Arquitectura monolitica vs microservicios

Monoliticos:

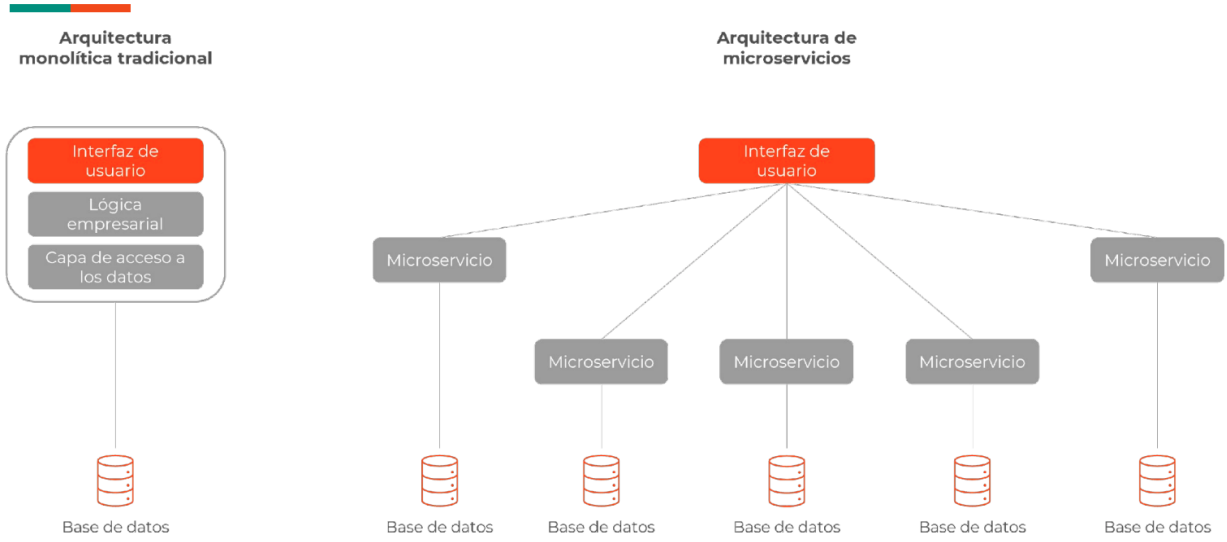
Aplicación desarrollada en único componente. Las principales características son:

- Alto acoplamiento.
- Escalabilidad limitada.
- Despliegue y cambios demorados.

Microservicios:

Aplicación desarrollada en servicios pequeños e independientes. Las principales características son:

- Modularidad y desacoplamiento
- Favorece la escalabilidad
- Mayor complejidad en gestión.



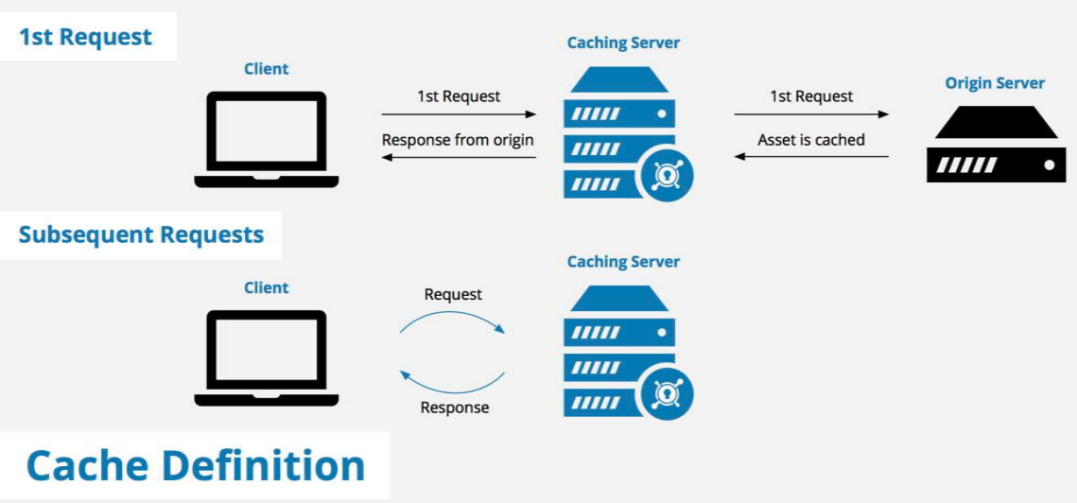
	Microservicios	**Monoliticos**
Acoplamiento	Bajo	Alto
Escalabilidad	Horizontal	Vertical
Mantenibilidad	Mas practico	Mas desafiante
Despliegue	independiente	Unico proceso
Comunicacion	interfaces	Llamadas internas
Complejidad de gestión	Alta	Menor
Tolerancia a fallos	Mayor aislamiento	Puede propagarse
Tiempo de desarrollo	Alto	Bajo

Equipo de desarrollo	Autonomo	Dependiente

Cache - Implementar parte del proyecto con Redis

Que es el cache:

Un área o tipo de memoria de computadora en la que la información que se usa con frecuencia se puede almacenar temporalmente y acceder a ella con especial rapidez.



Tipos de cache:

- Servidor: se utiliza en redes de entrega de contenido o servidores proxy web
- Navegador: Almacenan localmente para que se pueda acceder a ellos más rápido.
- Memoria: La memoria caché almacena ciertas partes de los datos en la RAM estática (SRAM).
- Disco: La caché del disco almacena datos que se han leído recientemente.

Clasificación uso en Angular:

- Caché de archivos estáticos: Uso de service workers, que permiten almacenar en caché los archivos HTML, CSS, JavaScript y otros recursos estáticos necesarios para cargar la página web.
- Caché de datos: LocalStorage o SessionStorage para guardar datos en el lado del cliente. Esto es útil para almacenar información que no cambia con frecuencia y que puede ser reutilizada en diferentes partes.
- Cache de solicitudes HTTP: Angular proporciona un módulo llamado HttpClient que se utiliza para realizar solicitudes HTTP a servidores.
- Cache de vistas: Angular utiliza un enfoque basado en componentes . Podes implementar técnicas de cache de vistas para evitar renderizar y cargar componentes.

Ventajas	Desventajas
Rendimiento mejorado (reduce la cantidad de tiempo que se tarda en recuperar datos)	Posible contaminación (si se llena el caché de datos inútiles puede pasar que al desbordar se pierdan datos)
Uso de ancho de banda reducido (se reduce la	Posibles datos incoherentes (datos incoherentes

cantidad de datos que deben transferirse entre el servidor y el cliente)	entre diferentes servidores o clientes)
Carga del servidor reducida (los servidores no tienen que recuperar la información de su fuente original con tanta frecuencia)	Posibles datos obsoletos (los datos del caché pueden volverse desactualizados si cambian con frecuencia y no tienen asignado un tiempo de expiración)

Políticas de reemplazo de caché: T = m * Tm + Th + E

- m: Tasa de fallos
- Tm : tiempo de cuando hay error
- Th: Latencia (tiempo para hacer referencia a la caché, debe ser el mismo para los aciertos y errores)
- E: extras (varios efectos secundarios, como efectos de cola en sistemas multiprocesador)

Politicas:

- Algoritmo de Belady: El algoritmo de almacenamiento en caché más eficiente sería descartar siempre la información que no será necesaria durante más tiempo en el futuro.
- FIFO: Primero en entrar, primero en salir
- LIFO o FILO: Last in first out, first in last out.
- LRU: Usado menos recientemente
- MRU: Usado mas recientemente

Redis: Remote Dictionary Server
Clave:valor, en memoria, open source.
¿Por qué usar Redis? Rápido, sencillo, escalable, bien documentado.
Como comunicarse con Redis? SET GET INCR HSET LPOP DEL DECR HGET RPUSH
Como implementar Redis? Utilizando Docker para crear los nodos del cluster, Redis Commander, Conectar con Redis

Redis es una muy buena alternativa para el uso de caché a nivel de servidor. Tiene las ventajas de que aceleran mucho las búsquedas pero para estructuras de memoria sencillas y limitadas a clave valor.
El cache es algo interno de cierta forma al programar apps web a alto nivel ya que muchos frameworks poseen ya integrado el manejo del cache.

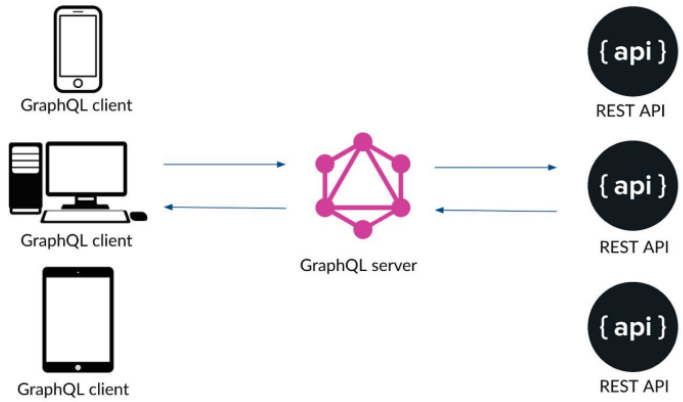
GraphQL/gRPC

GraphQL:
Solucionar los problemas de:

- (-) Under Fetching: Se necesitan muchas llamadas a la API para poder obtener los datos necesarios
- (+) Over Fetching: Se obtienen muchos datos que no son necesarios.
- GraphQL utiliza un solo endpoint y sintaxis de consulta para solicitar y recibir datos
- Permite realizar cambios sin afectar a los clientes existentes.

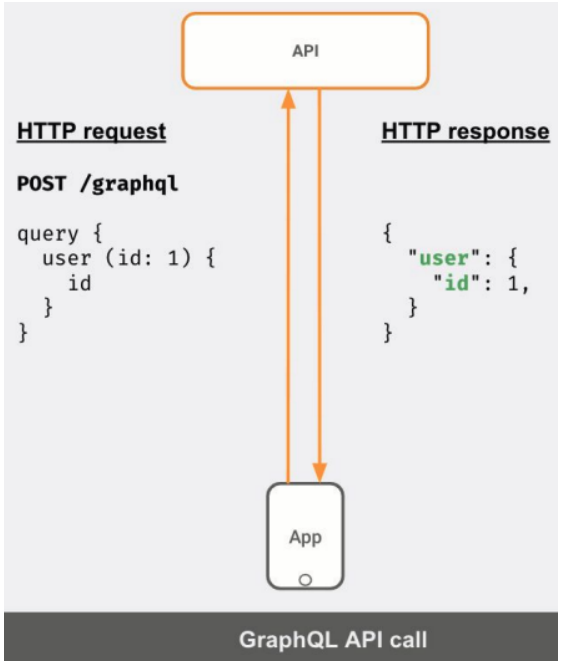
- GraphQL resulta más eficiente en escenarios complejos. REST cuando son específicos y estáticos.

Puede integrarse con API REST



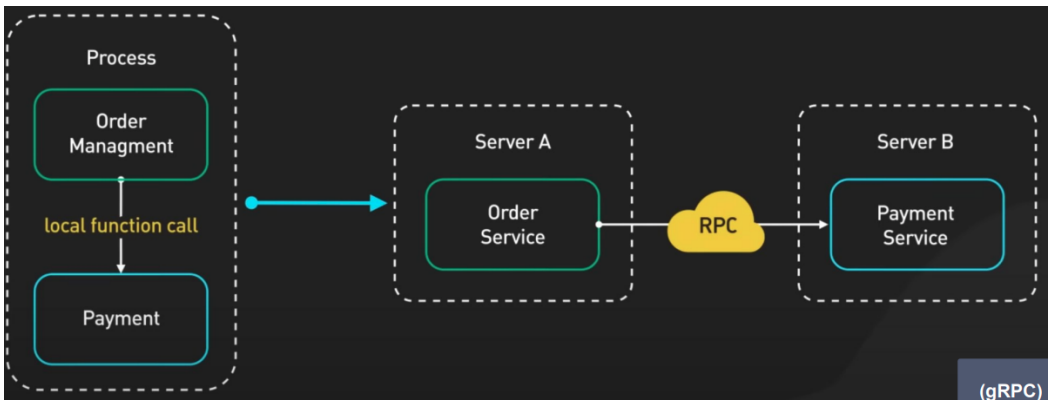
Flujo:

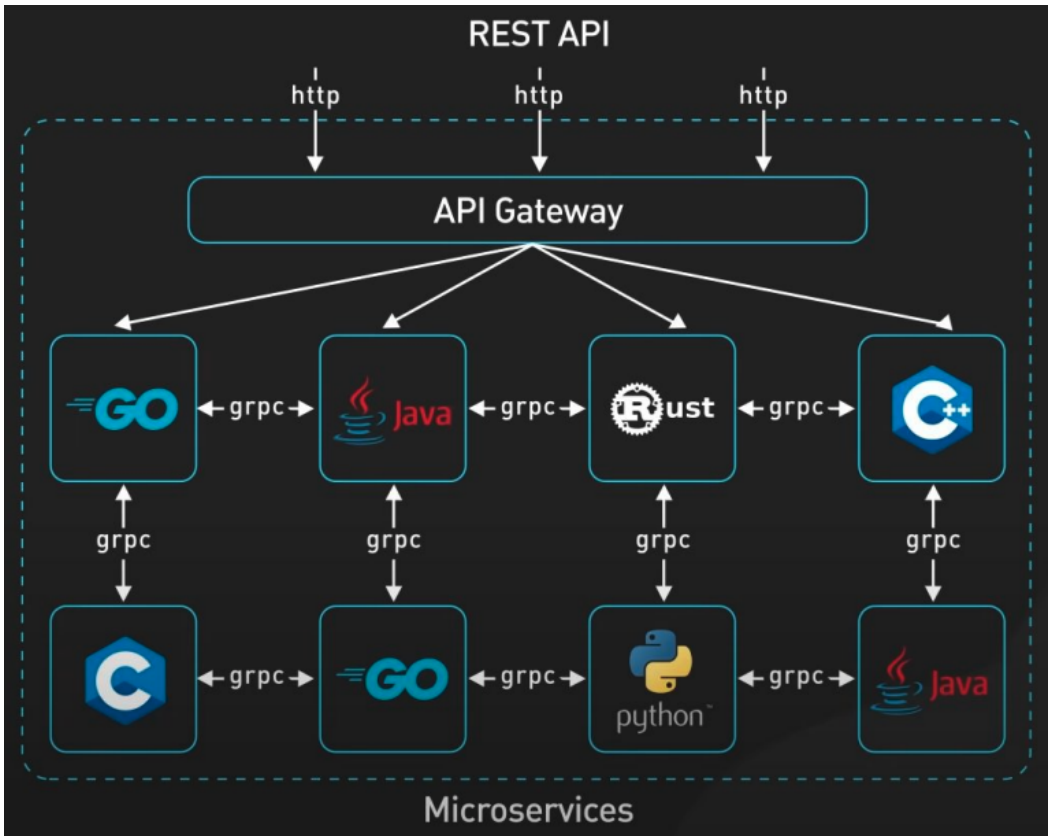
1. El cliente realiza la consulta al servidor con un JSON (al igual que API REST) indicando los datos que solicita.
2. El servidor obtiene el objeto JSON y extrae la cadena de consulta. Según la sintaxis GraphQL y el esquema, el servidor procesa y valida la consulta.
3. El servidor solicita las llamadas a la base de datos y a los servicios necesarios para obtener los datos.
4. Finalmente el servidor devuelve el objeto JSON con los datos solicitados.



gRPC:

Local Procedure Call vs Remote Procedure Call





Protocol buffers:

- Serialización Eficiente
- Definición de interfaces
- Generación de código
- Versionado y evolución
- Compatibilidad con múltiples lenguajes

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

EOA (Arquitectura orientada a eventos) y SOA (Arquitectura Orientada a Servicios)

Arquitectura orientada a eventos

Patrón de diseño de software que se basa en la comunicación asíncrona entre componentes mediante eventos. Los eventos representan sucesos significativos en el sistema y pueden ser generados por diferentes fuentes, como usuarios, sistemas externos o cambios en el estado interno. Es muy poderoso para construir sistemas flexibles, escalables.
Componentes principales:

- Evento: Suceso significativo en el sistema
- Productor de eventos: Genera y emite eventos
- Consumidor de eventos: Recibe y procesa eventos
- Bus de eventos: Canal de comunicación entre los productores y consumidores de eventos.

Flujo de trabajo:

- El productor de eventos genera un evento y lo emite en el bus de eventos
- El bus de eventos distribuye en evento a todos los consumidores interesados
- Los consumidores reciben los eventos y ejecutan acciones en respuesta a ellos.
- Los consumidores pueden generar nuevos eventos, que a su vez son emitidos en el bus y procesados por otros consumidores.

Ejemplos:

- Sistemas de mensajería instantánea: Notificación de mensajes nuevos, actualización de la lista de contactos
- Aplicaciones IoT: recepción de datos de sensores, control de dispositivos.
- Sistemas de comercio electrónico: Procesamiento de órdenes, notificación de cambios de estado.
- Integración de sistemas: Sincronización de datos entre aplicaciones.

Arquitectura orientada a servicios
Es un servicio en lugar de componente, tiene interoperabilidad, bajo acoplamiento y alta abstracción.
Servicio:

- Creado por terceros
- Reutilizable
- Persistente
- escalable

Ventajas y desventajas de EDA:

Ventajas	Desventajas
● Mayor capacidad de respuesta en tiempo real: Permite detectar y reaccionar rápidamente a	● Complejidad: Implementar EDA puede resultar más complejo que otros enfoques, ya que

eventos, lo que es beneficioso en aplicaciones que requieren un procesamiento ágil. ● Desacoplamiento: Los componentes en una arquitectura orientada a eventos están menos acoplados, lo que facilita la modificación y la extensión del sistema. ● Escalabilidad: Al utilizar eventos como mecanismo de comunicación, EDA permite escalar el sistema horizontalmente agregando más instancias del componente que procesa los eventos.	requiere la implementación y el manejo adecuado de los componentes de eventos y los mecanismos de comunicación. ● Coordinación: La coordinación de eventos y las garantías de entrega pueden ser desafiantes en sistemas distribuidos y de alta concurrencia. ● Mayor consumo de recursos: El procesamiento de eventos en tiempo real puede requerir más recursos computacionales y de red en comparación con otros enfoques.

Ventajas y desventajas de SOA:

Ventajas	Desventajas
● Reutilización: Los servicios pueden ser compartidos y reutilizados en diferentes aplicaciones y contextos. ● Interoperabilidad: permite la integración de sistemas heterogéneos y la comunicación entre diferentes plataformas tecnológicas. ● Flexibilidad y modularidad: Los servicios pueden ser actualizados o reemplazados sin afectar a otros componentes del sistema.	● Complejidad inicial: El diseño e implementación de una arquitectura SOA puede ser complejo y requerir un esfuerzo adicional. ● Gestión de servicios: La gestión y el monitoreo de los servicios pueden ser un desafío, especialmente en sistemas grandes con muchos servicios.

Combinar SOA y EDA:
Flexibilidad, reactividad, desacoplamiento, escalabilidad

Arquitectura en capas y cliente-servidor

Cliente-servidor:

Un modelo de diseño de software y redes en el que las tareas se distribuyen entre dos tipos de entidades, clientes y servidores. Los clientes se definen como dispositivos o programas que solicitan y envían peticiones y consumen recursos o servicios, y los servidores son los dispositivos o programas que proporcionan y administran dichos servicios.

Servidor: Diseñado para ser confiable, escalable, capaz de manejar múltiples solicitudes de clientes simultáneamente. Pueden proporcionar una variedad de servicios tales como almacenamiento de datos, procesamiento de datos, servicios web, correo electrónico, etc.

Cliente: Son dispositivos como computadoras personales, teléfonos móviles, que se comunican con servidores. Estos clientes pueden tener interfaces de usuario para permitir que los usuarios interactúen con los servidores y vean las respuestas obtenidas.

Pros	Contras
● Escalable añadiendo más servidores para los clientes o más clientes para un servidor. ● Procesamiento y gestión de recursos centralizado, facilita control y administración de los mismos. ● Es posible reutilizar un mismo servidor para brindar servicios a varios clientes distintos (App Mobile, App Web).	● Depende de una red para comunicar a los clientes con los servidores. ● Gran cantidad de clientes actuando sobre un mismo servidor puede sobrecargarlo. ● Es costoso adquirir y mantener servidores.

Casos de uso:

- Aplicaciones web: Navegadores web actúan de clientes y solicitan los recursos a un servidor remoto.
- Aplicaciones Mobile: Similar al caso anterior, pero con dispositivos mobile actuando como clientes, es posible compartir el servidor de este caso con el de una app web.
- Database and UI: Es útil cuando es necesario tener una interfaz de usuario que se comunique con una base de datos, donde el usuario realiza operaciones sobre la UI, que actúa como cliente, y manda las requests a la data-base.

Arquitectura en capas

Se divide la lógica del programa en capas separadas lógicamente en niveles diferenciados por sus responsabilidades que se comunican por interfaces comunes. SRP, OCP, DIP. Cada capa interactúa por medio de interfaces bien definidas con las capas inferiores, y cada nivel refiere a la implementación física de las capas. Las capas deben comunicarse de alguna manera estandarizada.

Modelo n-capas: Se determinan la cantidad de capas necesarias, en general 3. Presentación, lógica de negocio, acceso de datos. Otras pueden ser caches, balanceadores de carga.
Por apertura:

- Capa caja negra: no se conoce ni accede nada más allá de las interfaces.
- Capa caja blanca: más permisiva de acceso a su interior.

Por interacción:

- Capa cerrada: Solo se puede acceder a la capa inferior.
- Capa abierta: Se pueden acceder a capas inferiores salteando intermedias

Casos de uso: Es una arquitectura sumamente versátil y abstracta, por lo que al menos que el proyecto no amerita la complejidad necesaria para implementarla, es útil.

- Servicios web: Cada servicio se puede abstraer como parte o su propia capa individual, y ser sustituidas fácilmente por implementaciones nuevas o diferentes.
- Aplicaciones complejas: Permite descomponer el desarrollo en áreas definidas una vez son acordadas las interfaces necesarias.

Pros	Contras
● Modular: Cada capa es individual y cohesiva, y se pueden trabajar de manera independiente y ser fácilmente sustituidas por diferentes implementaciones, que son altamente reutilizables. ● Escalable: Se pueden agregar y sacar capas en medida de necesidad.	● Consumo de Recursos: Al necesitar algún medio de comunicación entre las capas, se requieren más recursos para soportarlos. ● Complejidad: La necesidad de interfaces y comunicación ordenada y correcta complejiza el sistema, y puede aumentar el tiempo de desarrollo. ● Fallas en Cadena: En caso de una falla de una de las capas inferiores, puede causar el colapso de todas las superiores.

Combinacion:
Consiste en dividir un servidor en varias capas con funciones únicas definidas, de esta manera el servidor proporciona las ventajas de la modularidad conseguidas gracias a la arquitectura en capas y la centralización obtenida desde el modelo cliente-servidor, obteniendo un sistema que goza de lo mejor de los dos mundos y tiene una excepcional escalabilidad.