

## **Notas exposición**

### **Aplicaciones monolíticas**

Antes de los microservicios, la mayoría de las aplicaciones eran monolíticas, lo que significa que todos los componentes (UI, lógica de negocio y base de datos) estaban empaquetados en una sola aplicación.

Esto pues que ventajas traía: simplicidad de desarrollo y despliegue. Sin embargo, presentaba un sinfín de problemas: dificultad para escalar, falta de flexibilidad y un tiempo de despliegue largo en caso de modificaciones, aparte de generar la necesidad de una gran capacidad de control.

### **Servicios Web**

Para facilitar la comunicación entre sistemas, surgieron los servicios web, que permiten que diferentes aplicaciones se comuniquen a través del internet. Un ejemplo... pues una tienda en línea donde el frontend (Angular, React) se comunica con un backend en Java mediante una API REST.

¿Qué ventajas nos trae esto? Una interoperabilidad, ya que, no importa el lenguaje o tecnología utilizado en el “servidor”, cualquier cliente puede conectarse y utilizar el servicio por medio de una solicitud. Sin embargo, viendo todo el “servidor” como un backend, pues nos trae un problema, ya que, al ser una sola pieza, puede ser difícil de mantener y manejar en caso de cambios o errores.

Además, si un endpoint recibe demasiadas solicitudes, se puede sobrecargar toda la aplicación, ya que, al no ser independiente el backend, un error en algún sector puede repercutir en todo el sistema.

### **Nacimiento de los Microservicios**

Ahora bien, para solucionar estas limitaciones de los servicios web llegan los microservicios, los cuales, dividen una aplicación en pequeños servicios independientes, cada uno con su propia lógica y base de datos si es necesario.

Un ejemplo de esto puede ser Netflix, que en lugar de tener un solo backend monolítico, tienen microservicios separados para manejar usuarios, recomendaciones, historial de vistas, pagos, etc. Así, si el microservicio de pagos falla, el resto del sistema sigue funcionando.

Con lo cual, ¿qué ventajas nos ofrecen? Brindan la capacidad de escalar el sistema por medio de los módulos o microservicios según sea la demanda de cada uno. También, un desarrollo flexible, ya que, se puede trabajar en diferentes módulos sin afectar a otros y, con esto, desplegar una parte del sistema es posible sin afectar a la aplicación completa.

Pero bueno, definamos que es un microservicio formalmente.

Es una arquitectura de software donde una aplicación se divide en pequeños servicios independientes que se comunican entre sí, así, cada microservicio es autónomo, tiene su propia lógica y, en muchos casos, su propia base de datos.

Imaginen que tienes un restaurante y, en lugar de que todo lo haga un solo cocinero (monolito), tienen diferentes especialistas: uno hace pizzas, otro postres y otro bebidas. Cada uno trabaja de forma independiente, pero juntos ofrecen el servicio completo. Eso son los microservicios en software.

Estos presentan las siguientes características:

- **Autonomía** → Cada microservicio funciona de manera independiente.
- **Escalabilidad** → Puedes aumentar los recursos de un solo microservicio sin afectar a los demás.
- **Desarrollo distribuido** → Diferentes equipos pueden trabajar en diferentes microservicios.
- **Comunicación a través de APIs** → Se usa REST, gRPC, o mensajería como Kafka o RabbitMQ.
- **Despliegue independiente** → No es necesario actualizar toda la aplicación si cambias un solo microservicio.
- **Manejo de fallos** → Si un microservicio falla, el resto sigue funcionando.

Si en Netflix falla el servicio de pagos, los usuarios aún pueden ver sus series, porque el microservicio de streaming sigue funcionando.

## Tecnologías para microservicios

Lenguajes:

- Java (Spring Boot)
- Python (FastAPI, Flask)
- Node.js (Express.js)
- GoLang

Comunicación entre microservicios:

- REST (JSON sobre HTTP)
- gRPC (mejor rendimiento que REST). Es un protocolo de comunicación rápida entre microservicios desarrollado por Google. Usa protobufs (Protocol Buffers) en lugar de JSON o XML, lo que lo hace más eficiente. Es más rápido que REST porque reduce el tamaño de los mensajes.

### ♦ Ventajas de reducir el tamaño

- ✓ Menos consumo de ancho de banda.
- ✓ Mayor velocidad de transmisión.
- ✓ Menos latencia en la comunicación entre microservicios.

- RabbitMQ / Kafka (mensajería para microservicios asíncronos). Son mensajeros para microservicios. En lugar de que un servicio haga una solicitud directa a otro, se envían mensajes a una "cola" y otros microservicios los leen cuando pueden procesarlos.
- Rabbit -> FIFO. Kafka -> basado en logs, en eventos.

#### ✚ Ejemplo de uso:

- Si un usuario hace un pedido en Amazon, el servicio de "Pedidos" envía un mensaje a RabbitMQ en lugar de llamar directamente a "Pagos".
- Luego, el servicio de "Pagos" recoge el mensaje cuando esté listo y lo procesa.
- Así se evitan bloqueos y los microservicios trabajan de forma independiente.

### Orquestación y despliegue:

- Docker (contenedores). Es una plataforma que permite crear contenedores (una especie de "mini máquinas virtuales" que incluyen todo lo necesario para ejecutar una aplicación). Facilita el despliegue porque el código siempre se ejecuta igual en cualquier máquina.
- Kubernetes (gestión de contenedores). Es una herramienta que gestiona múltiples contenedores Docker. Se encarga de la escalabilidad automática (por ejemplo, si una app tiene mucho tráfico, Kubernetes crea más instancias). También maneja fallos: si un microservicio falla, Kubernetes lo reinicia automáticamente. Si tienes un servicio de pagos y el tráfico aumenta en Black Friday, Kubernetes duplica las instancias de ese servicio para que todo siga funcionando sin problemas.
- ¿Qué es la orquestación? Cuando tenemos muchos microservicios, necesitamos una forma de coordinarlos automáticamente, para asegurarnos de que se ejecuten correctamente, se comuniquen bien y escalen cuando sea necesario. Por ejemplo, imaginemos que tenemos 10 microservicios, pero si la demanda sube, necesitamos más instancias de algunos de ellos. La orquestación se encarga justamente de crear, monitorear y balancear estos microservicios automáticamente. Para esto se usan los Docker u kubernetes.

### Comunicación en microservicios

## ✖ 2. ¿Cómo funciona la comunicación en microservicios?

En microservicios, **NO HAY UN SERVIDOR CENTRAL QUE LLAME A TODOS**. En su lugar, hay varias formas de manejar la comunicación:

### ✅ 1. API Gateway (Arquitectura más común)

- En lugar de que cada cliente llame a cada microservicio, hay un **único punto de entrada** que distribuye las solicitudes.
- **Ejemplo:** Si un usuario entra a Netflix, el API Gateway recibe la solicitud y la distribuye a los microservicios de "Catálogo", "Recomendaciones" y "Usuarios" según sea necesario.

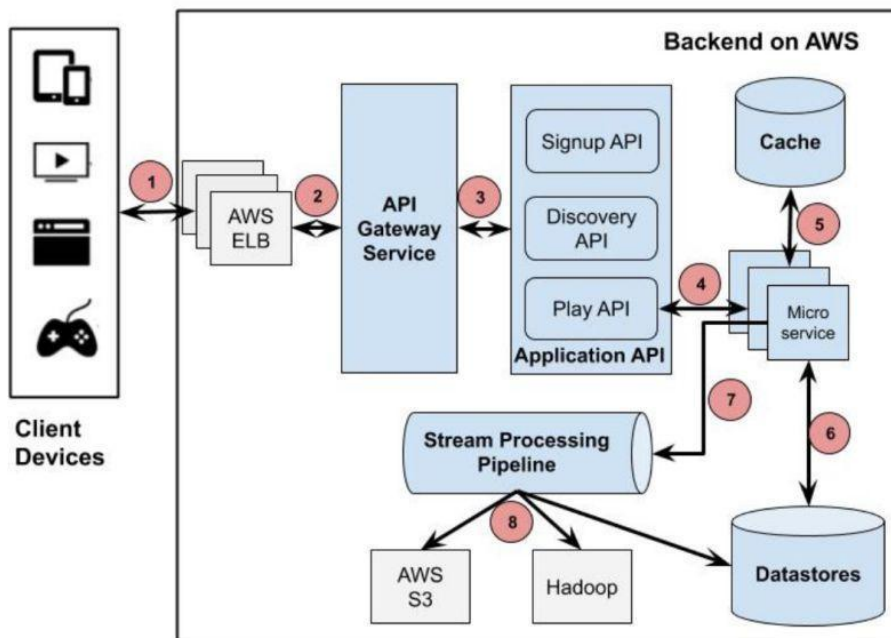
### ✅ 2. Comunicación Directa entre Microservicios

- Un microservicio puede llamar a otro directamente usando **gRPC o HTTP**.
- **Ejemplo:** En Uber, el microservicio de "Viajes" consulta al microservicio de "Pagos" para verificar si un usuario tiene saldo.

### ✅ 3. Comunicación Asíncrona con RabbitMQ/Kafka

- Un servicio **no llama directamente** a otro, sino que **deja un mensaje en una cola** para que otro lo recoja cuando esté listo.
- **Ejemplo:** En una tienda online, el servicio de "Pedidos" envía un mensaje a "Envíos" cuando un pago se confirma, en lugar de hacer una llamada directa.

## Microservices Architecture at **NETFLIX**



## Desafíos y desventajas

- **Mayor complejidad** → Hay que gestionar la comunicación entre servicios.
- **Monitoreo más difícil** → Se necesitan herramientas como Prometheus o ELK.
- **Latencia en la comunicación** → Si hay muchas llamadas entre microservicios, puede haber demoras.

Si en una tienda en línea hay demasiados microservicios haciendo peticiones entre sí, puede aumentar el tiempo de carga si no se optimiza bien.

En los servicios web tradicionales (monolíticos), **toda la aplicación depende de una conexión HTTPS** para que funcione. Si hay un problema en la red o en el servidor, **toda la app puede fallar**.


✓ Microservicios solucionan esto de varias maneras:

1 **Comunicación interna sin HTTPS:** Los microservicios pueden comunicarse entre sí mediante protocolos internos como gRPC, RabbitMQ o Kafka, que no dependen exclusivamente de HTTP.

2 **Redundancia y fallos controlados:** Si un microservicio falla, el sistema puede seguir funcionando sin afectar toda la aplicación. Por ejemplo, si el servicio de "recomendaciones" de Netflix se cae, el servicio de streaming sigue funcionando.

3 **Balanceo de carga:** Se pueden distribuir los microservicios en múltiples servidores o regiones, asegurando que si uno falla, otro tome el control.

🔥 **Ejemplo real:**

- Si Amazon tuviera un sistema monolítico y se cae su base de datos central, **toda la tienda dejaría de funcionar**.
- Con microservicios, cada parte (pagos, catálogo, usuarios) es independiente. Si se cae "recomendaciones", los usuarios aún pueden  ar.