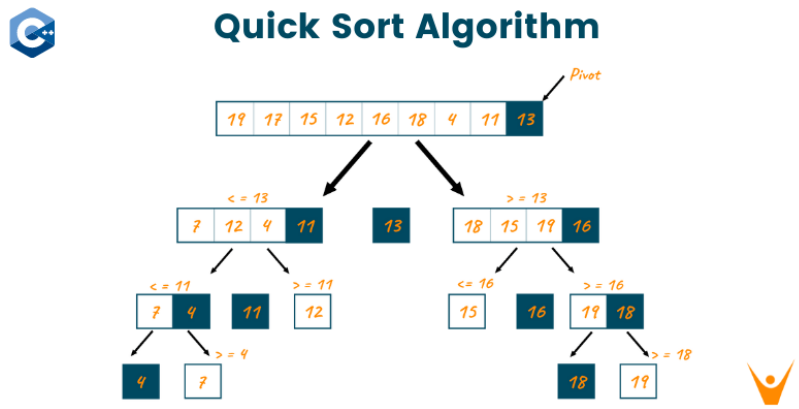
PSC. Conceptos generales

KISS: Keep it simple. Principio que aboga por mantener diseños sencillos y evitar complejidades innecesarias para que las soluciones sean más fáciles de entender y de mantener.

Divide y vencerás: Principio algorítmico que trata de dividir/descomponer un problema complejo en varios subproblemas más sencillos hasta que la solución de estos sea obvia. Los subproblemas más básicos se llaman casos base, una vez resuelto este caso base, se resuelve el subproblema que lo ha originado. De esta manera, se resuelven recursivamente todos los subproblemas hasta que el problema original también se haya resuelto. Ejemplos: algoritmo de búsqueda binaria, Quicksort...



SOLID: Un conjunto de 5 principios de diseño software para escribir código limpio, escalable y mantenible.

- S: Single Responsibility Principle (Responsabilidad única). Una clase debería tener solo una responsabilidad.
- O: Open/Closed Principle (Abierto/Cerrado). Una clase debe estar abierta a la extensión pero no a la modificación.
- L: Liskov Substitution Principle (Sustitución de Liskov). Una clase hija no debería sobrescribir el comportamiento de la clase padre. Un programa debe poder usar una clase padre e hija de manera intercambiable, sin que se altere su correcto funcionamiento.
- I: Interface Segregation Principle (Segregación de Interfaces). Las interfaces deben ser lo más pequeñas que sea posible. Una clase no debe implementar métodos o interfaces que no vaya a usar
- D: Dependency Inversion Principle (Inversión de Dependencias). Las clases de alto nivel no deben depender de clases de bajo nivel, ambas deberían depender de abstracciones. Esto permite separar la lógica de un programa de una implementación concreta.

Convención > configuración: Este principio sostiene que es mejor mantener unas configuraciones predeterminadas en lugar de configuraciones explícitas. De esta manera solo se deben especificar detalles concretos, haciendo el desarrollo más fácil y rápido. Un ejemplo de framework que sigue este principio es Spring Boot.

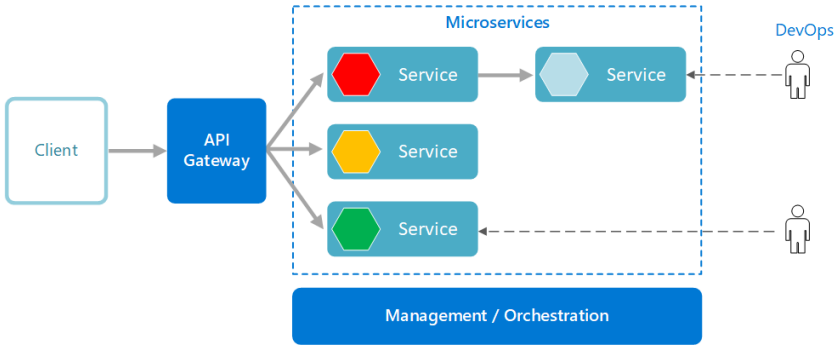
Share Nothing: Una arquitectura en la que cada componente es independiente y no comparte estado ni memoria con otros. Esto mejora la escalabilidad y la tolerancia a fallos.

Actor Model: Un modelo de programación que se basa en actores como unidades básicas. Los actores se comunican mediante mensajes y no comparten estado, siguiendo el principio "Share Nothing".

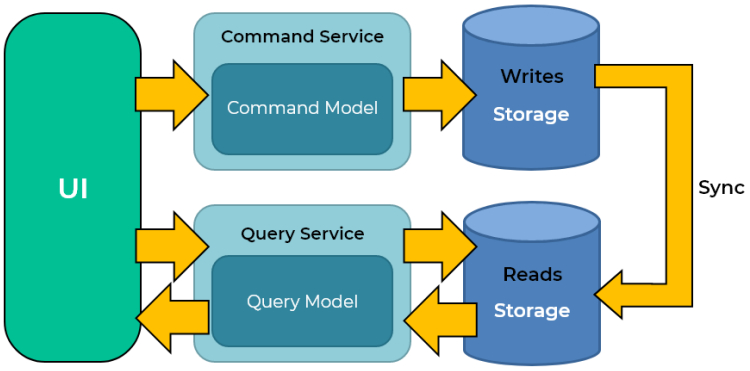
Stateless Server: Un servidor que no guarda información sobre el estado de la sesión de un cliente. Esto facilita la escalabilidad horizontal, ya que cualquier servidor puede manejar cualquier solicitud.

Brokerless: Una arquitectura sin intermediarios (brokers) en las comunicaciones entre servicios, lo que reduce latencia y complejidad.

Microservicios: Arquitectura que divide una aplicación en un conjunto de servicios pequeños e independientes, cada uno con una responsabilidad específica. Facilita el desarrollo, la implementación y el escalado de aplicaciones complejas.



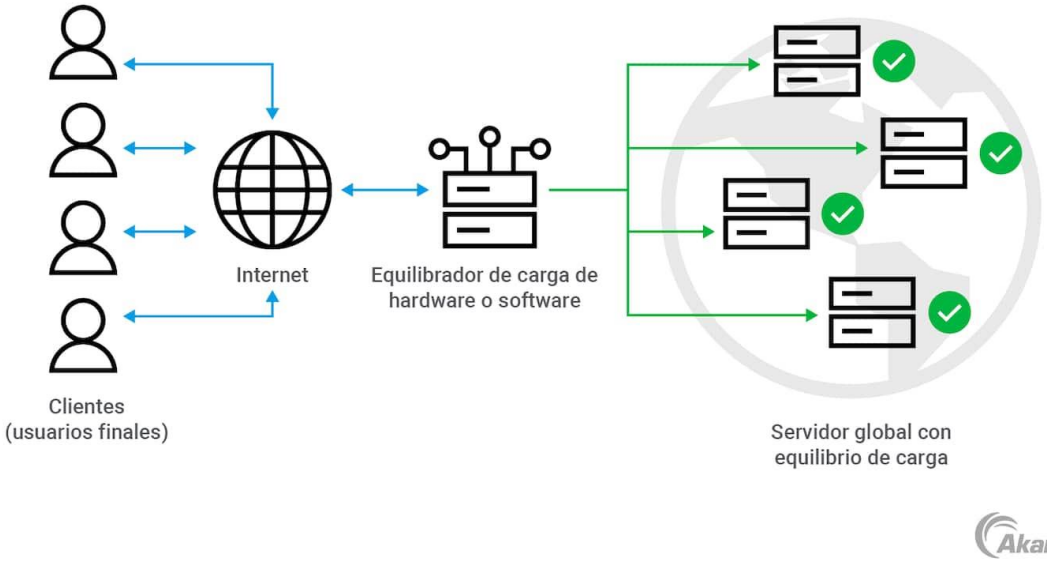
Command Query Responsibility Segregation (CQRS): Patrón de diseño que separa las operaciones de lectura (queries) a base de datos de las operaciones de escritura (commands).



Escalabilidad: La escalabilidad es la capacidad de un sistema de adaptarse a una carga de trabajo creciente proporcionando un rendimiento aceptable. Hay dos tipos de escalabilidad.

- Escalabilidad vertical: consiste en aumentar las prestaciones de la máquina donde se alojan nuestros servicios o aplicación. Es un enfoque simple, aunque limitado, ya que no se pueden escalar las capacidades de una máquina de manera infinita.
- Escalabilidad horizontal: consiste en aumentar el número de instancias de nuestro servicio para así poder distribuir la carga. Tiene mayor potencial, pero se requiere de un diseño arquitectónico más complejo.

Load Balancing: Consiste en distribuir de manera uniforme la carga de un servicio que se haya escalado horizontalmente entre varias instancias. De esta forma se logra evitar la sobrecarga de una de las instancias del servicio y se ofrece rendimiento óptimo de cara al usuario. El balanceo de carga aumenta la tolerancia a fallos de nuestro servicio ya que las instancias caídas del servicio se detectan y excluyen automáticamente del grupo, redirigiendo las peticiones a las otras instancias disponibles.



Tolerancia a fallos: Es la capacidad de un sistema para detectar, manejar y recuperarse de fallos en sus componentes sin interrumpir significativamente su funcionamiento.

- **Idempotencia**: Las operaciones idempotentes son aquellas que pueden repetirse varias veces sin causar efectos colaterales ya que siempre generan el mismo resultado. Esto permite manejar fallos mediante un mecanismo de reintentos.
- **Stateless server**: El hecho de que un servidor no guarda información de sus sesiones con los usuarios hace que cualquiera de sus replicas pueda manejar las solicitudes en caso de que uno de ellos se falle.
- **Let it crash**: Una filosofía que asume que los errores ocurrirán y se enfoca en recuperarse rápidamente en lugar de evitarlos.

Consistencia:

Garantía de que todos los nodos del sistema presenten el mismo estado o información a los usuarios, incluso frente a fallos o concurrencia, es decir, que todos los nodos tienen una visión coherente y sincronizada del estado del sistema.

Tipos de consistencia:

- **Consistencia Fuerte:** Garantiza que, después de una operación de escritura, todas las lecturas devolverán el resultado más reciente.
- **Consistencia Eventual:** Los nodos convergen al mismo estado con el tiempo, pero no garantizan que las lecturas sean consistentes de inmediato.
- **Consistencia Débil**: Permite lecturas inconsistentes y no garantiza que los nodos converjan, excepto bajo condiciones específicas.

CAP theorem: Un principio que establece que un sistema distribuido no puede garantizar simultáneamente consistencia (C), disponibilidad (A) y tolerancia a particiones (P).

"The Log": Concepto utilizado para mantener un registro secuencial de eventos, común en sistemas como Apache Kafka.

Event Sourcing: Un patrón de diseño en el que se almacena cada cambio en el estado como un evento, lo que permite reconstruir estados pasados y auditar el sistema fácilmente.