**Microservice Design Patterns – 2024**

**Aggregator Microservice Design Pattern**

**Proxy Microservice Design Pattern**

**Chained Microservice Design Pattern (Chain of Responsibility)**

**Branch Microservice Design Pattern**

**Shared Data Microservice Design Pattern**

**Asynchronous Messaging Microservice Design Pattern**

**Circuit Breaker Pattern**

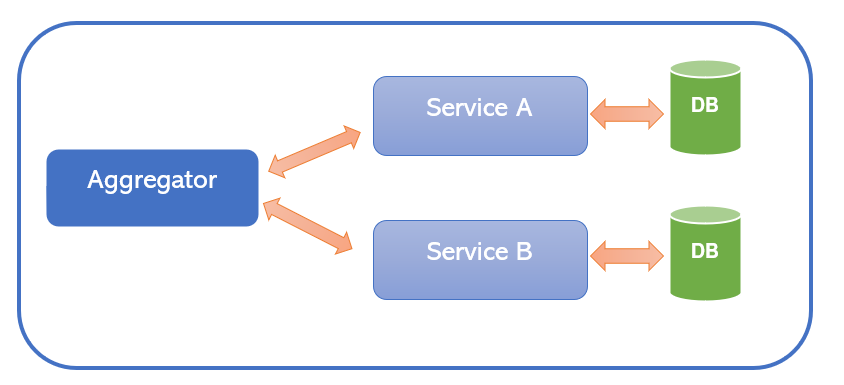
**CQRS Pattern**

**Event Sourcing Pattern**

**API Gateway Pattern**

Aggregator Microservice Design Pattern

**Aggregator Design Pattern is a service that receives a request, then makes requests to multiple services, combines the results and responds to the initiating request. This pattern is particularly useful when a client request involves multiple services to provide the required response.**



**Use this pattern when:**

* **A client needs to communicate with multiple backend services to perform an operation.**
* **The client may use networks with significant latency, such as cellular networks.**

@RestController

public class AggregatorController {

@Autowired

private OrderClient orderClient;

@Autowired

private PaymentClient paymentClient;

@GetMapping("/aggregate/{id}")

public AggregateData aggregate(@PathVariable String id) {

Order order = orderClient.getOrderById(id);

Payment payment = paymentClient.getPaymentByOrderId(id);

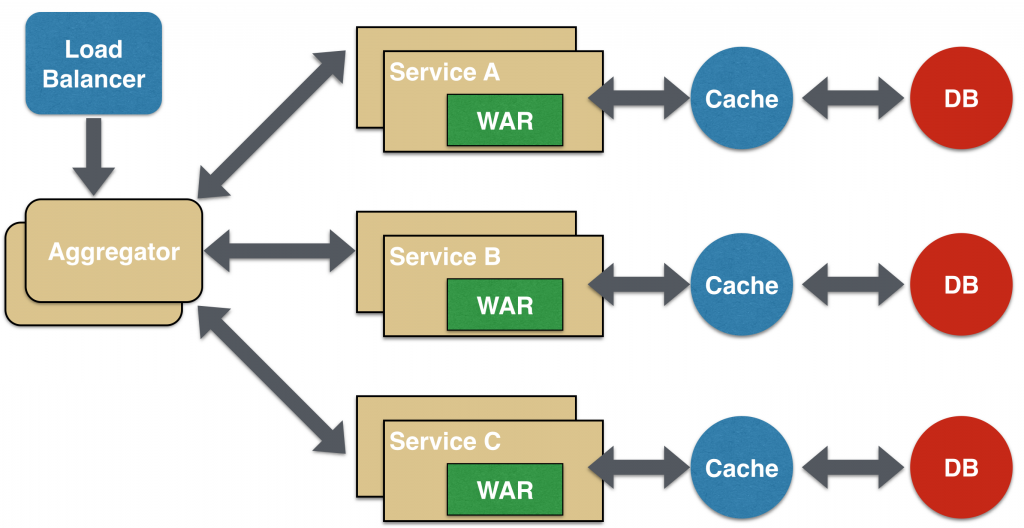
return new AggregateData(order, payment);

}

}

**Performance**

* **Aggregator services can be prone to performance bottlenecks and can add latency if not properly designed.**



Proxy Microservice Design Pattern

* This pattern acts as an intermediary between the client and the service.
* The proxy microservice design pattern is slightly similar to the aggregator where we create a microservice to invoke other services based on the business requirements.
* The Proxy pattern provides an additional layer of abstraction and control, which can be beneficial in a microservices architecture. I
* t allows you to manage and configure services independently without impacting the client.

There are two categories of proxy service pattern namely;

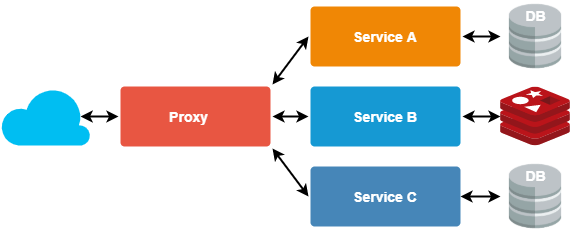
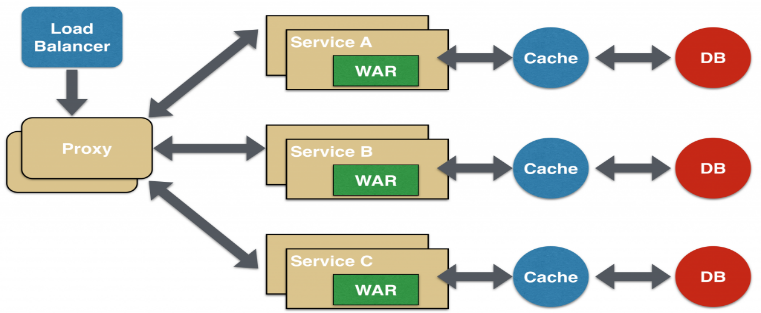
1. Dumb Proxy: Responsible for delegating a request to another service.
2. Smart Proxy: Responsible for applying transformation, logic, filtering, rejection before delegating a request to another service.

Usage of this Design Pattern

1.    Use this pattern whenever you want introduce a new api and also have to serve existing consumer with old api.

2.    Use this pattern whenever you want to implement device or channel specific apis.

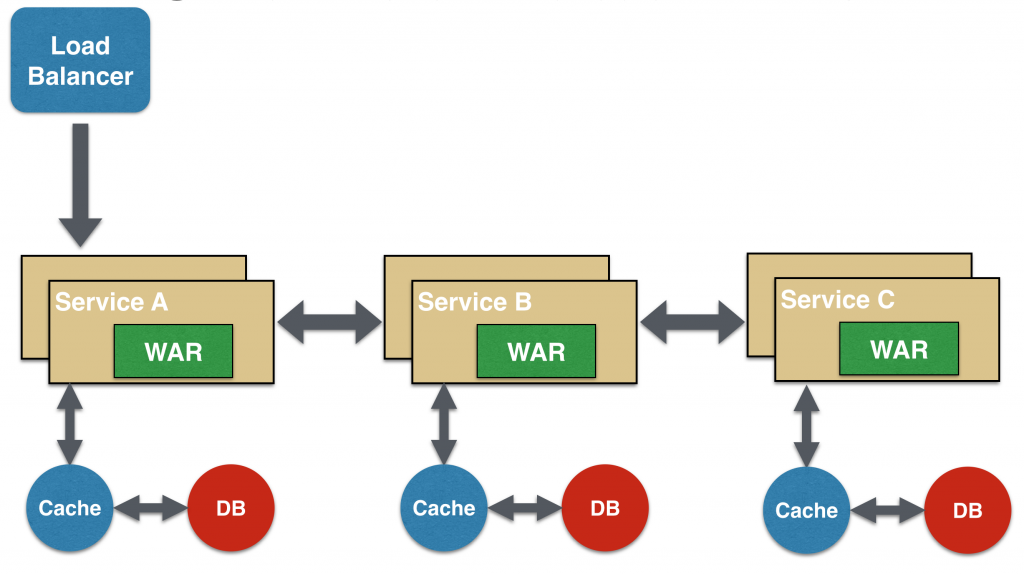
3.    Use this pattern whenever you want to filter, transform request / response.

### Chained Microservice Design Pattern (Chain of Responsibility)

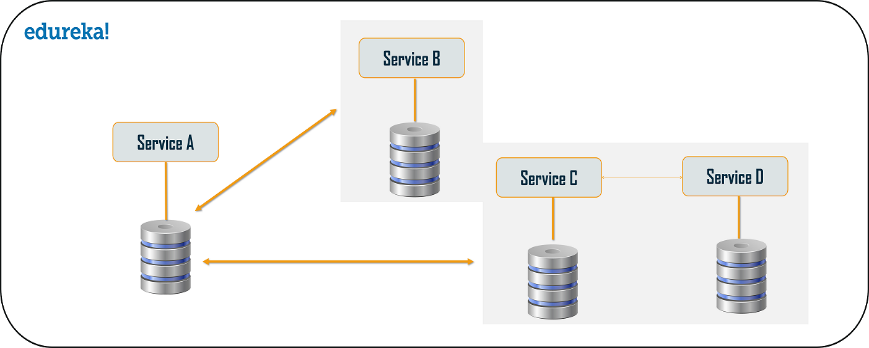
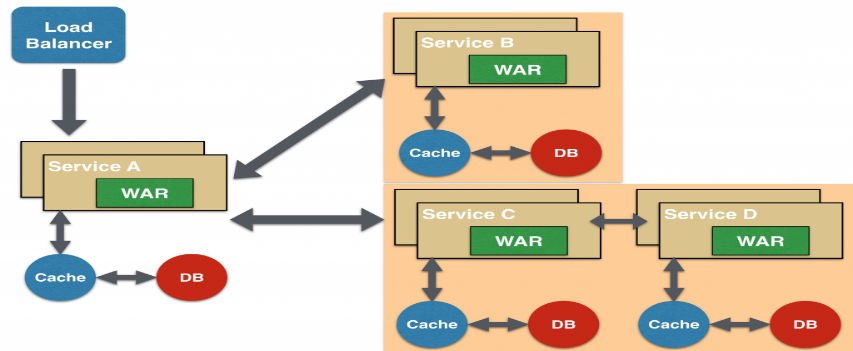
The chain of responsibility pattern is a behavioral design pattern where a request is processed sequentially by a chain of handlers until one of them handles it. A common use case is applying a series of filters or validations to a request before processing it. This pattern enhances reusability, maintainability, and modularity while allowing for easy addition or removal of handlers. In this pattern, each service in the chain processes the request and decides whether to pass it on to the next service.

**Service A calls Service B and Service B calls Service C**. All the services are likely using a synchronous HTTP request/response messaging. The key part to remember is that the client is blocked until the complete chain of request/response. Another important aspect to understand here is to not make the chain too long.

[](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-chain.png)

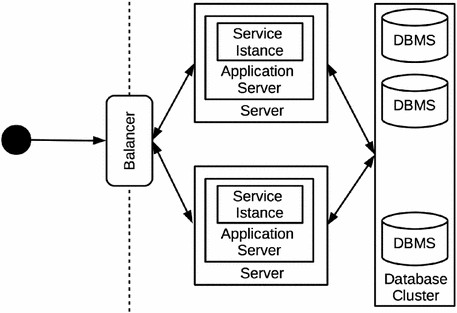
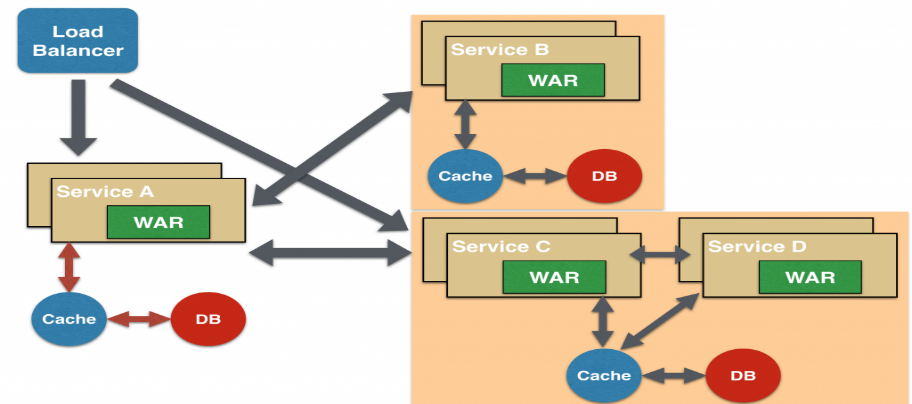
## Branch Microservice Design Pattern

* The Branch pattern, also known as the Fork-Join pattern, is a structural design pattern.
* The request is passed to two or more mutually exclusive microservices chains, processes them concurrently and combines the results. This pattern can significantly improve the performance of your application by leveraging parallel processing.
* Branch microservice design pattern extends Aggregator design pattern and allows simultaneous response processing of multiple services. **Service A concurrently calls Service B and Service C**.

[](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-branch.png)

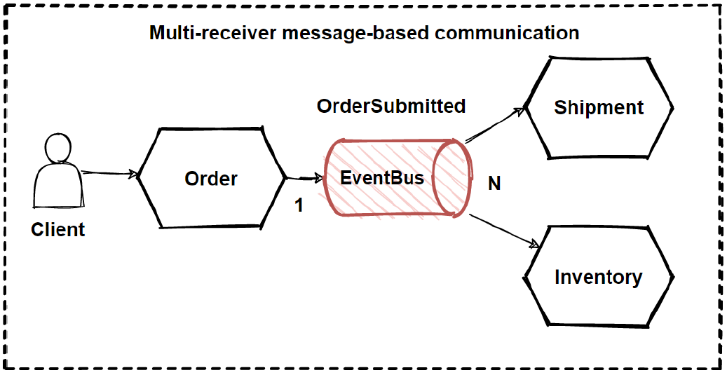
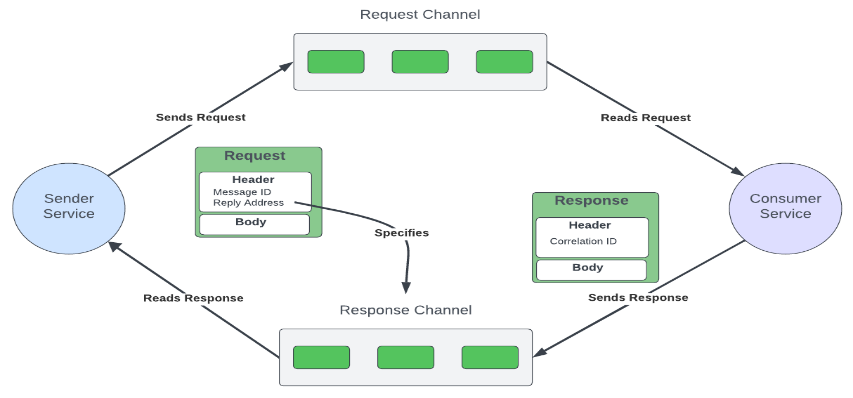
## Shared Data Microservice Design Pattern

The Shared Database pattern allows multiple services to share the same database. It simplifies data management and ensures data consistency across services. However, the Shared Database pattern contradicts the core principle of microservices, which advocates for one database per service. Therefore, it should be used judiciously and only when the benefits outweigh the risks.

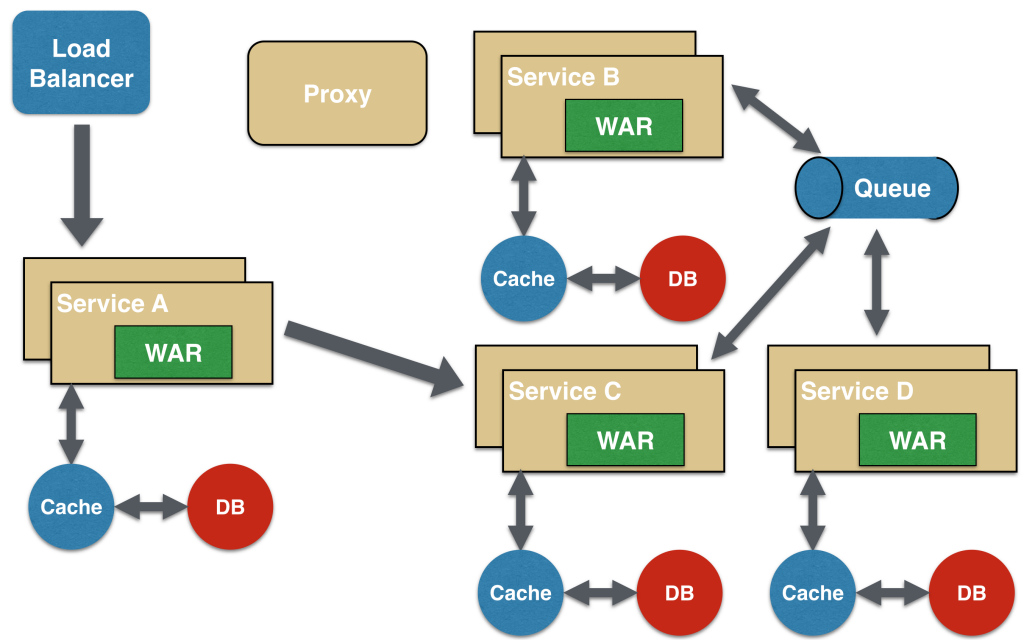
 [](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-branch-shared-data.png)

## Asynchronous Messaging Microservice Design Pattern

* This pattern is a communication approach in which microservices interact using message-based protocols.
* Asynchronous communication is typically implemented using message brokers or event-driven architectures, such as Apache Kafka or RabbitMQ.
* This pattern allows services to operate independently, improving scalability, and resilience.

‍ 

In this design pattern, Service A may call Service C synchronously which is then communicating with Service B and D asynchronously using a shared message queue. You can use Kafka or RabbitMQ to achieve.

[](http://blog.arungupta.me/wp-content/uploads/2015/04/microservices-async-messaging.png)

## Circuit Breaker Pattern

* The circuit breaker pattern is a fault-tolerance mechanism that prevents cascading failures in a microservices
* It monitors requests to a service and detects failures or timeouts.
* When the failure rate exceeds a threshold, the circuit breaker "opens," stopping requests to the failing service.
* After a specified period, the circuit breaker "closes" partially, allowing a few requests to test service health. If successful, the circuit breaker fully "closes," resuming normal operation.
* This pattern promotes resilience, enables graceful degradation, and minimizes downtime.

## CQRS

* CQRS (Command Query Responsibility Segregation) is an architectural pattern that separates read (query) and write (command) operations, enabling them to be scaled and optimized independently.
* This pattern simplifies the codebase and reduces contention between read and write operations, but introduces additional complexity in handling eventual consistency and synchronization between write and read models
* In a microservices architecture, CQRS can be combined with Event Sourcing to create highly scalable and performant systems.

‍

## Event Sourcing

* Event sourcing is a design pattern that captures all changes to an application’s state as a sequence of events.
* Event sourcing is a data persistence pattern where changes to application state are stored as a sequence of events. Instead of updating a single record, each state change is recorded as a new event.
* This pattern is useful in systems where you need a high level of auditability or where the history of events is a crucial part of the business domain.

## API Gateway Pattern

* The API Gateway design pattern provides a single-entry point for all client requests.
* It routes these requests to appropriate microservices. This pattern is essential in a microservices architecture, where you have numerous services with different interfaces.
* Without an API Gateway, a client would need to interact with each service directly, which would be complex and inefficient.
* The gateway can handle requests in one place, perform necessary transformations and route them to the appropriate services.

