Sistemas distribuidos

IT BOARDING

BOOTCAMP

Índice

01 Qué es un microservicio

02 Monolito vs. Microservicios

03 Patrones de Comunicación entre Microservicios

04 Microservicios en Meli antes y después

IT BOARDING

BOOTCAMP

SISTEMAS DISTRIBUIDOS

// Microservicios

IT BOARDING

BOOTCAMP

Qué son los microservicios

Son un enfoque arquitectónico y organizativo para el desarrollo, donde el software está compuesto por **pequeños servicios independientes** que se **comunican** a través de **API** bien definidas.

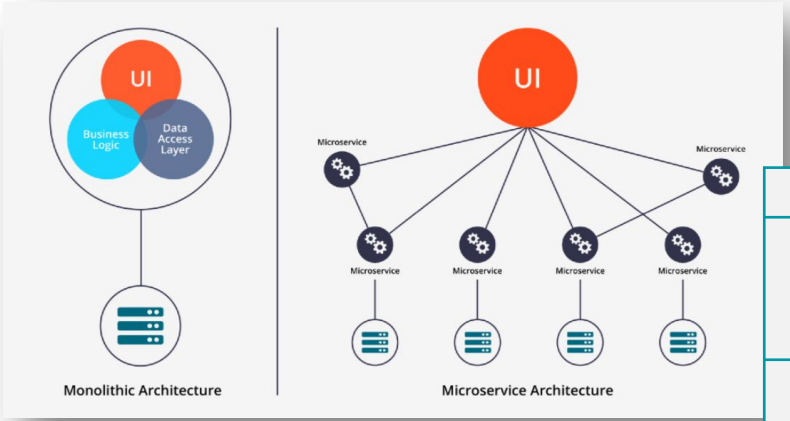
Características:

Autónomos y desacoplados: Se ejecutan de forma independiente, cada servicio se puede actualizar, implementar y escalar para satisfacer la demanda de funciones específicas de una aplicación.

Especializados: Cada servicio está diseñado para un conjunto de capacidades de negocio y se enfoca en resolver un problema específico.

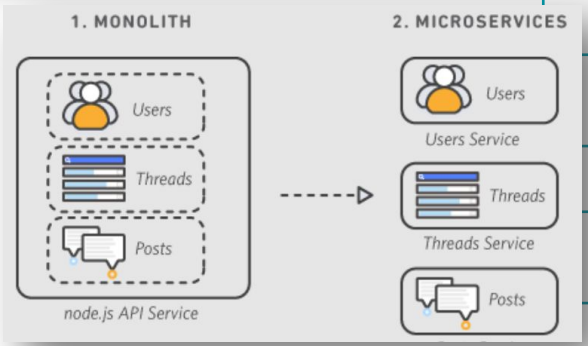
Utilizan un Single Source of Truth (SSOT) distribuido: Cada microservicio tiene su Base de Datos propia que no se comparte con otros.

Arquitectura Monolítica vs. Microservicios



Para definir qué es un **microservicio** es útil **compararlo** con la arquitectura **monolítica**.

	Monolito	Microservicios
Arquitectura	Diseñado como un solo ejecutable, del lado del servidor suele tener una arquitectura de 3 capas cliente-servidor-base de datos	Diseñado como un conjunto de pequeños servicios cada uno corriendo independientemente y comunicándose de forma ágil.
Modularidad	Basada en características de la tecnología.	Basada en las capacidades del negocio.
Agilidad	Los cambios implican buildear y deployar una nueva versión de toda la aplicación.	Los cambios pueden ser aplicados a cada servicio de forma independiente.
Escalamiento	La aplicación se escala horizontalmente atrás de un loadbalancer.	Cada servicio es escalado independientemente según sea necesario.
Implementación	Generalmente escritos en un mismo lenguaje.	Escritos en el lenguaje que mejor se adapte a las necesidades.
Mantenibilidad	Base de código muy grande, intimidante para nuevos desarrolladores.	Base de código más pequeña, cada equipo tiene ownership de algunos microservicios.



Ventajas y desventajas

- **Agilidad**: fomentan una organización de equipos pequeños e independientes con ownership de un microservicio. Esto acorta los tiempos del ciclo de desarrollo.
- **Escalabilidad**: cada servicio escala de forma independiente para satisfacer la demanda.
- **Implementación sencilla**: permiten la integración y la entrega continuas, lo que facilita probar nuevas ideas y mejorar las existentes sin alto riesgo.
- **Versatilidad:** es posible elegir la mejor herramienta para resolver problemas específicos.
- **Resistencia**: al tratarse de servicios independientes el error en uno no afecta al todo. En una arquitectura monolítica, un error en un solo componente, puede provocar un error en toda la aplicación.
- **Mantenimiento**: se pueden hacer cambios en un módulo a la vez sin afectar el resto.

- **Mayor consumo de memoria**: cada microservicio tiene sus propios recursos y bases de datos, consumen más memoria y CPU.
- **Inversión de tiempo inicial**: a veces puede llevar más tiempo diseñarlos.
- **Complejidad en la gestión**: si hay gran cantidad, será más complicado controlar la gestión e integración.
- **Dificultad en la realización de pruebas**: como los componentes de la aplicación están distribuidos, las pruebas y test globales son más complicados de realizar.

SISTEMAS DISTRIBUIDOS

// Patrones de comunicación entre Microservicios

IT BOARDING

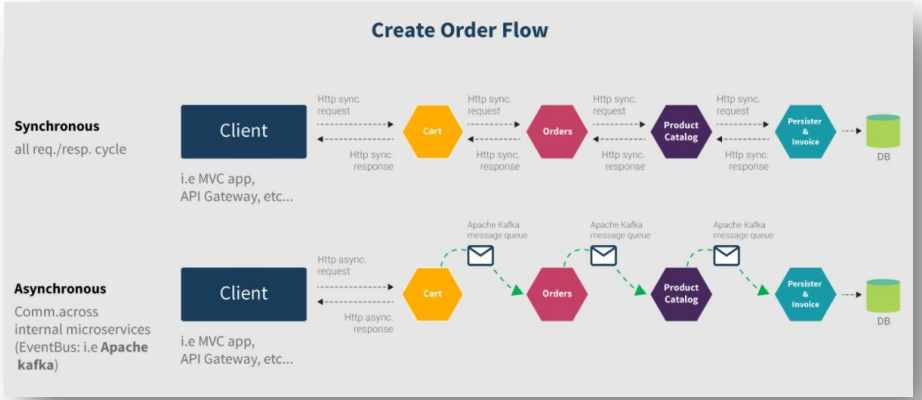
BOOTCAMP

Comunicación entre microservicios

En una arquitectura **monolítica** los componentes se comunican entre ellos mediante llamadas a nivel del lenguaje.

Con la arquitectura de **microservicios**, los servicios tienen que comunicarse mediante un **protocolo** de **comunicación** por ejemplo **HTTP o AMQP**.

No existe un única solución de comunicación entre microservicios. Dependiendo el protocolo puede ser **síncrona** o **asíncrona**.



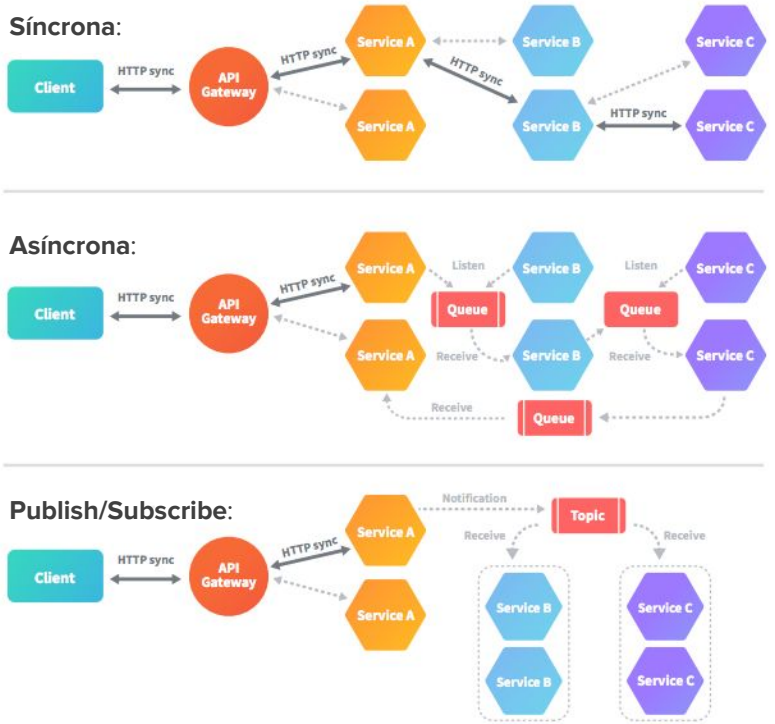
Diferencia entre Comunicación Síncrona y Asíncrona

Síncrona:

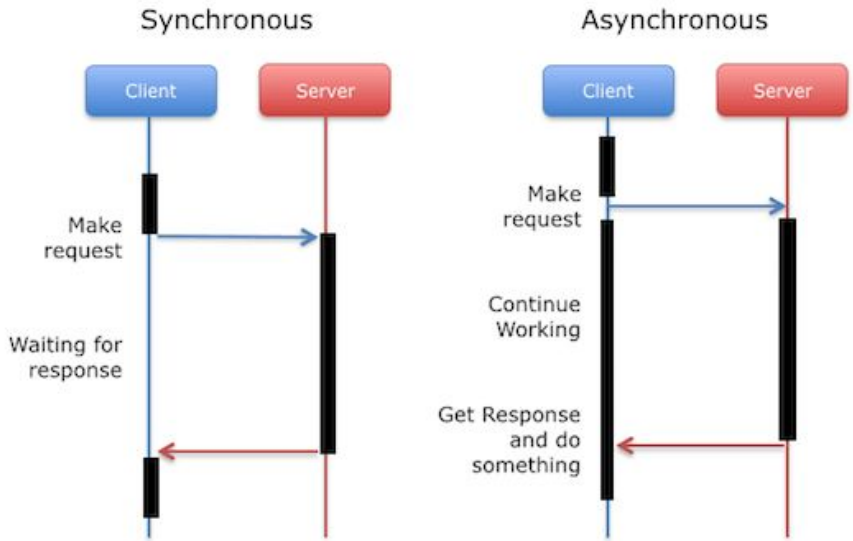
Se requiere una dirección de servicio de origen predefinida, hacia dónde enviar la **Request**, y ambos (remitente y destinatario de la llamada) deben estar en funcionamiento en este momento. El cliente únicamente puede continuar su tarea en el momento que recibe una **Response** del servidor. El enfoque de Request/Response suele utilizar el protocolo HTTP e incluye **REST, GraphQL y gRPC.**

Asíncrona:

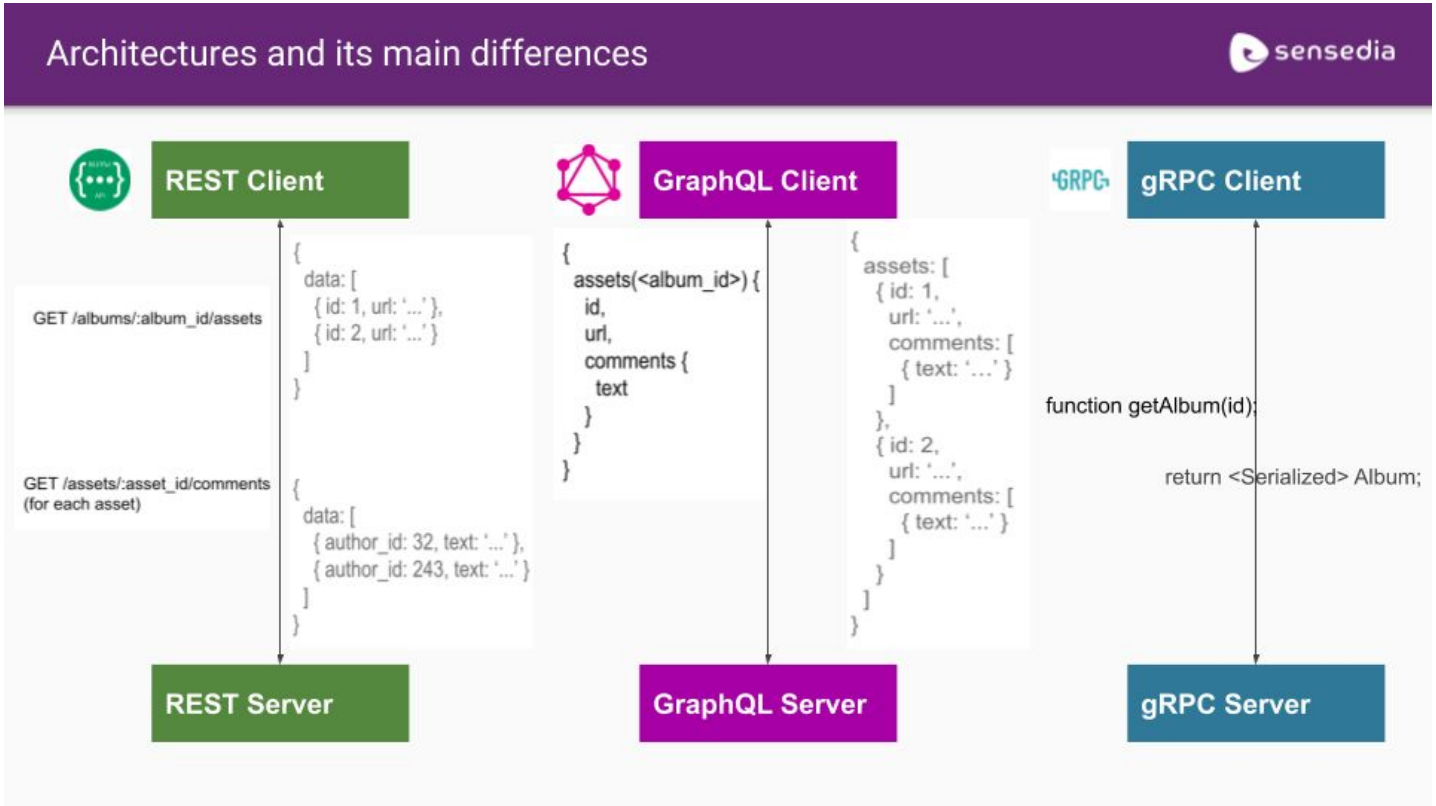
Se envía un mensaje a una **cola** o agente de mensajes, y el mensaje se pone en cola si el servicio de recepción está inactivo y continúa más tarde cuando está activo. El remitente del mensaje no espera ninguna respuesta. Los protocolos asincrónicos como **MQTT, STOMP, AMQP** son manejados por plataformas como **Apache Kafka Stream, RabbitMQ**.



Sync vs Async - Otro layer



Comparación de protocolos



Comparación de protocolos

Messaging protocols comparison

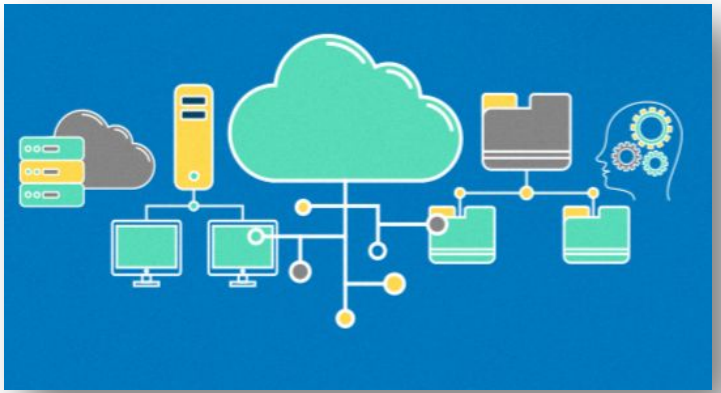
	AMQP	MQTT	XMPP	STOMP
goal	replacement of proprietary protocols	messaging for resource-constrained devices	instant messaging, adopted for wider use	Message-oriented middleware
format	binary	binary	XML-based	text-based
API	divided into classes (> 40 methods in RabbitMQ)	simple (5 basic operations with 2-3 packet types for each)	different XML items with multiple types	~ 10 basic commands
reliability	publisher/subscriber acknowledgements, transactions	acknowledgements	Acknowledgments and resumptions (XEP-198)	Subscriber acknowledgements and transactions
security	SASL, TLS/SSL	no built-in TLS/SSL, header authentication	SASL, TLS/SSL	depending on message broker
extensibility	extension points	none	extensible	depending on message broker

¿Qué elegir?

La comunicación **Rest/HTTP** funciona para patrones de **Request/Response síncronos**, para arquitecturas orientadas a servicios (SOA) y APIs expuestas al público.

Algunas **desventajas** son:

- **Baja performace:** La Request no obtiene una Response hasta que todas las llamadas internas han terminado esto puede resultar en tiempos de respuesta más lentos. También puede bajar si hay muchas llamadas HTTP.
- **Pérdida de autonomía:** Si los microservicios se conectan a través de HTTP y dependen de la respuesta de otro, no pueden ser totalmente autónomos.
- **Manejo de fallas complejo:** Si hay una cadena de llamadas HTTP y un microservicio intermedio falla, toda la cadena falla. Para esto se utilizan los retries y los circuit breakers.



Suele recomendarse para la **comunicación interna entre microservicios**, un patrón **asíncrono** para disminuir la cantidad de llamadas en cadena, e independizarse del ciclo de Request/Response.

Gracias.

IT BOARDING

BOOTCAMP