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)
6. Two Phase commit (Handling distributed transactions)

* Two Phase commit
* Saga pattern

# 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

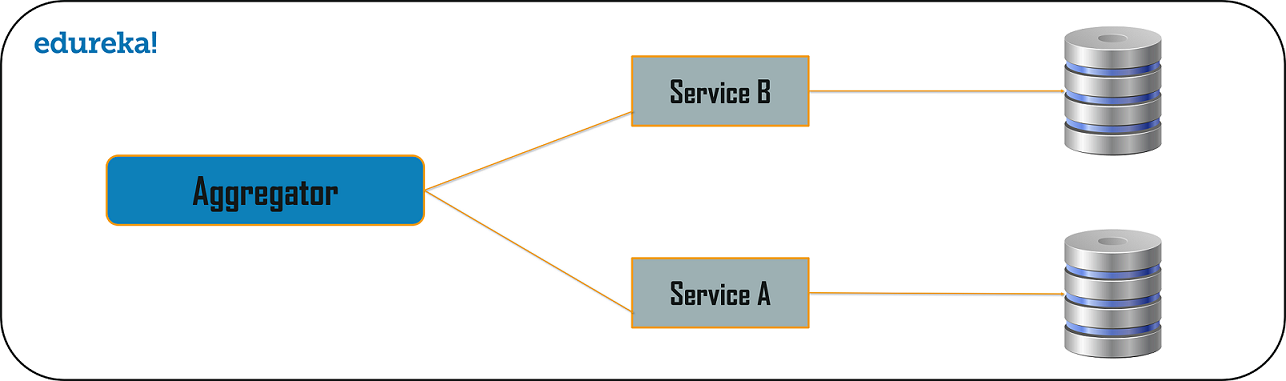
## Aggregator Pattern

## 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 Microservices, 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 microservice 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.

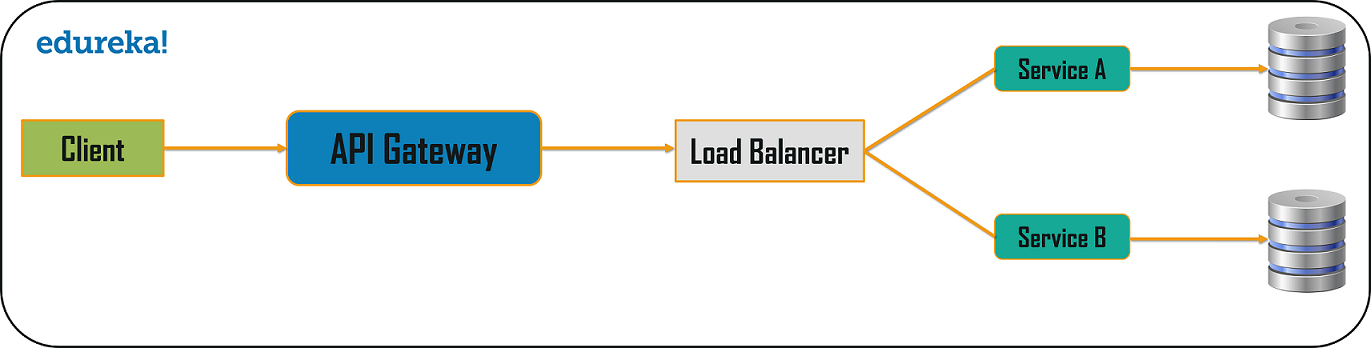


## API Gateway Design Pattern

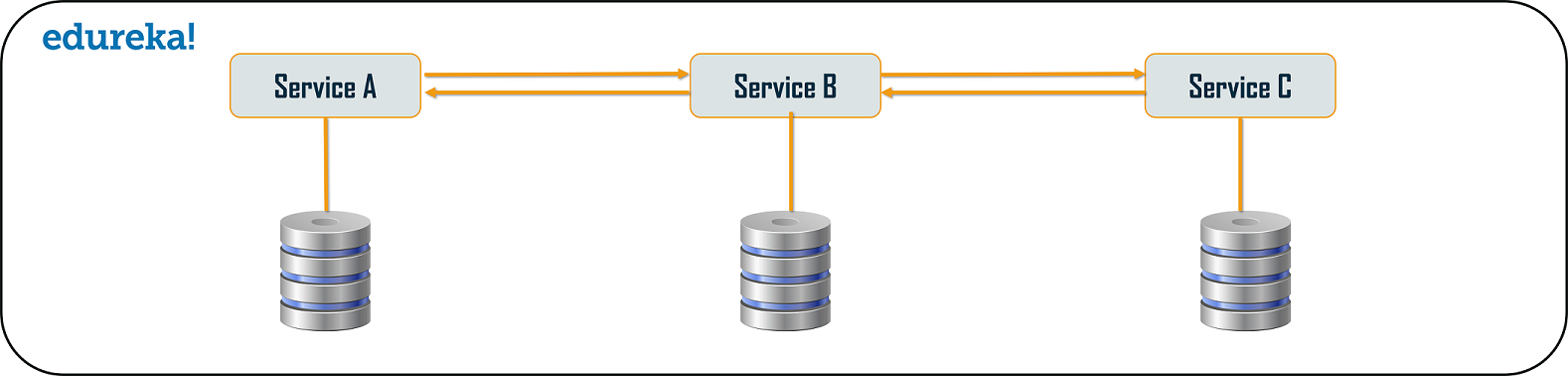
Microservices are built in such a way that each service has its own functionality. But, when an application is broken down into small autonomous services, then there could be few problems that a developer might face. The problems could be as follows:

1. How can I request information from multiple microservices?
2. Different UI require different data to respond to the same backend database service
3. How to transform data according to the consumer requirement from reusable Microservices
4. How to handle multiple protocol requests?

Well, the solution to these kinds of problems could be the API Gateway Design Pattern.  The API Gateway Design Pattern address not only the concerns mentioned above but it solves many other problems. This microservice design pattern can also be considered as the proxy service to route a request to the concerned microservice. Being a variation of the Aggregator service, it can send the request to multiple services and similarly aggregate the results back to the composite or the consumer service. API Gateway also acts as the entry point for all the microservices and creates fine-grained APIs’ for different types of clients.

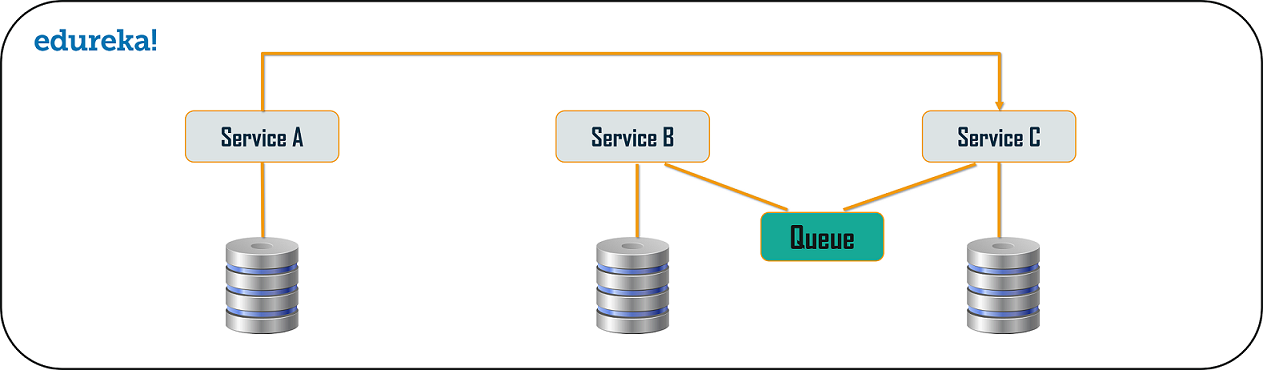


## Chained or Chain of Responsibility Pattern

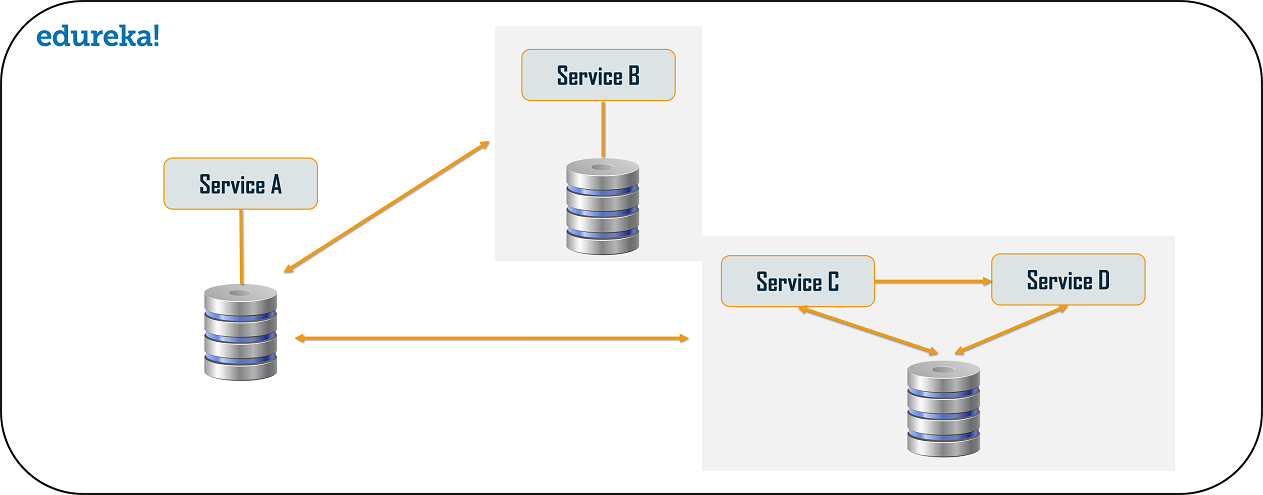


## Asynchronous Messaging Design Pattern

## 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



## Event Sourcing Design Pattern

The event sourcing design pattern creates events regarding the changes in the application state. Also, these events are stored as a sequence of events to help the developers track which change was made when. So, with the help of this, you can always adjust the application state to cope up with the past changes. You can also query these events, for any data change and simultaneously publish these events from the event store. Once the events are published, you can see the changes of the application state on the presentation layer.

### Event Sourcing - Microservices Design Patterns - Edureka

## Branch Pattern

Branch microservice design pattern is a design pattern in which you can simultaneously process the requests and responses from two or more independent microservices. So, unlike the chained design pattern, the request is not passed in a sequence, but the request is passed to two or more mutually exclusive microservices chains. This design pattern extends the Aggregator design pattern and provides the flexibility to produce responses from multiple chains or single chain. For example, if you consider an e-commerce application, then you may need to retrieve data from multiple sources and this data could be a collaborated output of data from various services. So, you can use the branch pattern, to retrieve data from multiple sources.

### Branch - Microservices Design Patterns - Edureka

## Command Query Responsibility Segregator (CQRS) Design Pattern

Every microservices design has either the database per service model or the shared database per service. But, in the database per service model, we cannot implement a query as the data access is only limited to one single database. So, in such scenario you can use the CQRS pattern. According to this pattern, the application will be divided into two parts: Command and Query. The command part will handle all the requests related to CREATE, UPDATE, DELETE while the query part will take care of the materialized views. The materialized views are updated through a sequence of events which are creating using the event source pattern discussed above.

### CQRS - Microservices Design Patterns - Edureka

## Circuit Breaker Pattern

As the name suggests, the Circuit Breaker design pattern is used to stop the process of request and response if a service is not working. So, for example, let’s say a client is sending a request to retrieve data from multiple services. But, due to some issues, one of the services is down. Now, there are mainly two problems you will face: first, since the client will not have any knowledge about a particular service being down, the request will be continuously sent to that service. The second problem is that the network resources will be exhausted with low performance and bad user experience.

So, to avoid such problems, you can use the Circuit Breaker Design Pattern. With the help of this pattern, the client will invoke a remote service via a proxy. This proxy will basically behave as a circuit barrier. So, when the number of failures crosses the threshold number, the circuit breaker trips for a particular time period. Then, all the attempts to invoke the remote service will fail in this timeout period. Once that time period is finished, the circuit breaker will allow a limited number of tests to pass through and if those requests succeed, the circuit breaker resumes back to the normal operation. Else, if there is a failure, then the time out period begins again.

### Circuit Breaker - Microservices Design Patterns - Edureka

# 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 |