**Micro-Services Design Principles & Patterns**

**Design Patterns vs Design Principles**

* Design Principles are general guidelines that can guide your class structure and relationships. On the other hand, Design Patterns are proven solutions that solve commonly reoccurring problems.
* Having said that, most of the practical implementations of these design principles are mostly done using one or more design patterns.
* FYI: <https://bootcamp.uxdesign.cc/software-design-principles-every-developers-should-know-23d24735518e>

**Some of these Design Principles are:**

1. **Encapsulate what varies:**

* This principle suggests, Identify the aspects of your applications that vary and separate them from what stays the same. Most design patterns like Strategy, Adapter, Facade, Decorator, Observer, Factory etc follow this principle.

**2. Favor composition:**

* Object oriented programming provides 2 types of relationships between classes and its instances. has-a (composition) and is-a (inheritance). This design principle essentially suggests us that “has-a relationship should be preferred over is-a relationship”.

3. **Program to interfaces:**

* Interface Segregation Principle

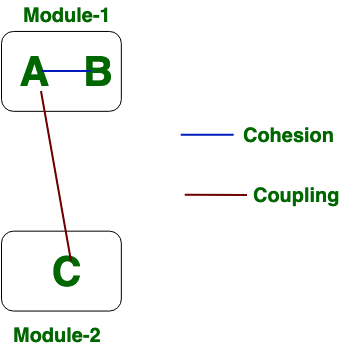
4.**Loose coupling:**

difference between coupling and cohesion in java?

* Cohesion is an indication of how **related and focused the responsibilities** of an software element are.
* Coupling refers to how **strongly a software element** is **connected** to other **elements**.
* The software element could be **class, package, component, subsystem or a system**. And while designing the systems it is recommended to have software elements that have **High** cohesion and support Low coupling.
* Low cohesion results in monolithic classes that are **difficult to maintain**, **understand and reduces re-usablity.**
* Similarly, **High Coupling results in classes that are tightly coupled** and changes tend not be non-local, difficult to change and reduces the reuse.

**Cohesion**: Cohesion is the indication of the relationship within the module. It is the concept of intra-module. Cohesion has many types but usually, high cohesion is good for software. Cohesion represents the functional strength of modules. Highly cohesive gives the best software. In cohesion, the module focuses on a single thing. Cohesion is created between the same module.

**Coupling**: Coupling is also the indication of the relationships between modules. It is the concept of the Inter-module. The coupling has also many types but usually, the low coupling is good for software. Coupling represents the independence among modules. Whereas loosely coupling gives the best software. In coupling, modules are connected to the other modules. Coupling is created between two different modules.



**5.SOLID principles**

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

**Principles Used to Design Microservice Architecture:**

1. Independent & Autonomous Services /Single Responsibility Principle /Loose Coupling
2. Scalability /Auto Scaling / High Availability
3. Decentralization/Decentralized Data Management
4. Resilient Services/ Fault Tolerance
5. Real-Time Load Balancing
6. Continuous delivery through DevOps Integration
7. Seamless API Integration and Continuous Monitoring
8. Isolation from Failures
9. Auto -Provisioning/Infrastructure Automation

* **Independent & Autonomous Services:**

Autonomous. Each component service in a microservices architecture can be developed, deployed, operated, and scaled without affecting the functioning of other services. Services do not need to share any of their code or implementation with other services. It will communicate through API calls only.

* **Scalability:**

A scalable, performant micro-service is one that is driven by efficiency, one that can not only handle a large number of tasks or requests at the same time, but can handle them efficiently and is prepared for tasks or requests to increase in the future. The ability to increase or decrease IT resources as needed to meet changing demand

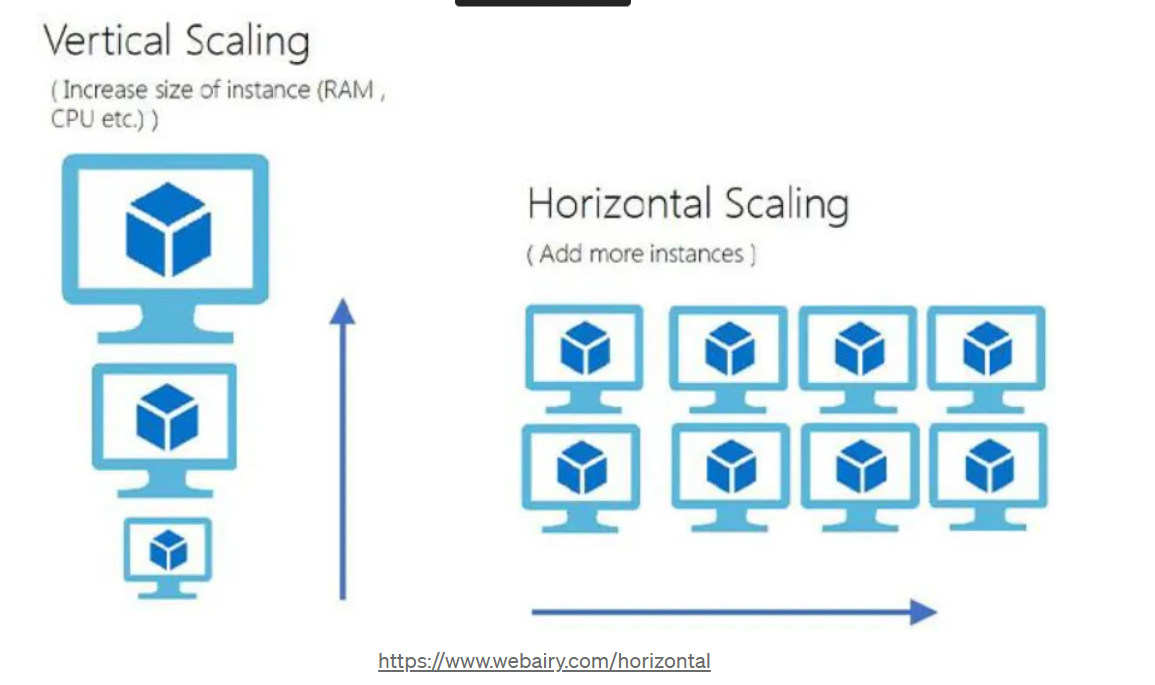
**Scalability**is simply measured by the number of requests an application can handle successfully. It can be measured by the **number of requests** and it can effectively support **simultaneously**. Once the application can no longer handle any more **simultaneous requests**, it has reached its scalability limit.

When your business to grow, in order to prevent downtime, and **reduce latency**, you must **scale your resources accordingly**. You can scale these resources through a combination of the **network bandwidth**, **CPU**and **physical memory** requirements, and hard disk adjustments.

**Horizontal scaling** and **Vertical scaling**both involve adding resources to your computing infrastructure, you must decide which is right for your application. Scaling horizontally and scaling vertically are similar in that they both involve **adding computing resources**to your infrastructure. There are distinct differences between the two in terms of implementation and performance.

**What’s the main difference?**  
**Horizontal**scaling means scaling by **adding more machines**to your pool of resources (also described as “**scaling out**”), whereas **Vertical**scaling refers to scaling by **adding more power**(e.g. CPU, RAM) to an existing machine (also described as “**scaling up**”).

<https://medium.com/design-microservices-architecture-with-patterns/scalability-vertical-scaling-horizontal-scaling-adb52ff679f>



* **Decentralization:**

Decentralization means each and every component of the system have the same authority, same power. They should share equal responsibility and control.

Decentralization refers to the distribution of decision-making authority across different entities within a system rather than being centralized in a single point. In the context of micro-services, decentralization can be achieved by allowing each micro-service to have its own separate domain of responsibility, data storage, and logic. This empowers teams to develop, deploy, and scale their services independently, without being tightly coupled to a central system.

FYI: <https://admin.headless.ly/posts/Microservices_Decentralization%3A_Enabling_Flexibility_in_Go_Headless>

FYI: <https://www.bemyaficionado.com/principles-of-microservices-decentralize-all-the-things/>

FYI: <https://www.bemyaficionado.com/principles-of-microservices-decentralize-all-the-things/>

**Orchestration VS Choreography**

What is Choreography?

Transferring this into IT architecture, choreography refers to the coordination and organization of interactions between different software components or systems. It is a design pattern that enables the development of distributed systems where the components interact with each other through a set of well-defined interfaces, without requiring a central authority or coordinator.

A choreographed system uses by definition event-driven communication, whereas orchestration uses command-driven communication.

Reference link : <https://camunda.com/blog/2023/02/orchestration-vs-choreography/>

### Benefits of Choreography

After having explained the term itself let us take a closer look at the benefits of using a choreographed pattern for a microservice architecture. The most important ones are outlined in the bulleted list below:

* **Loose coupling:** Choreography allows microservices to be loosely coupled, which means they can operate independently and asynchronously without depending on a central coordinator. This can make the system more scalable and resilient, as the failure of one microservice will not necessarily affect the other microservices.
* **Ease of maintenance:** Choreography allows microservices to be developed and maintained independently, which can make it easier to update and evolve the system.
* **Decentralized control:** Choreography allows control to be decentralized, which can make the system more resilient and less prone to failure.
* **Asynchronous communication:** Choreography allows microservices to communicate asynchronously, which can be more efficient and scalable than synchronous communication.

Overall, choreography can be a useful design pattern for building scalable, resilient, and maintainable microservice architectures. Though some of these benefits can actually turn into drawbacks.

### Downsides of Choreography

* **Complexity**: Choreography can be more complex to implement and maintain than orchestration, as it requires the development of well-defined interfaces and the coordination of interactions between the microservices.
* **Lack of central control**: Without a central coordinator, it can be more difficult to monitor and manage the interactions between the microservices in a choreographed system.
* **Lack of visibility**: Choreography can make it more difficult to get a holistic view of the system, as there is no central coordinator that has visibility into all of the interactions between the microservices.
* **Difficulty troubleshooting**: Without a central coordinator, it can be more difficult to troubleshoot issues in a choreographed system. So called event-chains can easily occur when multiple services are required for a certain domain. These are typically not visible and hence troubleshooting gets even trickier.

## What is Orchestration?

Orchestration refers to the coordination and management of the interactions and dependencies between different software components or systems. It is a design pattern that enables the development of distributed systems where a central authority or coordinator is responsible for managing and coordinating the actions of the different components.

An orchestrated system uses command driven communication.

### Benefits of Orchestration

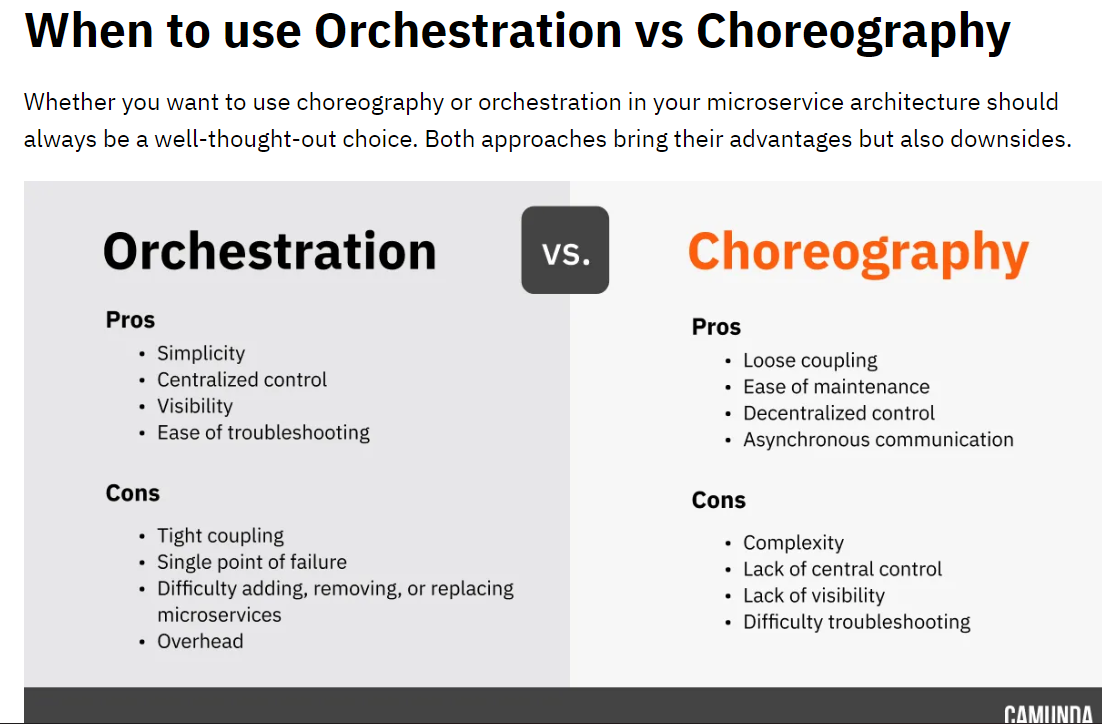
Orchestration in microservice architectures can lead to some nice benefits which compensate for the drawbacks of a choreographed system. A few of them are explained below:

* **Simplicity**: Orchestration can be simpler to implement and maintain than choreography, as it relies on a central coordinator to manage and coordinate the interactions between the microservices.
* **Centralized control**: With a central coordinator, it is easier to monitor and manage the interactions between the microservices in an orchestrated system.
* **Visibility**: Orchestration allows for a holistic view of the system, as the central coordinator has visibility into all of the interactions between the microservices.
* **Ease of troubleshooting**: With a central coordinator, it is easier to troubleshoot issues in an orchestrated system.

### Downsides of Orchestration

This pattern also also some general downsides. Let’s take a closer look at them and also try to find out how Camunda tackles them with their process orchestration platform.

* **Tight coupling:** Orchestration is typically known for being tighter coupled which can make the system less scalable and resilient. This is in theory correct but you could also overcome this issue by using asynchronous communication styles. A command itself could for instance also be transported via a message broker.
* **Single point of failure:** The central coordinator in an orchestrated system can be a single point of failure, which can make the system less resilient. Though there is luckily the possibility of deploying the orchestrator in a high availability fashion.
* **Difficulty adding, removing, or replacing microservices:** Changing the configuration of an orchestrated system can be more difficult, as it requires updating the central coordinator.
  + In the context of [**Camunda**](https://camunda.com/) this is not necessarily true. By using the [**BPMN 2.0 standard**](https://camunda.com/bpmn/) it is very easy to change this configuration. It can be even easier than in a choreographed system. For more information on this topic I can recommend reading [**this article**](https://blog.bernd-ruecker.com/how-to-tame-event-driven-microservices-5b30a6b98f86).
* **Overhead**: The central coordinator in an orchestrated system can add overhead to the system.



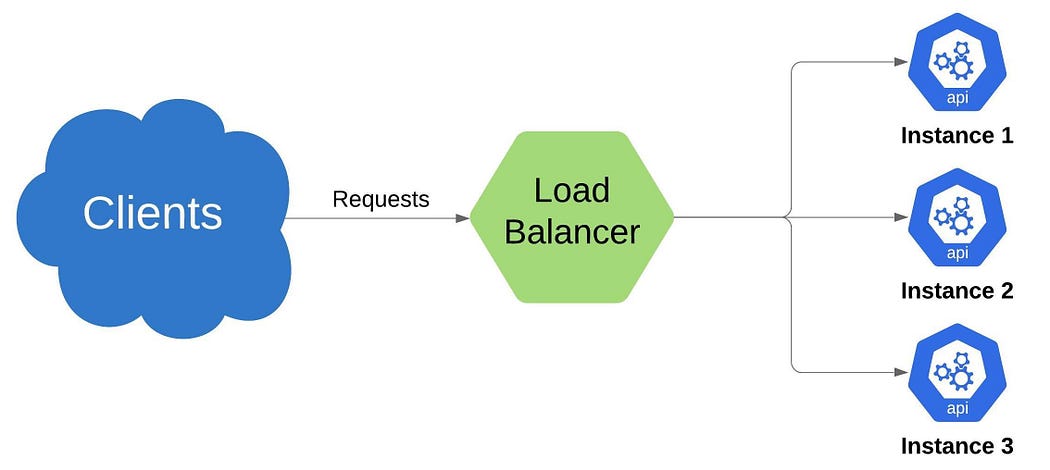
**Resilient Services:**

* Resilience is often referred to as the ability to 'bounce back' in the face of adversity (difficulty situation).
* A microservice needs to be resilient to failures and to be able to restart often on another machine for availability.
* This resiliency also comes down to the state that was saved on behalf of the microservice, where the microservice can recover this state from, and whether the microservice can restart successfully
* Common resiliency patterns used in application development include the **bulkhead pattern** and **circuit breaker pattern**. These patterns often incorporate other resiliency techniques like **fallback**, **retry**, and **timeout**.
* Reference Link: <https://medium.com/aspnetrun/microservices-resilience-and-fault-tolerance-with-applying-retry-and-circuit-breaker-patterns-c32e518db990>

**Real-Time Load Balancing:**

* Load balancing is the process of distributing traffic among different instances of the same application. To create a fault-tolerant system, it's common to run multiple instances of each application. Thus, whenever one service needs to communicate with another, it needs to pick a particular instance to send its request.
* Load balancing is the process of sharing, incoming network traffic in concurrent or discrete time between servers called a server farm or server pool. This sharing process can be evenly scale or can be performed according to certain rules. Rules like Round Robin, Least Connections etc.
* In microservices architecture there are two type of load balancing; they are
  + - 1. server side load balancing
      2. client side load balancing.

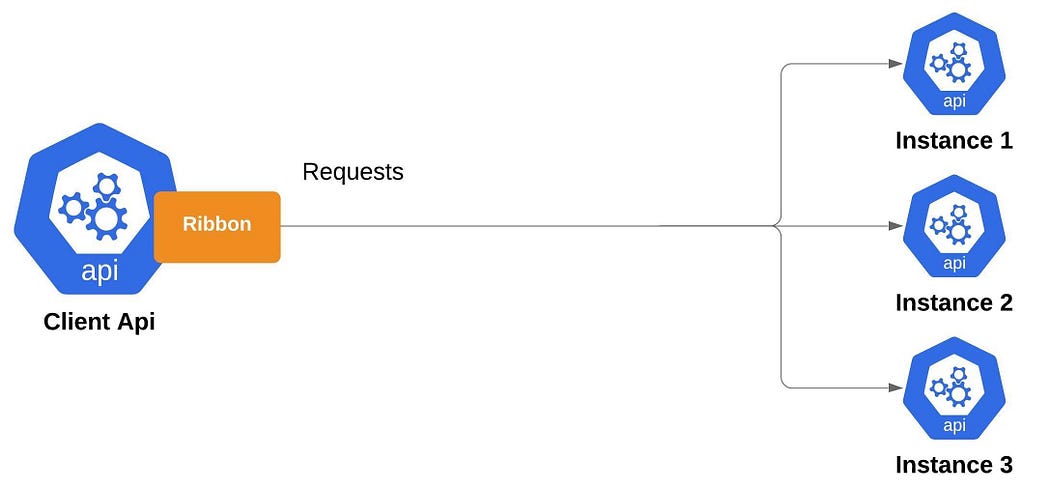
Server Side Load Balancing



Server side load balancing is a classical load balancing. The traffic is distributed by a load distributor placed in **front of the servers** and distributed to the servers that will perform the main work equally or according to certain rules. As examples most common used server side load balancers **nginx, netscaler etc.**

**Client Side Load Balancing**

In client side load balancing, the client handles the entire [load balancing](https://spring.io/guides/gs/spring-cloud-loadbalancer/) and the client API is expected to know all the instances of server API addresses that are available hard-coded within a service registry.



## Client Side Load Balancer vs Server Side Load Balancer

| **Client Side Load Balancer** | **Server-Side Load Balancer** |
| --- | --- |
| If you are keeping the load balancer on the client side and giving the load balancing responsibility to the client, then it’s called Client-Side Load Balancing. | If you are keeping the load balancer on the server side and giving the load balancing responsibility to the server, then it’s called Server-Side Load Balancing. |
| No more single point of failure in Client Side Load Balancer. | The main disadvantage of Server-Side Load Balancing is the single point of failure. |
| Less network latency as the client can directly call the backend servers. | Network latency rises in Server-Side Load Balancing. |
| Cost Reduction as there is no need for server-side load balancing. | The cost is high to implement Server-Side Load Balancing in comparison to Client-Side Load Balancing. |
| In this case, the microservice code is combined with the load balancer’s logic. | Since each microservice will have a particular load balancer, the complexity of the system rises and it is hard to manage. |
| Client-Side Load Balancer Provided by Spring Cloud: **Spring Cloud Load Balancer, Netflix Ribbon** | Server-Side Load Balancer Provided by Spring Cloud: **Spring Cloud Gateway, Netflix Zuul** |

**Continuous delivery through DevOps Integration**

**Continuous Integration vs Continuous Delivery vs Continuous Deployment:**

**Continuous Integration/Phase1:**

When a developer commits the code and it has to integrate with different type of platform to build and make ready for release such as Git hub, Sonar Qube(code quality check), Maven Build(for clean and build the package) and we need to follow the process as like some stages. To build the stages we need pipelines helps.

**Continuous Delivery:/Phase2**

Continuous Delivery (CD) is a DevOps practice that refers to the building, testing, and delivering improvements to the software code. The phase is referred to as the extension of the **Continuous Integration** phase to make sure that new changes can be released to the customers quickly in a **substantial** manner.

**Continuous Deployment:/Phase3**

When the step of Continuous Delivery is extended, it results in the phase of **Continuous Deployment**. Continuous Deployment (CD) is the **final stage** in the pipeline that refers to the automatic releasing of any developer changes from the repository to the production.

Continuous Deployment ensures that any change that passes through the stages of production is released to the end-users. There is absolutely no way other than any failure in the test that may stop the deployment of new changes to the output.

Reference link: <https://www.simplilearn.com/tutorials/devops-tutorial/continuous-delivery-and-continuous-deployment>

**Seamless API Integration and Continuous Monitoring:**

Continuous Monitoring:

Continuous monitoring is the process and technology used to detect **compliance and risk issues** associated with an organization's financial and operational environment

Through continuous monitoring, developers can detect issues with their services as soon as they arise and take measures to **prevent significant damage**. It also provides insights into how your services are **performing**, allowing you to make informed decisions.

ReferenceLink:<https://navendu.me/posts/introduction-to-monitoring-microservices/#:~:text=Without%20proper%20monitoring%2C%20microservices%20can,measures%20to%20prevent%20significant%20damage>.

**Tools used:** Splunk, Kibana, Nagios, Elastic Search Engine, etc.

**Isolation from Failures**

This is very important when designing and running reliable systems, especially in the world of microservice architectures. Fundamentally, fault tolerance means that a system can continue to operate and provide service even if something goes wrong. Believe me, problems can occur in many ways, including hardware malfunctions, software glitches, network issues, and even human error.

**Why is fault tolerance so important in microservice architectures?** In this type of setup there are many different services that all work independently. This means that each service can experience its own failure, creating a domino effect that can bring down the entire system. not good.

**Temporary and Permanent Failures.**

Temporary outages are temporary and are usually caused by **network problems** or **system congestion**. This issue is not permanent and can often be resolved by retrying the process or waiting for the issue to be resolved automatically.

Permanent disability, on the other hand, is more severe. These can be caused by hardware malfunctions, software bugs, or human errors that require manual intervention to correct. In the event of such a failure, a backup system or redundancy mechanism should be put in place to keep things running smoothly.

In Micro-services, Fault Tolerance can be achieved by implementing below techniques:

* + - 1. Partition:
      2. Circuit breaker:
      3. Graceful Degradation:

**Auto-Provisioning/Infrastructure Automation:**

What is provisioning?

* Provisioning is the process of creating and setting up IT infrastructure, and includes the steps required to manage user and system access to various resources.

Benefits of automated provisioning:

* Today, most provisioning tasks can easily be handled through automation, using infrastructure-as-code (IaC). With IaC, infrastructure specifications are stored in configuration files, which means that developers just need to execute a script to provision the same environment every time. Codifying infrastructure gives IT teams a template to follow for provisioning, and although this can still be accomplished manually, automation tools can make this process far more efficient.

Example: Ancible, pippet, chuf, etc.

**Micro-Services Design Patterns:**

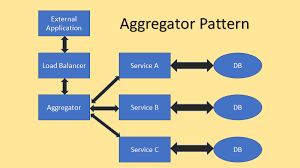
1. Aggregator
2. API Gateway
3. Chained or Chain of Responsibility
4. Asynchronous Messaging
5. Database or Shared Data
6. Event Sourcing
7. Branch
8. Command Query Responsibility Segregator(CQRS)
9. Circuit Breaker
10. Decomposition
11. SAGA

Reference link: <https://dzone.com/articles/design-patterns-for-microservices>

1. **Aggregator:**

In IT industry, aggregator refers to a website or program that collects related items of data and displays them.

In Microservices Architecture, we split a large, complex application into small, autonomous, independently deployable services. Therefore, it’s necessary to think about how to collaborate the data returned by each service. In IT industry, aggregator refers to a website or program that collects related items of data and displays them. So, in microservices the Aggregator Design Pattern is a service that receives a request, then makes requests of multiple services, combines the results and responds to the initiating request.

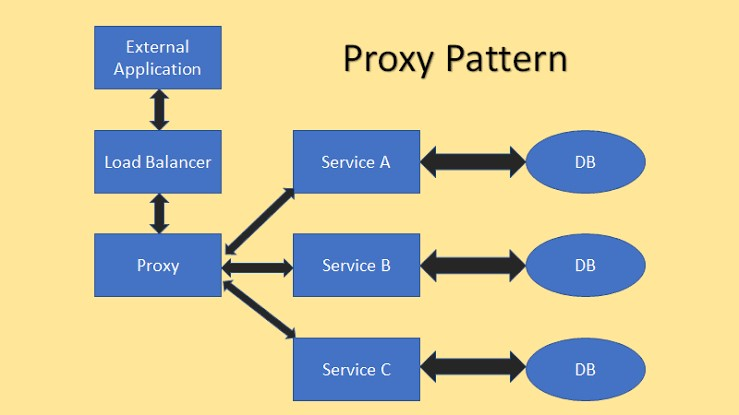


In Aggregator Pattern, there are 3 ways to implement it within Microservices application.

1. Scatter Gather Pattern/Asynchronous Communication/Proxy Pattern
2. Chained Pattern/Synchronous Communication
3. Branch Pattern/ Combination of Synchronous and Asynchronous

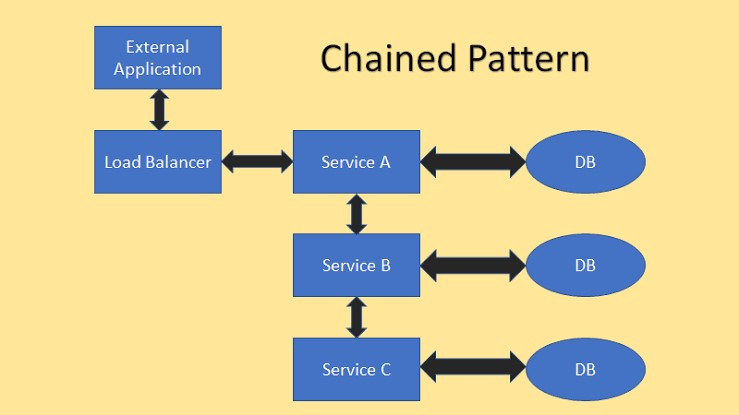
**Scatter Gather Pattern/Asynchronous Communication/Proxy Pattern:**

|  |
| --- |
| public class AsyncAggregatorMicroservice {  private final ExecutorService executorService;  private final Microservice1Client microservice1Client;  private final Microservice2Client microservice2Client;  private final Microservice3Client microservice3Client;  public AsyncAggregatorMicroservice(ExecutorService executorService, Microservice1Client microservice1Client, Microservice2Client microservice2Client, Microservice3Client microservice3Client) {  this.executorService = executorService;  this.microservice1Client = microservice1Client;  this.microservice2Client = microservice2Client;  this.microservice3Client = microservice3Client;  }  public CompletableFuture<AggregatedResponse> processRequest(Request request) {  CompletableFuture<Response1> response1Future = CompletableFuture.supplyAsync(() -> microservice1Client.processRequest(request), executorService); CompletableFuture<Response2> response2Future = CompletableFuture.supplyAsync(() -> microservice2Client.processRequest(request), executorService); CompletableFuture<Response3> response3Future = CompletableFuture.supplyAsync(() -> microservice3Client.processRequest(request), executorService); return CompletableFuture.allOf(response1Future, response2Future, response3Future) .thenApply(v -> new AggregatedResponse(response1Future.join(), response2Future.join(),  response3Future.join()));  }  } |



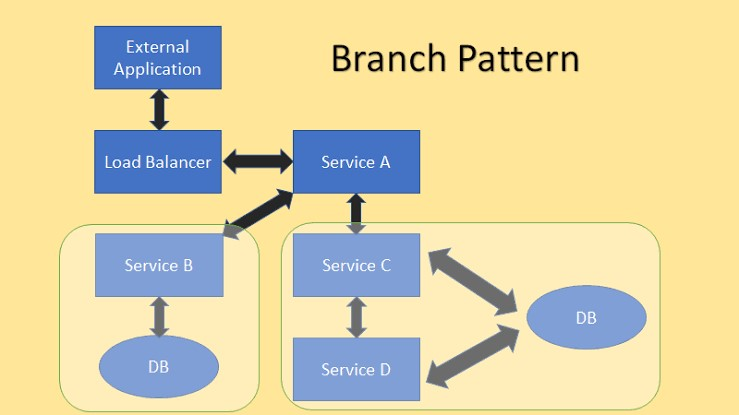
**Chained Pattern/Synchronous Communication:**

|  |
| --- |
| public class SyncAggregatorMicroservice {  private final Microservice1Client microservice1Client;  private final Microservice2Client microservice2Client;  private final Microservice3Client microservice3Client;  public SyncAggregatorMicroservice(Microservice1Client microservice1Client, Microservice2Client microservice2Client, Microservice3Client microservice3Client) {  this.microservice1Client = microservice1Client;  this.microservice2Client = microservice2Client;  this.microservice3Client = microservice3Client;  }  public AggregatedResponse processRequest(Request request) {  Response1 response1 = microservice1Client.processRequest(request);  Response2 response2 = microservice2Client.processRequest(request);  Response3 response3 = microservice3Client.processRequest(request);  return new AggregatedResponse(response1, response2, response3);  }  } |



**Branch Pattern/ Combination of Synchronous and Asynchronous:**

|  |
| --- |
| public class HybridAggregatorMicroservice {  private final ExecutorService executorService;  private final Microservice1Client microservice1Client;  private final Microservice2Client microservice2Client;  private final Microservice3Client microservice3Client;  public HybridAggregatorMicroservice(ExecutorService executorService, Microservice1Client microservice1Client, Microservice2Client microservice2Client, Microservice3Client microservice3Client) {  this.executorService = executorService;  this.microservice1Client = microservice1Client;  this.microservice2Client = microservice2Client;  this.microservice3Client = microservice3Client;  }  public AggregatedResponse processRequest(Request request) {  CompletableFuture<Response1> response1Future = CompletableFuture.supplyAsync(() -> microservice1Client.processRequest(request), executorService); Response2 response2 = microservice2Client.processRequest(request);  CompletableFuture<Response3> response3Future = CompletableFuture.supplyAsync(() -> microservice3Client.processRequest(request), executorService); return CompletableFuture.allOf(response1Future, response3Future) .thenApply(v -> new AggregatedResponse(response1Future.join(), response2, response3Future.join()));  } |



ReferenceLink:

* <https://medium.com/nerd-for-tech/design-patterns-for-microservices-aggregator-pattern-99c122ac6b73>
* <https://www.java67.com/2023/01/aggregator-microservice-pattern-in-java.html>

Basically, The Service aggregator design pattern receives a request from the client or API Gateways, then dispatches requests of multiple internal backend microservices, and then combines the results and responds back to the initiating request in 1 response structure.

The Aggregator Microservice Pattern in Java is a useful design pattern for composing complex services by aggregating the responses of multiple independent microservices. In Java, this pattern can be implemented using asynchronous communication, synchronous communication, or a combination of both, depending on the requirements of the system.

**Usage of this Design Pattern:**

1. Use this pattern when requires low transaction rate/low demand/low expected increased rate of demand activities.

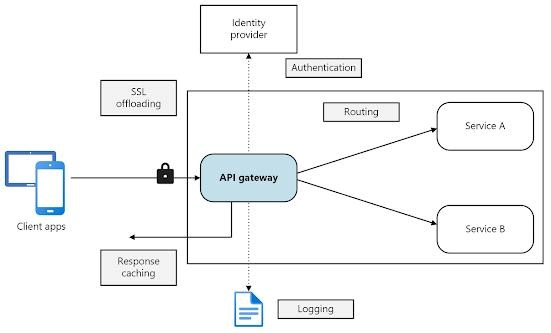
2. Use this pattern when project requires more than one team to implement it in parallel.

**Advantages of Aggregator Design Pattern:**

* we can reduce chattiness and communication overhead between the client and microservices.
* This pattern is easy to understand and implement.
* This pattern provides single access point for microservices.

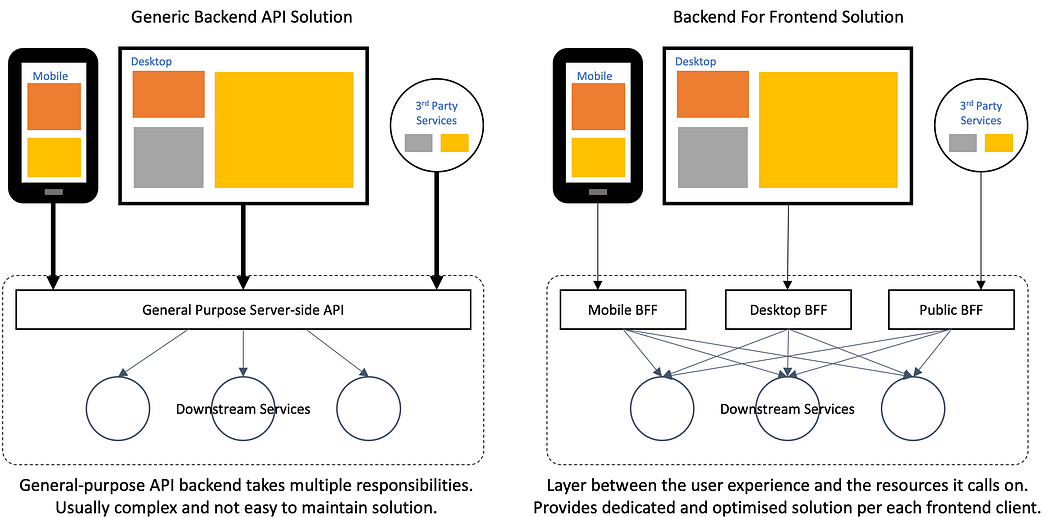
1. **API Gateway:**

* The API gateway pattern is recommended if you want to design and build complex large microservices-based applications with multiple client applications.
* It provides a single entry point to the APIs with encapsulating the underlying system architecture.
* In summary, the API gateway locate between the client apps and the internal microservices.
* It is working as a reverse proxy and routing requests from clients to backend services.
* It is also provide cross-cutting concerns like authentication, SSL termination, and cache.
* So there are several client applications connect to single API Gateway in here.
* We should careful about this situation, because if we put here a single API Gateway, that means its possible to single-point-of-failure risk in here.
* If these client applications increase, or adding more logic to business complexity in API Gateway, it would be anti-pattern.
* API Gateway service can be growing and evolving based on many different requirements from the client apps.
* That’s why the best practices is splitting the API Gateway in multiple services or multiple smaller API Gateways.
* We will make use BFF-Backend-for-Frontend pattern for above scenario.
* In summary, we need to careful about using single API Gateway, it should be segregated based on business boundaries of the client applications and not be a single aggregator for all the internal microservices.



**Backend for Frontend (BFF) design pattern:**

You need to think of the user-facing application as being two components — a client-side application living outside your perimeter and a server-side component (BFF) inside your perimeter. BFF is a variant of the API Gateway pattern, but it also provides an additional layer between microservices and each client type separately. Instead of a single point of entry, it introduces multiple gateways. Because of that, you can have a tailored API that targets the needs of each client (mobile, web, desktop, voice assistant, etc.), and remove a lot of the bloat caused by keeping it all in one place. The below image describes how it works.



ReferenceLink:<https://medium.com/mobilepeople/backend-for-frontend-pattern-why-you-need-to-know-it-46f94ce420b0>

**Main Features of API Gateway Pattern**

1- Reverse proxy or gateway routing

* Another benefit is abstracting internal operations, API GW provide abstraction over the backend microservices, so even there is changes on backend microservices, it wont be affect to client applications. That means don’t need to update client applications when changing backend services.

2- Requests aggregation

* This is part of gateway aggregation pattern features. API Gateway can aggregate multiple internal microservices into a single client request. With this approach, the client application sends a single request to the API Gateway
* After that API Gateway dispatches several requests to the internal microservices and then aggregates the results and sends everything back to the client application in 1 single response. The main benefit of this gateway aggregation pattern is to reduce chattiness communication between the client applications and the backend microservices.

3- Cross-cutting concerns and gateway offloading

* This is part of gateway offloading pattern features. Since API Gateway handle client request in centralized placed, its best practice to implement cross cutting functionality on the API Gateways.
* Authentication and authorization
* Service discovery integration
* Response caching
* Retry policies, circuit breaker, and QoS
* Rate limiting and throttling
* Load balancing
* Logging, tracing, correlation
* Headers, query strings, and claims transformation
* IP allowlisting

**Tools Using to Achieve API Gate Way:**

* Kong Gateway.
* Apache APISIX.
* Tyk.
* KrakenD.
* Gravitee.io.
* Apigee.
* Amazon API Gateway.
* Azure API Management.
* Ambassador.
* Gloo.

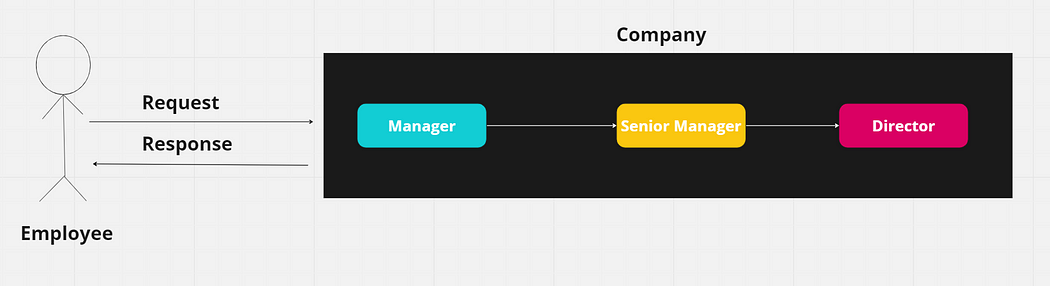
**How to select your API Gateway**

Here are some characteristics to consider when you choose an API Gateway or API Management solution that perfectly fits your need. Note that the following list of attributes is not organized in order of priority:

* Primary edge functionalities.
* Security.
* Simple configuration.
* Installation and deployment Options.
* Self-hosted vs Cloud-hosted.
* Customization.
* Integration.
* Performance.
* Features.
* Community.
* Price.

1. **Chained or Chain of Responsibility(Handler Design Pattern)**

* The Chain of Responsibility design pattern is a behavioral pattern that allows an object to pass a request along a chain of potential handlers until it is handled by an appropriate object. This pattern promotes loose coupling between the sender of a request and its receivers, and it allows multiple objects to have a chance to handle the request without explicitly specifying the receiver.
* Suppose you are an employee in a company now you give a request of some approval to your manager if they can approve it , then you can get response from them otherwise the request is passed to senior manager , if they can approve it , then you can get response otherwise it goes to next person or level , so here you as an employee is completely unaware of how your request is getting approved and how multiple people are handling your request , this is Chain Responsibility Pattern .



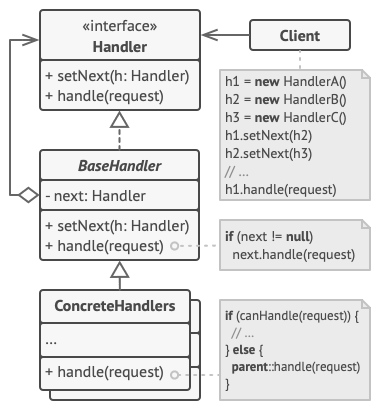
ref link: <https://medium.com/@ngneha090/beginners-guide-to-chain-responsibility-design-pattern-in-java-e1e0ddac2cb6>

**Components of Chain Responsibility Pattern:**

**Handler**: The Handler is an interface or abstract class that defines the common interface for handling requests. It typically includes a method like handleRequest().

**ConcreteHandler**: ConcreteHandler classes implement the Handler interface and provide specific implementations for handling requests. Each ConcreteHandler has a reference to the next handler in the chain. It decides whether to handle the request or pass it to the next handler in the chain.

**Client**: The Client initiates the request and starts the chain of handling. It is responsible for creating the chain of handlers and linking them together.



|  |
| --- |
| public abstract class RequestHandler {  String name;  RequestHandler nextHandler;  private RequestHandler(){  }  public RequestHandler(String name){  this.name=name;  }  abstract void setNext(RequestHandler nextHandler);  void approve (int id)  {  if(this.nextHandler != null)  this.nextHandler.approve(id);  else  System.out.println("request cannot be approved");  }  } |

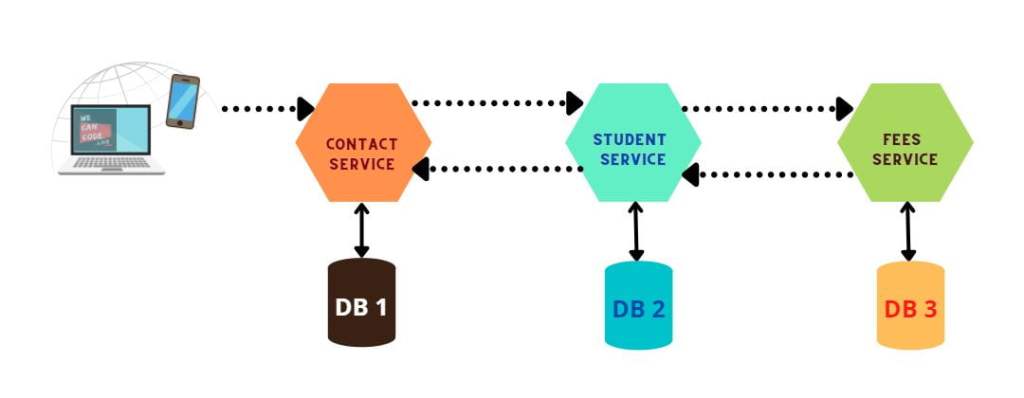
|  |
| --- |
| public class Manager extends RequestHandler{  public Manager(){  super("manager");  }  @Override  void setNext(RequestHandler nextHandler){  this.nextHandler=nextHandler;  }  @Override  void approve(int id){  if(id>=1 && id<=20)  {  System.out.println("Request Approved");  }  else{  super.approve(id);  }  } |

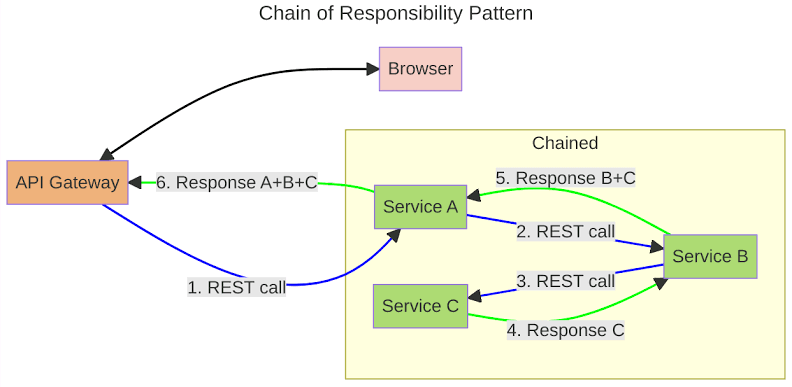
|  |
| --- |
| public class SeniorManager extends RequestHandler{  public SeniorManager(){  super("Senior manager");  }  @Override  void setNext(RequestHandler nextHandler){  this.nextHandler=nextHandler;  }  @Override  void approve(int id){  if(id>=21 && id<=40)  {  System.out.println("Request Approved");  }  else{  super.approve(id);  }  }  } |

|  |
| --- |
| public class Director extends RequestHandler{  public Director(){  super("Director");  }  @Override  void setNext(RequestHandler nextHandler){  this.nextHandler=nextHandler;  }  @Override  void approve(int id){  if(id>=41 && id<=80)  {  System.out.println("Request Approved");  }  else{  super.approve(id);  }  }  } |

|  |
| --- |
| public class Main {  public static void main(String[] args) {  RequestHandler manager=new Manager();  RequestHandler seniorManager=new SeniorManager();  RequestHandler director=new Director();  manager.setNext(seniorManager);  seniorManager.setNext(director);  manager.approve(19);  manager.approve(90);  }  } |

**Chained or Chain of Responsibility in Microservices:**

****

****

* Consider we have Contact, Student & Fees three microservices, each web service communicates with the following or next service as in waterfall.
* Yes, Contact service calls Student service and Student service calls Fees service. Response of Fees and Student will be returned back to Contact service.

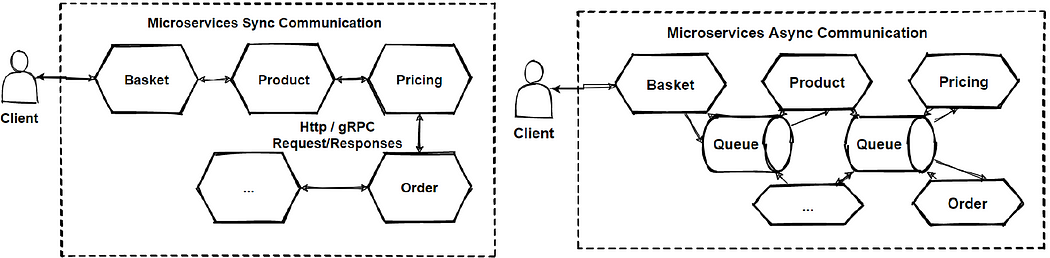
What is Chained or Chain of Responsibility Design Pattern:

* Chained Or Chain Of Responsibility Design Pattern in Microservice
* Here the Client must wait till the response comes back after the consolidated response is available, meaning the Client will be waiting for the chain of response to be returned back as a response.
* Use this service where the client is not expecting any response. (Something like calling a batch job, synchronous tasks or orchestrated flow of actions.
* All these services use the HTTP request or response for communicating,
* Note: Each service request and response looks different from the chained flow.

1. **Asynchronous Messaging Design Pattern:**

Microservices Asynchronous Message-Based Communication between Backend Internal Microservices:

* Synchronous communication is good if your communication is only between a few microservices. But when it comes to several microservices need to call each other and wait some long operations until finished, then we should use async communication. Otherwise that dependency and coupling of microservices will create bottleneck and create serious problems of the architecture.
* So we should understand that isolation is important between microservices, we should isolate services as much as possible. Since microservices are distributed system running on multiple processes, services required to interact with each other with using an inter-process communication protocols like sync **HTTP, gRPC or async AMQP protocols.**
* If we have a few interaction with querying microservices then we should use HTTP request/response with resource APIs.
* But when it comes to busy interactions in communication across multiple microservices, then we should use asynchronous messaging platforms like message broker systems.
* We said Asynchronous protocols, which is **AMQP** protocol for performing async message transmissions. In this **AMQP** protocol, the producer send a message and doesn’t wait a response. It only send message and expects that it will consume by subscriber services via to message broker systems.



**synchronous communication:**

Let’s see the image, You can see in the synchronous communication is becoming chain of request and highly coupled depended between microservices. So this is pain point of architecture and we can say this is anti-pattern and need to re-design system.

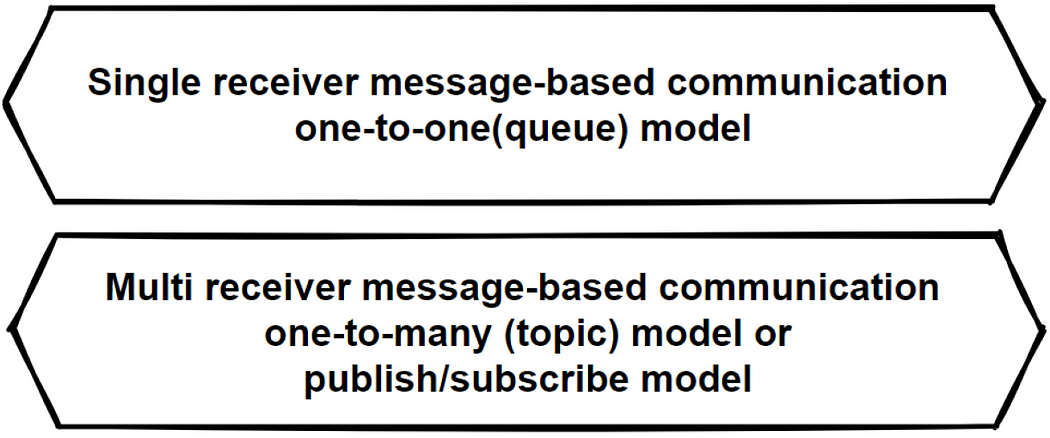
**asynchronous communication:**

With the asynchronous communication, this problem can be solve. Microservices communicate with each other over the message broker system in async way. Mostly its good to use **publish/subscribe pattern** with **message broker systems** that we will see in the upcoming sections.

**Asynchronous Message-Based Communication in Microservices Architecture:**

If you have multiple microservices are required to interact each other and if you want to interact them without any dependency or make loosely coupled, than we should use Asynchronous message-based communication in Microservices Architecture. Because Asynchronous message-based communication is providing works **with events**. So events can place the communication between microservices. We called this communication is a **event-driven communication**.eg: Kafka

2 Type of Asynchronous Messaging Communication:

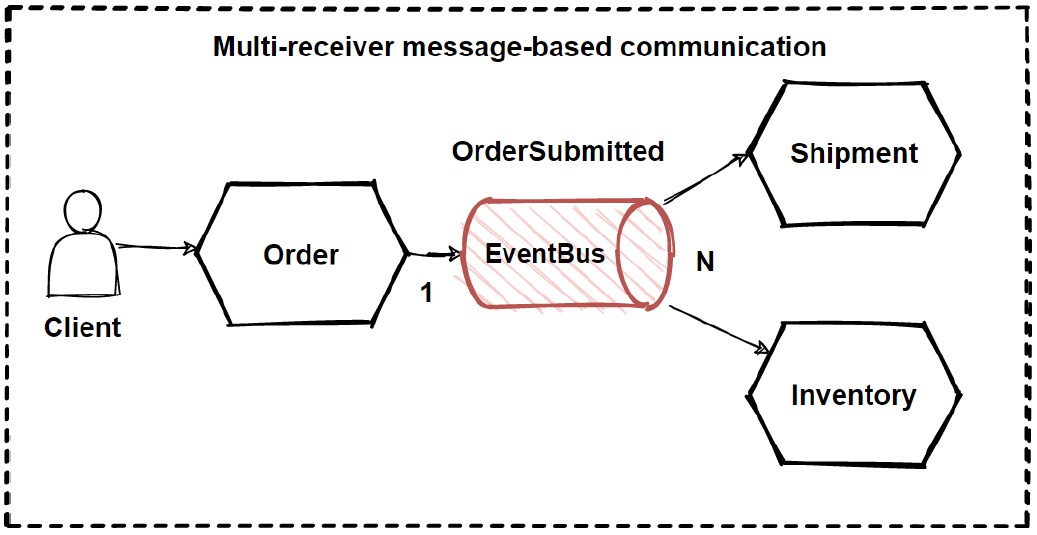


**Single-receiver message-based communication**

This communication is basically for performing one-to-one or point-to-point communications. If we will send 1 request to the specific consumer, and this operation will take long time, then its good to use this Single-receiver async one-to-one communication.

**Multiple-receiver message-based communication:**

* This communication is basically for performing publish/subscribe mechanisms that has multiple receivers. So in this communication, the producer service publish a message and it consumes from several microservices which’s are subscribing this message on the message broker system. These publish/subscribe operations should require an event bus interface to publish events to any subscriber.
* Mostly this async communication is using in event-driven architectures. In Asynchronous event-driven communication, microservice publishes an event when something happens. Example use case can be like a price change in a product microservice. Price Changed event can subscribed from **Shopping Cart microservice** in order to update **basket price asynchronously**.

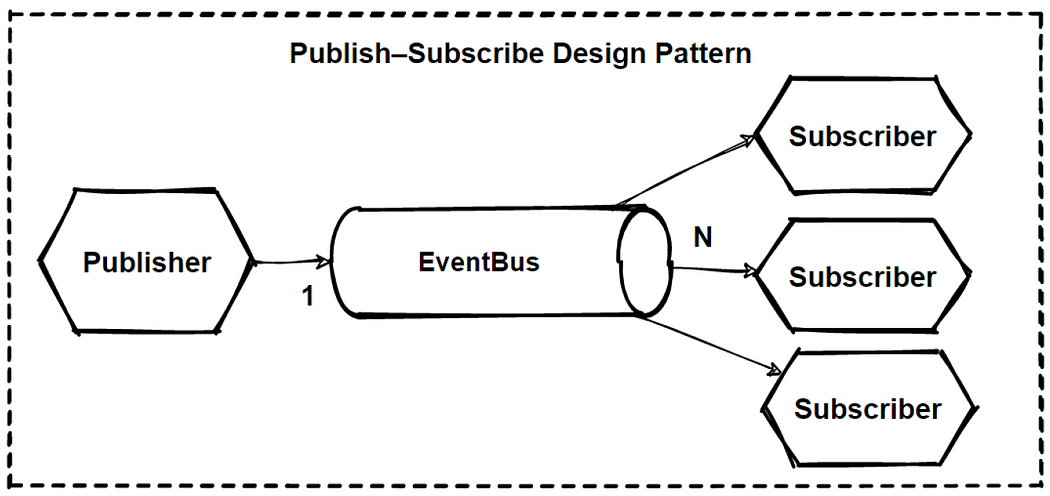


|  |
| --- |
| * Lets check the above image here, This is an example of publish/subscribe messaging based on event-driven communication. * The Order microservices publishes OrderSubmitted events to an event bus and Shipment and Inventory microservices can subscribe to this event, and take their internal actions according to event details. |

* This publish/subscribe patterns implement by using event bus**. And the event bus can also have implementations with messaging broker systems that** supports asynchronous communication and a publish/subscribe model like **Kafka and Rabbitmq.**

**Publish–Subscribe Design Pattern:**

* Actually we already saw how Publish–subscribe pattern is works with async communication but let me explain here also, Publish–subscribe is a messaging pattern, that has sender of messages which’s are called publishers, and has specific receivers which’s are called subscribers.

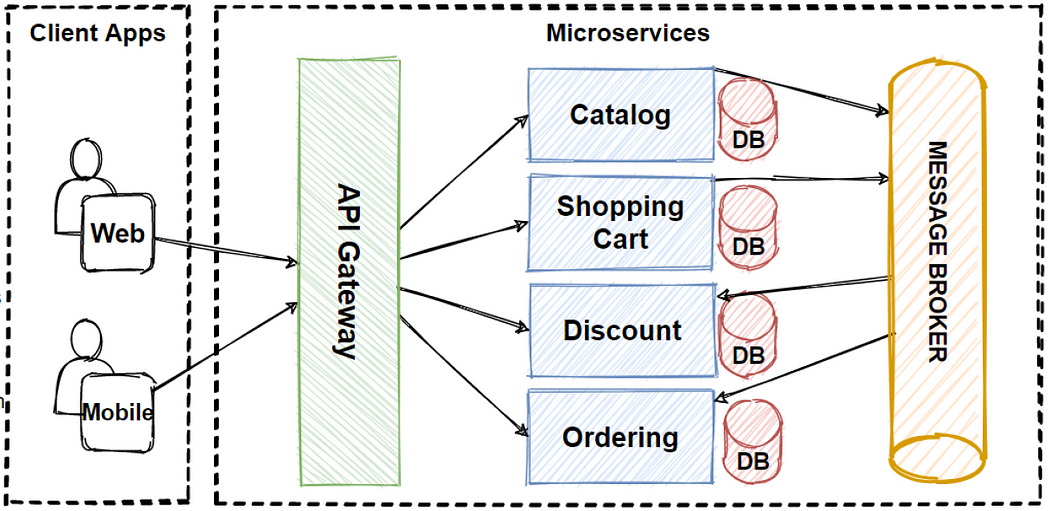


* So publishers don’t send the messages directly to the subscribers. Instead categorize published messages and send them into message broker systems without knowledge of which subscribers are there. Similarly, subscribers express interest and only receive messages that are of interest, without knowledge of which publishers send to them.
* By this way, Publishers and subscribers communicate each other without coupling or any dependency of each other. It decouples microservices communications, so that microservices can be managed independently, and scale independently without worrying about communications. So It increases scalability and improves responsiveness of the system.

We will use Principles and Patterns are;

* Dependency Inversion Principles (DIP)
* Publish–Subscribe Design Pattern

So we will add a “Pub/Sub Message Broker” in order to communicate aync between microservices to provide loosely coupling.



**Technology Choices — Adapting Technology Stack:**

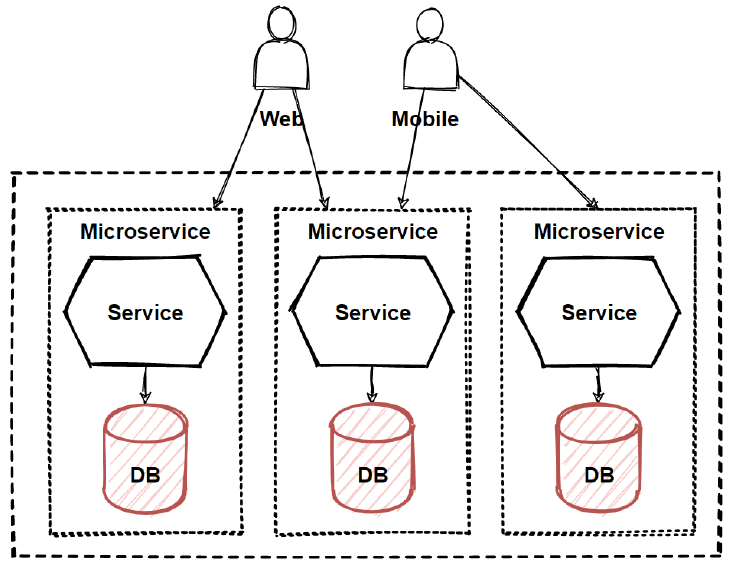
1. Kafka -
2. RabbitMQ
3. JMS(Java Messaging System)
4. Google Pub/Sub
5. Amazon Services
6. ActiveMQ
7. Azure Services
8. Azure Service Bus -
9. Amazon Kinesis for event streaming.
10. **Database or Shared Data Design Pattern/** **shared-database-per-service pattern:**

Microservices architecture is constantly growing. It brings a lot of benefits, especially over obsolete monolith architecture. On the other hand, there are multiple challenges while developing a project using microservices. One of the most important concerns is database design. There are two crucial questions as it comes to data design. How to organize the data and where to store it?

There are two main options for organizing the databases when using microservices architecture:

* Database per service
* Shared database
* Ref link: <https://waytoeasylearn.com/learn/database-per-service-pattern/>

**Database per service**



**Explanation:**

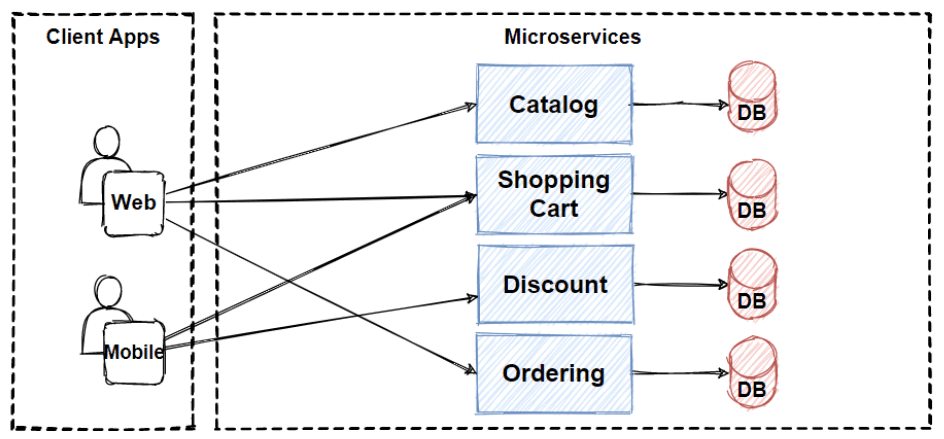
* One of the core characteristics of the microservices architecture is the **loose coupling of services**.
* For that reason every service must have its own databases**, it can be polyglot persistence among microservices.**

Let’s think about our e-commerce application.

* We will have Product — Ordering, and Shopping Cart microservices that each services data in their own databases.
* Any changes to one database don’t impact other microservices.
* The service’s database can’t be accessed directly by other microservices.
* Each service’s persistent data can only be accessed via Rest APIs.
* So database per microservice provides many benefits, especially for evolving rapidly and supporting massive scale systems.

Data schema changes made easy without impacting other microservices

* Each database can scale independently
* Microservices Domain data is encapsulated within the service
* If one of the database server is down, this will not affect to other services



So if we see the image that example of the microservice architecture of e-commerce application;

* The product microservice using NoSQL document database for storing catalog related data which is storing JSON objects to accommodate high-volumes of read operations.
* The shopping cart microservice using a distributed cache that supports its simple, key-value data store.
* The ordering microservice using a relational database to accommodate the rich relational structure of its underlying data.

Because of the ability of massive scale and high availability, NoSQL databases are getting high popularity and becoming widely used in an enterprise applications. Also, their schema-less structure gives flexibility to developments on microservices. We discuss NoSQL databases later in upcoming tutorials.

**Drawbacks:**

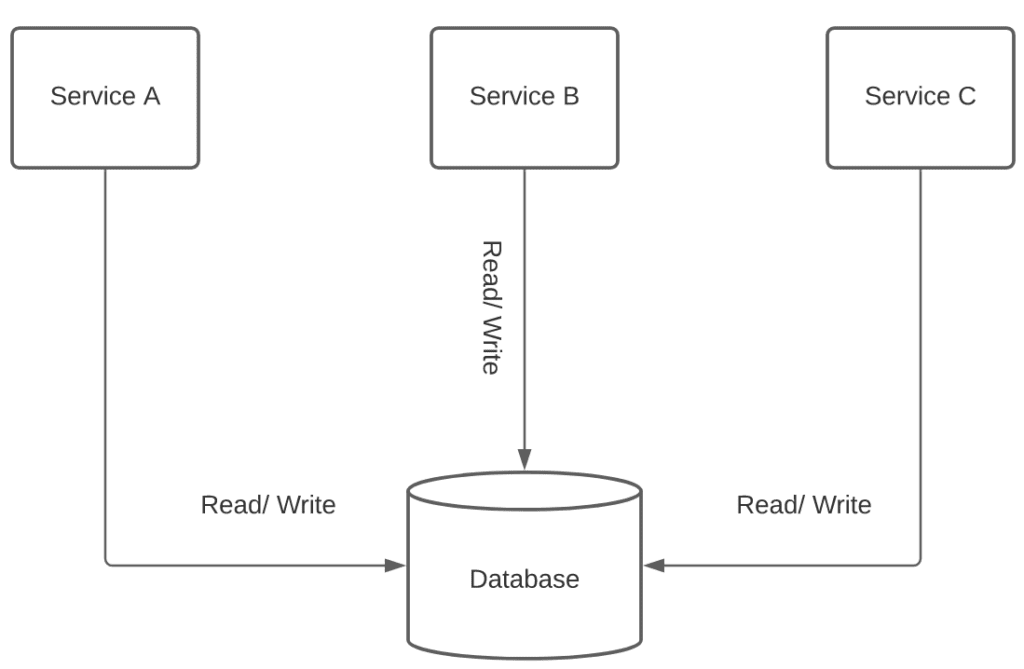
* Despite all of those benefits, there are some serious drawbacks and challenges regarding the database per service approach.
* As we mentioned earlier, each microservice can only access directly its own data store.
* Therefore, services need a communication method to exchange data. So, each service must provide a clear API.
* Consequently, there is a need for a failure protection mechanism in case the communication fails.
* Let’s say we send payment requests from service A to service B.
* Service A awaits for the response to perform appropriate action based on the result.
* During that, service B goes offline.
* We need to handle the situation and inform service A about the result when B is back online.
* The circuit breaker mechanism can help out here.

2. **Shared Database:/SAGA:**

A shared database is considered an anti-pattern. Although, it’s debatable. The point is that **when using a shared database, the microservices lose their core properties: scalability, resilience, and independence.** Therefore, a shared database is rarely used with microservices.

When a shared database seems to be the best option for the microservices project, we should rethink if we really need the microservices.

**Maybe the monolith would be the better choice**. Let’s see how a shared database approach looks like:



* Another important thing is no need to exchange stored data between microservices.
* Microservices with **shared databases can’t easily scale.**
* The use cases of using a shared database with microservices aren’t common.
* An example could be a temporary state while migrating the monolith to microservices.
* The primary benefit of the shared database over per service is transaction management.
* There is no need to span the transactions over the services.
* Another important thing is no need to exchange stored data between microservices.
* So, the API is simplified, and there is no problem with the consistency of data and state in case the communication fails.
* There are some serious drawbacks though.
* Microservices with shared databases can’t easily scale.
* What is more, the database will be a single point of failure.
* Changes related to the database could impact multiple services.
* Besides, microservices won’t be independent in terms of development and deployment as they connect to and operate on the same database.

1. **Command(write/update/delete) and Query(read) Responsibility Segregator (CQRS):**

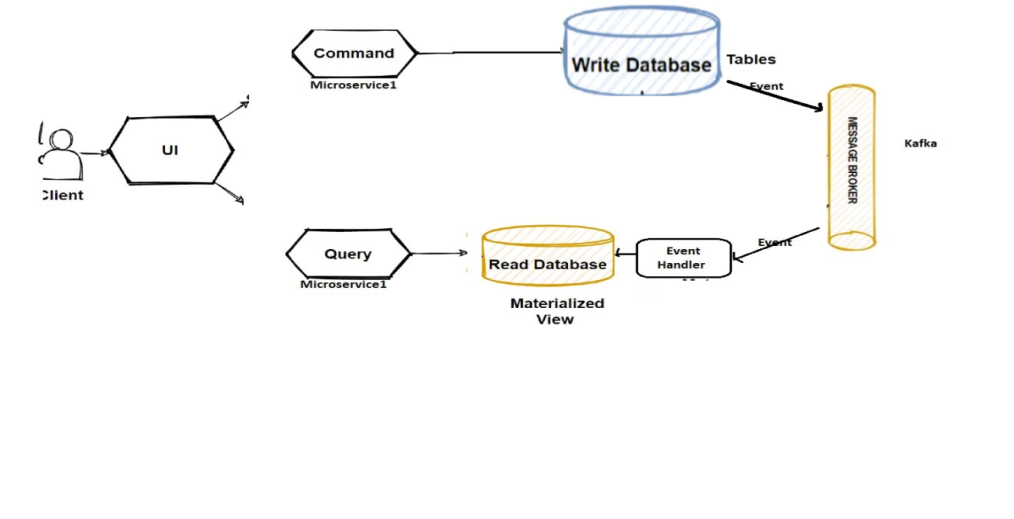
|  |
| --- |
| Command = create/update/delete operations which change the state of entity.  Query = To get the exact state of the entity by applying some queries. |

* The Command Query Responsibility Segregator (CQRS) pattern is a software design pattern that **separates the responsibilities of reading and writing data**.
* It separates the read model, which is responsible **for retrieving data**, from the write model, which is responsible for **modifying data**.

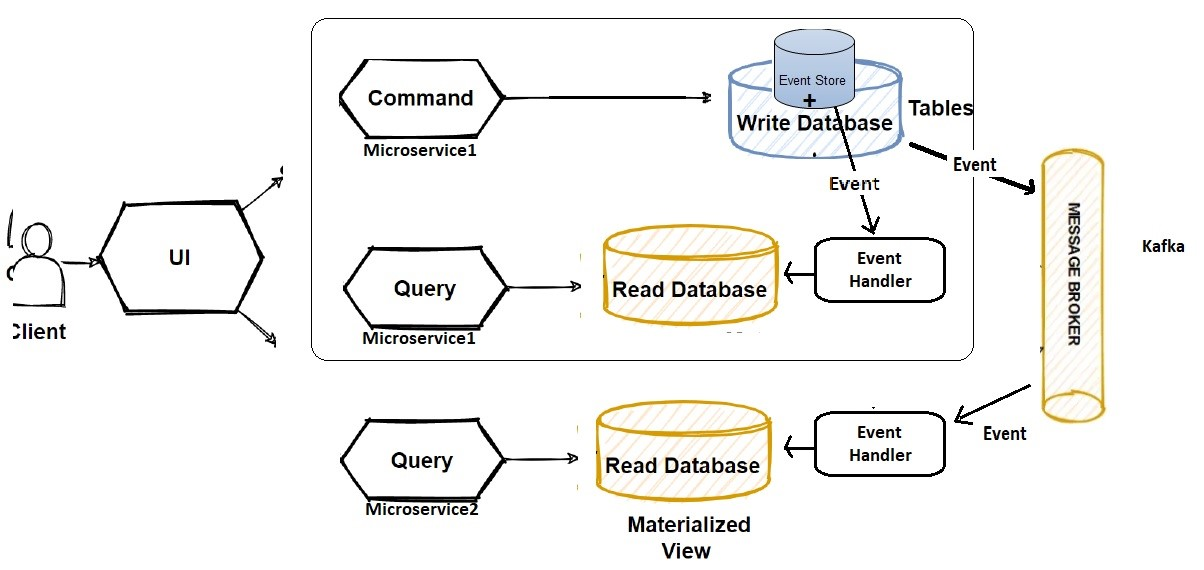
**Why CQRS design pattern came?**

* Normally, in monolithic applications, most of time we have 1 database and this database should respond both **query and update operations**. That means a database is both working for **complex join queries**, and also perform **CRUD** operations. But if the application goes more complex this query and crud operations will be also is going to be **un-manageable situation**.
* In example of reading database, if your application required some query that needs to join more than 10 table, this will lock the database due **to latency of query computation**.
* Also if we give example of writing database, when performing crud operations we would need to make complex validations and process long business logics, **so this will cause to lock database operations**.

**CQRS without Event Sourcing:**

****

**CQRS with Event Sourcing:**



**CQRS “separation of concerns” strategy :**

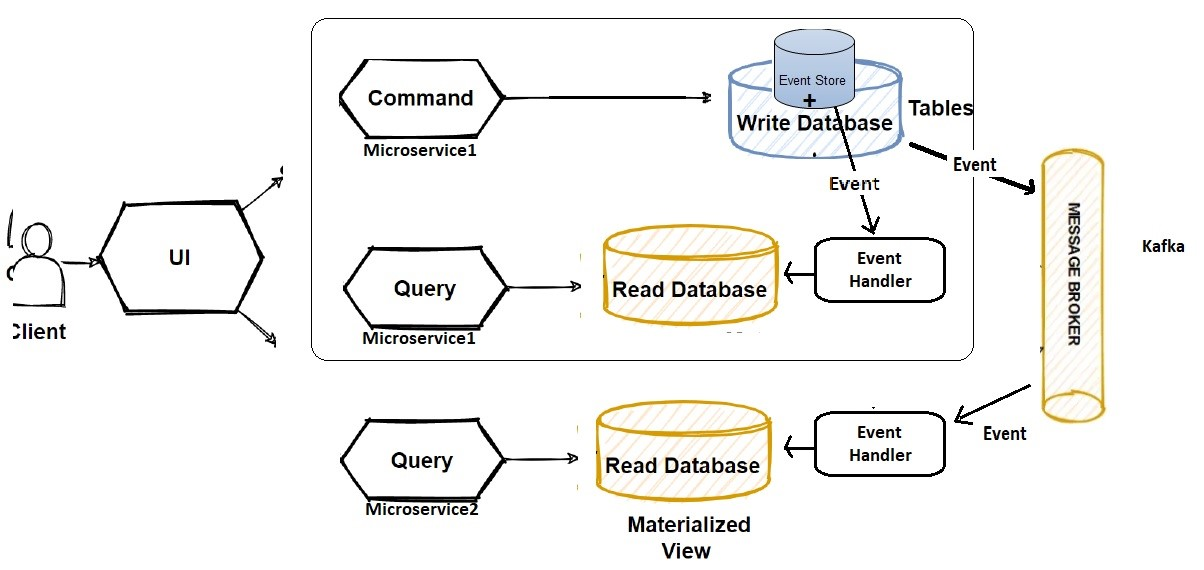
* CQRS offers to use “**separation of concerns”** principles and separate reading database and the writing database with 2 databases.
* By this way we can even use different database for reading and writing database types like using **no-sql for reading** and using relational database **for crud operations for writting.**
* Another consideration is we should understand our application use case behaviors, if our application is **mostly reading** use cases and **not writing so much**, we can say our application is **read-incentive application**. So we should design our architecture as per our reading requirements with focusing reading databases.
* So we can say that CQRS separates reads and writes into different databases, **Commands performs update data**, **Queries performs read data**.
* **Commands** should be actions with task-based operations like “**add item into shopping** cart” or “**checkout order**”. So commands can be handle with message broker systems that provide to process commands in async way.
* **Queries** is never modify the database. Queries always return the **JSON data with DTO** objects. By this way, we can **isolate the Commands and Queries**.

**Article Reference Link**:

* <https://medium.com/design-microservices-architecture-with-patterns/cqrs-design-pattern-in-microservices-architectures-5d41e359768c>
* <https://www.ibm.com/cloud/architecture/architectures/event-driven-cqrs-pattern/>

1. **Event Sourcing Microservice Design Pattern:**

* **Event sourcing** is a microservice design pattern that involves capturing all changes to an application’s state as a **sequence of events**, rather than simply updating the state itself.
* **Each event** represents a discrete/detailed change to the system and is stored in an **event log**, which can be used to **reconstruct the system’s state** at any point in time.



* **Event Sourcing talk about its own System state (all their own micro services are upto date with latest updates)**

**Event sourcing has several benefits:**

* It solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes. Because it persists events rather than domain objects, it mostly avoids the object‑relational **impedance** mismatch problem.

**What is the difference between event driven pattern and event sourcing patter in microservice?**

|  |  |
| --- | --- |
| **Event Sourcing** | **Event-Driven Architecture** |
| Event sourcing is a **persistence pattern**, ensuring that every data change is recorded as an **immutable event**. These events serve as the **single source** **of** **truth** for the **system's state**. | With event-driven architecture, these **events** are passed on to various parts of the system in a **loosely coupled** manner. |
| One captures every data change in an event | Uses events to communicate changes between different parts of a system |
| This keeps all components up to date with the latest state changes. | Event-driven architecture is ideal for microservices-based systems. Each microservice can publish events to a message broker and interested microservices can subscribe to those events. |
| Event sourcing adds to this resilience as it can replay the events to recreate the system's state in case of a failure or to support temporal queries. | Event-driven architectures are inherently more resilient and fault-tolerant. If a component fails to process an event, it can be retried or processed by another component without affecting the overall system. |
| Event sourcing is a unique system design pattern where all modifications to the application state are stored in sequence as events. However, in the context of event sourcing, an "event" signifies a "state change" more than a "notification". |  |

**Note:**

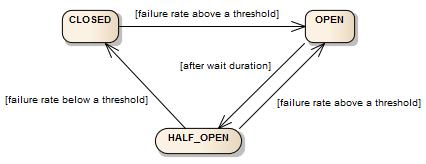
* They're different tools, but when combined, they can be extremely powerful in addressing complex problems. Let's see what can be achieved through this combination:
* When combined with event sourcing, microservices can receive events that represent changes to data and update their state accordingly.
* The core components of event sourcing are the event stores – specialized storage spaces where the sequence of events is preserved. This system doesn't overwrite **old events with new one**s. Instead, it adds new events to the sequence to maintain a complete history.
* Ref link: <https://estuary.dev/event-driven-vs-event-sourcing/#what-is-event-sourcing>

**Important notes about Event Sourcing Pattern:**

* Event store is related with database or we can have separate event sourcing system.
* An event store is a type of database optimized for storage of events.
* Event sourcing will track the status of events
* Event Sourcing can trigger the events either internal system or external end points ( Service A in network 1 to Service B in network2)
* Through Event Sourcing we can maintain data consistency and we can make use the versioning to make data more consistence.

1. **Circuit Breaker Micro-service Design Pattern and Retry Mechanisam(Spring)**

The CircuitBreaker is implemented via a finite state machine with three normal states: **CLOSED, OPEN and HALF\_OPEN** and two special states **DISABLED and FORCED\_OPEN**.



Reference Link:

* <https://resilience4j.readme.io/docs/circuitbreaker>
* <https://jstobigdata.com/architecture/circuit-breaker-pattern-in-microservices/>
* <https://www.java67.com/2023/04/what-is-circuit-breaker-design-pattern.html>

The CircuitBreaker uses a sliding window to store and aggregate the outcome of calls. You can choose between a count-based sliding window and a time-based sliding window.

1. The **count-based sliding window** aggregrates the outcome of the last N calls.
2. The **time-based sliding window** aggregrates the outcome of the calls of the last N seconds.

**Count-based sliding window :** Aggregates the outcome of the last N calls. For instance, if the count window(total number of requests) size is 10 and failure threshold **is 50%,** when circuit breaker detect **5 failure out of last 10** calls, it changes from **CLOSED to OPEN**.

**Time-based sliding window :** Aggregates the outcome of the calls of the last **N** seconds. For instance, if the time window size is 10 seconds**(given requests flow time)** and failure **threshold is 50%,** when circuit breaker detect **5 failure out of last 10** seconds calls, it changes from **CLOSED to OPEN.**

**Failure rate threshold :** The state of the Circuit Breaker changes from CLOSED to OPEN when the failure rate is equal or greater than a configurable threshold. For instance, when more than **50%** of the recorded calls have failed.

**Slow call rate threshold :** The circuit breaker changes from CLOSED to OPEN when the percentage of slow calls is equal or greater than a configurable threshold. For instance, when more than 50% of the recorded calls took longer than 5 seconds. This helps to reduce the load on an external system before it is actually unresponsive.

**Minimum number of calls :** The failure rate and slow call rate can only be calculated if a minimum number of calls are recorded. For instance, if the minimum number of required calls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated.

If only 9 calls have been evaluated the circuit breaker will not trip open even if all 9 calls have failed.

**Circuit Breaker Explanation:**

* Circuit breakers are a design pattern to create resilient microservices by limiting the impact of service failures and latencies. The major aim of the Circuit Breaker pattern is to prevent any cascading failure in the system. In a microservice system, failing fast is critical.
* A cascading failure is a failure in a system of **interconnected parts in which the failure of one or few parts leads to the failure of other parts**, growing progressively as a result of positive feedback.
* The Circuit Breaker pattern is a design pattern used in software engineering to handle failures in distributed systems.
* It is used to detect and handle faults in communication between services, preventing them from **cascading and causing further damage**.
* The Circuit Breaker pattern works by wrapping a potentially dangerous or faulty operation in a circuit breaker object.
* The circuit breaker is designed to detect when the **operation is failing** or **taking too long to complete**.
* Once a threshold is reached, the circuit breaker will “trip” and stop the operation from executing, returning a pre-configured fallback value instead.
* This helps prevent further damage by stopping the faulty operation from cascading through the system.

**The Circuit Breaker pattern has three states: Closed, Open, and Half-Open.**

1. **Closed** : In the closed state, the circuit breaker allows requests to flow through and execute the operation as normal.

2. **Open**: In the open state, the circuit breaker returns a pre-configured fallback value instead of executing the operation.

3. **Half-Open** : In the half-open state, the circuit breaker allows a limited number of requests to pass through to test if the operation is functioning correctly.

If these requests succeed, the circuit breaker returns to the closed state. If they fail, the circuit breaker returns to the open state.

* The CircuitBreaker rejects calls with a **CallNotPermittedException** when it is OPEN.
* After a wait time duration has elapsed, the CircuitBreaker state changes from OPEN to HALF\_OPEN and permits a configurable number of calls to see if the backend is still unavailable or has become available again.
* Further calls are rejected with a CallNotPermittedException, until all permitted calls have completed. If the failure rate or slow call rate is then equal or greater than the configured threshold, the state changes back to OPEN. If the failure rate and slow call rate is below the threshold, the state changes back to CLOSED.

**Real-time Use case Scenario**: (Assume A, B, C are 3 microservices)

**Usecase 1:** C is available

request --------> A -> B -> C (Fully Closed State)

**Usecase2:** C is not available

Request A to B, B to C failed (note: tried 5 times C is not responsded)

Then,

Assume that B configured the Circuit Breaker Pattern, then B service will call the fallback method after it reached configured threshold limit value

This state called as "Opened State"

**Usecase3**: C is available

Now one more request came from A to B, and circuit is fully opened state now, then it will prefer the fallback method to execute and return to service A.

**Note:**

1. we have configured wait period/threshold in service B after that period tried to call service C and if failure rate above the configured threshold value then it goes back to “**OPEN**” state.

This state is called as - **Half Open** state

2. we can configure the how many requests can try to goes to “HALF OPEN” state to recheck the health check of service C by using property called “ **permittedNumberOfCalls**” -

3. If failure rate is above the threshold value then it will go to **OPEN** state again.

4. Let's assume you got positive/failure rate below the threshold value (http response code is 200 series) response from service **C,** then service B will close the circuit again.

**Usecase4:** C is available

request -------->A -> B -> C (Fully Closed State)

**Usecase5**: C is not available

Usecase2 scenario will repeated again -- Circuit will open again after reached threshold limit.

**Circuit Breaker Implementation in Java:**

**Ref link:**

* [**https://stackoverflow.com/questions/65690371/nosuchmethodexception-in-resilience4j-fallback-with-spring-boot**](https://stackoverflow.com/questions/65690371/nosuchmethodexception-in-resilience4j-fallback-with-spring-boot)
* [**https://medium.com/bliblidotcom-techblog/resilience4j-circuit-breaker-implementation-on-spring-boot-9f8d195a49e0**](https://medium.com/bliblidotcom-techblog/resilience4j-circuit-breaker-implementation-on-spring-boot-9f8d195a49e0)
* **Resilience4j**: It is a lightweight, easy-to-use fault tolerance library inspired by Netflix Hystrix, but designed for Java 8 and functional programming.
* **Resilience4j** comes with an in-memory CircuitBreakerRegistry based on a ConcurrentHashMap which provides thread safety and atomicity guarantees.
* Maven repo : <https://mvnrepository.com/artifact/io.github.resilience4j/resilience4j-retry>

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/io.github.resilience4j/resilience4j-retry -->  <dependency>  <groupId>io.github.resilience4j</groupId>  <artifactId>resilience4j-retry</artifactId>  <version>2.1.0</version>  </dependency> |

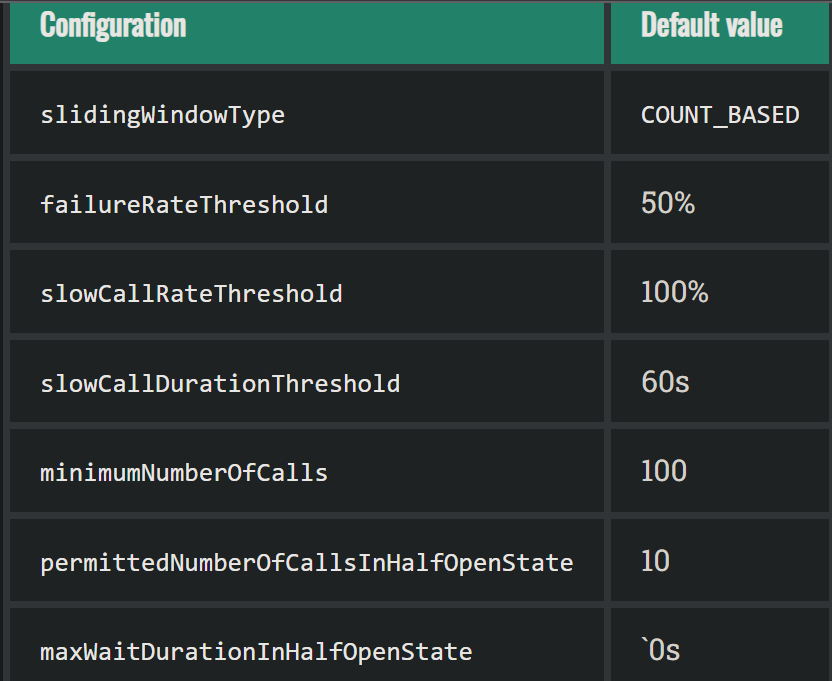
you can use the **CircuitBreakerRegistry** to manage (create and retrieve) CircuitBreaker instances. You can create a **CircuitBreakerRegistry** with a global default CircuitBreakerConfig for all of your CircuitBreaker instances as follows.

* **Create a CircuitBreakerRegistry**

|  |
| --- |
| **CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry.ofDefaults();** |

You can provide your own custom global **CircuitBreakerConfig**. In order to create a custom global **CircuitBreakerConfig**, you can use the CircuitBreakerConfig builder. You can use the builder to configure the following properties.

* failureRateThreshold
* slowCallRateThreshold
* slowCallDurationThreshold
* permittedNumberOfCallsInHalfOpenState
* maxWaitDurationInHalfOpenState
* slidingWindowType
* slidingWindowSize
* minimumNumberOfCalls
* waitDurationInOpenState
* automaticTransitionFromOpenToHalfOpenEnabled
* recordExceptions
* ignoreExceptions
* recordFailurePredicate
* ignoreExceptionPredicate

****

**Step2:**

|  |
| --- |
| // Create a custom configuration for a CircuitBreaker  CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()  .failureRateThreshold(50)  .slowCallRateThreshold(50)  .waitDurationInOpenState(Duration.ofMillis(1000))  .slowCallDurationThreshold(Duration.ofSeconds(2))  .permittedNumberOfCallsInHalfOpenState(3)  .minimumNumberOfCalls(10)  .slidingWindowType(SlidingWindowType.TIME\_BASED)  .slidingWindowSize(5)  .recordException(e -> INTERNAL\_SERVER\_ERROR  .equals(getResponse().getStatus()))  .recordExceptions(IOException.class, TimeoutException.class)  .ignoreExceptions(BusinessException.class, OtherBusinessException.class)  .build();  // Create a CircuitBreakerRegistry with a custom global configuration  CircuitBreakerRegistry circuitBreakerRegistry =  CircuitBreakerRegistry.of(circuitBreakerConfig);  // Get or create a CircuitBreaker from the CircuitBreakerRegistry  // with the global default configuration  CircuitBreaker circuitBreakerWithDefaultConfig =  circuitBreakerRegistry.circuitBreaker("name1");  // Get or create a CircuitBreaker from the CircuitBreakerRegistry  // with a custom configuration  CircuitBreaker circuitBreakerWithCustomConfig = circuitBreakerRegistry  .circuitBreaker("name2", circuitBreakerConfig); |

**Circuit Breaker design pattern will use internally AOP (Aspect Oriented Programming) concept.**

**Why? -**

AOP will be having some principles called @BeforeAdvide and @AfterAdvice

* **@AfterThrowingAdvice** will be used in circuit breaker, so we need add AOP dependency.
* Circuit Breaker uses after throwing advice from AOP, on top of it it implements its own functionality.
* In Circuit Breaker fall back method signature and end point method signature must be same expect one property i.e **Throwable** t
* We make make use actuator starter dependency - health end point to check status / information about circuit breaker.

**Status**: CLOSE/OPEN/HALD\_OPEN

|  |
| --- |
| management:    health:      circuitbreakers:        enabled: true    endpoints:      web:        exposure:          include: health    endpoint:      health:        show-details: always |

**Implementation using @CircuitBreaker:**

|  |
| --- |
| package com.example.demo.controller;  import java.util.Arrays;  import java.util.Objects;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.http.HttpStatus;  import org.springframework.http.ResponseEntity;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.client.RestTemplate;  import com.example.demo.resilience.PaytmCircuitBreakerRegistry;  import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;  @RestController()  public class PaytmController {    @Autowired  private PaytmCircuitBreakerRegistry paytmCircuitBreakerRegistry;    @GetMapping("/paytm-microservice")  public String healthCheck() {  return "Paytm Service {healthy:true}";  }    @GetMapping("/paytm-microservice/paytm-train-list")  @CircuitBreaker(name = "paytm-circuit-breaker", fallbackMethod = "irctcUnavailable")  public ResponseEntity<Object[]> getTrainList(Integer a){  System.out.println("Request Received to Paytm App");  RestTemplate restTemplate = new RestTemplate();  String env = System.getenv("IRCTC\_MICROSERVICE\_SERVICE\_HOST");  // env = Objects.nonNull(env)? env+"/irctc-microservice" :"http://localhost";  env = "http://localhost";  System.out.println("env==============================>" + env);  ResponseEntity<Object[]> responseEntity = restTemplate.getForEntity("http://localhost:9090/irctc-microservice/get-train-list", Object[].class);  Object[] objects = responseEntity.getBody();  Arrays.asList(objects).forEach(System.out::println);  return responseEntity;  }    @GetMapping("/paytm-microservice/paytm-train-list-with-resilience")  public ResponseEntity<Object[]> getTrainListWithResilienec(){  System.out.println("Request Received to Paytm App");  return paytmCircuitBreakerRegistry.getPaytmTrainList();  }      public ResponseEntity<Object[]> irctcUnavailable(Throwable t) {  String[] str = {"ContactsServersDown"};  return new ResponseEntity<Object[]>(str, HttpStatus.OK);  }    } |

Pom.xml

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <!--  https://mvnrepository.com/artifact/io.github.resilience4j/resilience4j-spring-boot2 -->  <dependency>  <groupId>io.github.resilience4j</groupId>  <artifactId>resilience4j-spring-boot2</artifactId>  <version>2.1.0</version>  </dependency>  <!--  https://mvnrepository.com/artifact/io.github.resilience4j/resilience4j-reactor -->  <dependency>  <groupId>io.github.resilience4j</groupId>  <artifactId>resilience4j-reactor</artifactId>  <version>2.1.0</version>  </dependency> |

**application.properties**

|  |
| --- |
| server.port=9092  management:  health:  circuitbreakers:  enabled: true  endpoints:  web:  exposure:  include: health  endpoint:  health:  show-details: always  resilience4j:  circuitbreaker :  configs:  shared :  register-health-indicator : true  sliding-window-size : 5  # failure-rate-threshold=40  # slow-call-rate-threshold=40  # permitted-number-of-calls-in-half-open-state=1  # max-wait-duration-in-half-open-state=10s  # wait-duration-in-open-state=10s  # slow-call-duration-threshold=2s  # writable-stack-trace-enabled=true  # automatic-transition-from-open-to-half-open-enabled=true    #resilience4j.circuitbreaker.configs.shared.sliding-window-type=count-based  #resilience4j.circuitbreaker.configs.shared.sliding-window-size=5  #resilience4j.circuitbreaker.configs.shared.failure-rate-threshold=40  #resilience4j.circuitbreaker.configs.shared.slow-call-rate-threshold=40  #resilience4j.circuitbreaker.configs.shared.permitted-number-of-calls-in-half-open-state=1  #resilience4j.circuitbreaker.configs.shared.max-wait-duration-in-half-open-state=10s  #resilience4j.circuitbreaker.configs.shared.wait-duration-in-open-state=10s  #resilience4j.circuitbreaker.configs.shared.slow-call-duration-threshold=2s  #resilience4j.circuitbreaker.configs.shared.writable-stack-trace-enabled=true  #resilience4j.circuitbreaker.configs.shared.automatic-transition-from-open-to-half-open-enabled=true  #  #resilience4j.circuitbreaker.instances.example.base-config=shared |

**BRANCH Design Pattern in Microservices**

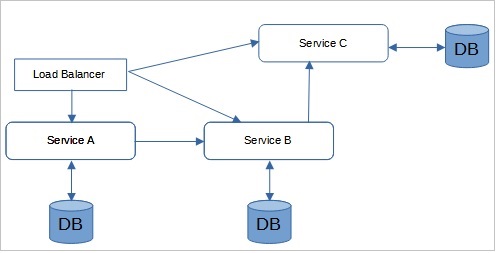
Branch microservice pattern is the extended version of aggregator pattern and chain pattern. In this design pattern, the client can directly communicate with the service. Also, one service can communicate with more than one services at a time. Branch microservice design pattern in which you can simultaneously process the requests and responses from two or more independent microservices.

**Problem Statement**

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. Now consider a case where one service needs output of another service as dependency and client can call any service.

**Solution**

We can use Branch Microservices Design Pattern here. Branch microservice pattern is the extended version of aggregator pattern and chain pattern. In this design pattern, the client can directly communicate with the service. Also, one service can communicate with more than one services at a time. Following is the diagrammatic representation of Branch Microservices.



**Advantages**

Branch microservice pattern allows the developer to configure service calls dynamically. All service calls will happen in a concurrent manner, which means service A can call Service B and C simultaneously.

**Usage of this Design Pattern:**

1. Use this pattern to achieve loose coupling where a request from the client is passed to a Branch.

2. Use this pattern when Multiple services have to handle a request.

3. Use this pattern whenever one service needs output of another service as dependency and client can call any service.

4. Use this pattern whenever to make parallel calls to multiple services or chained services.

**Advantages of this Design Pattern:**

1. Branch pattern avoid coupling between sender of a request to multiple services.

2. Branch pattern simplifies the request and response for client to make call to either one service or multiple services simultaneously.

3. Branch pattern support single responsibility principle to make sure each service is doing their job.

4. It is easy to add new microservice in chain if requires in future.

**Decomposition Design Pattern in Microservice**

Microservice decomposition by **subdomain** is a process of breaking down a monolithic system into smaller**, independent microservices** based on the corresponding **subdomains** defined by Domain-Driven Design (DDD).

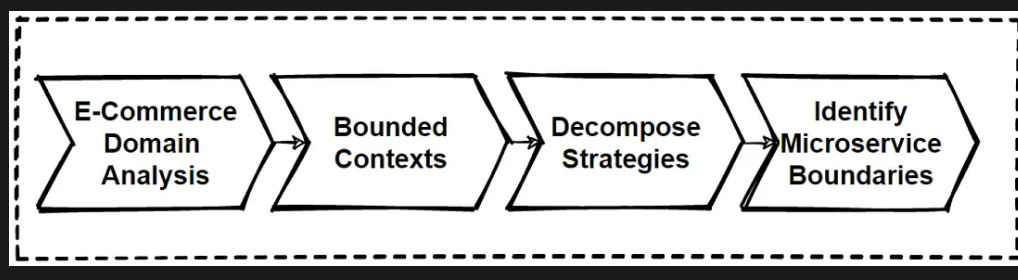
decomposition in java?

The process of breaking **down large problems into smaller problems**, each with a method that defines a **subproblem** in the larger problem.

**Green Projects:** Developing from scratch all the microservices by following DDD principles (Domain Driven Design)

**Brown Projects :** Existing legacy/monolithic applications converting into microservice applications

**Decomposition Microservices Architecture Path**



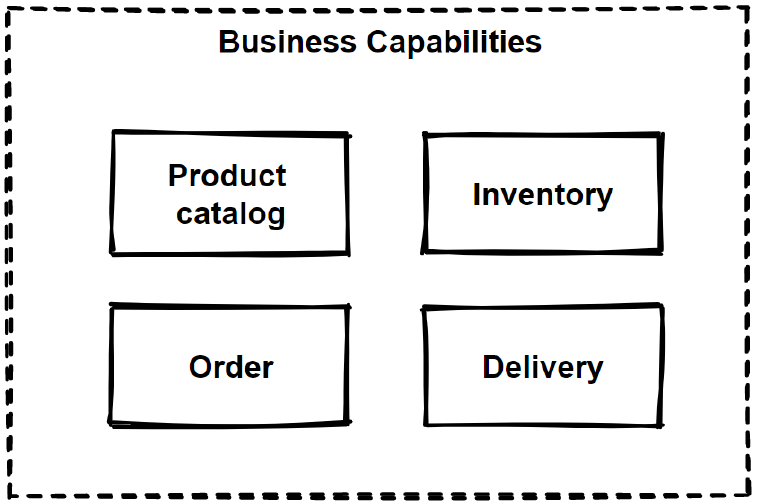
* We will make E-Commerce Domain analysis in order to define your microservice boundaries.
* After that, we will use DDD, Bounded Contexts, and other Decompose strategies like decompose by business capability, Decompose by subdomains microservices patterns
* Lastly, we will Identify microservice boundaries and refactor our current design.

**Microservices Decomposition Pattern — Decompose by Business Capability**

* Think about that our monolithic application is large and complex application and want to use the microservice architecture.
* With the microservice architecture, we are going to split the application as a set of loosely coupled services in order to accelerate software development processes.

**How to decompose an application into services?** There are some Prerequisite of decomposition of microservices.

* Services must be **cohesive**.
* A service should implement a small set of strongly related functions.
* Services must be **loosely coupled** — each service as an API that encapsulates its implementation.



If we look at our domain which is e-commerce application, The business capabilities can be like on the image included:

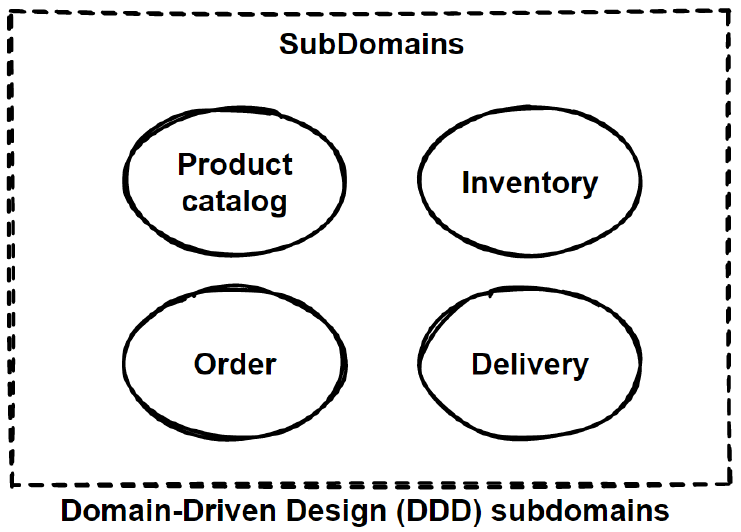
* Product catalog management
* Inventory management
* Order management
* Delivery management and so on.

So we can decompose our microservices as per their business capabilities

**Microservices Decomposition Pattern — Decompose by Subdomain**

In order to apply Decompose by Subdomain model, we should define services corresponding to Domain-**Driven Design (DDD) subdomains.**

* **DDD** refers to the application’s problem space the business as the domain.
* A domain is consists of multiple subdomains.
* Each subdomain corresponds to a different **part of the business**.



If we look at our domain which is e-commerce application, The subdomains of an online store include:

* Product catalog
* Inventory management
* Order management
* Delivery management

But to identify these domains and subdomains, we should understand the DDD and Bounded Context patterns properly.

**Bounded Context Pattern (Domain-Driven Design — DDD)**

We are going to see DDD — Bounded Context Pattern which is one of the main pattern that we mainly use when decomposing microservices. One of the best way of design microservices is using the DDD-Bounded Context pattern. So we should understand Bounded Context properly.

**Before we start lets clarify that What is DDD** ?

**What is DDD ?**

We can say that Domains are require high cooperation and have a certain complexity by nature are called collaborative domains. In general, there is a good option that DDD is a more suitable solution for domains with this characteristic.

**DDD has two phases, strategic and tactical DDD.**

**Strategic DDD**

In strategic DDD, we define the large-scale model of the system, defining to the business rules. Strategic DDD identifies disciplines that allow designing loosely coupling units and the context map between them.

**Tactical DDD**

Tactical DDD focuses on implementation and provides design patterns that we can use to build the software implementation. These design patterns include concepts such as entity, aggregate, value object, repository, and domain service. We are not going to detail in here. Because we are focus on the DDD and microservices.

DDD is increase collaboration between large technology teams by creating a common language on changing business rules. DDD domain defines an approach that has its own common language and divides boundaries into specific, independent components.

This common language is called ubiquitous language, and independent units are called Bounded Context.

**Reference links:**

* <https://medium.com/design-microservices-architecture-with-patterns/decomposition-of-microservices-architecture-c8e8cec453e>
* <https://www.linkedin.com/pulse/unveiling-power-decomposition-patterns-microservices-architecture-a6ikc>

**SAGA Design Pattern in Microservice**

* The **saga** pattern is a failure management pattern that helps establish consistency in distributed applications, and coordinates transactions between multiple microservices to maintain data consistency.
* The term “saga” refers **to Long Lived Transactions (LLT**). The name “SAGA” comes from the concept of a long story with many parts, just like a distributed transaction. In a SAGA, each part of the story is a local transaction, and together, they form the complete story.

**Problem:**

Microservices come with their own advantages and disadvantages. One such disadvantage is managing distributed transactions.

Let’s say your transaction spans 4 different microservices.

How do you ensure that your transaction either commits successfully with all the 4 tasks succeeding or fails successfully if any of the tasks is not completed ( the completed ones are rolled back)?

Spring Boot provides the annotation @Transactional to manage transactions.But this works only within a single method and within a single project.

**Solution:**

There is a design pattern which solves the issue with distributed transactions in microservices.

It was originally proposed by the computer **scientist Hector Gracia Molina** and **Kenneth Salem** as mentioned in their paper here.

As suggested in the paper , they created this for **Long Lived Transactions (LLT**) and not for microservices but it serves **well for microservices**.

A long lived transaction is a transaction which takes a longer time , may be minutes , hours or even days. You can’t lock a database until all the tasks in such transaction completes for it will severely affect the performance of the application. Hence they came up with the design pattern SAGA (probably named SAGA because they created it for dealing with long transactions – SAGA means a very long story).

It goes like this :

* If your transaction contains , lets say 4 tasks,
* You create a compensating task for each task except the last.
* So that if any task fails then you run the compensating tasks of the previous tasks to rollback the original effect.
* So if there are four tasks T1, T2 , T3 and T4,
* Then you have three corresponding compensating tasks C1,C2,C3.
* If , for example , T1 and T2 succeeds and T3 fails , then you run C2 and C1 to rollback the effects of T1 and T2 in Last In First Out order.
* So the sequence of the transaction goes like this : T1 -> T2 -> T3 (failed) -> C2 -> C1.
* You don’t need a compensating task for the last task because if the last task fails you just need to roll back the previous tasks.
* This is called Backward Recovery since you go back and execute the compensating tasks of already completed successful tasks.
* You can also try Forward Recovery by retrying T3 and then T4 if your business use case requires it.
* Backward Recovery is more common though.
* On a high level this is what SAGA design pattern is.

**It can be implemented in two ways:**

* Choreography
* Orchestration

**1. Choreography:**

* Choreography means the tasks **execute independently**. Once one task is completed , it invokes the next tasks in the sequence. In case if the next task fails then it invokes the compensating tasks for the previous tasks.
* Orchestration means the tasks are invoked by another **parent task**.
* It plays the role of an orchestrator.
* It calls each tasks in sequence and based on their response **decides whether to call the next task** or the compensating tasks.
* Let us implement SAGA using Choreography in this example.
* It is simpler and **neat compared to Orchestration**.

Reference link: <https://fullstackdeveloper.guru/2023/05/11/how-to-implement-saga-design-pattern-in-spring-boot/?utm_content=cmp-true>

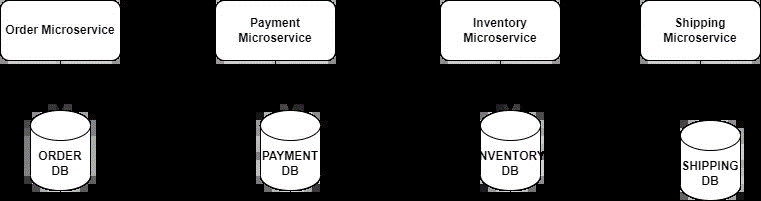
### Implementation:

Let’s consider the example of a ecommerce application.A customer places an order and the order gets shipped. This is the business use case. Let’s say there are four different microservices to take care of this flow.

* An order microservice which handles the customer orders.
* A payment microservice which handles payments for the orders.
* An inventory microservice which updates the inventory once orders are placed.
* A shipping microservice which deals with delivering the orders.

Note that in real case , separating these functionalities into four different apps may not be a good design . If your user base is small you can do all the above in a single monolith instead of four different microservices which is going to increase your network calls and infrastructure cost. Also handling transactions in a monolith is way more easier.

For this example though , we will go with this design to understand SAGA design pattern.



Now let’s consider the below functions in each microservice when a customer places an order:

1. createOrder() – Oder microservice
2. processPayment() – Payment microservice
3. updateInventory() – Inventory microservice
4. shipOrder() – Shipping Microservice

When a customer places an order and createOrder() , processPayment() methods succeed and updateInventory() method fails then the system will have a wrong inventory information. And the customer won’t get her order shipped!

So all these tasks have to be part of a single transaction.

You can use SAGA design pattern to implement distributed transaction.

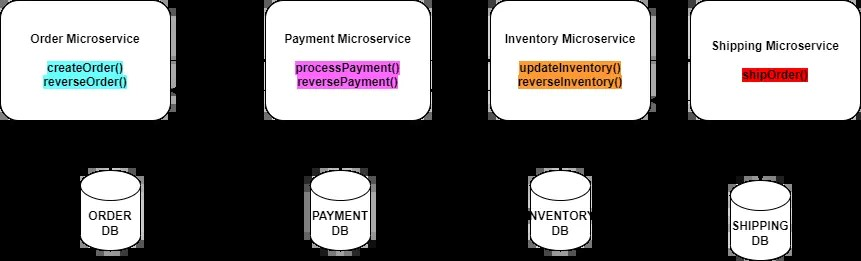
To resolve the above issue , you can rollback the entire transaction using backward recovery.

You can have a compensation task for each of the tasks above.

Here **are the compensating tasks**

1. reverseOrder() – Order microservice
2. reversePayment() – Payment microservice
3. reverseInventory() – Inventory microservice

Here is the updated flow:



Now if **updateInventory**() method fails, then you call **reversePayment**() and then **reverseOrder**() and the **order** will be **rolled** **back**!

**Code:**

**ORDER MICROSERVICE:**

|  |  |  |
| --- | --- | --- |
| Here is the controller class to add new orders:   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | package com.order.microservice;    import java.util.List;  import java.util.Optional;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.http.ResponseEntity;  import org.springframework.kafka.core.KafkaTemplate;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.PostMapping;  import org.springframework.web.bind.annotation.RequestBody;  import org.springframework.web.bind.annotation.RestController;    @RestController  public class OrderController {        @Autowired      private OrderRepository repository;        @Autowired      private KafkaTemplate<String, OrderEvent> kafkaTemplate;        @PostMapping("/orders")      public void createOrder(@RequestBody CustomerOrder customerOrder) {            OrderEntity order = new OrderEntity();          try {              // save order in database                order.setAmount(customerOrder.getAmount());              order.setItem(customerOrder.getItem());              order.setQuantity(customerOrder.getQuantity());              order.setStatus("CREATED");              order = this.repository.save(order);                customerOrder.setOrderId(order.getId());                // publish order created event for payment microservice to consume.                OrderEvent event = new OrderEvent();              event.setOrder(customerOrder);              event.setType("ORDER\_CREATED");              this.kafkaTemplate.send("new-orders", event);          } catch (Exception e) {                order.setStatus("FAILED");              this.repository.save(order);            }        }      } | |

**Explanation:**

* As you see when an order is placed , we create an entry in database and mark it as success.
* And then we publish an order created event using Apache Kafka so that payment microservice can receive it and process the payment.
* If the above logic fails we mark the order status as failed and save it to the database.

**THE COMPENSATING TASK:**

Now let’s see the compensating task for order microservice.

If payment microservice publishes an event to reverse the order then this task will be executed.

**Let’s create a class to impl**

|  |
| --- |
| package com.order.microservice;    import java.util.Optional;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.kafka.annotation.KafkaListener;  import org.springframework.stereotype.Component;  import org.springframework.web.bind.annotation.PathVariable;    import com.fasterxml.jackson.databind.ObjectMapper;    @Component  public class ReverseOrder {    @Autowired  private OrderRepository repository;    @KafkaListener(topics = "reversed-orders", groupId = "orders-group")  public void reverseOrder(String event) {    try {  OrderEvent orderEvent = new ObjectMapper().readValue(event, OrderEvent.class);    Optional<OrderEntity> order = this.repository.findById(orderEvent.getOrder().getOrderId());    order.ifPresent(o -> {  o.setStatus("FAILED");  this.repository.save(o);  });  } catch (Exception e) {    e.printStackTrace();  }    }  } |

**Explanation:**

* As you see , we listen for the “reversed-orders” topic in Apache Kafka and once we receive any event under that topic , we fetch it , extract the order details from it and mark the status in the database as failed (You could also remove the row entirely but marking it as failed gives us a chance to retry the order later and also do analytics).

For our example , we are using in memory database and Apache Kafka running in local. Here is the configuration for order microservice:

|  |
| --- |
| Application.properties file  spring.h2.console.enabled=true  spring.datasource.url=jdbc:h2:mem:ordersdb  spring.kafka.bootstrap-servers=localhost:9092  spring.kafka.consumer.group-id=orders-group  spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer  spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer |

* You need to add required dependencies for in memory database and apache kafka in pom.xml.
* Here is the link to the entire code: <https://github.com/vijaysrj/ordermicroservice>

**PAYMENT MICROSERVICE:**

* Once an order event is triggered , payment microservice picks it up , extracts the order details and then the payment information from it and then stores it in payment database.
* And then it triggers a payment event which will be picked up by inventory microservice.

**Here is the code for it:**

|  |
| --- |
| package com.payment.microservice;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.kafka.annotation.KafkaListener;  import org.springframework.kafka.core.KafkaTemplate;  import org.springframework.stereotype.Controller;    import com.fasterxml.jackson.core.JsonProcessingException;  import com.fasterxml.jackson.databind.JsonMappingException;  import com.fasterxml.jackson.databind.ObjectMapper;    @Controller  public class PaymentController {    @Autowired  private PaymentRepository repository;    @Autowired  private KafkaTemplate<String, PaymentEvent> kafkaTemplate;    @Autowired  private KafkaTemplate<String, OrderEvent> kafkaOrderTemplate;    @KafkaListener(topics = "new-orders", groupId = "orders-group")  public void processPayment(String event) throws JsonMappingException, JsonProcessingException {    System.out.println("Recieved event" + event);  OrderEvent orderEvent = new ObjectMapper().readValue(event, OrderEvent.class);    CustomerOrder order = orderEvent.getOrder();  Payment payment = new Payment();  try {    // save payment details in db  payment.setAmount(order.getAmount());  payment.setMode(order.getPaymentMode());  payment.setOrderId(order.getOrderId());  payment.setStatus("SUCCESS");  this.repository.save(payment);    // publish payment created event for inventory microservice to consume.    PaymentEvent paymentEvent = new PaymentEvent();  paymentEvent.setOrder(orderEvent.getOrder());  paymentEvent.setType("PAYMENT\_CREATED");  this.kafkaTemplate.send("new-payments", paymentEvent);  } catch (Exception e) {    payment.setOrderId(order.getOrderId());  payment.setStatus("FAILED");  repository.save(payment);    // reverse previous task  OrderEvent oe = new OrderEvent();  oe.setOrder(order);  oe.setType("ORDER\_REVERSED");  this.kafkaOrderTemplate.send("reversed-orders", orderEvent);    }    } } |

* Also , as you see if the payment fails for some reason , you mark the status as “failed” and update the database.
* You also trigger the compensating task for order service by publishing an event to Kafka.

**THE COMPENSATING TASK:**

To reverse a payment we create a compensating task.

Here is the code:

|  |
| --- |
| package com.payment.microservice;    import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.kafka.annotation.KafkaListener;  import org.springframework.kafka.core.KafkaTemplate;  import org.springframework.stereotype.Component;  import org.springframework.web.bind.annotation.PathVariable;    import org.springframework.web.bind.annotation.PutMapping;  import org.springframework.web.bind.annotation.RestController;    import com.fasterxml.jackson.databind.ObjectMapper;    @Component  public class ReversePayment {    @Autowired  private PaymentRepository repository;    @Autowired  private KafkaTemplate<String, OrderEvent> kafkaTemplate;    @KafkaListener(topics = "reversed-payments", groupId = "payments-group")  public void reversePayment(String event) {    try {    PaymentEvent paymentEvent = new ObjectMapper().readValue(event, PaymentEvent.class);    CustomerOrder order = paymentEvent.getOrder();    // do refund..    // update status as failed    Iterable<Payment> payments = this.repository.findByOrderId(order.getOrderId());    payments.forEach(p -> {    p.setStatus("FAILED");  this.repository.save(p);  });    // reverse previous task  OrderEvent orderEvent = new OrderEvent();  orderEvent.setOrder(paymentEvent.getOrder());  orderEvent.setType("ORDER\_REVERSED");  this.kafkaTemplate.send("reversed-orders", orderEvent);    } catch (Exception e) {    e.printStackTrace();    }    }    } |

**Explanation:**

* When the compensating task above is triggered by inventory microservice when it fails , we update the status as failed in the payment database for the particular order. We also need to reverse the order created for this payment , so we trigger the compensating task of order service as well.
* Here is the link to the code: <https://github.com/vijaysrj/paymentsmicroservice>

**Advantages:**

* If you want to execute a series of tasks spanning different microservices SAGA helps you achieve it.
* SAGA lets you do a transaction in an asynchronous way , this can improve application performance

**Disadvantages:**

* SAGA makes your code more complex to implement and understand
* SAGA only ensures eventual consistency of data so if your transaction has tightly coupled tasks then it is not an option

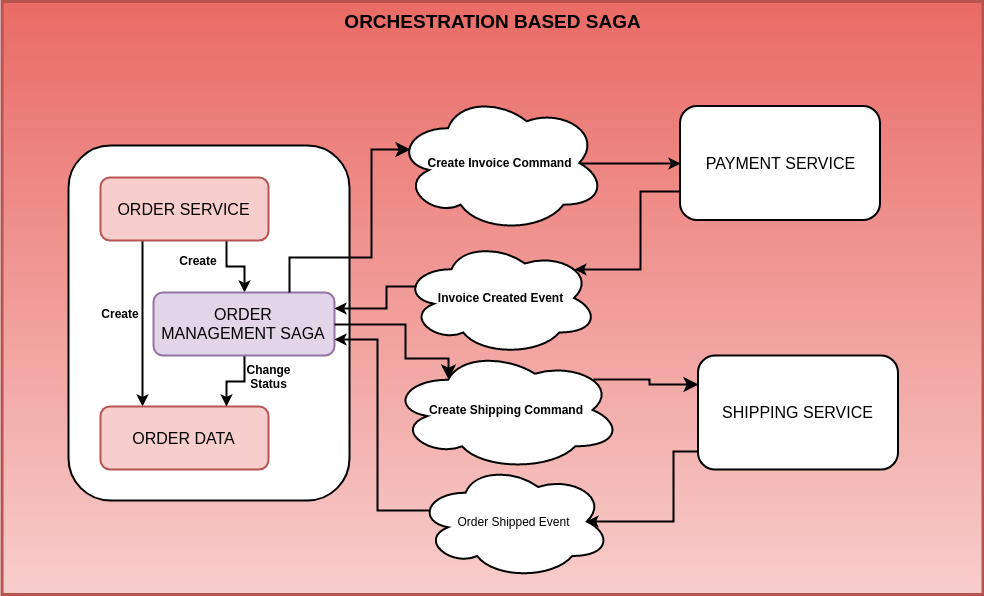
**Conclusion:**

We saw how SAGA can help in implementing distributed transactions with sample code.

**Reference Link:**

* [**https://fullstackdeveloper.guru/2023/05/11/how-to-implement-saga-design-pattern-in-spring-boot/?utm\_content=cmp-true**](https://fullstackdeveloper.guru/2023/05/11/how-to-implement-saga-design-pattern-in-spring-boot/?utm_content=cmp-true)

**Orchestration Saga Pattern**



Reference Link:

* With work flows - [**https://www.vinsguru.com/orchestration-saga-pattern-with-spring-boot/**](https://www.vinsguru.com/orchestration-saga-pattern-with-spring-boot/)
* <https://helpezee.wordpress.com/2019/08/04/orchestration-based-saga-pattern/>