-message queue được dùng trong hệ thống microservices với chức năng như 1 hàng đợi tin nhắn theo cơ chế FIFO(first in first out) , chứa các message tượng trưng cho các tác vụ bất đồng bộ cần thưc hiện

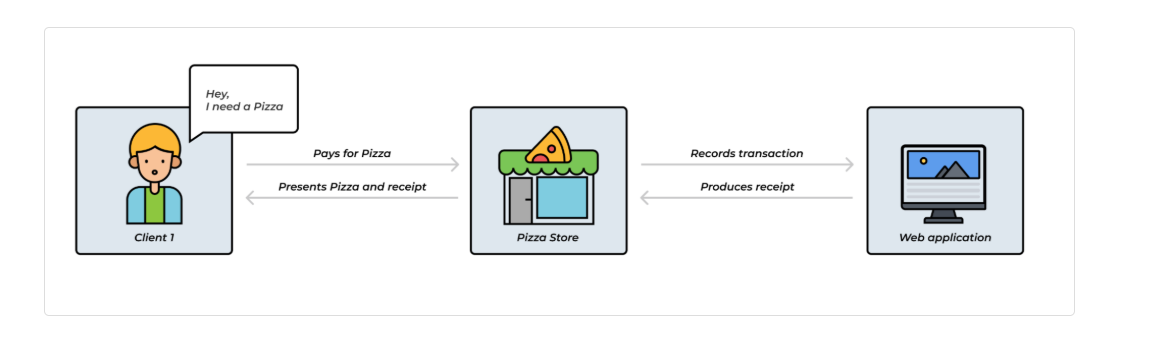
-Tưởng tượng bạn có 1 hệ thống order pizza hoạt động theo quy trình sau :   
  
+Khách hàng order pizza từ cửa hàng

+Cửa hàng tiếp nhận order thực hiện gửi yêu cầu tạo giao dịch đến web app

+Web app tiếp nhận giao dịch và xử lý , lưu trữ các thông tin cần thiết vào database

+web app gửi lại đơn order về cho cửa hàng sau khi đã xác thực để làm pizza

+cửa hàng nhận đơn , làm pizza và giao cho khách hàng

+khách hàng nhận được hàng , cửa hàng kết thúc phiên làm việc với khách hàng và chờ đợi phiên làm việc với khách hàng tiếp theo  
  


-Giả sử mỗi lần ra hệ thống web app thực hiện lưu trữ , xử lý order cho 1 khách hàng sẽ tốn 2s

-với 1 hệ thống như trên : hoạt động theo cơ chế đồng bộ , nếu bình thường chỉ có 5,10 khách hàng cùng 1 lúc , cửa hàng hoàn toàn có thể thực hiện các công việc một cách bình thường

-Giả sử vào 1 thời điểm nào đó cửa hàng ra một chương trình sale pizza giảm giá 1 nửa và số lượng khách hàng đặt hàng tăng đột biến : từ 5,10 khách hàng -> 1000 , 5000 khách hàng cùng lúc : khi này với mỗi khách hàng , hệ thống web app phải xử lý rất tốn thời gian , khách hàng sẽ phải đợi hàng giờ đồng hồ cho việc order pizza , tất nhiên khách hàng không thích điều đó , họ sẽ rời đi và bạn sẽ bị mất khách hàng , thậm chí với lượng req đồng thời quá lớn có thể gây quá tải và sập server -> thiệt hại cho cửa hàng.

-Hướng xử lý sẽ là :

+ khi 1 khách hàng đặt hàng, họ sẽ điền tất cả thông tin cần thiết vào 1 tờ phiếu , gửi nó cho cửa hàng

+cửa hàng sẽ nhận tờ phiếu đó và nói với khách hàng : “ok! Cửa hàng đã nhận được đơn của bạn , bạn hãy về trc và chờ đợi thông tin liên lạc từ phía cửa hàng “

+Sau đó cửa hàng sẽ đưa tờ phiếu điền tất cả thông tin của khách hàng giao cho web app để xử lý (lúc này khách hàng đã về). Giao theo cơ chế ACK-ban đầu cửa hàng sẽ giao cho web app 1 lượng req cố định để không bị quá tải ví dụ chỉ giao 50 đơn mặc dù có 1000 đơn . Mỗi khi web app xử lý xong 1 đơn nó sẽ gửi 1 bản hóa đơn đã xử lý cho cửa hàng (ACK) cửa hàng biết là web app đã xử lý thành công , lại tiếp tục gửi thêm 1 đơn sao cho số lượng đơn không vượt quá lượng giới hạn cho phép của web app => web app không bị quá tải

+web app xử lý xong trả về đơn cho cửa hàng , cửa hàng sẽ thực hiện việc giao pizza dựa trên thông tin web app đã xử lý

* Khách hàng sẽ không phải chờ đợi cho đến khi mọi tác vụ được thực hiện xong , họ sẽ làm việc của họ trong lúc chờ đợi theo cơ chế “I will call you when I done”
* Hệ thống sẽ không phải chịu 1 lượng req quá cao dẫn đến quá tải và sập server.

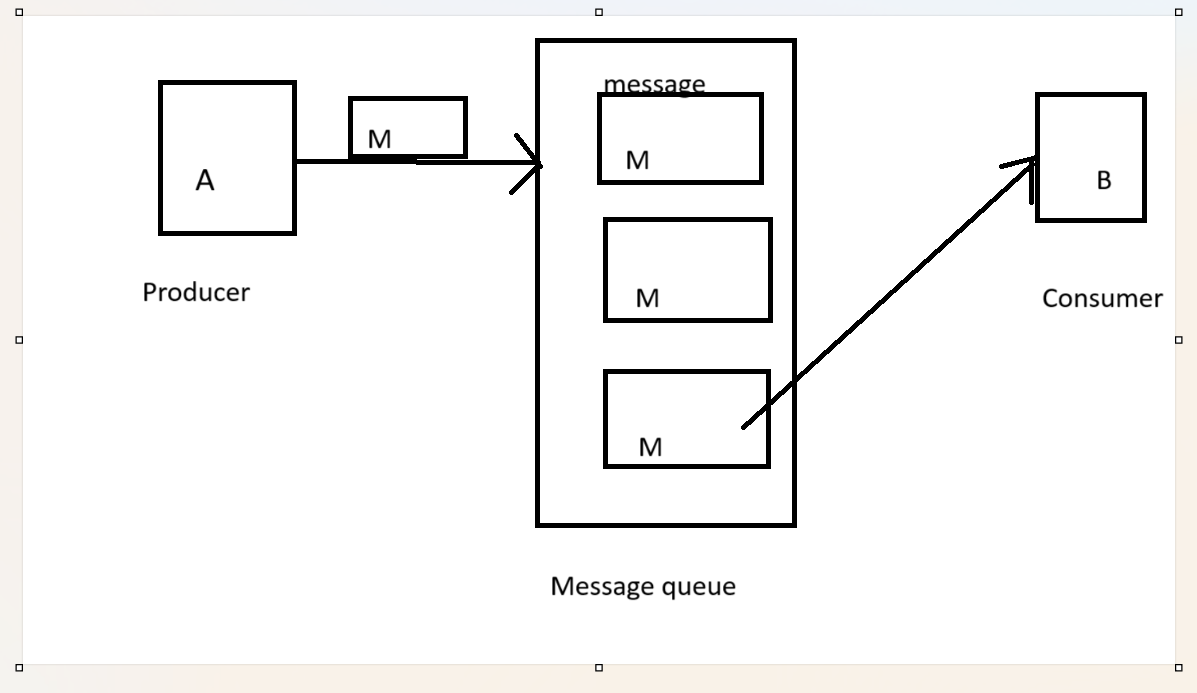
Và đây chính là ý tưởng cho mô hình message queue

\*)Mô hình message queue (RabbitMQ) :

-giả sử có 2 server giao tiếp với nhau thông qua 1 hàng đợi tin nhắn

-Server A sẽ là server tạo ra các message (task yêu cầu thực hiện) gửi vào message queue – Producer

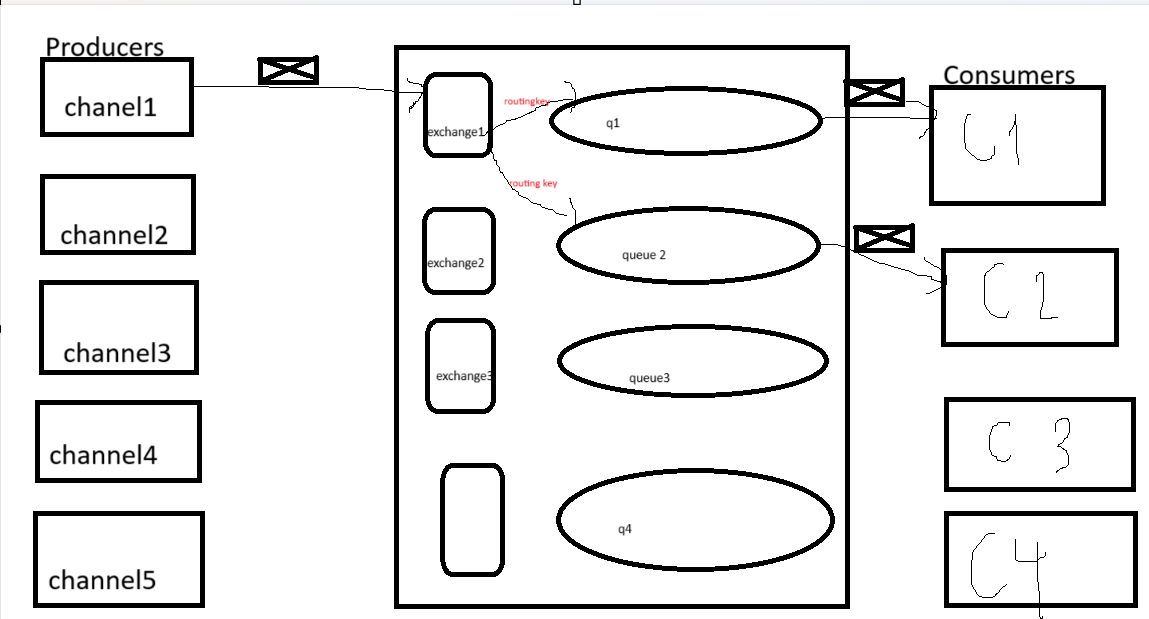
-Server B sẽ là server nhận các message từ queue để thực hiện – Consumer



\*) RabbitMQ :

-Exchange : producer không trực tiếp gửi message vào queue mà cần thông qua 1 exchange , exchange này sẽ được binding với queue. Và tùy theo từng type (direct , fanout , header , topic) mà exchange sẽ sử dụng routing key để có thể đưa message vào đúng hàng đợi

-Trong RabbitMQ có thể có cùng lúc nhiều queue và exchange , channel cùng 1 lúc.



-work flow của một message pub/sub là :

+ Producer sẽ tạo ra 1 exchange , tùy loại mà có dùng routingKey hay không

+Producer gửi message vào exchange

+Consumer kiểm tra sự tồn tại của exchange

+Consumer tạo ra một queue mới

+Consumer binding queue được tạo ra với exchange tùy theo loại mà có routingKey hay không

+Consumer Subscribe queue nhận message từ queue để xủ lý

-Các loại exchange :

+fanout : gửi message đến tất cả các queue binding với nó (bỏ qua routing key)

+direct : gửi message đến các queue được binding với nó và trùng routingKey với exchange

+Topic : **Wildcard**:

* \* → khớp một phần
* # → khớp nhiều phần

Ví dụ : routing\_key = "game.win"

Binding:

* Queue A bind với "game.\*" → nhận game.win, game.lose, v.v.
* Queue B bind với "game.#" → nhận game.win, game.score.update, ...

+Headers : Thay vì dùng routing key, exchange sẽ định tuyến dựa trên các **header** trong message.

-chi tiết hơn về wildcard :

**\* (sao) – khớp đúng một từ (word)**

* Khớp đúng **một phần** (one word) trong routing key, được phân tách bằng dấu chấm (.).
* Ví dụ:
  + a.\*.c sẽ khớp a.b.c, nhưng **không** khớp a.b.d.c hoặc a.c

✅ Ví dụ:

Binding key: game.\*.win

Routing key: game.boss.win ✅ MATCH

Routing key: game.win ❌ NOT MATCH

Routing key: game.boss.1.win ❌ NOT MATCH

**🔸 # (thăng) – khớp 0 hoặc nhiều từ**

* Khớp từ **0 đến nhiều phần** (words), nên rất linh hoạt.
* Có thể nằm ở đầu, giữa, hoặc cuối binding key.

✅ Ví dụ:

Binding key: game.#.win

Routing key: game.win ✅ MATCH (0 từ giữa)

Routing key: game.boss.win ✅ MATCH (1 từ giữa)

Routing key: game.boss.1.win ✅ MATCH (2 từ giữa)

Routing key: game ❌ NOT MATCH

-Triển khai trong NodeJs :

Producer.js

const amqp = require("amqplib") // import thư viện

const main = async () => {

    const conn = await amqp.connect("amqp://localhost") // tạo connection

    const channel = await conn.createChannel() // tạo channel

    const exchange = "my-exchange" // tạo tên cho exchange

    channel.assertExchange(exchange, "direct") // tạo exchange

    channel.publish(exchange, "createPdf", Buffer.from(JSON.stringify({ /

        task: "doing",

        version: 1.0

    })))// push message vào exchange , và routingKey cho message là “createPdf”

    setTimeout(() => {

        conn.close()

        process.exit(0)

    }, 200); // đóng connect và chương trình sau 200ms

}

main()

\*) RPC :