Event Store.
- Choreography-based SAGA pattern can work well for small number of steps in a transaction (e.g. 2 to 4 steps). When number of steps in a transaction is increasing, it is difficult to track which services listen to which events.

Orchestration-based SAGA: A coordinator service (Saga Execution Orchestrator, SEG) is responsible for sequencing transactions according to business logic. Orchestrator decides to which operation should be performed. If an operation fails, Orchestrator undo the previous steps. It is called as compensation operation. Compensations are the actions to apply when a failure happens to keep the system in consistent state.

- Undoing changes may not be possible already when data has been changed by a different transaction.
- Compensations must be **idempotent** because they might be called more than once within the retry mechanism.
- Compensations should be designed carefully.

> *There are some available frameworks to implement the Saga orchestration pattern e.g. Camunda, Apache Camel.*

Benefits of SAGA

- Perform **non-blocking** operations running on local atomic transactions

- Offer **no deadlocks** between transactions
- Offer **no single point of failure**

Drawbacks of SAGA

- **Eventual** data consistency
- Does not have **read isolation**, needs extra effort (e.g. the user could see the operation being completed, but in a few second, it is cancelled due to a compensation transaction.)
- Difficult to debug, when participant service count is increased
- Increased development cost (actual service developments plus compensations service developments are required)
- Design is **complex**

> *Data consistency between the distributed data stores can be extremely difficult to maintain. There needs to be a different mindset in design of the new applications.*
>
> *We can say that **responsibility for data consistency moves from database to application level.***

Which Solution to Choose

The solution depends on the use case and consistency requirements.

In general, the following design considerations should be taken into account.

1. **Avoid** using **distributed transactions** across microservices if possible. Working with distributed transactions brings more complex problems.
2. Design your system that doesn't require distributed consistency as much as possible. To achieve this, identify **transaction boundaries** as following;

- Identify the operations that have to work in same unit of work. Use strong consistency for this type of operations

- Identify the operations that can able to tolerate possible latencies in terms of consistency. Use eventual consistency for this type of operations

3. Consider using **event-driven architecture** for asynchronous non-blocking service calls

4. Design **fault-tolerant systems** by compensations and reconciliation processes to keep the system consistent

5. Eventual consistent patterns requires a **change in mindset** for design and development

Conclusion

Microservices architecture has great features such as high availability, scalability, automation, autonomous teams etc. A number of changes in traditional methods are required to obtain maximum efficiency of the microservice architectural style. Data and consistency management is one of the topics that needs to be designed carefully.

Have fun in your microservice journey.

References

- O'Reilly, Monolith to Microservices by Sam Newman https://www.oreilly.com/library/view/monolith-to-microservices/9781492047834/
- O'Reilly, Migrating to Microservice Databases by Edson Yanaga https://www.oreilly.com/library/view/migrating-to-microservice/9781492048824/
- Chris Richardson. Using Saga Patterns to Maintain Data Consistency in a Microservice Architecture. https://www.youtube.com/watch?v=YPbGW3Fnmbc
- https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga

- https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction
- https://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/
- https://www.baeldung.com/transactions-across-microservices

Microservices Data Consistency Transaction Management Eventual Consistency Software Architecture

Follow

Written by Dilfuruz Kizilpinar

482 Followers

Principal Solution Architect www.linkedin.com/in/dilfuruz-kizilpinar

More from Dilfuruz Kizilpinar

Dilfuruz Kizilpinar

Key Benefits and Success Factors of Microservices

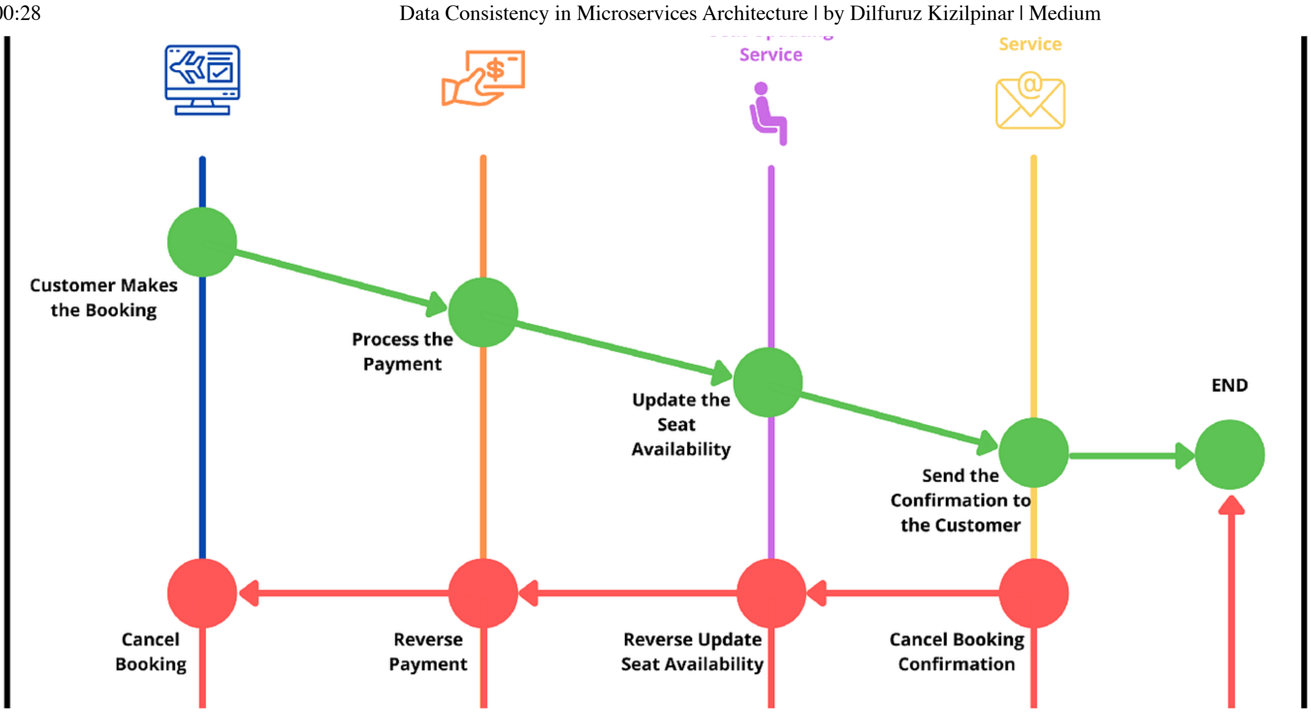
Microservices and organizational structures

5 min read · Jun 1, 2021

75 1

See all from Dilfuruz Kizilpinar

Recommended from Medium



Service
Service
Customer Makes the Booking
Process the Payment
Update the Seat Availability
Send the Confirmation to the Customer
END
Cancel Booking
Reverse Payment
Reverse Update Seat Availability
Cancel Booking Confirmation

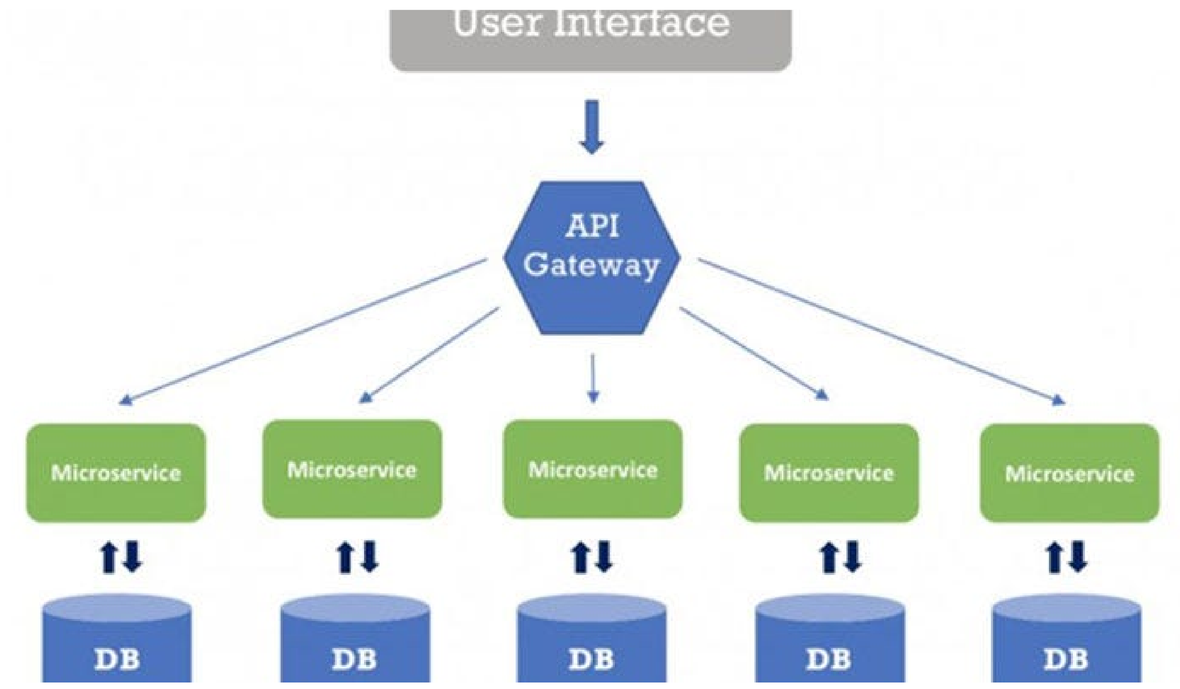
Soma in Javarevisited

What is SAGA Pattern in Microservice Architecture? Which Problem does it solve?

SAGA is an essential Micro service Pattern which solves the problem of maintaining data consistency in distributed system

✦ · 6 min read · Jan 31

338 1



Soma in Javarevisited

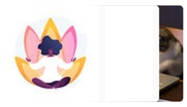
50 Microservices Design and Architecture Interview Questions for Experienced Java Programmers

Preparing for Senior Java developer role where Microservices skill is required? Here are 50 questions which you should know before going...

· 11 min read · Jan 14

458 8

Lists



Stories to Help You Grow as a Software Developer

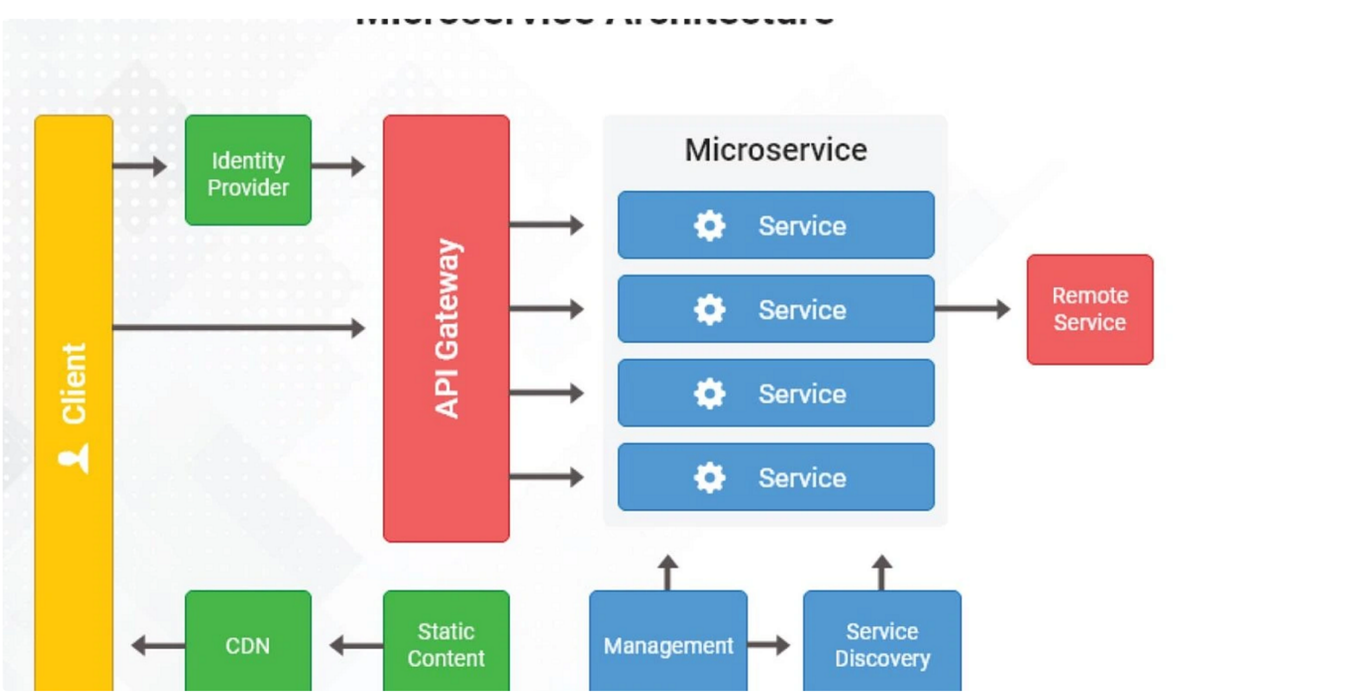
19 stories · 153 saves

General Coding Knowledge

20 stories · 37 saves

Now in AI: Handpicked by Better Programming

248 stories · 17 saves



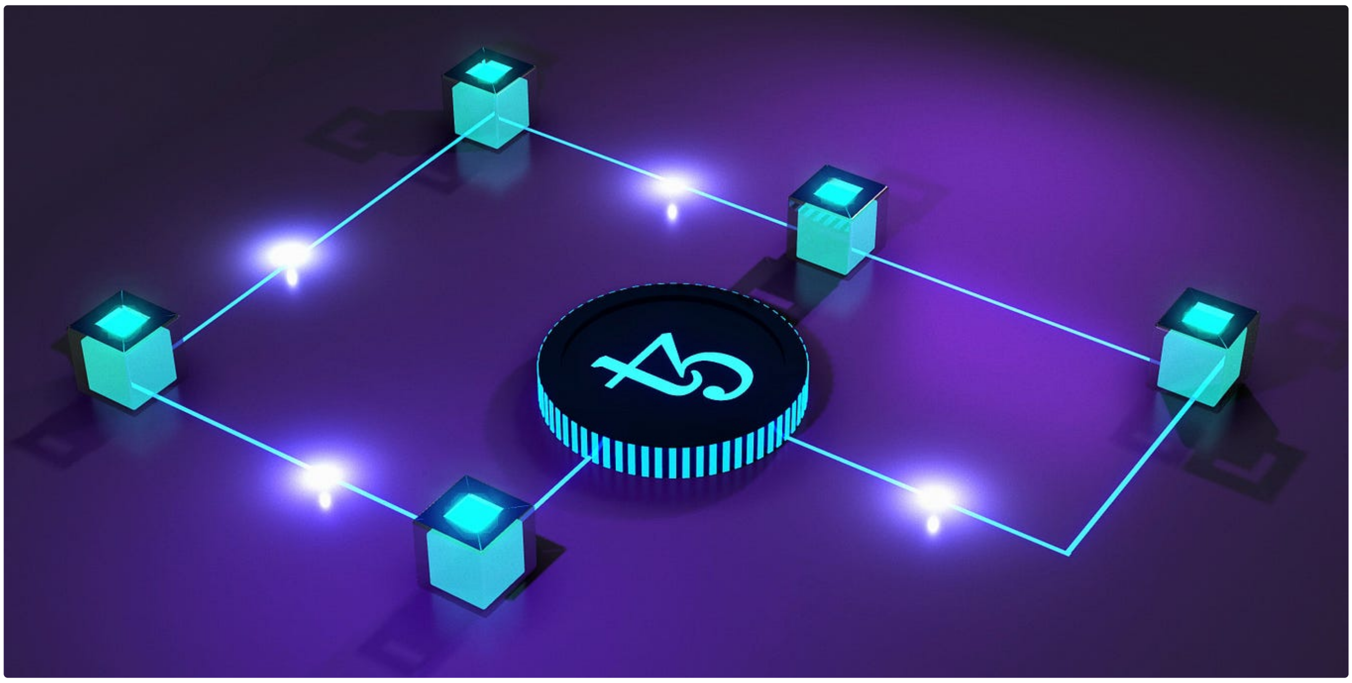
Dineshchandgr - A Top writer in Technology in Javarevisited

Do you know about Microservices and their Design Patterns?

Hello everyone. In this article, we are going to see about Microservice Architecture and how it is different from a Monolithic application...

· 8 min read · Feb 3

426 2



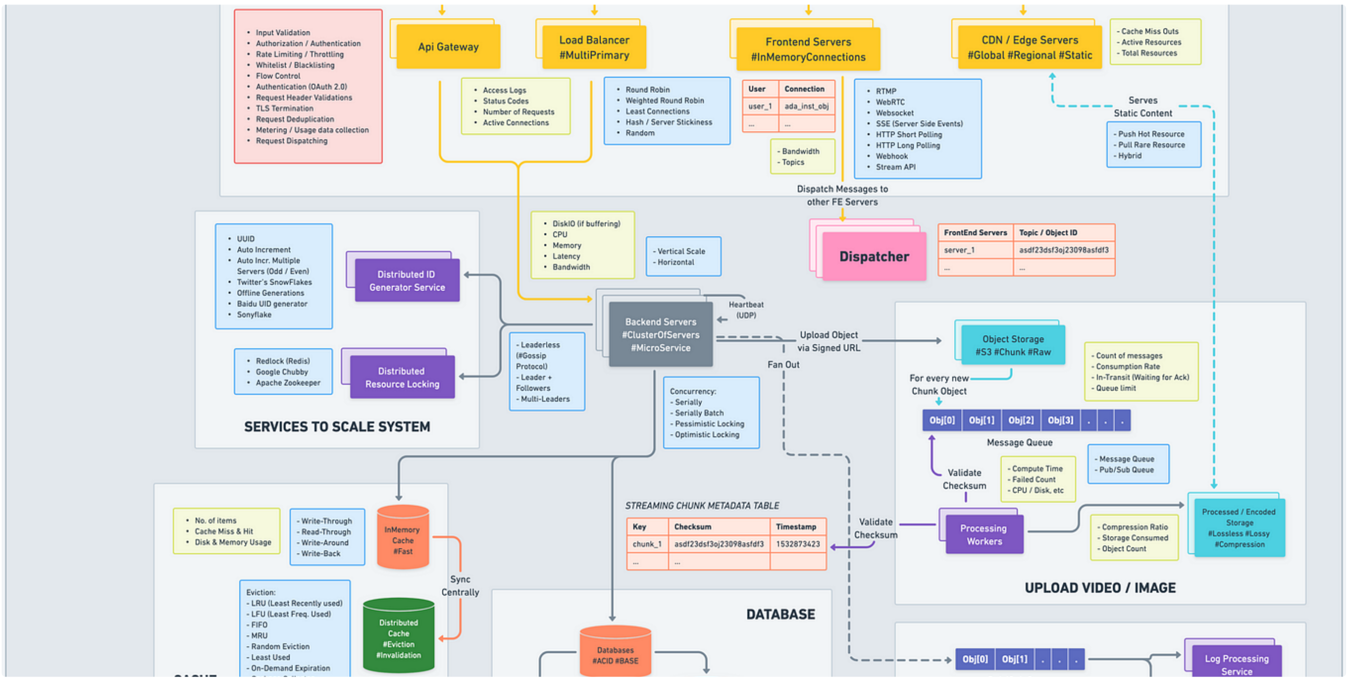
pandaquests in Level Up Coding

Two-Phase Commit (2PC) for data consistency in microservices

Two-Phase Commit (2PC) is a protocol that provides a mechanism for coordinating transactions across multiple participants in a distributed...

· 7 min read · Feb 14

53


Api Gateway
Load Balancer #MultiPrimary
Frontend Servers #InMemoryConnections
CDN / Edge Servers #Global #Regional #Static
Dispatcher
Backend Servers #ClusterOfServers #MicroService
Distributed ID Generator Service
Distributed Resource Locking
SERVICES TO SCALE SYSTEM
Object Storage #S3 #Chunk #Raw
Message Queue
Processing Workers
UPLOAD VIDEO / IMAGE
STREAMING CHUNK METADATA TABLE
DATABASE
InMemory Cache #Fast
Distributed Cache #Eviction #Invalidation
Log Processing Service

Love Sharma in Dev Genius

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

✦ · 9 min read · Apr 20

4.7K 32

Yasas Sandeepa in Level Up Coding

Microservices Explained with Node.js | Concepts and Hands on Experience

Build a server application in Node.js & Docker from scratch with microservice architecture.

· 23 min read · Feb 1

578 1

See more recommendations