

Por qué Microservicios

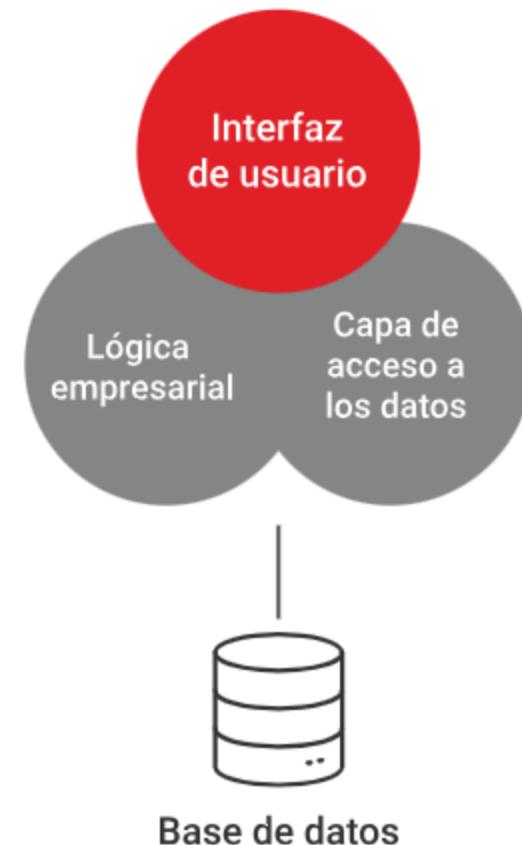
Respuesta al problema de mantenimiento y
evolución de **sistemas monolíticos**



Qué es un monolito

Aplicación de software en la que todas sus **capas** (interfaz de usuario, lógica de negocio y acceso a datos) están **combinadas** en un mismo programa y sobre una misma plataforma.

ARQUITECTURA MONOLÍTICA



Problemas de los monolitos

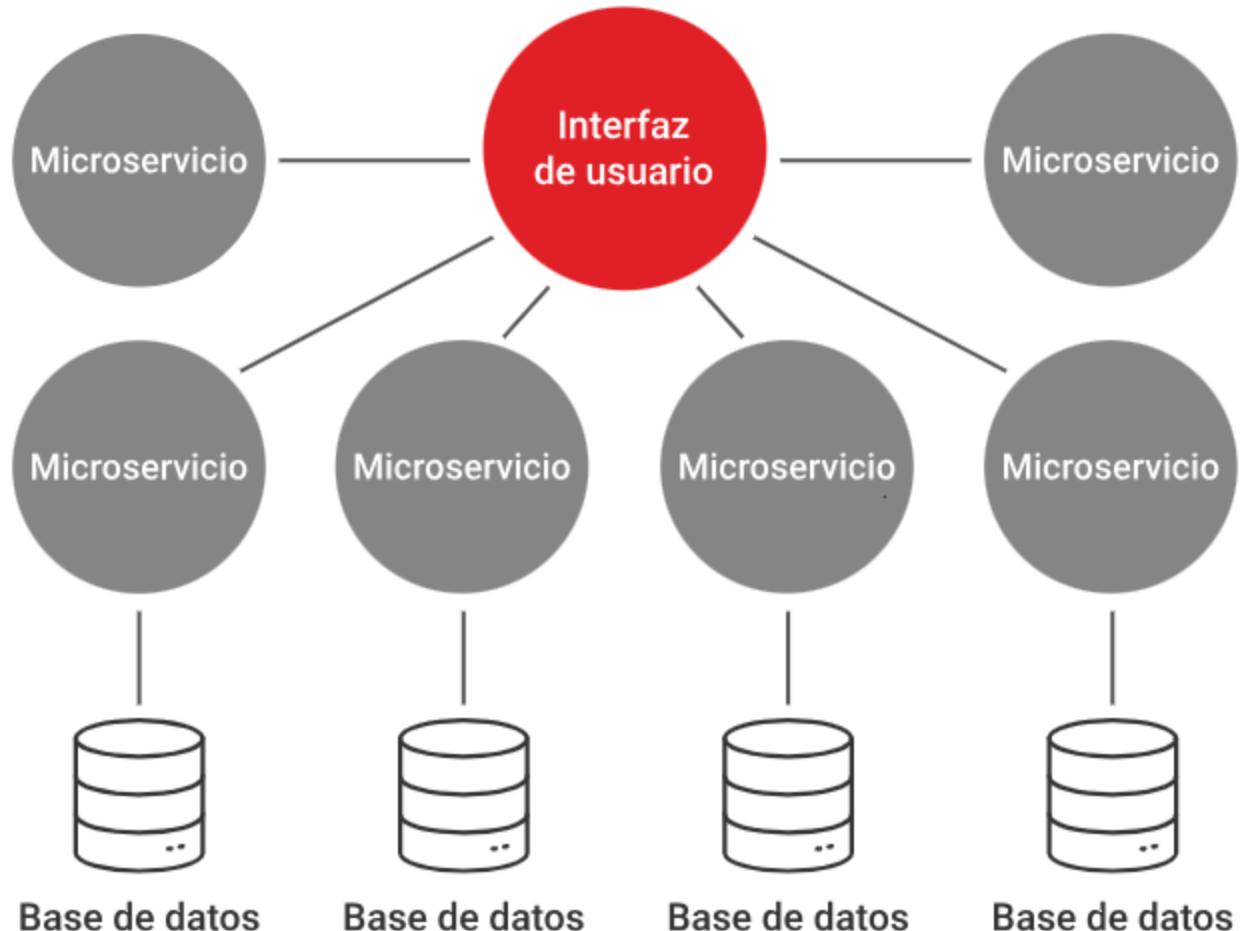
- Si se produce un fallo del sistema **se caen todos** los servicios
- Es difícil de escalar
- Imposibilidad de innovación tecnológica
- Despliegues o actualizaciones conflictivas
- **Complejo de gestionar** equipos de desarrollo
 - Alto número de desarrolladores
 - Nivel de conocimiento de todo el Sistema
 - Difícil desarrollar funcionalidades en paralelo



Qué son los microservicios

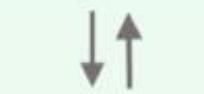
Pequeñas aplicaciones con una funcionalidad muy **concreta** y un alto nivel de **especialización** que trabajan en **conjunto**

ARQUITECTURA DE MICROSERVICIOS

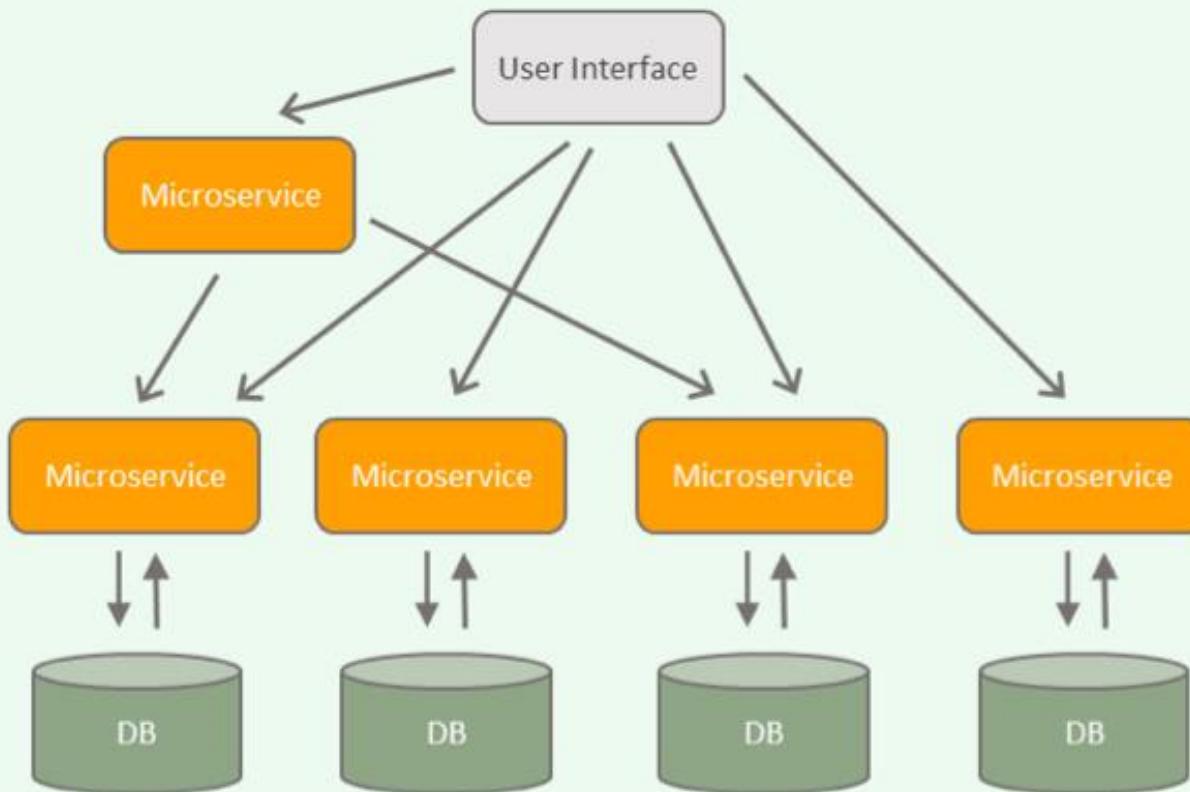


Monolito vs Microservicio

MONOLITHIC ARCHITECTURE

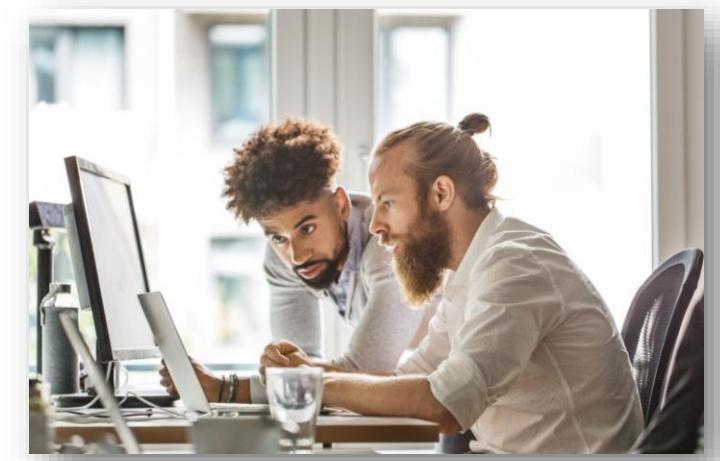


MICROSERVICES ARCHITECTURE



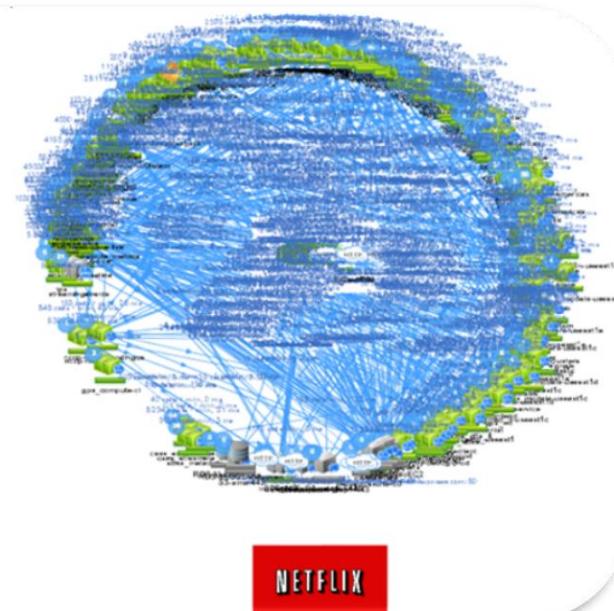
Ventajas de los microservicios

- Alta **tolerancia a fallos**. Si una parte del sistema se cae lo demás sigue funcionando (Circuit breaker)
- Permite escalabilidad del sistema
- Implementación **simple**
- Código más **mantenible** (menos interdependencias)
- Agilidad de cambios
- Gestión de equipos más simple
- Despliegues y actualizaciones con riesgos controlados
- Permite uso de distintas tecnologías



Problemas microservicios

- **Orquestación compleja** de los sistemas
- Aumento de la complejidad del entorno de desarrollo
- Los **contratos** de los microservicios (API) no son fáciles de cambiar
- Los equipos de desarrollo necesitan formación y/o experiencia

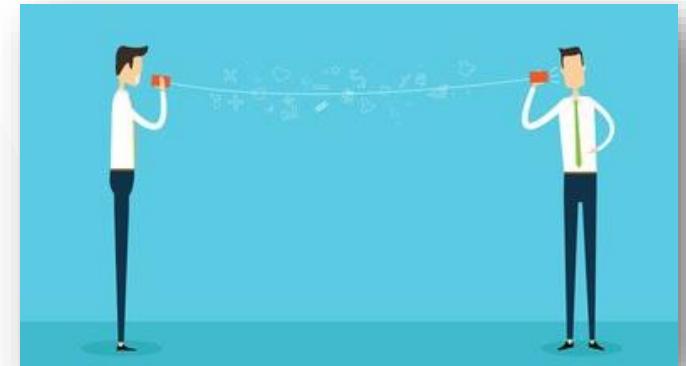


Comunicaciones entre microservicios

Se puede utilizar el tipo de comunicaciones que se quiera, pero lo recomendable es que sea algún **estándar**. De esta forma ya se tienen librerías para el desarrollo rápido y eficiente

Tipos de comunicaciones:

- REST
- gRPC
- SOAP
- Kafka
- Otras



Patrones de software utilizados en microservicios

Patrones para la descomposición	Patrones de infraestructura	Patrones de integración	Patrones de acceso a datos	Patrones de observabilidad	Patrones para servicios externos
Por capacidades de negocio	Configuración centralizada	API Gateway	Tabla de índices	Agregación de logs	Un backend por frontend
Por subdominios (enfoque DDD)	Descubrimiento y registro	Publicador/suscriptor	Sharding	Monitoreo de punto final	Capa anticorrupción
Estrangulador	Balanceo de carga	Coreografía	Una base de datos por servicio	Métricas de rendimiento	
Bulkhead	Reintentos	Orquestación	CQRS	Traceo distribuido	
	Circuit breaker	Saga			
	Azul/verde				

Circuit Breaker (1/2)

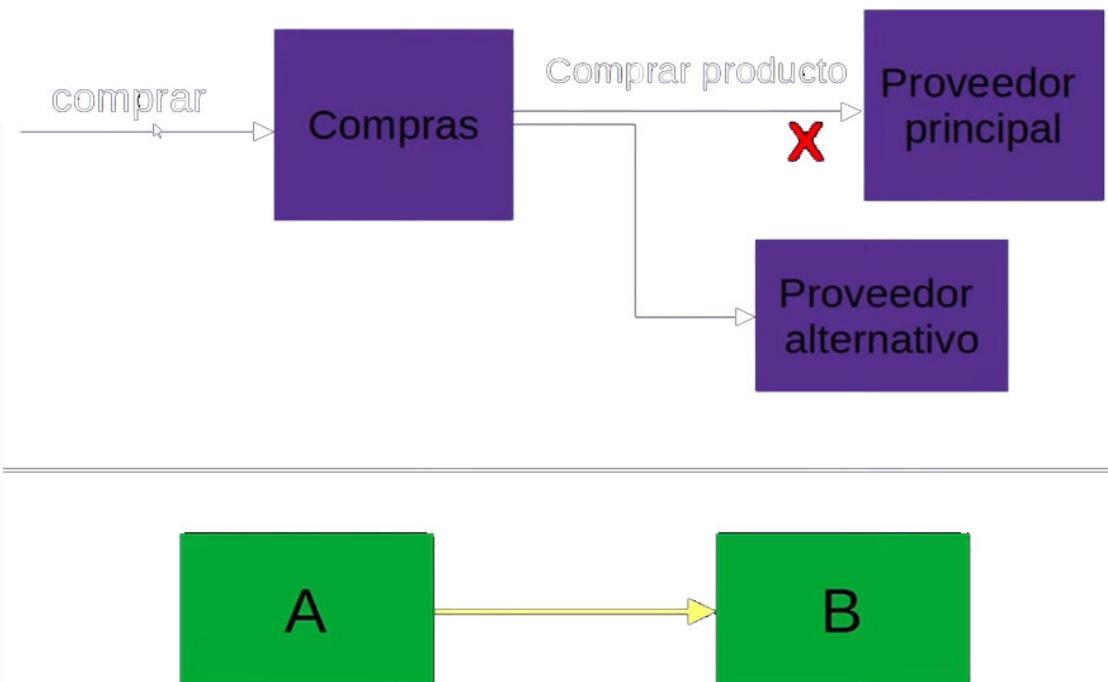
Objetivo: identificar cuando hay un fallo de un sistema y contener el problema.

Acciones:

- Informar del error
- Gestionar el problema (se abre el circuito)

Ventajas:

- **Monitorización:** el sistema está controlado en tiempo real. Ante un fallo se disparan las alarmas
- **Sobrecarga:** al abrir el circuito se corta un posible efecto de bola de nieve en todo el sistema, el problema está controlado
- **Tolerancia a fallos:** el Circuit Breaker puede redireccionar la petición al siguiente proveedor en caso de que alguno falle, evitando tener que enviarle el error al cliente.



Circuit Breaker (2/2)

La idea básica del circuit breaker es encapsular a una llamada a una función en un objeto CircuitBraker, quien maneja los fallos (mencionados anteriormente). Una vez que los fallos lleguen a una cantidad, el sistema cambia de dinámica, y a partir de ese momento devuelve errores a todas las llamadas hechas el CircuitBraker, sin siquiera ejecutar la función protegida. Un circuit breaker tiene principalmente tres estados:

Cerrado : Permite la realización de llamadas.

Semi-aberto: Se evalúa el tiempo transcurrido desde el último error de conexión, y se realiza o no una nueva llamada.

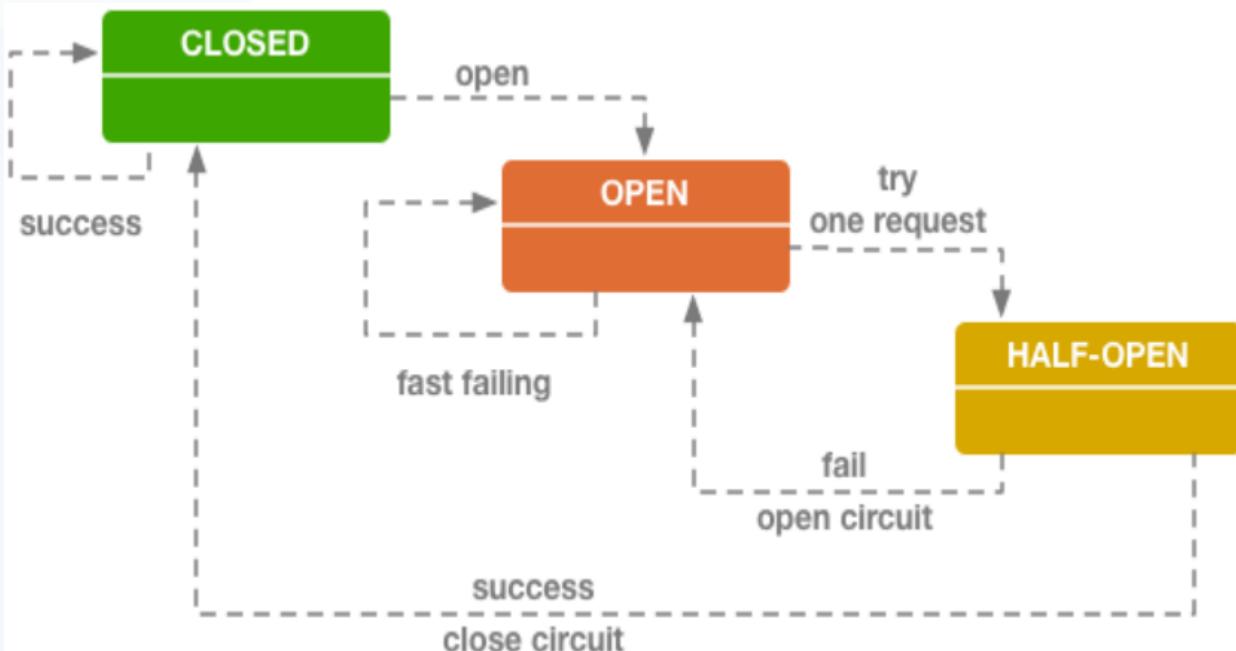
Abierto: Las llamadas no se realizan y se devuelven , cada vez que se intente realizar una, un error.

Mediante el registro de fallos podremos decidir cuando el “circuito” se abrirá impidiendo la realización de más llamadas.

A cerrado: cuando la conexión es exitosa, se establece el numero de fallos a cero y se pone el estado del CircuitBraker a “cerrado”.

A abierto: cuando numerosas conexiones han resultado en fallo (número acumulativo de fallos > número máximo de fallos) el circuito se abre, y se impiden las conexiones

A semi-Abierto: Si el estado del circuitBraker es “abierto” y la diferencia de tiempo (tiempo desde la última llamada con error > un valor arbitrario) se pone el estado a “semi-abierto”. La próxima vez que se intente realizar una llamada será como si el circuito estuviese “cerrado”, sin alterar el numero de fallos.



AUTHORIZATION SERVER

En la arquitectura de microservicios, un Authorization Server (Servidor de Autorización) es un componente crucial que se encarga de la seguridad, especialmente la autenticación y autorización de los usuarios y servicios. Funciona como un intermediario centralizado que emite tokens de acceso después de autenticar con éxito a los usuarios y validar sus permisos. Estos tokens luego se utilizan para acceder a los diferentes microservicios en el sistema.

Autenticación: Cuando un usuario intenta acceder a un recurso o servicio, primero se autentica en el Authorization Server. Esto suele hacerse mediante un nombre de usuario y contraseña, aunque también pueden utilizarse métodos más sofisticados como la autenticación multifactor.

Emisión de Tokens: Una vez autenticado, el servidor emite un token (generalmente un JWT - JSON Web Token) que contiene información sobre la identidad del usuario y sus permisos o roles.

Validación de Tokens: Cuando el usuario intenta acceder a un microservicio, este servicio valida el token con el Authorization Server para asegurarse de que es válido y que el usuario tiene los permisos necesarios.

Acceso a Recursos: Una vez validado el token, el usuario puede acceder a los recursos o servicios solicitados.

Este enfoque centralizado para la gestión de la autorización facilita el mantenimiento de la seguridad en sistemas complejos, donde diferentes microservicios pueden tener diferentes requisitos y niveles de acceso.



CONFIGURATION MANAGER

En un entorno de microservicios es una herramienta o componente clave que ayuda a manejar y centralizar la configuración de todos los microservicios en un sistema. Este concepto es especialmente importante en arquitecturas basadas en microservicios debido a la naturaleza distribuida y a menudo compleja de estos sistemas. Aquí te detallo sus funciones principales y ventajas:

Centralización de la Configuración: En sistemas de microservicios, cada servicio puede tener su propia configuración, como información de conexión a bases de datos, parámetros de entorno, URLs de servicios externos, entre otros. Un Configuration Manager centraliza estas configuraciones en un lugar único, facilitando su gestión y mantenimiento.

Consistencia y Sincronización: Asegura que todos los servicios estén sincronizados con la última versión de la configuración. Esto es crucial para mantener la consistencia en todo el sistema, especialmente cuando se realizan cambios o actualizaciones.

Manejo de Configuraciones por Entorno: Permite manejar diferentes configuraciones para distintos entornos, como desarrollo, prueba y producción, lo cual es una práctica común en el desarrollo de software.

Automatización y Dinamismo: Ofrece la capacidad de actualizar la configuración de manera dinámica y automática. En algunos sistemas, es posible cambiar la configuración sin necesidad de reiniciar los servicios, lo cual es una gran ventaja en términos de disponibilidad y flexibilidad.

Seguridad: Al centralizar la configuración, también se centraliza la gestión de secretos y credenciales, lo que permite implementar medidas de seguridad más robustas.

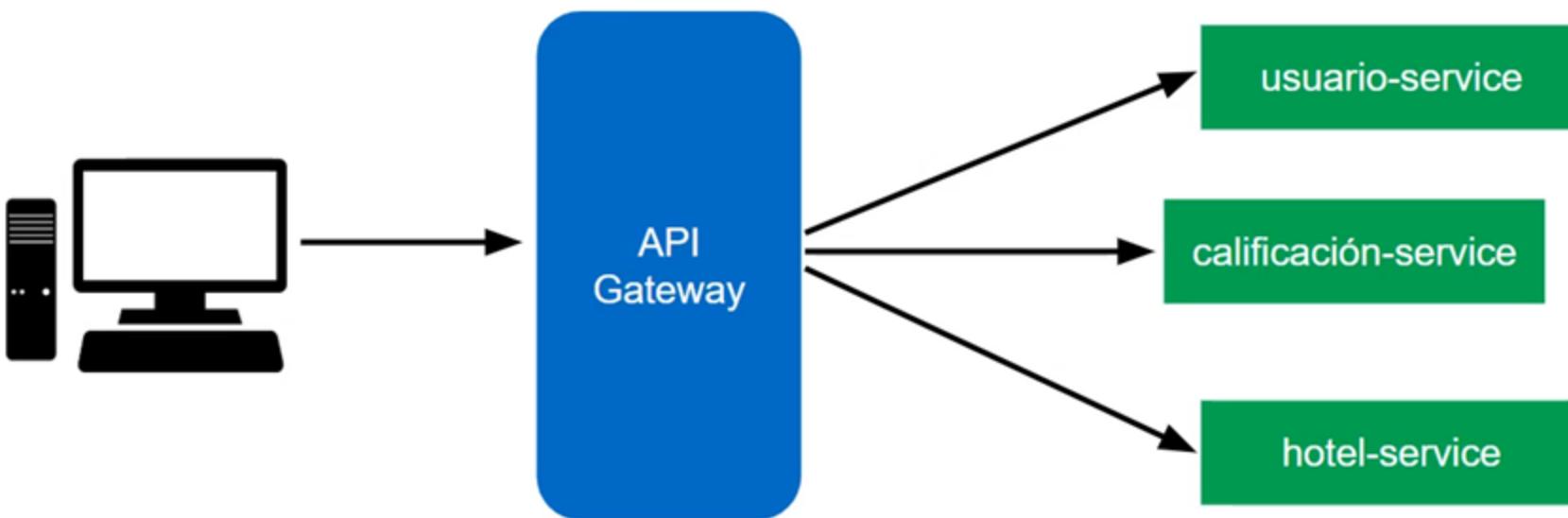
Facilidad de Escalado y Mantenimiento: Simplifica el escalado y mantenimiento del sistema, ya que cualquier cambio de configuración se hace en un solo lugar y se propaga automáticamente a todos los servicios necesarios.

Herramientas como Spring Cloud Config, Consul, etcd y Apache ZooKeeper son ejemplos de Configuration Managers utilizados en arquitecturas de microservicios. Estas herramientas proporcionan una interfaz para almacenar y recuperar configuraciones, manejar versiones, y a veces incluyen características adicionales como cifrado de datos y autenticación para acceder a las configuraciones.

API GATEWAY

¿Qué es API Gateway?

Un **API Gateway** es el gestor de tráfico que interactúa con los datos o el servicio backend real y aplica políticas, autenticación y control de acceso general para las llamadas de una API para proteger datos valiosos. Un API Gateway es la forma en que usted controla el acceso a sus sistemas y servicios de back-end y fue diseñado para optimizar la comunicación entre los clientes externos y sus servicios de back-end .



KAFKA-BUS DE EVENTOS

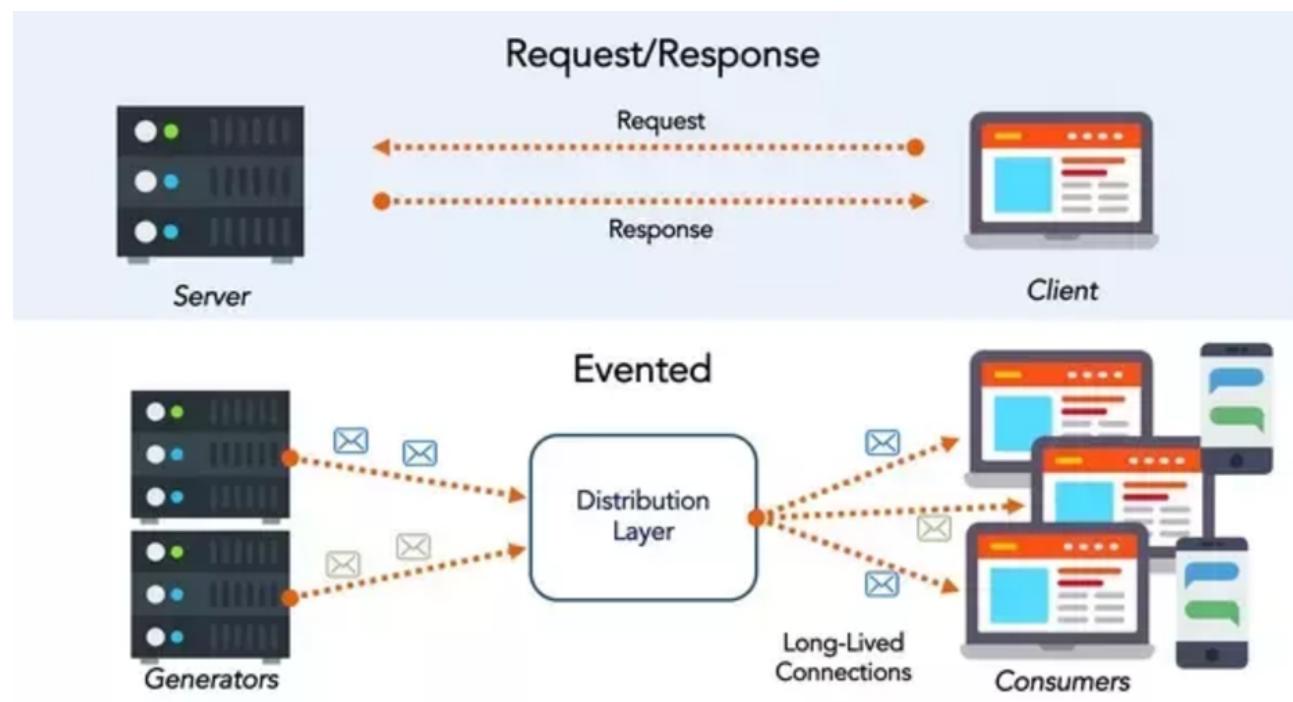
Apache Kafka es una plataforma de procesamiento de streams de datos en tiempo real y de alto rendimiento, utilizada comúnmente para construir sistemas de mensajería, rastreo de actividades, métricas, monitoreo de operaciones y agregación de datos. Fue desarrollada originalmente por LinkedIn y luego se convirtió en un proyecto de código abierto gestionado por la Apache Software Foundation.

En un ambiente de microservicios, Kafka actúa como un sistema de mensajería que permite a los diferentes servicios comunicarse entre sí de manera eficiente y escalable. Los microservicios producen y consumen mensajes que se envían a través de Kafka, lo que facilita la desacoplación de los procesos y mejora la escalabilidad del sistema en general.

Producer. producen datos y los envían a un tópico específico dentro del sistema Kafka. Por ejemplo, un servicio de microservicios que genera eventos de usuario, como clics o transacciones, actuaría como un publicador al enviar estos eventos a Kafka.

Consumer. es el proceso inverso al del publicador. Los consumidores leen los mensajes de Kafka. Pueden suscribirse a uno o varios tópicos y procesar los datos que se envían a esos tópicos. Por ejemplo, un servicio que analiza los eventos de usuario y genera informes a partir de estos datos sería un consumidor, ya que lee la información del tópico correspondiente en Kafka.

Topic. en Kafka es una categoría o canal de mensajes. Puedes pensar en un tópico como una cola de mensajes a la que los publicadores envían datos y de la cual los consumidores leen. Los tópicos en Kafka están particionados y distribuidos a través de diferentes nodos en el clúster de Kafka, lo que permite una alta disponibilidad y escalabilidad. Los tópicos se utilizan para separar diferentes tipos de mensajes; por ejemplo, podrías tener un tópico para eventos de clics de usuario y otro para transacciones de compra.



TEST UNITARIO

Un test unitario es un tipo de prueba en desarrollo de software que se enfoca en verificar la correctitud de una parte específica del código, generalmente a nivel de funciones o métodos. El objetivo es asegurarse de que cada unidad funcional del software funcione como se espera. Esto es crucial para detectar errores en etapas tempranas del desarrollo, facilitando su corrección y mejorando la calidad del código.

Mockito y JUnit 5 son herramientas populares en el ecosistema de Java para realizar pruebas unitarias.

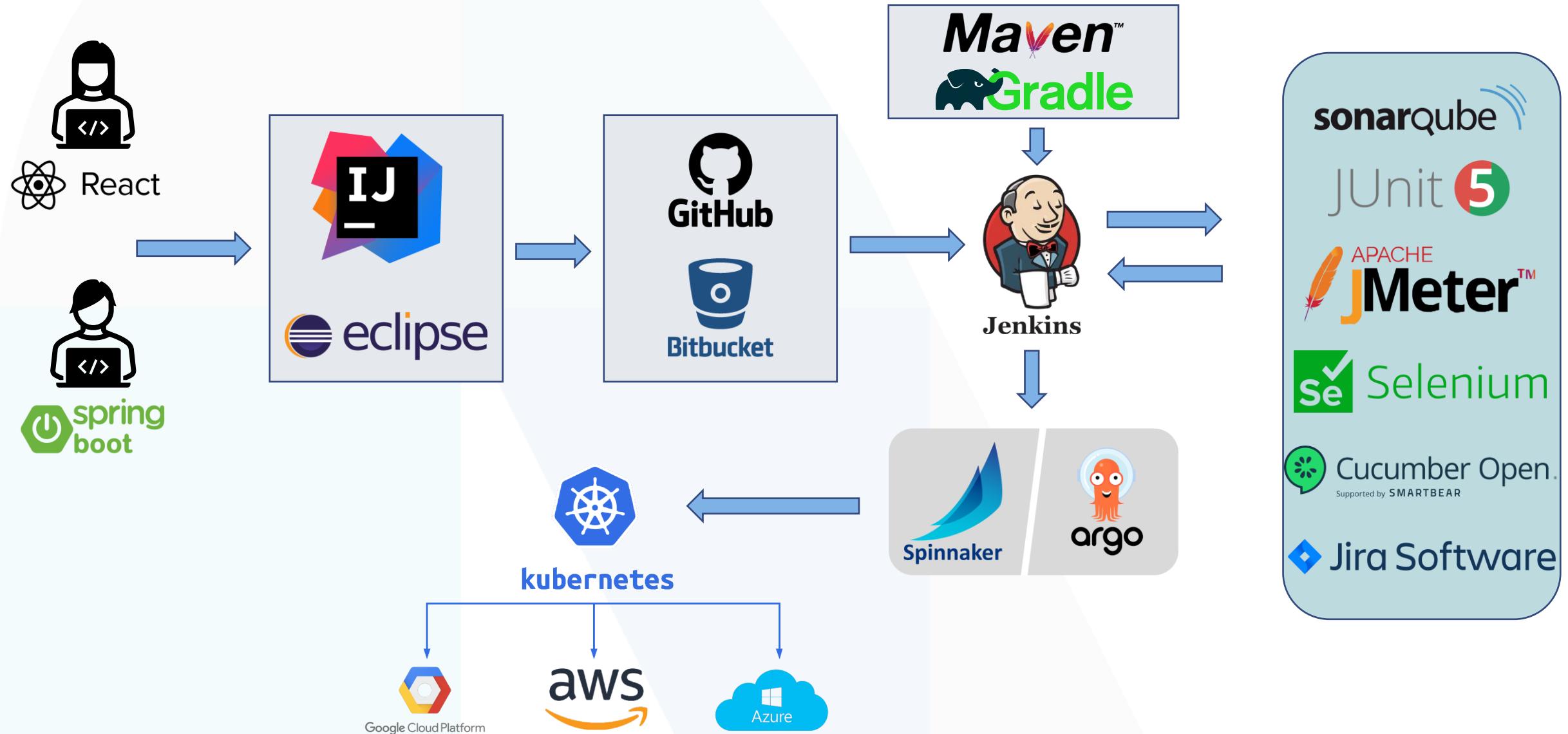
Mockito: Es un framework de pruebas para Java que permite la creación de objetos simulados (mocks). Estos objetos simulados pueden imitar el comportamiento de componentes reales del sistema, permitiendo al desarrollador controlar las respuestas de estos componentes en diferentes escenarios de prueba. Esto es especialmente útil cuando se quieren probar unidades de código que dependen de otros componentes externos o que tienen efectos secundarios complicados de replicar en un entorno de prueba.

JUnit 5: Es la quinta versión del framework JUnit, una de las herramientas más utilizadas para la escritura de pruebas unitarias en Java. Proporciona anotaciones y aserciones para definir pruebas y verificar los resultados esperados. JUnit 5 incluye varias mejoras respecto a sus versiones anteriores, como un modelo de extensión más potente, soporte para ejecución de pruebas en paralelo, y mejor integración con otras herramientas y frameworks.

Ambas herramientas, Mockito y JUnit, se complementan bien al escribir pruebas unitarias: mientras JUnit se utiliza para estructurar las pruebas y verificar los resultados, Mockito ayuda a simular y controlar las dependencias externas de las unidades de código que se están probando.



Diagrama despliegue aplicación



PROYECTO DEL CURSO

