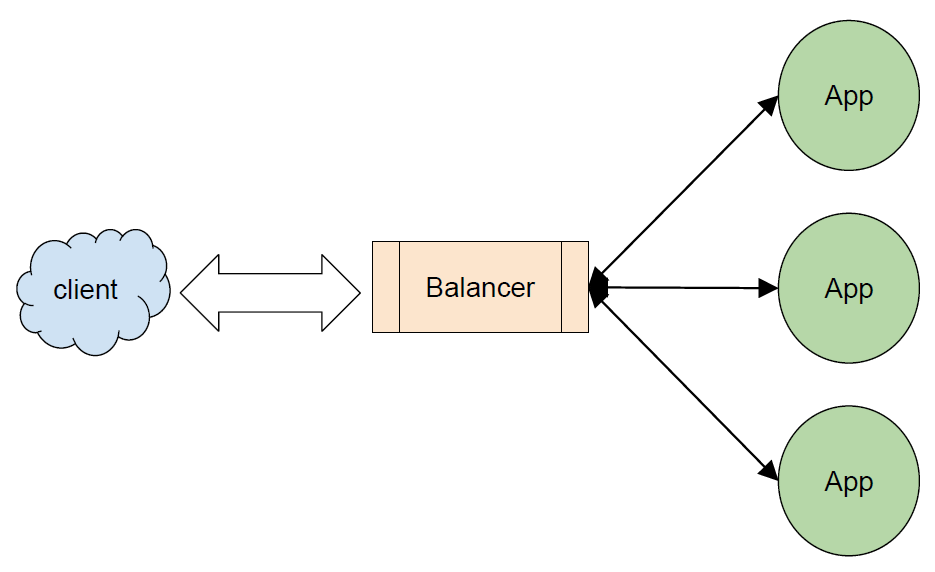
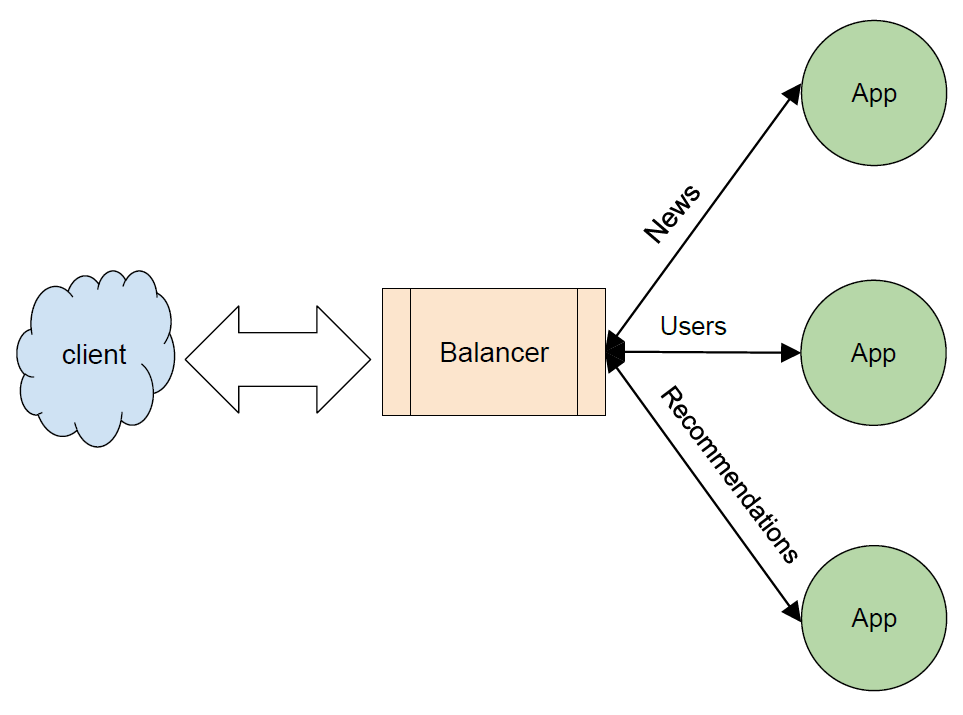
**Microservice Patterns**

**What is scalability of an application ?**

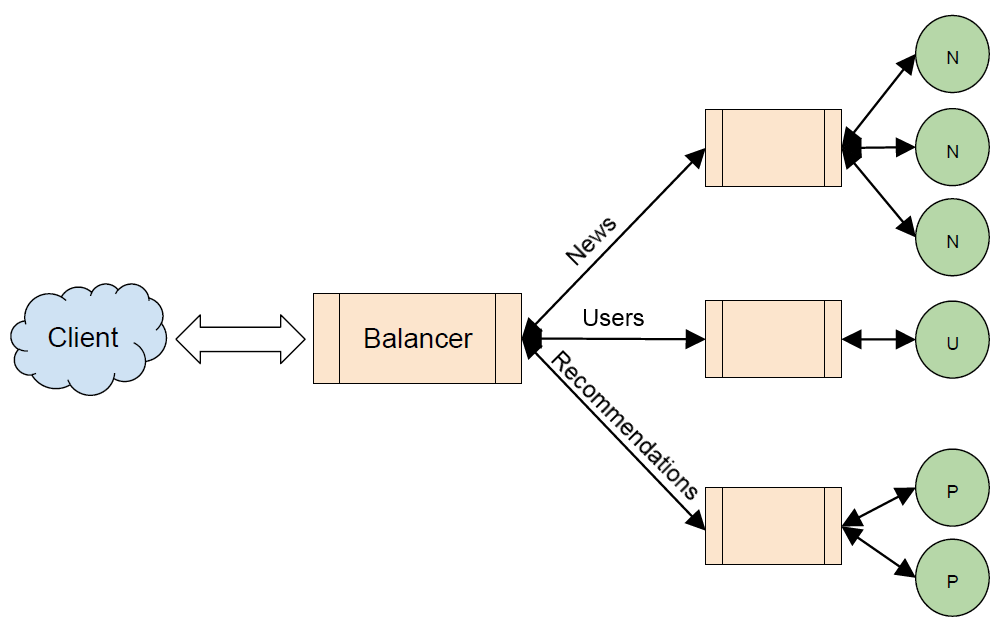
Scalability can be done by three axis x , y and z

X –axis : On the x-axis, this strategy targets the horizontal scalability with the same application server replicated n times in full and in a balanced order of 1/n



Y-axis : Functionality wise scaling

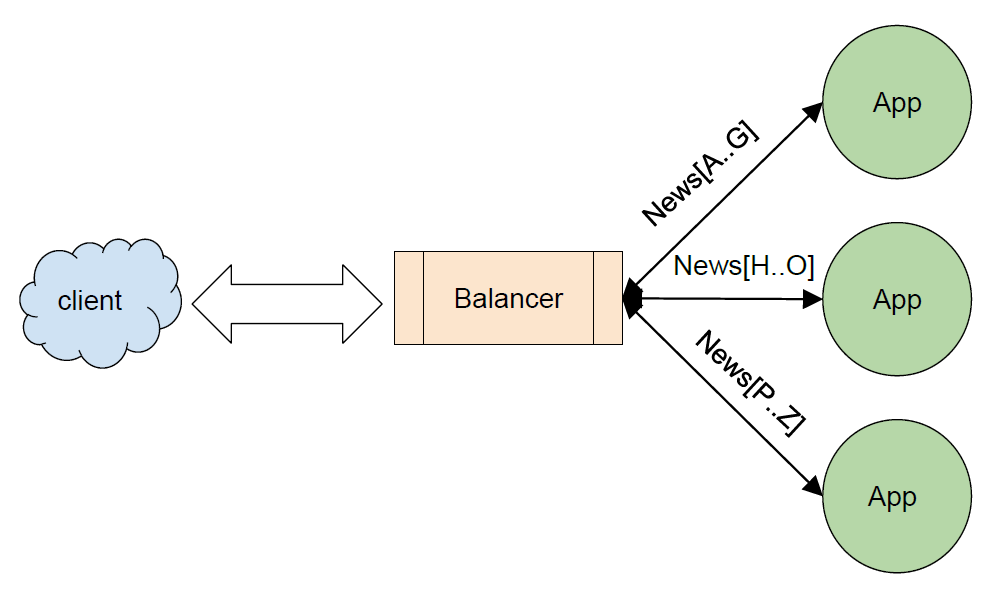
This join between *y*-axis and *x*-axis allows us, occasionally, to bring scalability to just part of the microservices. In the following diagram, it can be seen that News was the most scaled microservice, followed by Recommendations, but Users have no major changes. This type of scalability technique greatly reduces the drawbacks of shared resource access, as each microservice structure manages and uses only its own resources, such as caches and databases. Take a look at the following diagram:



Z axis : Data wise scaling

The z-axis is very similar to the x-axis when it comes to scalability structure, as it distributes exactly the same code on each server. The big difference is that each server responds to a specific subset of data.

This means that, in a global application, the database of a microservice is distributed by region and is preferably available for this region, that is, people who access the website in Europe will, preferably, see the European news.



**How communication between two microservices work ?**

Synchronous :

* HTTP
* TCP
* WebSockets
* Sockets
* RPC
* SOAP

Asynchronous :

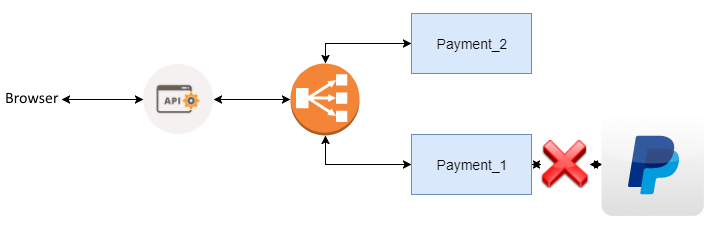
In some direct communications between microservices, timing may be important, but there are other times where the process can simply be asynchronous; there's no need for an immediate response or confirmation of success, all that is required is to simply run a task. For this approach, the message broker is just perfect.

Example of some message broker Kafka, RabitMQ

**What is fail fast ?**

In the case of the payment process in an online store, the microservice is the final step of the purchase, being responsible for completing the payment of the product requested by the client. Payment must be communicated using credit card gateways. If one of these gateways presents any outages, our microservices can start to fail due to dependency on external services.

The following diagram shows an example where our microservice payment is trying to establish a connection with a billing system, but there are problems:



We can't just wait for the connection to be restored; we must fail fast and take a decision on what to do. For this type of problem, the circuit breaker approach can be very interesting. We could set up a timeout or connection failure policy to offer some other form of payment, or submit a friendly message of failure. This avoids a possible widespread system failure due to exhaustion of resources.

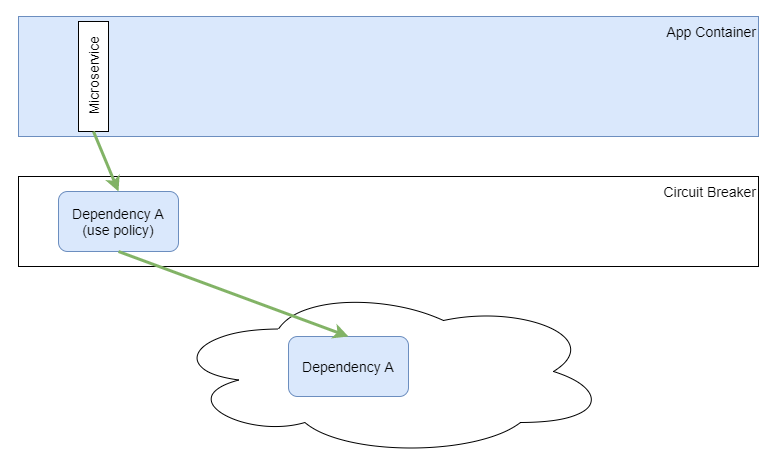
**What is Circuit breaker ?**

The circuit breaker is an automatic operating switch that turns itself off when there is an overload or short circuit. As well as the electric fuse, the purpose of the circuit breaker is failing quickly and protecting electrical installations. In the case of a microservice, it protects the general integrity of the application.

Imagine the situation where a microservice presents slowness. The requests keep coming, and it begins to be queued. At some point, collateral damage happens. Especially in the case of a microservice that has a dependency on communication with other microservices, we need to apply the circuit breaker.

The concept of the circuit breaker is relatively simple, possessed by only two states:

* **On**: Releasing the call to the external dependency
* **Off**: Failing the call immediately and taking a previously configured action
* In practice, instead of microservices directly accessing the external dependency, the circuit breaker will put itself in the middle of the call. In case of any failure according to predetermined parameters, which, for example, may be a timeout, the circuit breaker interrupts the communication with the dependency that is failing. Of course, something can be taken by the microservice side. The behavior of the circuit breaker can be seen in the following diagram:



* There are some frameworks that assist in the implementation of the circuit breaker. Currently, the framework with greater prominence is Hystrix, which was created by the Netflix development team

**What is design principals in microservice ?**

SOLID design principal

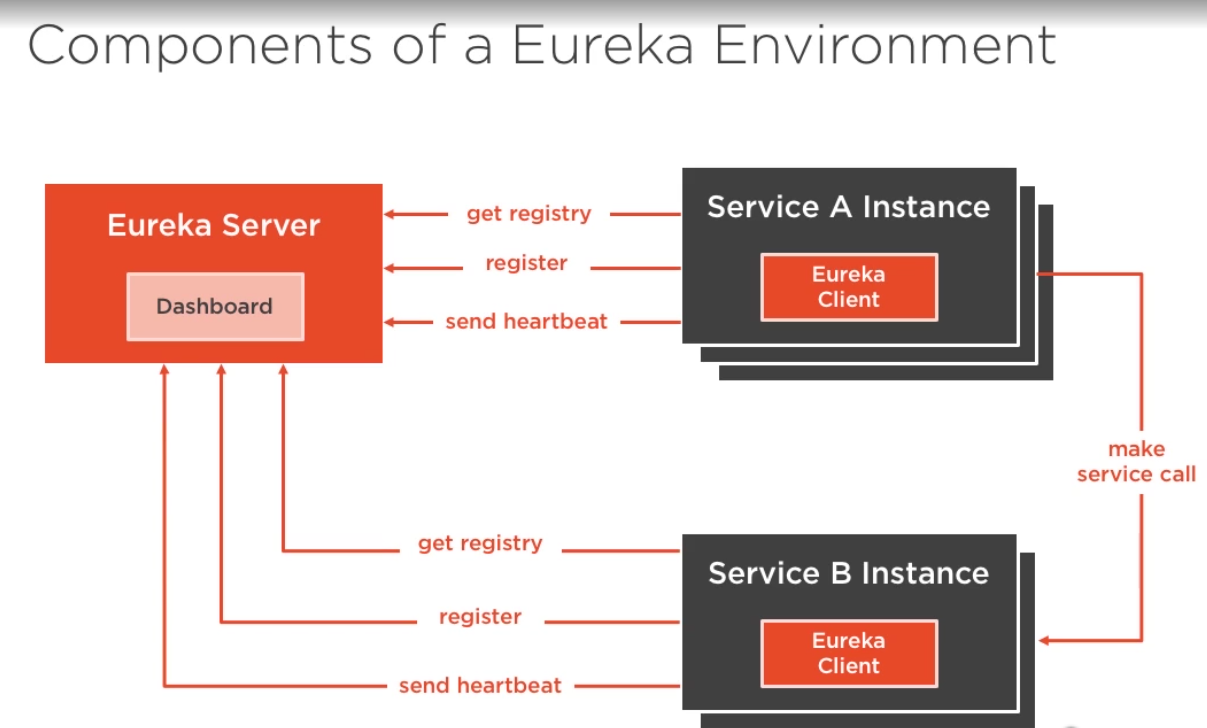
<https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#SRP>

**Core chracteristics of microservice ?**

* Components exposed as services it can be http, soap or event driven.
* Tied to specific domain, means each microservice represent a specific functionality, like booking service, payment service(Domain driven design)
* Each microservices are loosly coupled
* fault tolerance needs to be taken , as example circuit breaker.
* Every microservices are handled and deployed by separate team.

**What is service discovery ?**

It is a service directory or registry, it stores a key value pair of service id and the list of server containing the service. It has the responsibility to check healthstatus of of the registered services. Rhere is no database to store this information



Eureka is a spring boot started project from netfix which supports service discovery

**Eureka server**

Dependencies : Eureka server, actuator

Denote the main class with @EnableEurekaServer

define application.properties

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

**Register services with eureka server**

denote main class with @EnableDiscoveryClient this generic for all service discovery

@enableEurekaClient this is dedicated to eureka service registry

If we want to send the service name of the microservice at the bootstarp set the application name and server.port at bootstrap.properties file rather than application.properties file

The microservice sends heartbeat in regular interval to the eureka server

Dependencies : web, actuator, eureka discovery

bootstrap.propties

spring.application.name=guestservices

spring.cloud.config.uri=http://localhost:9000

application.properties

eureka-client-register-with-eureka=true

eureka-client-fetch-registry=true

eureka-instance.instance-id =

${spring.application.name}:${random.int}

server.port =0

eureka.client.serviceUrl.defaultZone=http://localhost:8090/eureka

**OAuth using spring boot**

[**https://www.javainuse.com/spring/spring-boot-oauth-introduction**](https://www.javainuse.com/spring/spring-boot-oauth-introduction)

**Refer this link**

[**https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/**](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/)