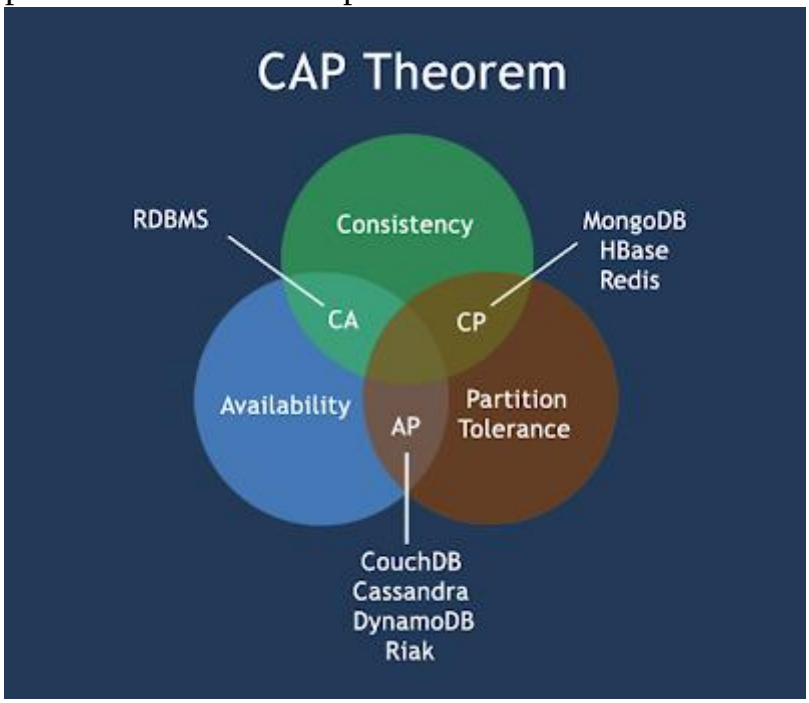
Tarea 0 Apuntes sobre conceptos generales

- **KISS:** "Keep it simple, stupid", cualquier Sistema va a funcionar mejor a la larga si es sencillo en vez de complejo, por eso una meta en el desarrollo ha de ser la sencillez del programa.
- **SOLID:** Son 5 principios que si se siguen se consigue un código mas limpio, mantenible, escalable a futuro y propenso a menos errores. Los 5 principios son
    1. "S" - Principio de Responsabilidad Única: Cada clase o componente debe tener una responsabilidad única y debe ser diseñado de manera tal que cumpla con esa responsabilidad de manera aislada.
    2. "O" - Principio Abierto-Cerrado: Las clases o componentes deben estar diseñadas de manera que puedan extenderse para añadir nuevas funcionalidades, pero sin modificar su código existente.
    3. "L" - Principio de Substitución de Liskov: Las clases derivadas deben poder ser utilizadas en cualquier lugar donde se utilicen las clases base sin provocar errores.
    4. "I" - Principio de Segregación de Interfaces: Las interfaces deben ser lo más específicas posible y no deben incluir métodos que no sean utilizados por los clientes.
    5. "D" - Principio de Inversión de Dependencias: Se basa en la idea de que las partes altamente abstractas de un sistema deben depender de las partes más concretas, en lugar de lo contrario.
- **Divide y vencerás**: Es un algoritmo, basado en la recursividad que trata de resolver un problema dividendo en 2 o mas subproblemas del mismo tipo, hasta que llegue a ser sencillo o trivial para resolverlos directamente y el conjunto de soluciones de todos los subproblemas dan la solución del problema original.
- **Convenio > Configuración:** Decrementar el numero de decisiones que tiene que realizar el programador y eliminar la complejidad de tener que configurar todas las áreas de desarrollo de una app.
- **Share nothing**: Es una arquitectura de sistemas en la que cada componente o nodo de un sistema tiene su propia memoria y almacenamiento y no comparte recursos con otros componentes. Se utiliza a menudo en sistemas distribuidos y en la nube, ya que permite una mayor escalabilidad y flexibilidad. Además, es menos propensa a fallos, ya que, si un componente falla, no afecta a otros componentes del sistema.

- **Modelo de actor:** Cada componente del sistema se conoce como actor y es responsable de realizar una tarea específica. Los actores no comparten datos ni memoria y no tienen acceso a los datos o a la memoria de otros actores. En su lugar, se comunican entre sí a través de mensajes y pueden realizar acciones sobre los mensajes que reciben.
- **Stateless Server:** Es un tipo de servidor que no mantiene el estado o la información de sesión de los clientes que lo utilizan. Esto significa que cada solicitud de un cliente es tratada de manera aislada. Un servidor stateless es más fácil de escalar y mantener que un servidor que mantiene el estado, ya que no hay necesidad de almacenar y gestionar información de sesión para cada cliente. Aun que puede ser menos eficiente a la hora de personalizar la experiencia de un usuario.
- **Brokerless:** Se refiere a un sistema o plataforma que funciona sin la necesidad de un intermediario. Se utilizan para eliminar la necesidad de un intermediario, lo que puede reducir los costos y aumentar la eficiencia.
- **Micro-services**: Un enfoque arquitectónico y organizativo para el desarrollo de software donde el software está compuesto por pequeños servicios independientes que se comunican a través de API bien definidas. El objetivo de los microservicios es permitir una mayor flexibilidad y escalabilidad en el desarrollo y mantenimiento de aplicaciones.
- **Command Query Responsibility Segregation:** Se trata de un patrón de diseño de software que se utiliza para separar las operaciones de escritura (comandos) y lectura (consultas) en aplicaciones. Esto permite una mayor flexibilidad y escalabilidad en el sistema, ya que las operaciones de escritura y lectura pueden manejarse de manera separada y optimizada para cada caso.
- **Escalabilidad:** La capacidad de un sistema para adaptarse y crecer de manera eficiente a medida que aumenta la carga o el uso. En general, se considera que un sistema es "escalable" si puede manejar un aumento en la carga sin disminuir significativamente su rendimiento o disponibilidad. La escalabilidad vertical implica aumentar la capacidad de un sistema existente mediante la adición de más recursos, como memoria o procesador, mientras que la escalabilidad horizontal implica aumentar la capacidad de un sistema mediante la adición de más instancias o "nodos" del sistema. Además, existe el término “scale cube” que se basa en tres dimensiones de escalabilidad: el "espacio", el "tiempo" y el "estado".

- **Load balancing:** Es una técnica utilizada en informática para distribuir de manera equilibrada la carga de trabajo entre varios recursos o servidores. El objetivo del load balancing es asegurar un rendimiento óptimo y una alta disponibilidad en un sistema informático, evitando la sobrecarga de uno o varios recursos y garantizando que todos los recursos estén utilizados de manera eficiente.
- **Tolerancia a fallos:** Se refiere a la capacidad de un sistema o componente de continuar funcionando de manera adecuada incluso cuando ocurre un fallo o un error. Esto es especialmente importante en sistemas críticos o que deben estar disponibles en todo momento.
    - **Idempotencia + repetición**: Se refieren a la capacidad de una operación o proceso de producir el mismo resultado independientemente de cuántas veces se realice.
    - **Stateless server:** Ya definido anteriormente
    - **"Let it crash"**: Es minimizar el tiempo de inactividad de un sistema y maximizar su disponibilidad a través de la recuperación automática de componentes o procesos fallados. En lugar de tratar de evitar el fallo a toda costa, se permite que el componente o proceso fallado se detenga y se recupere de manera automática, lo que permite al sistema recuperarse más rápidamente y evitar la propagación de errores a otros componentes.
- **Consistencia:** se refiere a la propiedad de un sistema o conjunto de datos de mantener un estado válido y coherente en todo momento. La consistencia es especialmente importante en sistemas que deben manejar grandes cantidades de datos o realizar operaciones críticas, ya que una falta de consistencia puede llevar a errores o resultados incorrectos. también puede ser un aspecto importante para considerar al diseñar sistemas distribuidos o en la nube, ya que pueden presentar desafíos adicionales para mantener la coherencia de los datos en distintos nodos o instancias.
    - **"Teorema de CAP":** Es un teorema en informática que describe los límites de la disponibilidad, la consistencia y la tolerancia a particiones en sistemas distribuidos. El teorema establece que es imposible garantizar simultáneamente la consistencia, la disponibilidad y la tolerancia a particiones en un sistema distribuido. se basa en la idea de que, en un sistema distribuido, es imposible garantizar que todos los nodos del

sistema estén siempre en comunicación y sincronizados. Por lo tanto, es imposible garantizar simultáneamente la consistencia, la disponibilidad y la tolerancia a particiones en un sistema distribuido. En lugar de tratar de cumplir con todos los criterios al mismo tiempo, el teorema de CAP sugiere que es necesario elegir dos de los tres criterios y optimizar el sistema para cumplir con ellos.



- "**The Log"**: Se refiere generalmente a un archivo o base de datos que registra eventos o sucesos ocurridos en un sistema. Los logs pueden utilizarse para registrar eventos críticos o importantes, como errores o fallos en el sistema, o para llevar un registro de la actividad del sistema. Los logs son una herramienta útil para la depuración y el diagnóstico de problemas en un sistema informático. Al registrar eventos ocurridos en el sistema, los logs pueden ayudar a identificar la causa de un fallo o error y proporcionar información valiosa para solucionar el problema.
- "**Event sourcing**": Es un patrón de diseño que consiste en almacenar todos los cambios a un sistema como una secuencia de eventos en lugar de almacenar el estado actual del sistema. Este patrón se utiliza a menudo en sistemas que deben manejar una gran cantidad de datos o que requieren un alto grado de flexibilidad y escalabilidad. El sistema puede reconstruir el

estado actual a partir de la secuencia de eventos en lugar de tener que almacenar todo el estado de manera explícita.