**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:**

4.Loose coupling:

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 microservice 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

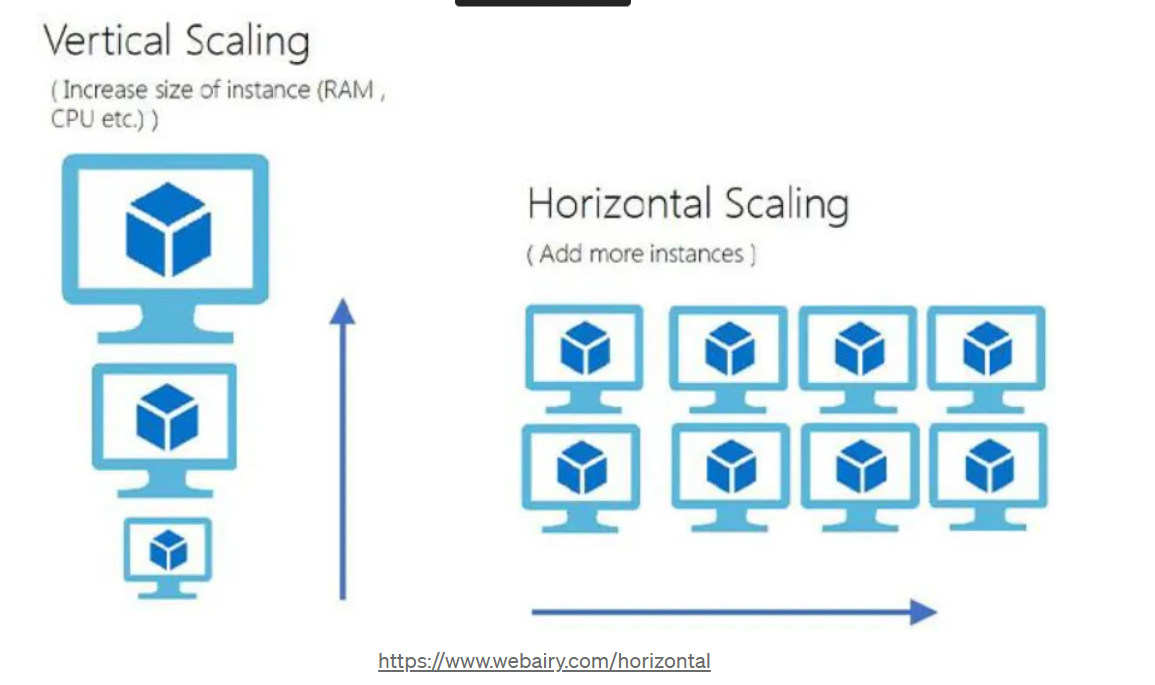
**Scalability**is simply measured by the number of requests an application can handle successfully. It can be measured by the **number of requests** and it can effectively support **simultaneously**. Once the application can no longer handle any more **simultaneous requests**, it has reached its scalability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

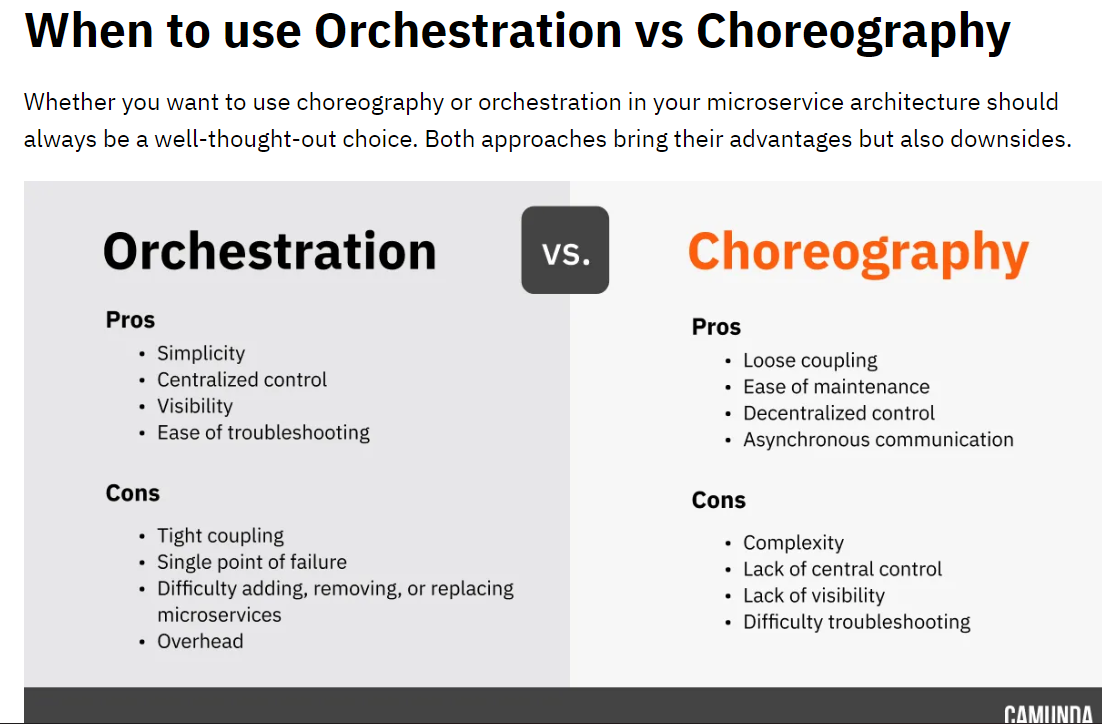
Orchestration in microservice architectures can lead to some nice benefits which compensate for the drawbacks of a choreographed system. A few of them are explained below:

* **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.



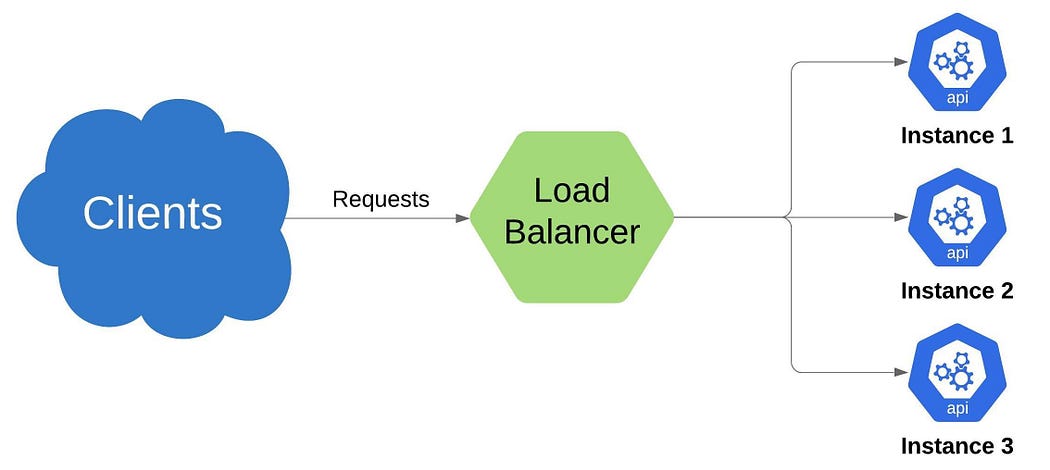
**Resilient Services:**

* 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.

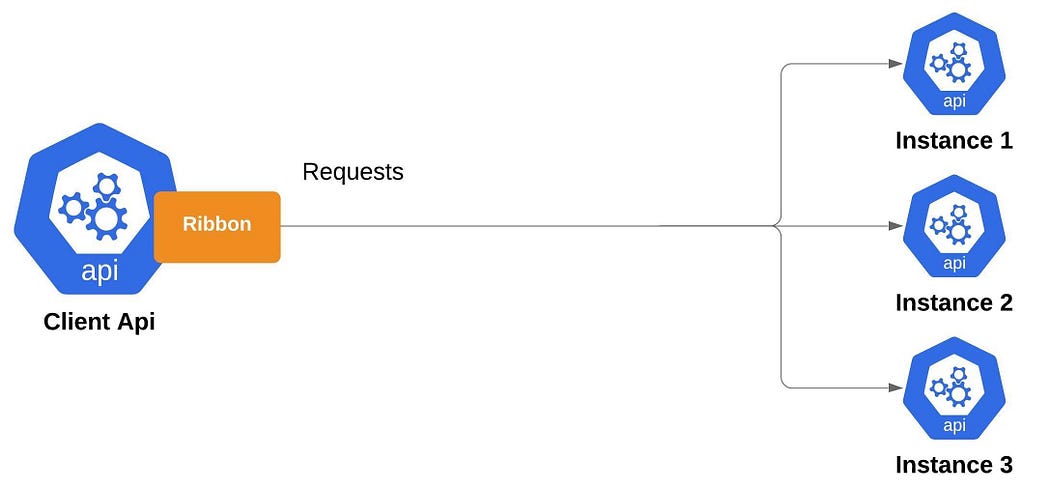
Server Side Load Balancing



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.



**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. This step is a gr

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, 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.

**Micro-Services Design Patterns:**

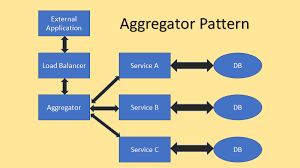
1. Aggregator
2. API Gateway
3. Chained or Chain of Responsibility
4. Asynchronous Messaging
5. Database or Shared Data
6. Event Sourcing
7. Branch
8. Command Query Responsibility Segregator(CQRS)
9. Circuit Breaker
10. Decomposition
11. SAGA

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 micr oservices 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.

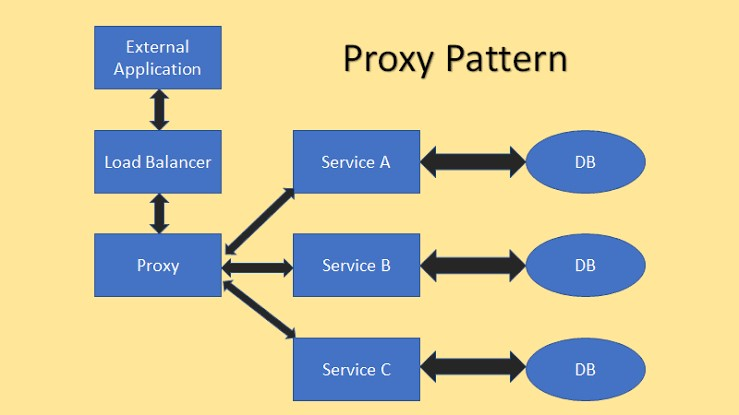


In Aggregator Pattern, there are 3 ways to implement it within Microservices application.

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

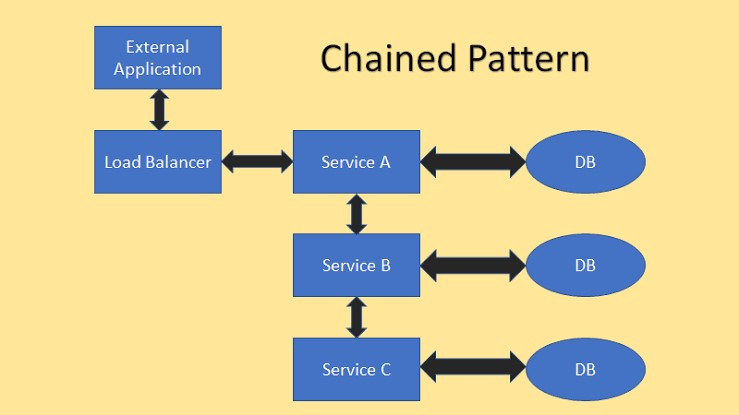
**Scatter Gather Pattern/Asynchronous Communication/Proxy Pattern:**

|  |
| --- |
| 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()));  }  } |



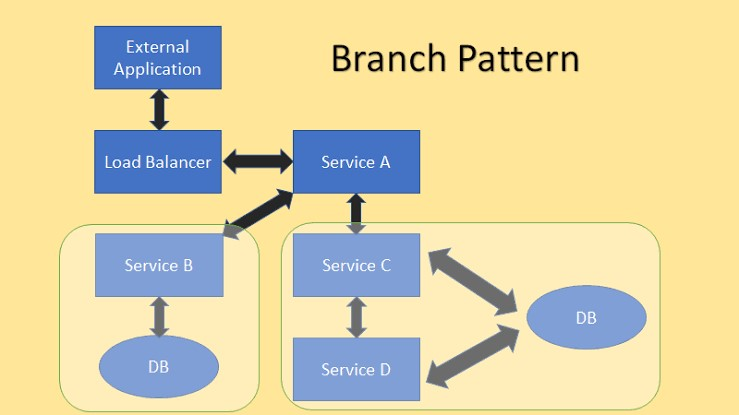
**Chained Pattern/Synchronous Communication:**

|  |
| --- |
| 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.

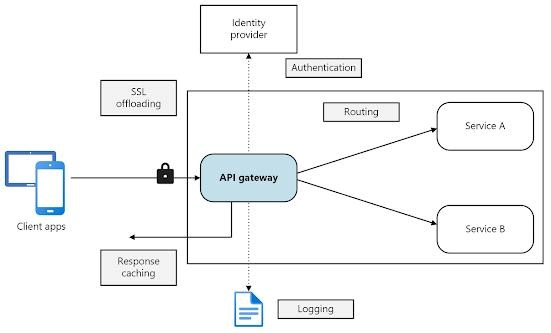
2. Use this pattern when project requires more than one team to implement it in parallel.

**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.

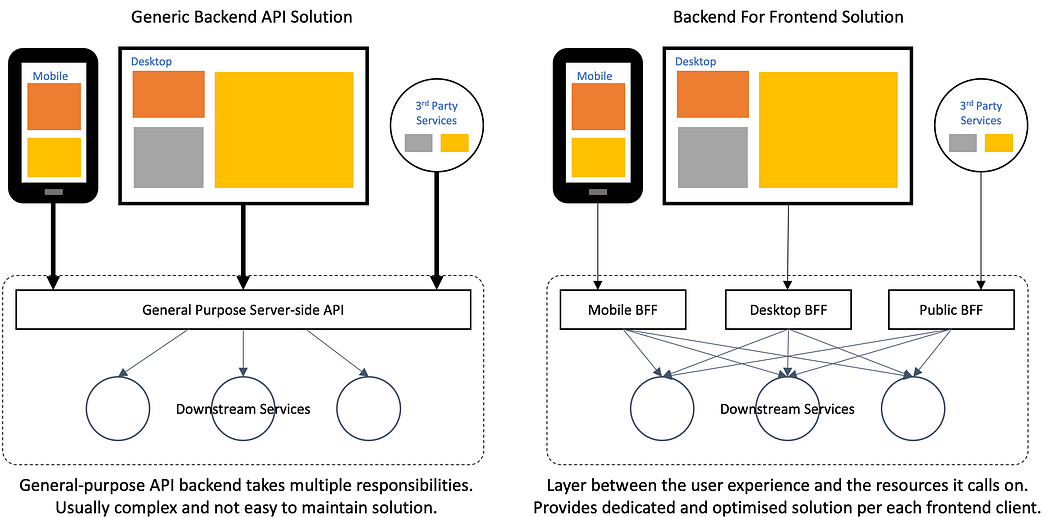
1. **API Gateway:**

* 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.

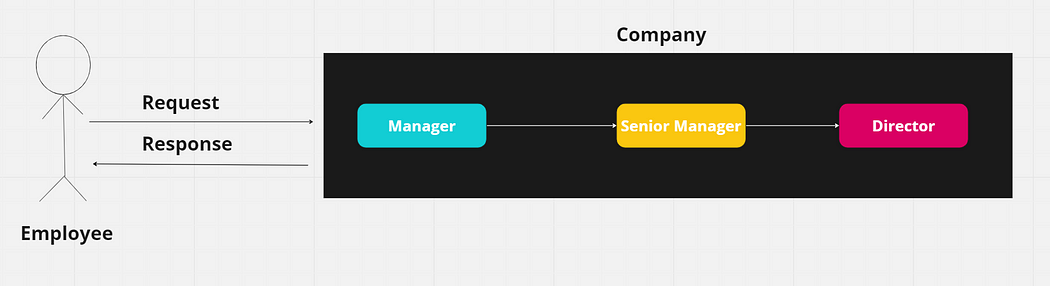
**How to select your API Gateway**

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**

* 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 .



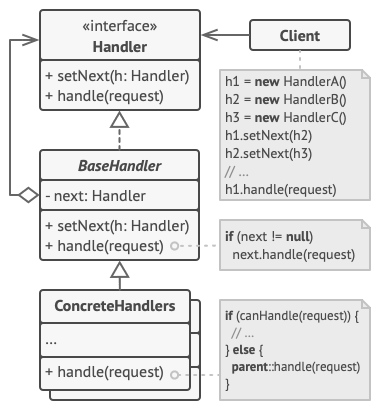
ref link: <https://medium.com/@ngneha090/beginners-guide-to-chain-responsibility-design-pattern-in-java-e1e0ddac2cb6>

**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");  }  } |

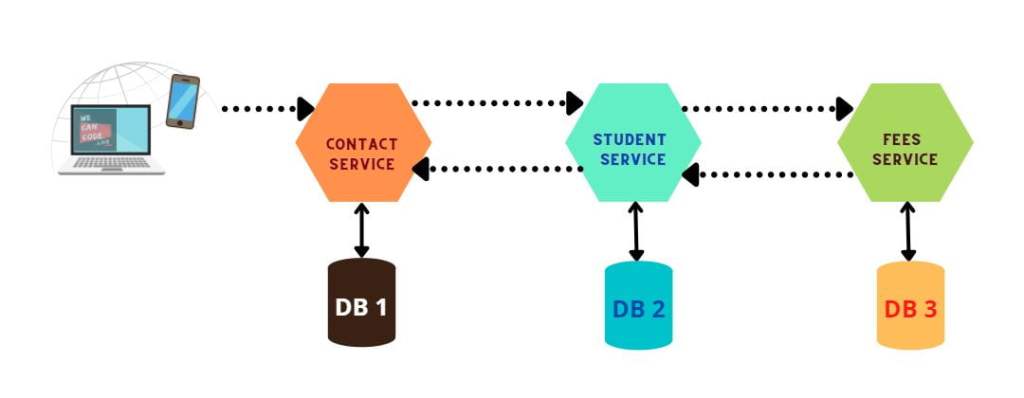
|  |
| --- |
| public class Manager extends RequestHandler{  public Manager(){  super("manager");  }  @Override  void setNext(RequestHandler nextHandler){  this.nextHandler=nextHandler;  }  @Override  void approve(int id){  if(id>=1 && id<=20)  {  System.out.println("Request Approved");  }  else{  super.approve(id);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

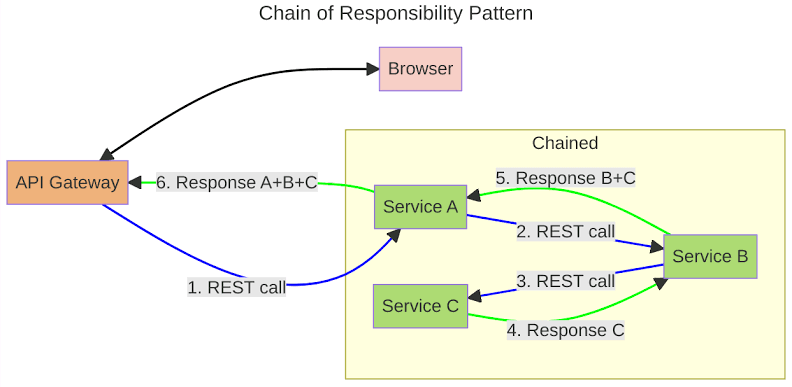
|  |
| --- |
| public class SeniorManager extends RequestHandler{  public SeniorManager(){  super("Senior manager");  }  @Override  void setNext(RequestHandler nextHandler){  this.nextHandler=nextHandler;  }  @Override  void approve(int id){  if(id>=21 && id<=40)  {  System.out.println("Request Approved");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

****

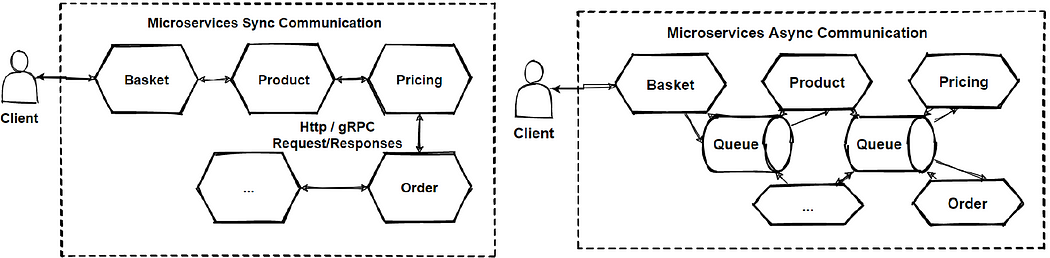
****

* 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.

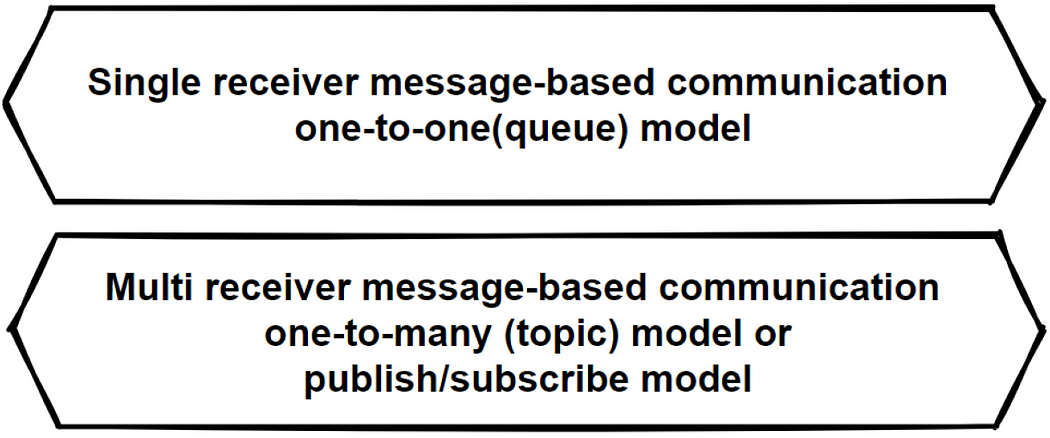
**asynchronous communication:**

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:

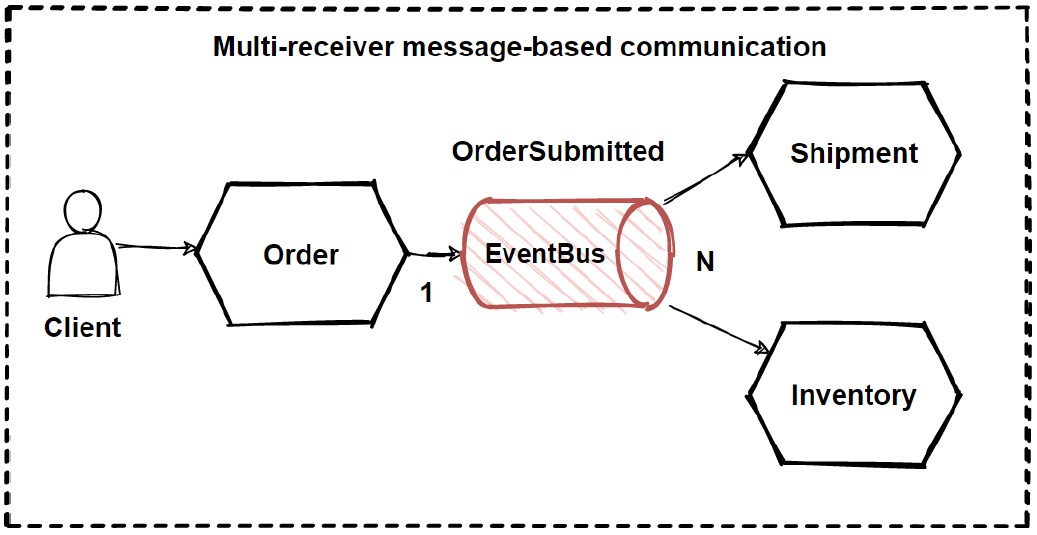


**Single-receiver message-based communication**

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 consumer 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**.

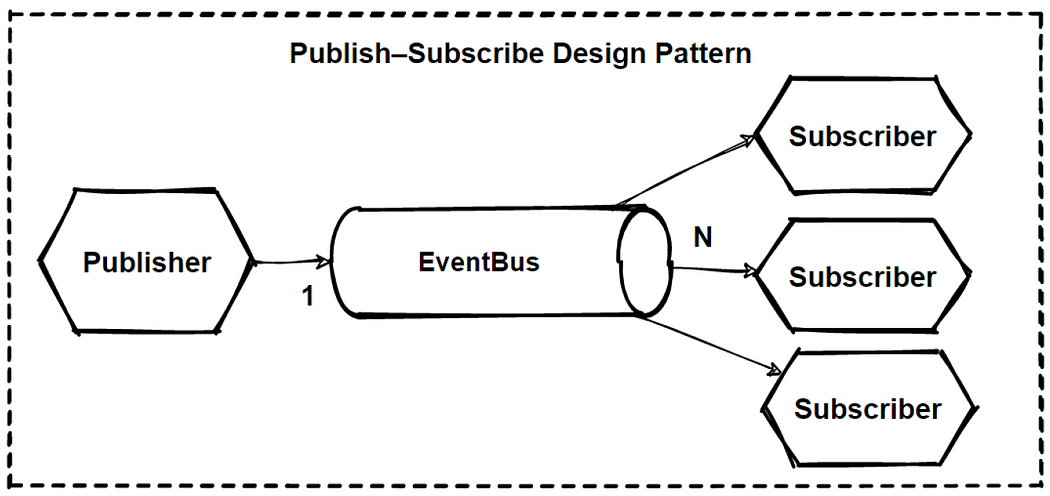


|  |
| --- |
| * Lets check the above image here, This is an example of publish/subscribe messaging based on event-driven communication. * The Order microservices publishes OrderSubmitted events to an event bus and Shipment and Inventory microservices can subscribe to this event, and take their internal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

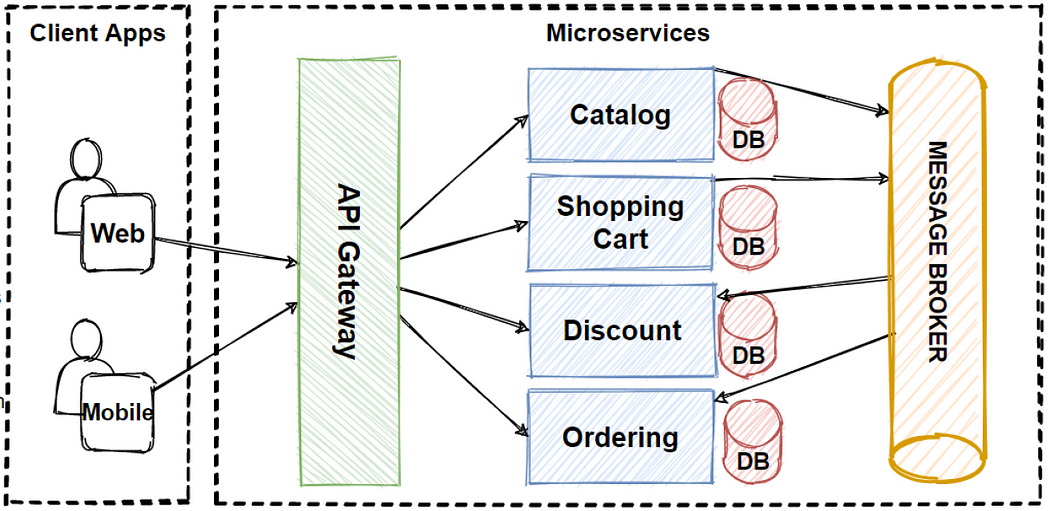


* So publishers don’t send the messages directly to the subscribers. Instead categorize published messages and send them into message broker systems without knowledge of which subscribers are there. Similarly, subscribers express interest and only receive messages that are of interest, without knowledge 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:**

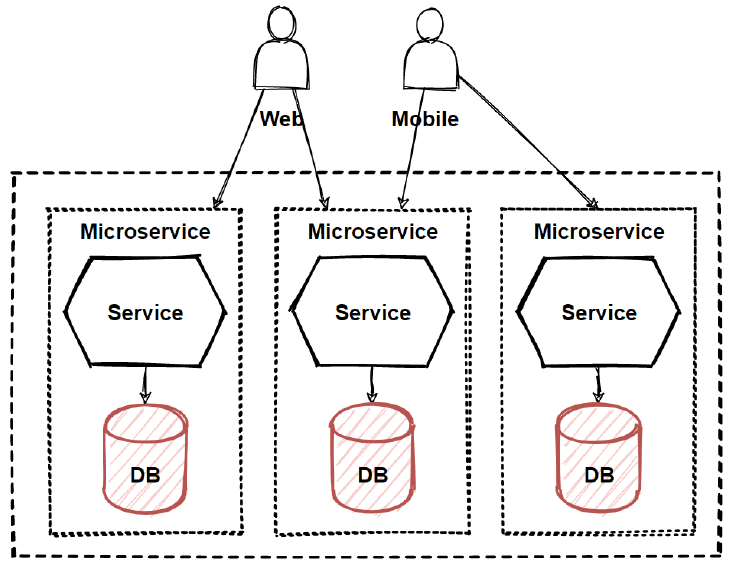
1. Kafka -
2. RabbitMQ
3. JMS(Java Messaging System)
4. Google Pub/Sub
5. Amazon Services
6. ActiveMQ
7. Azure Services
8. Azure Service Bus -
9. Amazon Kinesis for event streaming.
10. **Database or Shared Data Design Pattern/** **shared-database-per-service pattern:**

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**



Data schema changes made easy without impacting other microservices

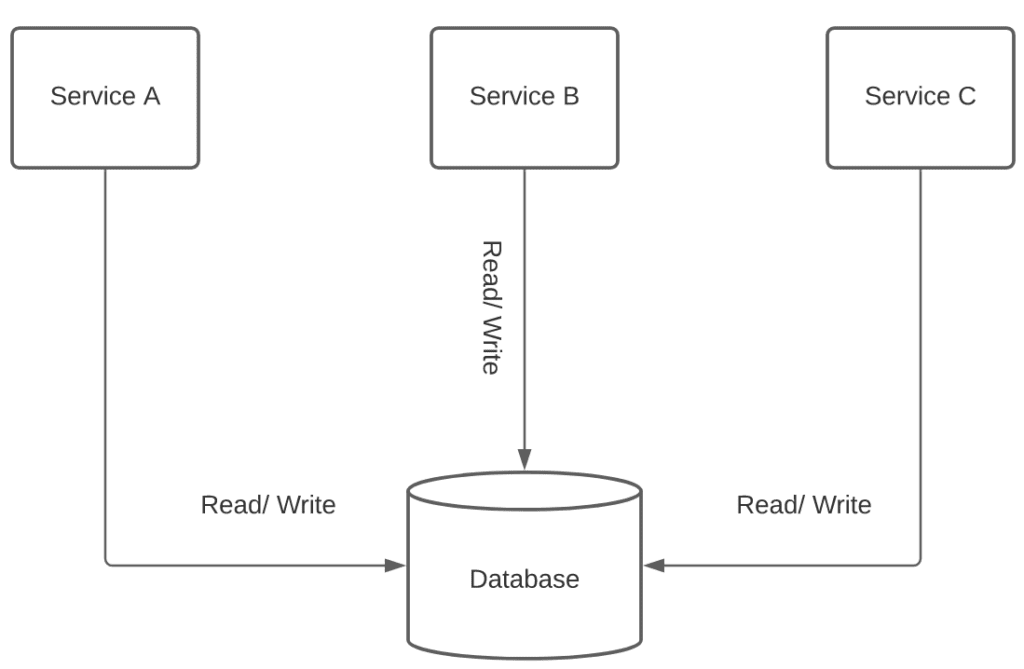
* Each database can scale independently
* Microservices Domain data is encapsulated within the service
* If one of the database server is down, this will not affect to other services

**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.



* Another important thing is no need to exchange stored data between microservices.
* Microservices with shared databases can’t easily scale.

1. **Command and Query Responsibility Segregator (CQRS)**

Command = create/update/delete operations which change the state of entity.

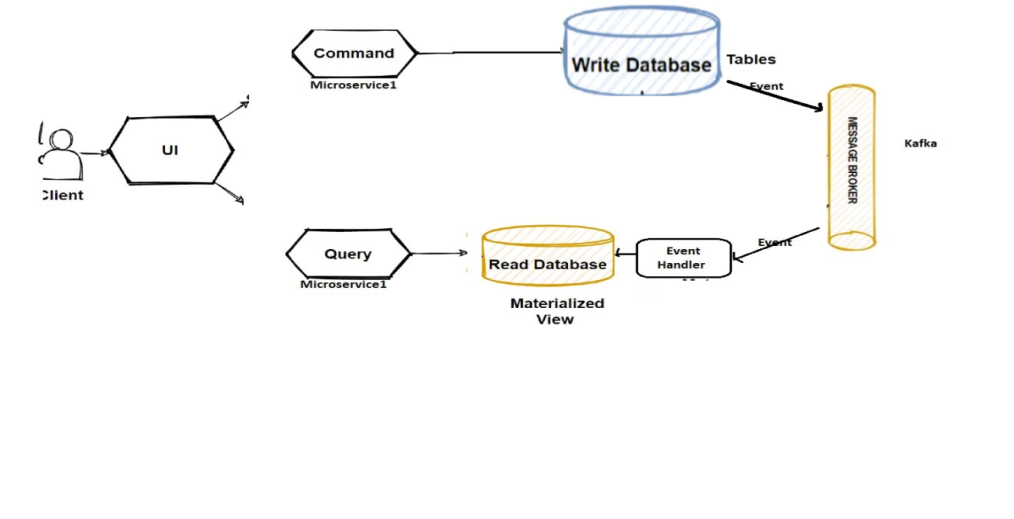
Query = To get the exact state of the entity by applying some queries.

* 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**.

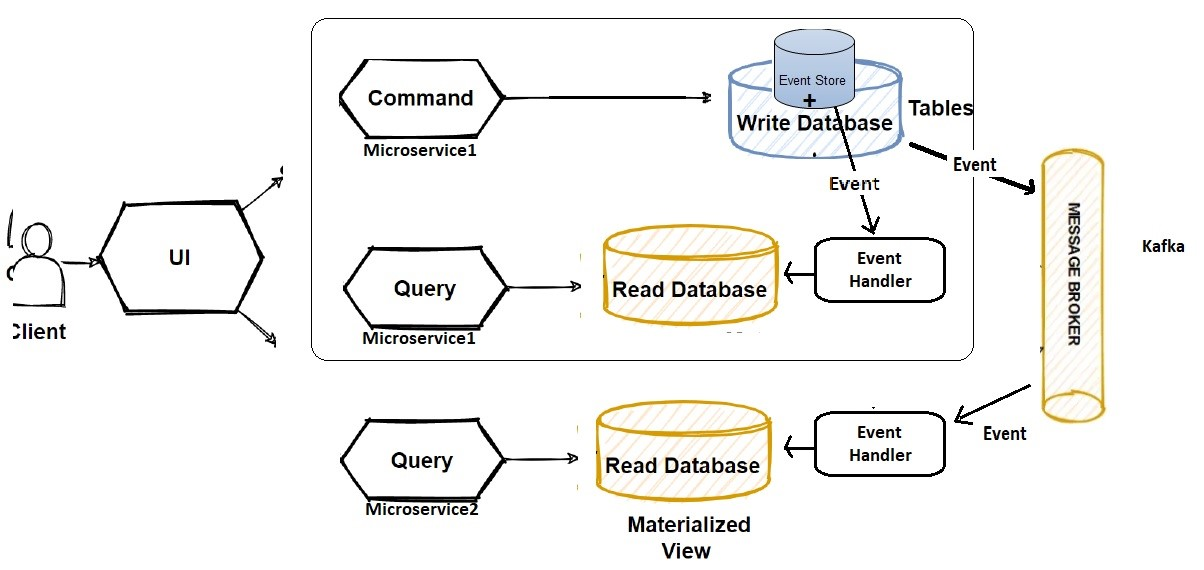
**Why CQRS design pattern came?**

* 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 :**

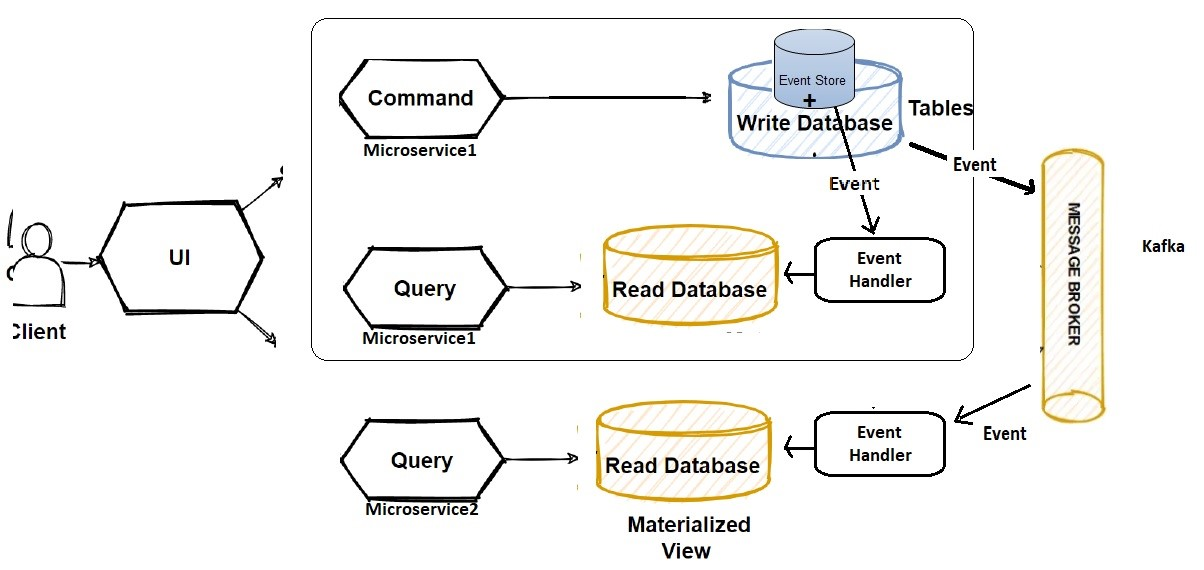
* CQRS offers to use “**separation of concerns”** principles and separate reading database and the writing database with 2 databases.
* By this way we can even use different database for reading and writing database types like using **no-sql for reading** and using relational database **for crud operations 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:**