Microservices

1. [What’s Microservices?](#_Microservices)
2. [Principles](#_The_principles_used)
3. [Advantages](#_Advantages)
4. [Microservices Design Patterns](#_Microservices_Design_Patterns)
   1. Aggregator
   2. API Gateway
   3. Chained or Chain of Responsibility
   4. Asynchronous Messaging
   5. Database or Shared Data
   6. Event Sourcing
   7. Branch
   8. Command Query Responsibility Segregator
   9. Circuit Breaker
   10. Decomposition
5. [Microservices vs Monolithic Architecture](#_Microservices_vs_Monolithic)

# Microservices

<https://medium.com/edureka/microservices-design-patterns-50640c7bf4a9>

<https://www.edureka.co/blog/microservices-design-patterns>

## **Microservices**, aka *microservice architecture*, is an architectural style that structures an application as a collection of small autonomous services, modeled around a **business domain.**In a Microservice Architecture, each service is self-contained and implements a single business capability.

# The principles used to design Microservices are as follows:

1. Independent & Autonomous Services
2. Scalability
3. Decentralization
4. Resilient Services (failure handling and retry)
5. Real-Time Load Balancing
6. Availability
7. Continuous delivery through DevOps Integration
8. Seamless API Integration and Continuous Monitoring
9. Isolation from Failures
10. Auto -Provisioning

# Advantages

* Technology diversity, e.g., Microservices can mix easily with other frameworks, libraries,  and databases
* Fault isolation, e.g., a process failure should not bring the whole system down.
* Greater support for smaller and parallel team
* Independent deployment
* Deployment time reduce

# Microservices Design Patterns

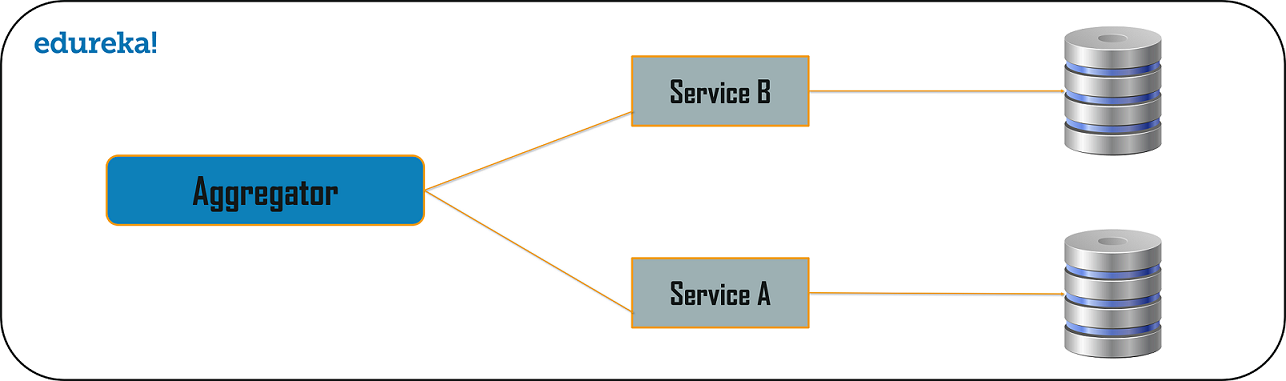
## Aggregator Pattern

## Aggregator in the computing world refers to a website or program that collects related items of data and displays them. So, even in Microservices patterns, Aggregator is a basic web page which invokes various services to get the required information or achieve the required functionality.

## Also, since the source of output gets divided on breaking the monolithic architecture to microservices, this pattern proves to be beneficial when you need an output by combining data from multiple services. So, if we have two services each having their own database, then an aggregator having a unique transaction ID, would collect the data from each individual microservice, apply the business logic and finally publish it as a REST endpoint. Later on, the data collected can be consumed by the respective services which require that collected data.

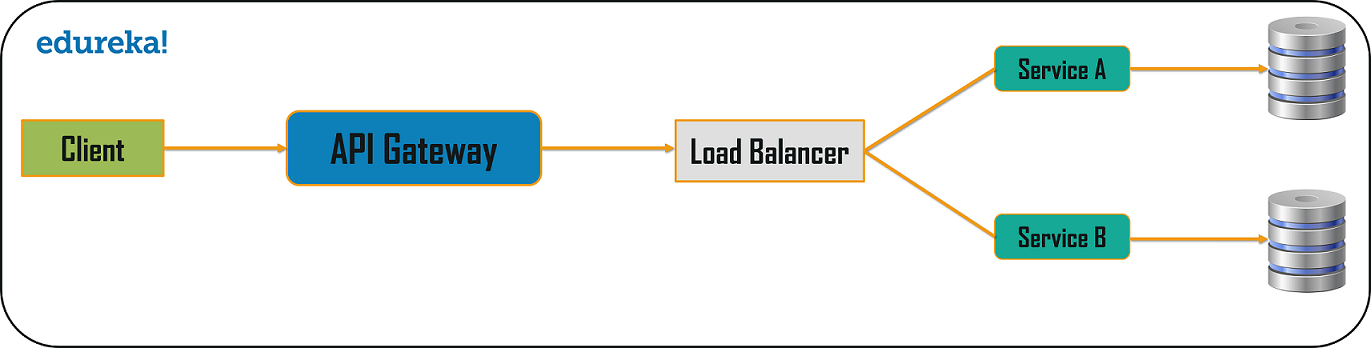
## The Aggregate Design Pattern is based on the DRY principle. Based on this principle, you can abstract the logic into a composite microservices and aggregate that particular business logic into one service.

## So, for example, if you consider two services: Service A and B, then you can individually scale these services simultaneously by providing the data to the composite microservice.

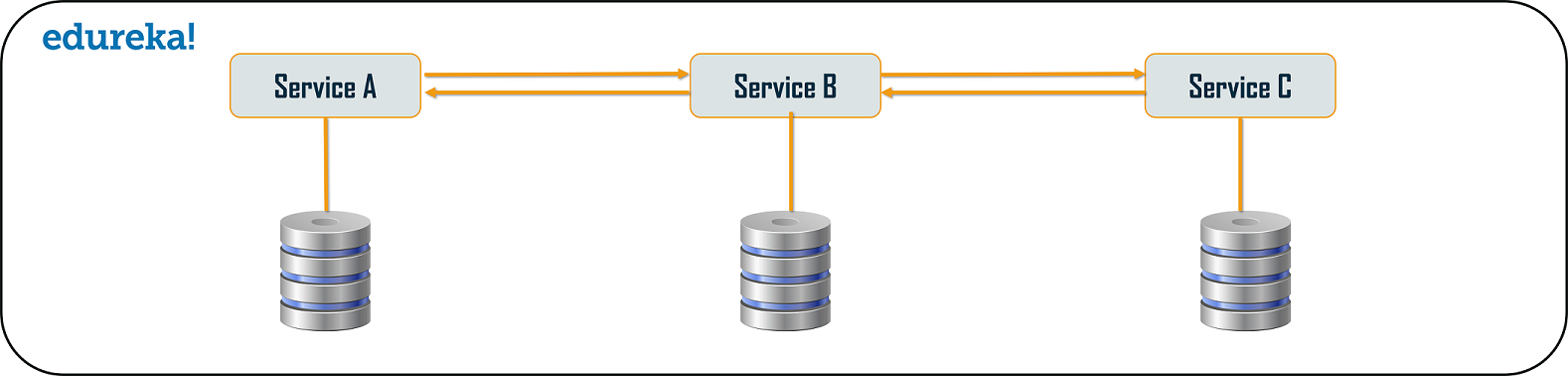


## API Gateway Design Pattern

## With the help of the API Gateway design pattern, the API gateways can convert the protocol request from one type to other. Similarly, it can also offload the authentication/authorization responsibility of the microservice.

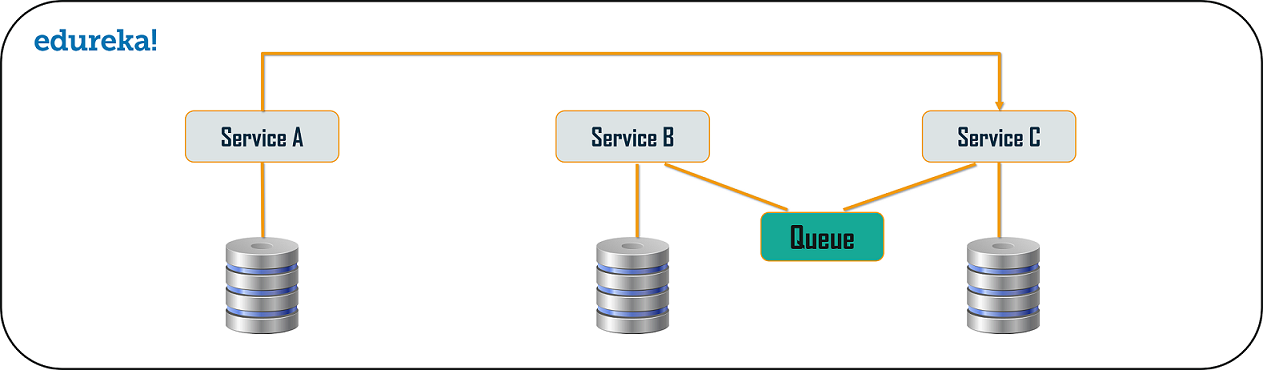


## Chained or Chain of Responsibility Pattern

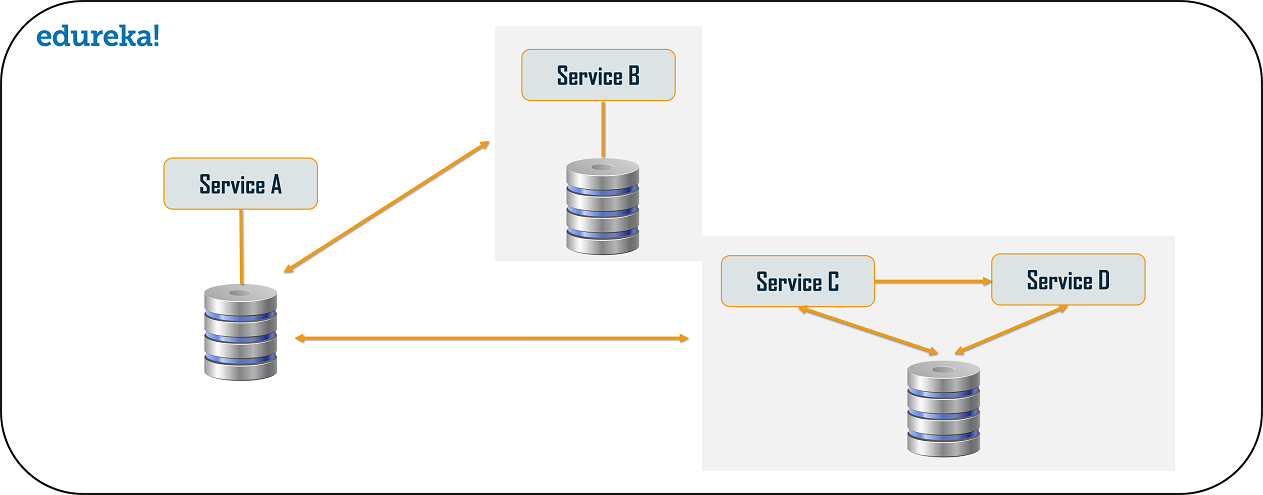


## Asynchronous Messaging Design Pattern

## From the above pattern, it is quite obvious that the client gets blocked or has to wait for a long time in synchronous messaging. But, if you do not want the consumer, to wait for a long time, then you can opt for the Asynchronous Messaging. In this type of microservices design pattern, all the services can communicate with each other, but they do not have to communicate with each other sequentially. So, if you consider 3 services: Service A, Service B, and Service C. The request from the client can be directly sent to the Service C and Service B simultaneously. These requests will be in a queue. Apart from this, the request can also be sent to Service A whose response need not have to be sent to the same service through which request has come.



## Database or Shared Data Pattern



# Microservices vs Monolithic Architecture

|  |  |
| --- | --- |
| Microservices | Monolithic Architecture |
| Service Startup is fast | Service startup takes time |
| Microservices are loosely coupled architecture. | Monolithic architecture is mostly tightly coupled. |
| Changes done in a single data model does not affect other Microservices. | Any changes in the data model affect the entire database |
| Microservices  focuses  on products, not projects | Monolithic put emphasize over the whole project |