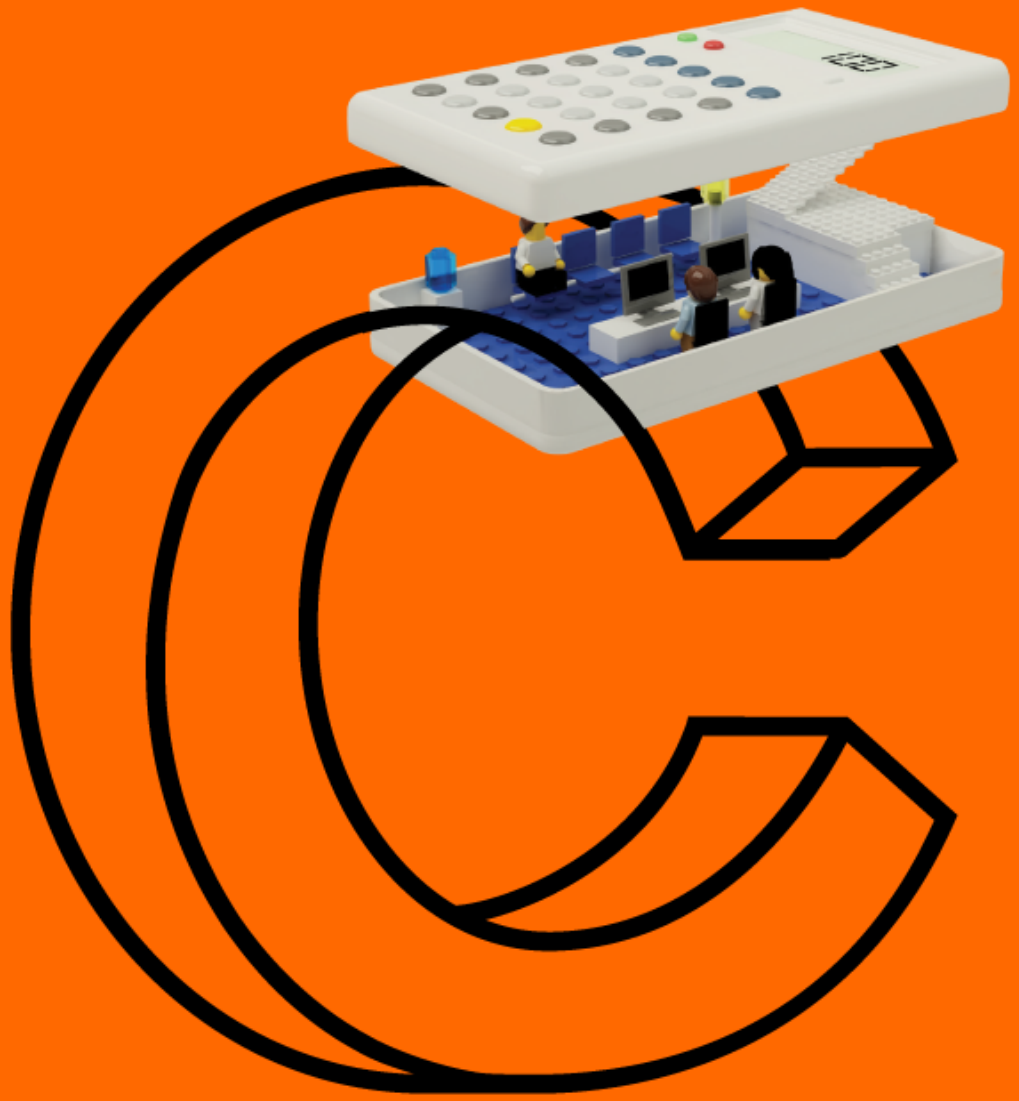
Análisis y Diseño de Sistemas Orientado a Objetos

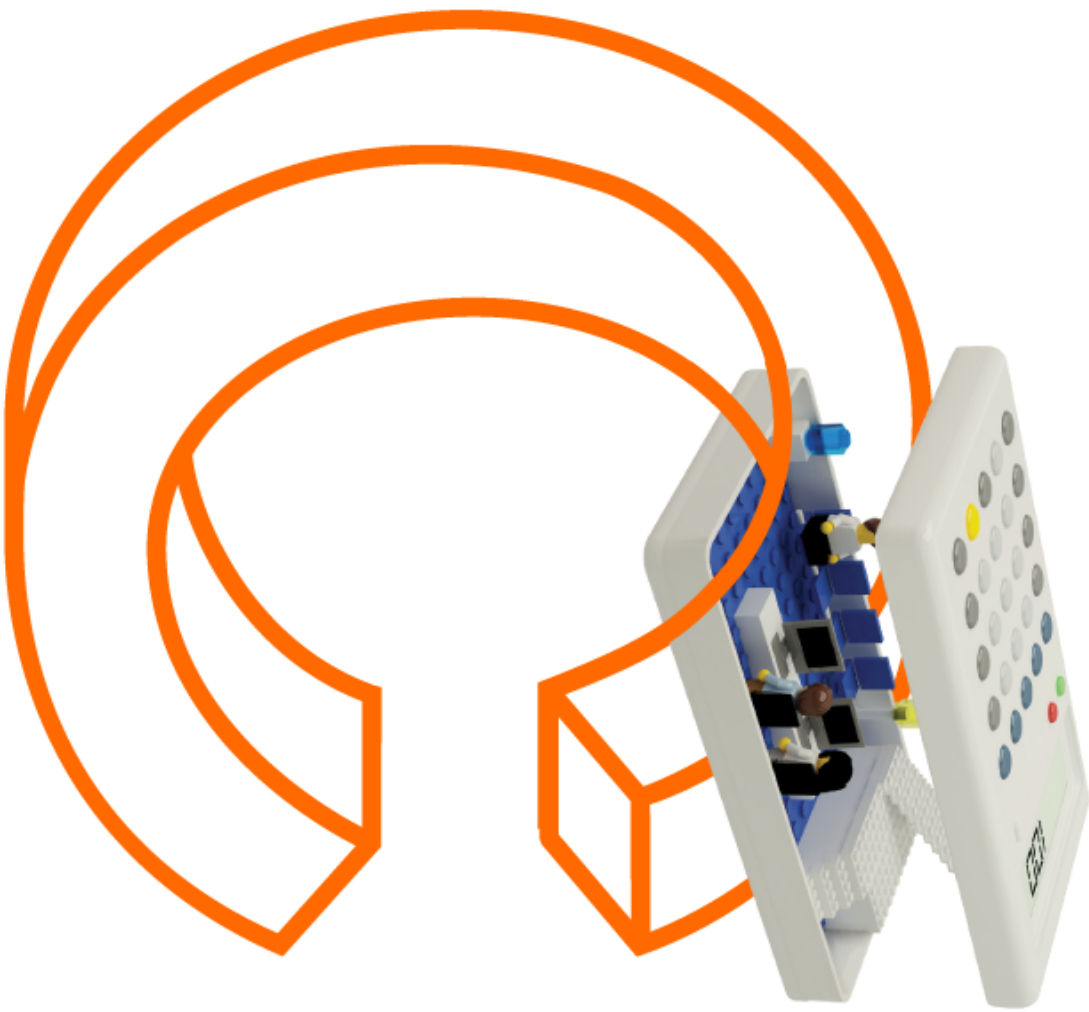


idat

Mg. Daniel Arias, **PMP™** y **Scaled Scrum Master™**

Analizar la Arquitectura como parte de la Disciplina de Análisis & Diseño

Semana 8



idat

Semana 8: Analizar la Arquitectura como parte de la Disciplina de Análisis & Diseño

Logro de aprendizaje

- Artefactos de Entrada
  - Glosario
  - Especificaciones complementarias
  - Modelo de Caso de Uso
- Artefacto de Salida
  - Realización de caso de uso (Diagrama de clases)
- Rol Diseñador

Actividades

Actividad 08:

- Elabora y presenta el Modelo de Diseño y de realización de caso de uso.

Aprendemos:

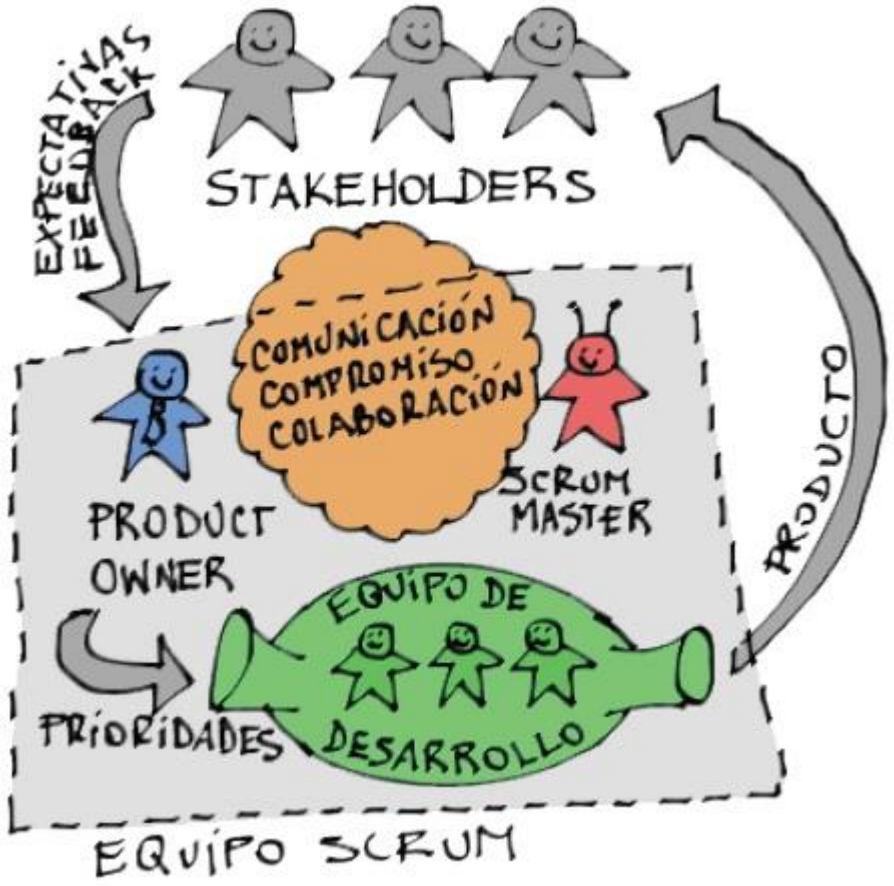
¿Qué aprendimos la clase pasada?

- Resolvemos la Segunda Evaluación Continua

Personas y roles en SCRUM

Un Equipo Scrum se compone de tres roles:

- Product Owner,
- Equipo de Desarrollo y
- Scrum Master.



http://www.menti.com

Aprendemos: Arquitectura de Software

¿Qué es?

El diseño arquitectónico representa la estructura de los datos y de los componentes del programa que se requieren para construir un sistema basado en computadora. Considera el estilo de arquitectura que adoptará el sistema, la estructura y las propiedades de los componentes que lo constituyen y las interrelaciones que ocurren entre sus componentes arquitectónicos.

¿Quién lo hace?

Aunque es un ingeniero de software quien puede diseñar tanto los datos como la arquitectura, es frecuente que, si deben construirse sistemas grandes y complejos, el trabajo lo realicen especialistas. El diseñador de una base de datos o data warehouse crea la arquitectura de los datos para un sistema. El “arquitecto del sistema” selecciona un estilo arquitectónico apropiado a partir de los requerimientos obtenidos durante el análisis de los datos.

Aprendemos: Arquitectura de Software

¿Por qué es importante?

¿Intentarías construir una casa sin un plano?, ¿o sí? Tampoco comenzarías los planos con el dibujo de la plomería del lugar. Antes de preocuparte por los detalles, necesitarías tener el panorama general: la casa en sí. Eso es lo que hace el diseño arquitectónico, da el panorama y asegura que sea el correcto.
Ocurre lo mismo con el Desarrollo de Software

¿Cuáles son los pasos?

El diseño de la arquitectura comienza con el diseño de los datos y continúa con la obtención de una o más representaciones de la estructura arquitectónica del sistema. Se analizan alternativas de estilos o patrones arquitectónicos para llegar a la estructura más adecuada para los requerimientos del usuario y para los atributos de calidad. Una vez seleccionada la alternativa, se elabora la arquitectura con el empleo de un método de diseño.

Aprendemos: Arquitectura de Software

¿Cuál es el producto final?

Durante el diseño arquitectónico se crea un modelo de arquitectura que incluye la arquitectura de los datos y la estructura del programa.
Además, se describen las propiedades y relaciones (interacciones) que hay entre los componentes.

¿Cómo me aseguro de que lo hice bien?

En cada etapa se revisan los productos del trabajo del diseño del software para que sean claros, correctos, completos y consistentes con los requerimientos y entre sí.

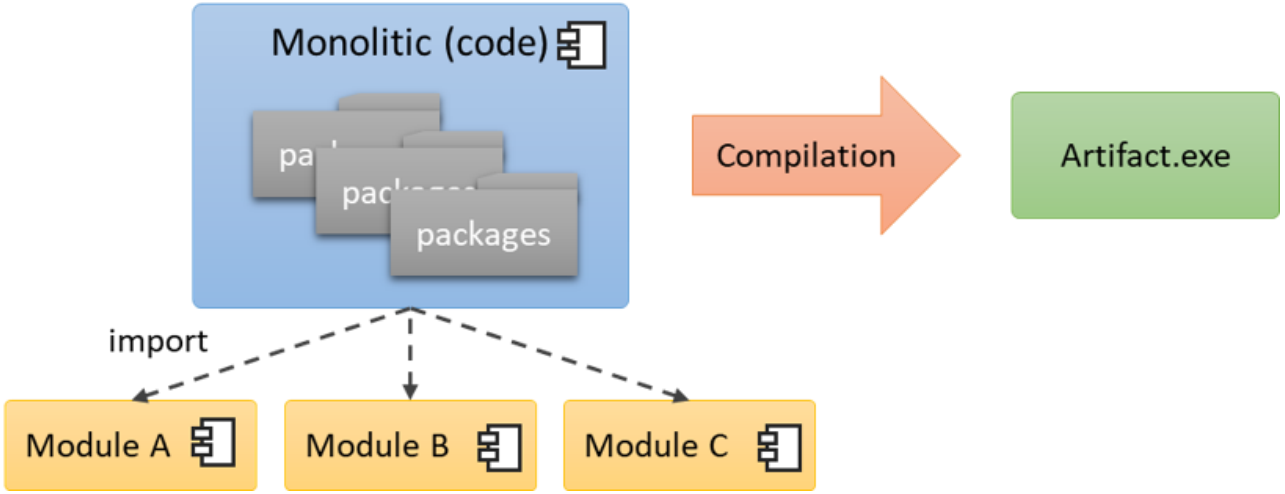
Estilos arquitectónicos

Un estilo arquitectónico establece un marco de referencia a partir del cual es posible construir aplicaciones que comparten un conjunto de atributos y características mediante el cual es posible identificarlos y clasificarlos.

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura Monolítica

El estilo arquitectónico monolítico consiste en crear una aplicación autosuficiente que contenga absolutamente toda la funcionalidad necesaria para realizar la tarea para la cual fue diseñada, sin contar con dependencias externas que complementen su funcionalidad. En este sentido, sus componentes trabajan juntos, compartiendo los mismos recursos y memoria. En pocas palabras, una aplicación monolítica es una unidad cohesiva de código.



Tip: Independencia

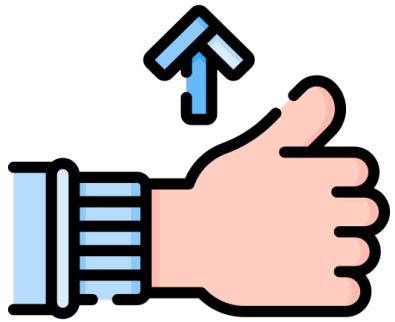
Quizás la palabra que mejor define a un monolítico es la Independencia, pues es la capacidad más notable que tiene.

Arquitectura Monolítica - Características

- Son aplicaciones autosuficientes (no requieren de nada para funcionar).
- Realizan de punta a punta todas las operaciones para terminar una tarea.
- Son por lo general aplicaciones grandes.
- Son por lo general silos de datos privados, es decir, cada instalación administra si propia base de datos.
- Todo el sistema corre sobre una solo plataforma.

Arquitectura Monolítica

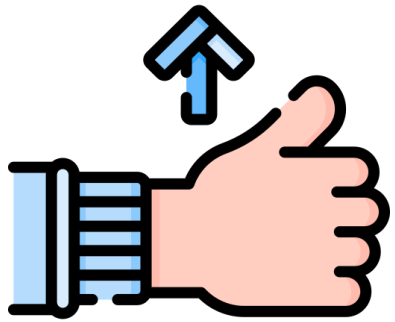
Ventajas	• **Fácil de desarrollar:** Debido a que solo existe un componente, es muy fácil para un equipo pequeño de desarrollo iniciar un nuevo proyecto y ponerlo en producción rápidamente. • **Fácil de escalar:** Solo es necesario instalar la aplicación en varios servidores y ponerlo detrás de un balanceador de cargar. • **Pocos puntos de fallo:** El hecho de no depender de nadie más, mitiga gran parte de los errores de comunicación, red, integraciones, etc. Prácticamente los errores que pueden salir son por algún bug del programador, pero no por factores ajenos.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

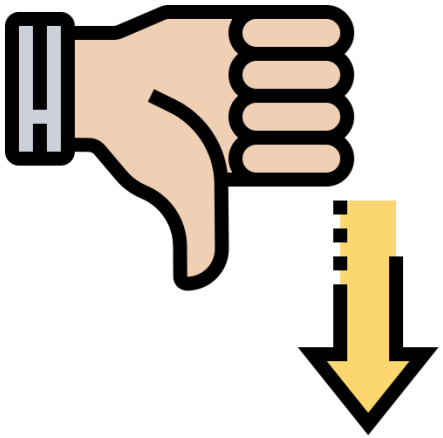
Arquitectura Monolítica

Ventajas	• **Autónomo:** Las aplicaciones Monolíticas se caracterizan por ser totalmente autónomas (auto suficientes), lo que les permite funcionar independientemente del resto de aplicaciones. • **Performance:** Las aplicaciones Monolíticas son significativamente más rápidas debido que todo el procesamiento lo realizan localmente y no requieren consumir procesos distribuidos para completar una tarea. • **Fácil de probar:** Debido a que es una sola unidad de código, toda la funcionalidad está disponible desde el inicio de la aplicación, por lo que es posible realizar todas las pruebas necesarias sin depender de nada más.



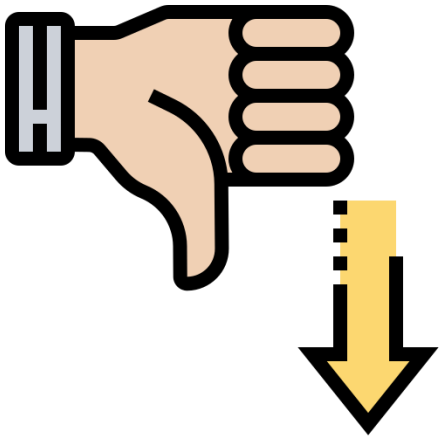
Arquitectura Monolítica

Desventajas	• **Anclado a un Stack tecnológico:** Debido a que todo el software es una sola pieza, implica que utilicemos el mismo Stack tecnológico para absolutamente todo. • **Escalado Monolítico:** Escalar una aplicación Monolítica implica escalar absolutamente toda la aplicación, gastando recursos para funcionalidad que quizás no necesita ser escalada. • **El tamaño sí importa:** sin albur, las aplicaciones Monolíticas son fácil de operar con equipo pequeños, pero a medida que la aplicación crece y con ello el equipo de desarrollo, se vuelve cada vez más complicado dividir el trabajo sin afectar funcionalidad.



Arquitectura Monolítica

Desventajas	• **Versión tras versión:** Cualquier cambio en la aplicación implicará realizar una compilación del todo el artefacto y una nueva versión que tendrá que ser administrada. • **Si falla, falla todo:** A menos que tengamos alta disponibilidad, si la aplicación Monolítica falla, falla todo el sistema, quedando totalmente inoperable. • **Es fácil perder el rumbo:** Por la naturaleza de tener todo en un mismo módulo es fácil caer en malas prácticas de programación, separación de responsabilidades y organización de las clases del código. • **Puede ser abrumador:** En proyectos muy grandes, puede ser abrumador para un nuevo programador hacer un cambio en el sistema.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura Monolítica

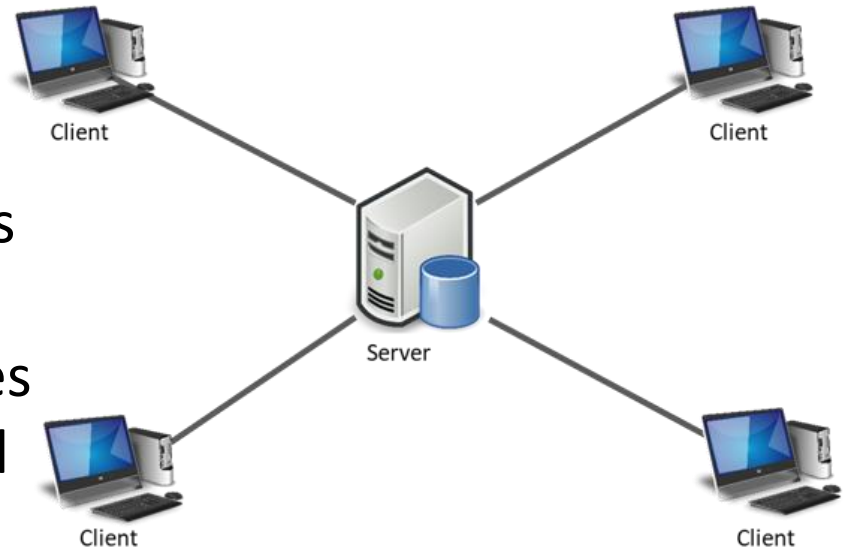
Conclusiones	A pesar de la mala fama que se les ha dado a las aplicaciones Monolíticas, la realidad es que tiene ventajas que hasta el día de hoy son difíciles de igualar, entre las que destacan su total independencia el performance que puede llegar a alcanzar. Utilizar un estilo Monolítico no es para nada malo, incluso en nuestra época, lo malo sería quedarnos siempre atascados en este estilo arquitectónico y no ir migrando a otros estilos más sofisticados a medida que nuestra aplicación va creciendo. Algunos síntomas de que el estilo arquitectónico ya nos comienza a quedar chico sería cuando, cada vez es más difícil dividir el trabajo entre los desarrolladores sin que tengan conflictos con el código, existen una necesidad de escalamiento más quirúrgica y se necesita escalar ciertas funcionalidades y no todo el proyecto, cuando el proyecto es tan grande que es complicado para los desarrolladores comprender el funcionamiento por la complejidad y la absurda cantidad de clases o archivos, y finalmente, cuando necesitamos empezar a diversificarnos con respecto al Stack tecnológico a utilizar.

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura Cliente-Servidor

Cliente-Servidor es uno de los estilos arquitectónicos distribuidos más conocidos, el cual está compuesto por dos componentes, el proveedor y el consumidor. El proveedor es un servidor que brinda una serie de servicios o recursos los cuales son consumido por el Cliente.

En una arquitectura Cliente-Servidor existe un servidor y múltiples clientes que se conectan al servidor para recuperar todos los recursos necesarios para funcionar, en este sentido, el cliente solo es una capa para representar los datos y se detonan acciones para modificar el estado del servidor, mientras que el servidor es el que hace todo el trabajo pesado.

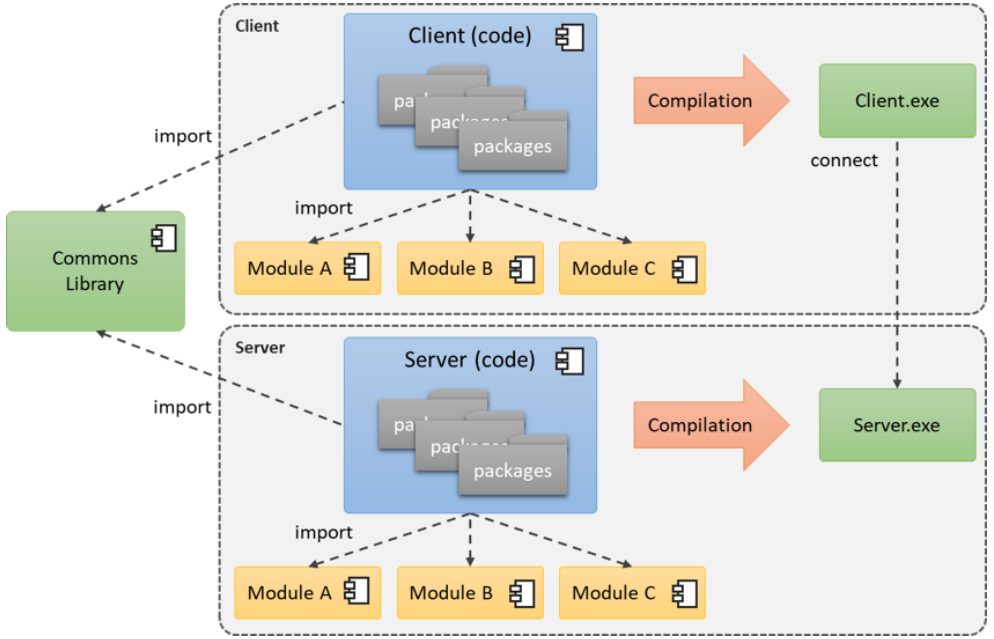


Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura Cliente-Servidor - Como se estructura

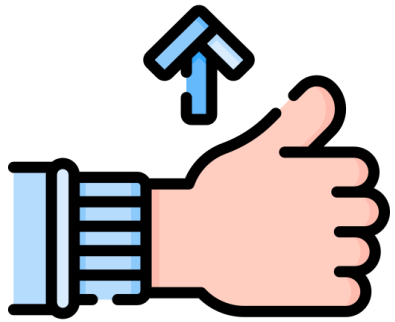
El cliente y el servidor son aplicaciones diferentes, por lo que pueden tener un ciclo de desarrollo diferente, así como usar tecnologías y un equipo diferente. Sin embargo, en la mayoría de los casos, el equipo que desarrolla el servidor también desarrolla el cliente, por lo que es normal ver que el cliente y el servidor están construidos con las mismas tecnologías.

El cliente y el servidor son construidos en general como Monolíticos, donde cada desarrollo crea su propio ejecutable único y funciona sobre un solo equipo, con la diferencia de que estas aplicaciones no son autosuficientes, ya que existe una dependencia simbiótica entre los dos.



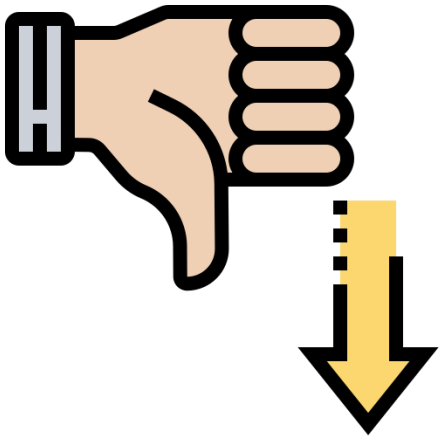
Arquitectura Cliente-Servidor

Ventajas	• **Centralización:** El servidor fungirá como única fuente de la verdad, lo que impide que los clientes conserven información desactualizada. • **Seguridad:** El servidor por lo general está protegido por firewall o subredes. • **Fácil de instalar (cliente):** El cliente es por lo general una aplicación simple que no tiene dependencias. • **Separación de responsabilidades:** Permite implementar la lógica de negocio de forma separada del cliente. • **Portabilidad:** Una de las ventajas de tener dos aplicaciones es que podemos desarrollar cada parte para correr en diferentes plataformas, por ejemplo, el servidor solo en Linux, mientras que el cliente podría ser multiplataforma.



Arquitectura Cliente-Servidor

Desventajas	• **Actualizaciones (clientes):** Una de las complicaciones es gestionar las actualizaciones en los clientes. • **Concurrencia:** Una cantidad no esperada de usuarios concurrentes puede ser un problema para el servidor. • **Todo o nada:** Si el servidor se cae, todos los clientes quedarán totalmente inoperables. • **Protocolos de bajo nivel:** Los protocolos más utilizados para establecer comunicación entre el cliente y el servidor suelen ser de bajo nivel, como Sockets, HTTP, RPC, etc. • **Depuración:** Es difícil analizar un error, pues los clientes están distribuidos en diferentes máquinas, incluso, equipos a los cuales no tenemos acceso.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

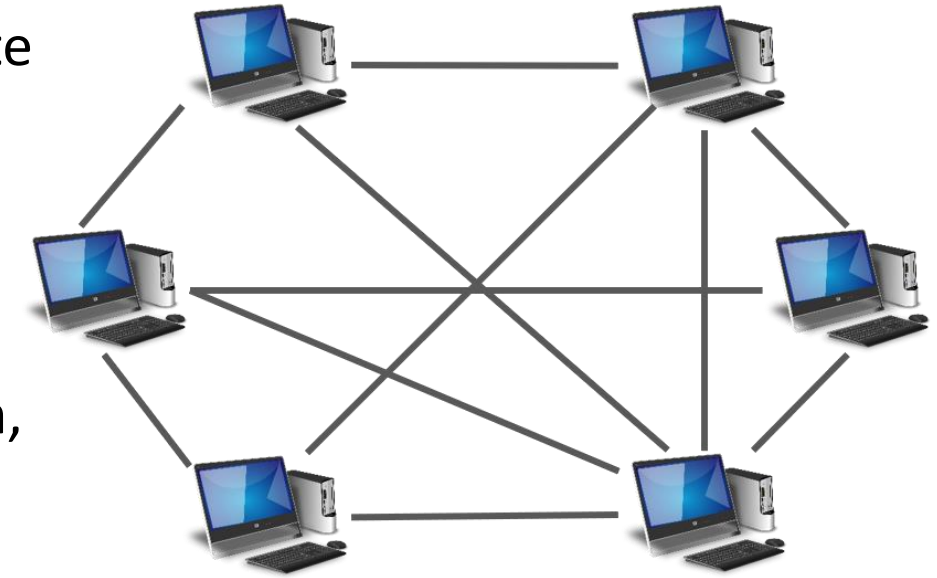
Arquitectura Cliente-Servidor

Conclusiones	A pesar de que el estilo Cliente-Servidor no es muy popular entre los nuevos desarrolladores, la realidad es que sigue siendo parte fundamental un muchas de las arquitecturas de hoy en día, solo basta decir que todo el internet está basado en Cliente-Servidor, sin embargo, no es común que como programadores o arquitectos nos encontremos ante problemáticas que requieran implementar un Cliente-Servidor, ya que estas arquitecturas están más enfocadas a aplicaciones CORE o de alto rendimiento, que por lo general es encapsulado por un Framework o API. A pesar de que puede que no les toque implementar una arquitectura Cliente-Servidor pronto, sí que es importante que entiendan cómo funciona, pues muchas de las herramientas que utilizamos hoy en día implementan este estilo arquitectónico, como podrían ser la base de datos, internet, sistemas de mensajería (JMS, MQ, etc), correo electrónico, programas de chat tipo Skype, etc. La realidad es que Cliente-Servidor es la base sobre la que está construida gran parte de la infraestructura tecnológica que hoy tenemos.

Arquitectura Peer To Peer (P2P)

El estilo arquitectónico Red entre iguales (Peer-to-peer, P2P) es una red de computadoras donde todos los dispositivos conectados a la red actúan como cliente y servidor al mismo tiempo. En esta arquitectura no es necesario un servidor central que administre la red (aunque puede existir), si no que todos los nodos de la red pueden comunicarse entre sí.

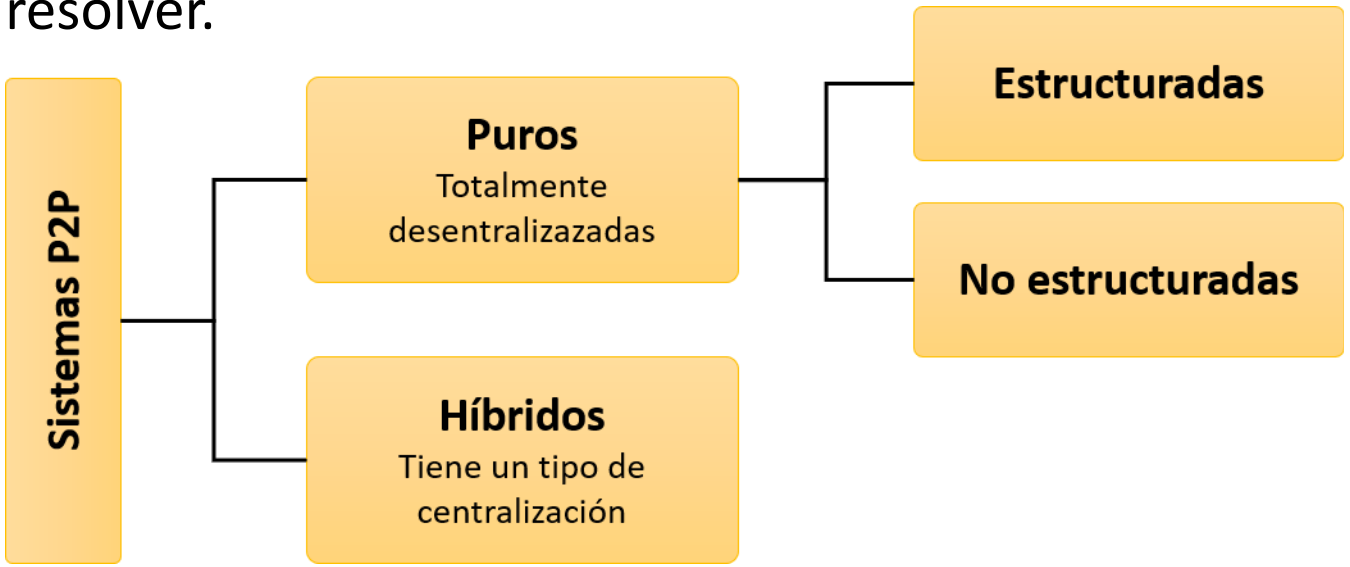
La arquitectura P2P es para muchos solo una variante de la arquitectura Cliente-Servidor, sin embargo, tiene una diferencia importante que hace que la podamos clasificar como un estilo arquitectónico independiente, y es que la arquitectura Cliente-Servidor tiene como punto medular la centralización, mientras la arquitectura P2P busca la descentralización.



Arquitectura Peer To Peer (P2P) - Como se estructura

Para comprender mejor como es que una aplicación P2P se estructura, es importante conocer las diferentes formas de implementarlo. A pesar de que la arquitectura P2P busca la descentralización y que mencionamos que no es necesario un servidor central, existen variantes que lo requieren, por lo que la arquitectura P2P puede variar en función del objetivo que estamos buscando resolver.

Para esto, podemos clasificar una arquitectura P2P basado en su grado de centralización:



Arquitectura Peer To Peer (P2P) - Como se estructura

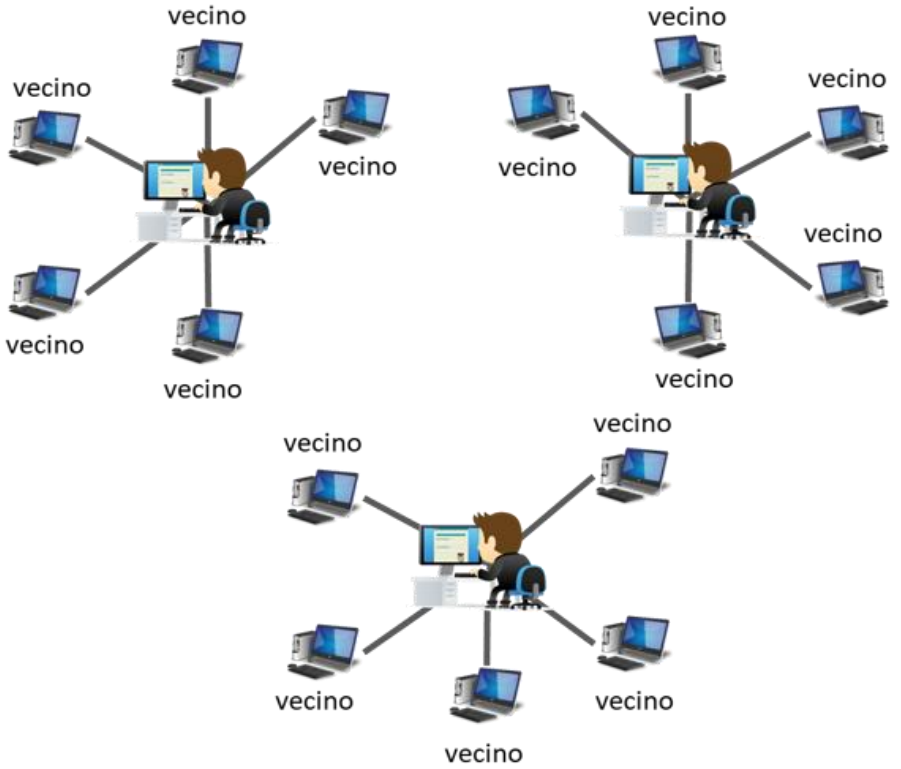
Arquitectura P2P Puro no estructuradas

Este tipo de arquitectura conforman una red totalmente descentralizada, donde no existe un servidor central ni roles donde algún nodo de la red tenga más privilegios o responsabilidades, además, todos los pares actúan como cliente y servidor al mismo tiempo, estableciendo una comunicación simétrica entre sí.

Arquitectura Peer To Peer (P2P) - Como se estructura

Arquitectura P2P Puro no estructuradas

En esta arquitectura un par solo se puede comunicar con los pares que conozca, es decir, que conoce la ubicación exacta. Ha estos pares conocidos se le conoce como vecinos, por lo que cada par puede tener un número de vecinos diferentes
Pueden existir muchas sub-redes donde un par conoce a algunos vecinos y otras sub-redes conocen a otros vecinos diferentes, y a pesar de que todos son partes de la red P2P no todos los pares se conocen entre sí.

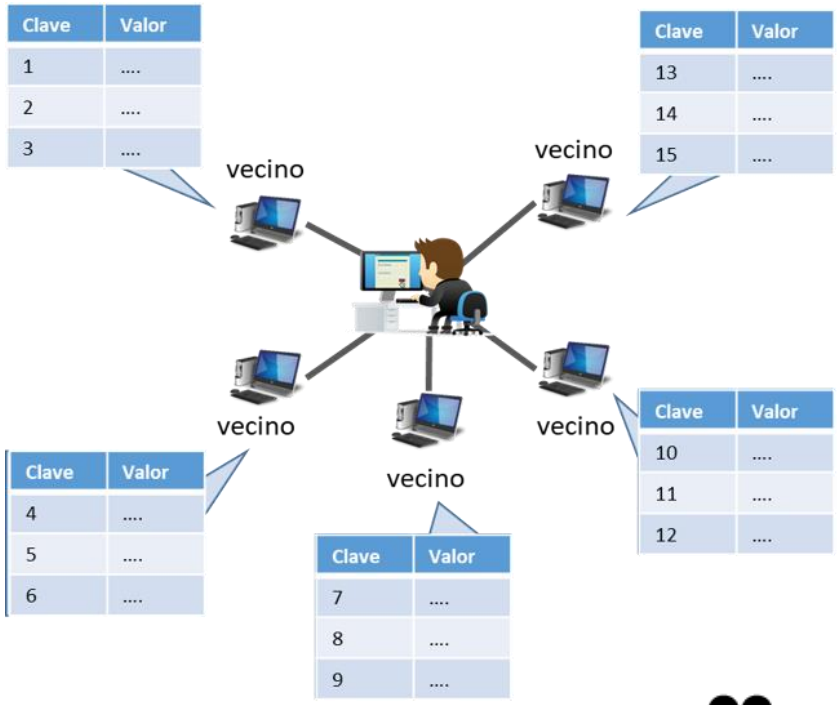


Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura Peer To Peer (P2P) - Como se estructura

Arquitectura P2P pura estructurada

Es una variante de la P2P no estructurada, por lo que mucho de la teoría planteada anteriormente aplica exactamente igual, sin embargo, la arquitectura P2P estructurada cambia la forma de buscar los recursos. En lugar de usar el método por inundación, busca los recursos basados en Tablas Hash Distribuidas (DHT por sus siglas en inglés). Esta tabla mantiene un registro de todos los recursos disponibles en la red en un formato Clave-Valor (Key-Value), donde la clave es un hash del recurso y el valor corresponde a la ubicación (nodo), de esta forma, solo es necesario buscar el clave hash del recurso en la tabla para conocer su ubicación, sin embargo, no están fácil como parece. Debido a que la tabla es distribuida, no existe físicamente un solo servidor, sino que está distribuida entre todos los nodos de la red, y cada nodo guarda un fragmento del índice total de los recursos compartidos.



Arquitectura Peer To Peer (P2P) - Como se estructura

Arquitectura P2P Híbridos

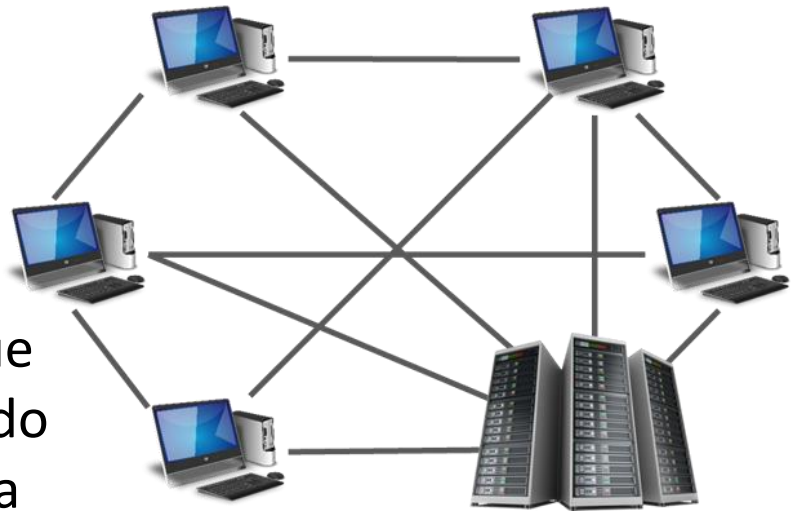
La arquitectura P2P híbrida o también conocida como Centralizada se caracteriza por tener un servidor central que sirve de enlace entre los nodos de la red, de tal forma que cualquier solitud para consumir los recursos de otros nodos deberán pasar primero por este servidor.

Arquitectura Peer To Peer (P2P) - Como se estructura

Arquitectura P2P Híbridos

La diferencia fundamental que tiene esta arquitectura con el Cliente-Servidor es que, este servidor no es el que proporciona los recursos, sino que solo sirve de enlace para conectar con otros nodos.

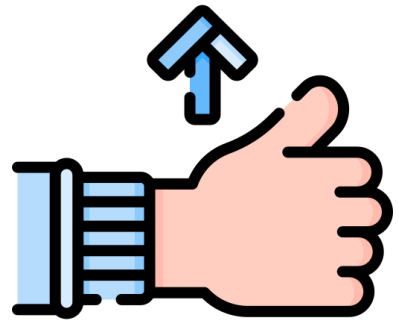
Un ejemplo claro de esta arquitectura es Napster. El servidor central de Napster no es quien almacena la información, sino que solo tiene el registro de todos los nodos conectados a la red y que archivos tiene cada nodo para compartir, de esta forma, cuando lanzamos una búsqueda, el servidor central de Napster sabe que nodos tiene ese archivo y nos dirige para que comencemos a descargar el archivo directamente sobre el nodo que tiene el archivo sin necesidad de un intermediario, de esta forma, Napster nunca tiene los archivos, pero sabe quién sí.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

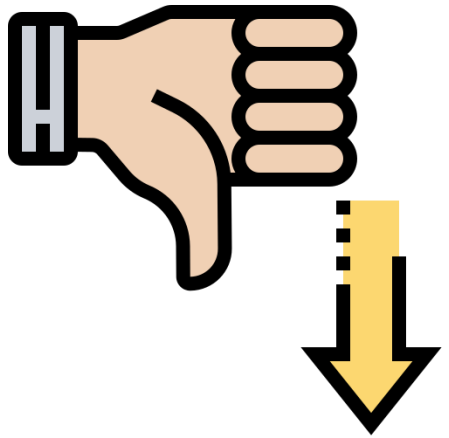
Arquitectura Peer To Peer (P2P)

Ventajas	• **Alta escalabilidad:** Permiten escalar fácilmente, pues a mayor número de nodos, más recursos hay disponibles y la carga de trabajo se divide entre todos los participantes. • **Tolerancia a fallas:** Tener los recursos distribuidos por varios nodos permite que los recursos estén disponibles sin importar si algunos de los nodos se caen. • **Descentralización:** Los sistemas P2P puros tiene una autonomía completa, por lo que pueden funcionar si un servidor central que organice toda la red. • **Privacidad:** Debido al sistema de búsqueda por inundación o tablas hash, es posible tener un alto nivel de privacidad, pues no es tan fácil saber que nodo fue el que realizo la petición inicial. • **Equilibrio de carga:** Todos los nodos de la red tienen el mismo rol y responsabilidades, por lo que toda la carga de trabajo se equilibra entre todos los nodos de la red.



Arquitectura Peer To Peer (P2P)

Desventajas	• **Alta complejidad:** Tiene una complejidad muy alta para desarrollarse, ya que tenemos que crear aplicaciones que funcionen como cliente y servidor, al mismo tiempo que tenemos que garantizar que las búsquedas de los recursos sean eficientes. • **Control:** A menos que tengamos un servidor central para administrar los recursos que se buscan y los usuarios que se conectan a la red, es muy fácil perder el control sobre el contenido que se puede compartir en la red. • **Seguridad:** Esta es una de las características menos implementadas en este tipo de arquitecturas, para evitar la intercepción de las comunicaciones, nodos maliciosos, contenido falso o adulterado o la propagación de virus o programas maliciosos. • **Tráfico:** En redes no optimizadas como las no estructuradas, podemos saturar la red con una gran cantidad de tráfico para propagar las peticiones a los nodos adyacentes.



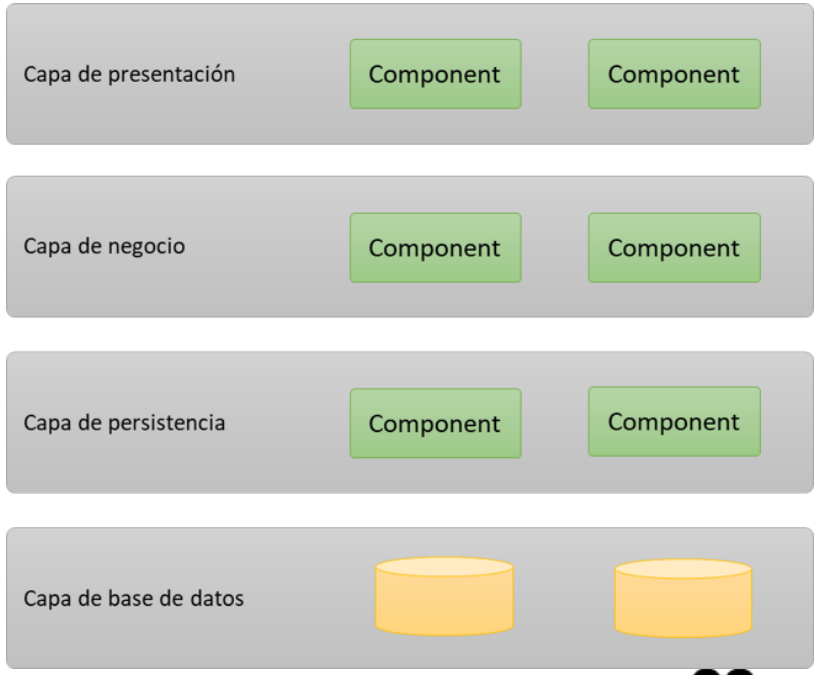
Arquitectura Peer To Peer (P2P)

Conclusiones	La arquitectura P2P se caracteriza por su capacidad de crecer a medida que más nodos entran a la red, de tal forma que entre más nodos se conectan, más poder de procesamiento de agrega a la red, al mismo tiempo que la hace más difícil de detener. No es común que se nos presente la necesidad de diseñar una aplicación con esta arquitectura, sin embargo, es importante entender cómo funcionan, pues seguramente utilizamos o estamos utilizando alguna aplicación que la utilice.

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura en Capas

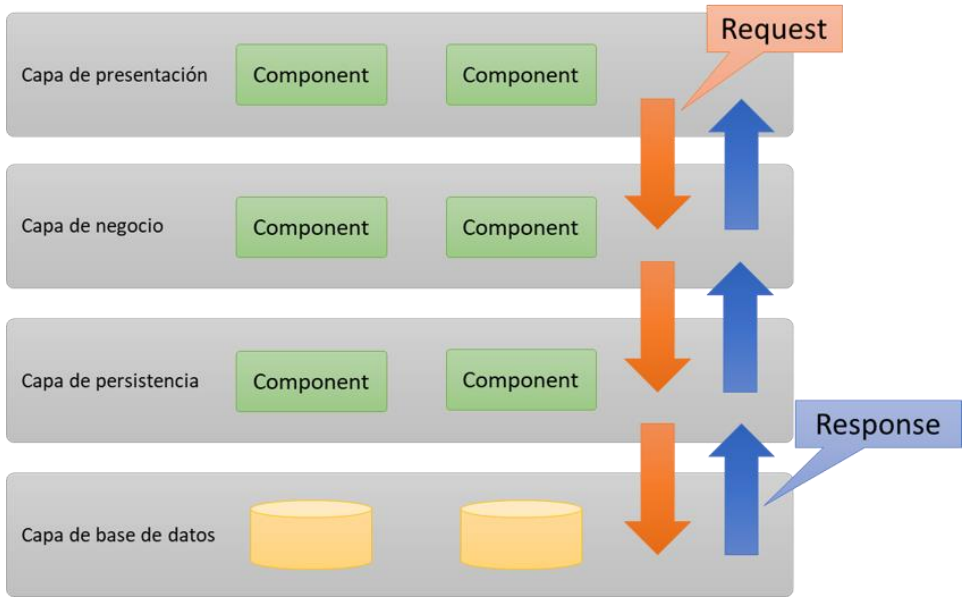
Es una de las más utilizadas, no solo por su simplicidad, sino porque también es utilizada por defecto cuando no estamos seguros de que arquitectura debemos de utilizar.
Consta en dividir la aplicación en capas, con la intención de que cada capa tenga un rol muy definido, como podría ser, una capa de presentación (UI), una capa de reglas de negocio (servicios) y una capa de acceso a datos (DAO), sin embargo, este estilo arquitectónico no define cuantas capas debe de tener la aplicación, sino más bien, se centra en la separación de la aplicación en capas (Aplica el principio Separación de preocupaciones (SoC)).
En la práctica, la mayoría de las veces este estilo arquitectónico es implementado en 4 capas, presentación, negocio, persistencia y base de datos, sin embargo, es habitual ver que la capa de negocio y persistencia se combinan en una solo capa, sobre todo cuando la lógica de persistencia está incrustada dentro de la capa de negocio.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura en Capas - Como se estructura

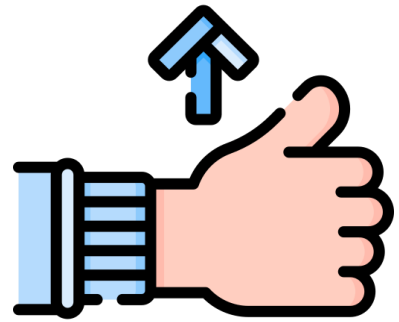
En una arquitectura en capas, todas las capas se colocan de forma horizontal, de tal forma que cada capa solo puede comunicarse con la capa que está inmediatamente por debajo, por lo que, si una capa quiere comunicarse con otras que están mucho más abajo, tendrán que hacerlo mediante la capa que está inmediatamente por debajo. Por ejemplo, si la capa de presentación requiere consultar la base de datos, tendrá que solicitar la información a la capa de negocio, la cual, a su vez, la solicitará a la capa de persistencia, la que, a su vez, la consultará a la base de datos, finalmente, la respuesta retornará en el sentido inverso hasta llegar la capa de presentación.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura en Capas

Ventajas	• **Separación de responsabilidades:** Permite la separación de preocupaciones (SoC), ya que cada capa tiene una sola responsabilidad. • **Fácil de desarrollar:** Este estilo arquitectónico es especialmente fácil de implementar, además de que es muy conocido y una gran mayoría de las aplicaciones la utilizan. • **Fácil de probar:** Debido a que la aplicación construida por capas es posible ir probando de forma individual cada capa, lo que permite probar por separada cada capa. • **Fácil de mantener:** Debido a que cada capa hace una tarea muy específica, es fácil detectar el origen de un bug para corregirlo, o simplemente se puede identificar donde se debe aplicar un cambio. • **Seguridad:** La separación de capas permite el aislamiento de los servidores en subredes diferentes, lo que hace más difícil realizar ataques.

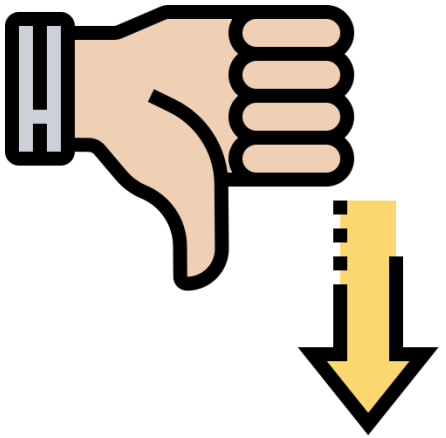


Separation of concerns (SoC)

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura en Capas

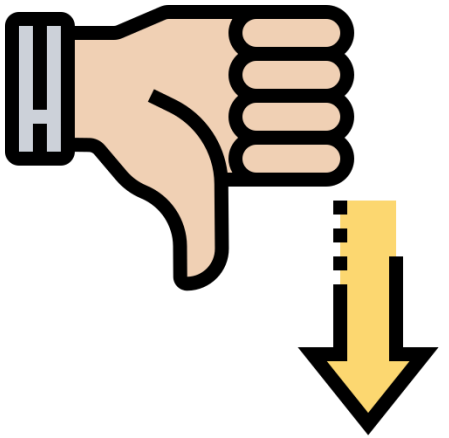
Desventajas	• **Performance:** La comunicación por la red o internet es una de las tareas más tardadas de un sistema, incluso, en muchas ocasiones más tardado que el mismo procesamiento de los datos, por lo que el hecho de tener que comunicarnos de capa en capa genera un degrado significativo en el performance. • **Escalabilidad:** Las aplicaciones que implementan este patrón por lo general tienda a ser monolíticas, lo que hace que san difíciles de escalar, aunque es posible replicar la aplicación completa en varios nodos, lo que provoca un escalado monolítico. • **Complejidad de despliegue:** En este tipo de arquitecturas es necesarios desplegar los componentes de abajo arriba, lo que crea una dependencia en el despliegue, además, si la aplicación tiende a lo monolítico, un pequeño cambio puede requerir el despliegue completo de la aplicación.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura en Capas

Desventajas	• **Anclado a un Stack tecnológico:** Si bien no es la regla, la realidad es que por lo general todas las capas de la aplicación son construidas con la misma tecnología, lo que hace amarremos la comunicación con tecnologías propietarias de la plataforma utilizada. • **Tolerancia a los fallos:** Si una capa falla, todas las capas superiores comienzan a fallar en cascada.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

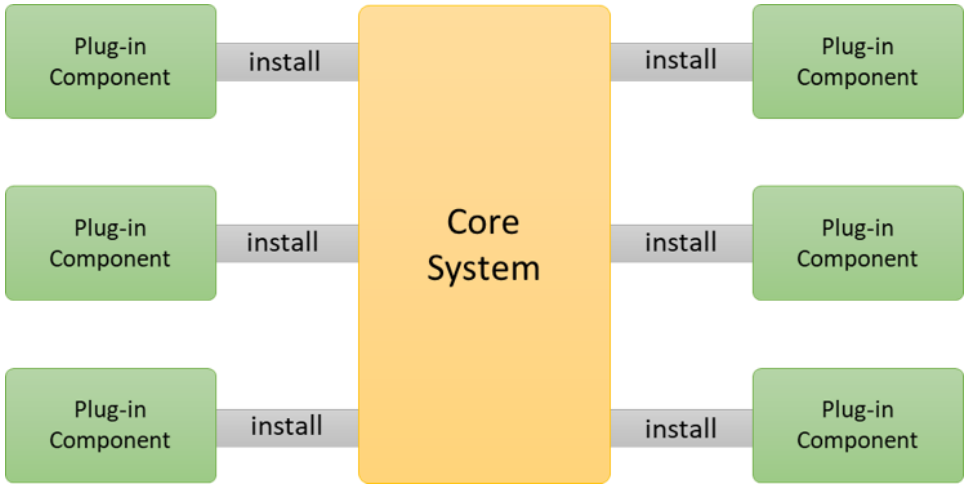
Arquitectura en Capas

Conclusiones	El estilo arquitectónico es uno de los más fácil de implementar, lo que lo hace unos de los patrones más versátiles y más ampliamente utilizados, lo que lo convierte en uno de los estilos arquitectónicos de referencia para muchas aplicaciones. Además, es un estilo que no representa mucha carga de mantenimiento para las empresas, lo que hace que pueda funcionar durante mucho tiempo de forma interrumpida. Por otra parte, hemos visto que este estilo tiene algunas desventajas que son importantes a tener en cuenta, sobre todo el performance y la escalabilidad, por lo que no es muy recomendable para aplicaciones de alto nivel de procesamiento de datos o que tiene una proyección de alto crecimiento a corto plazo.

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura de Microkernel

También conocido como arquitectura de Plug-in, permite crear aplicaciones extensibles, mediante la cual es posible agregar nueva funcionalidad mediante la adición de pequeños plugins que extienden la funcionalidad inicial del sistema.
En esta arquitectura las aplicaciones se dividen en dos tipos de componentes, en sistema Core (o sistema central) y los plugins (o módulos), el sistema Core contiene los elementos mínimos para hacer que la aplicación funcione y cumpla el propósito para el cual fue diseñada, por otra parte, los módulos o plugins con componentes periféricos que se añaden o instalan al componente Core para extender su funcionalidad. En este sentido, solo puede haber un componente Core y muchos Plugins.



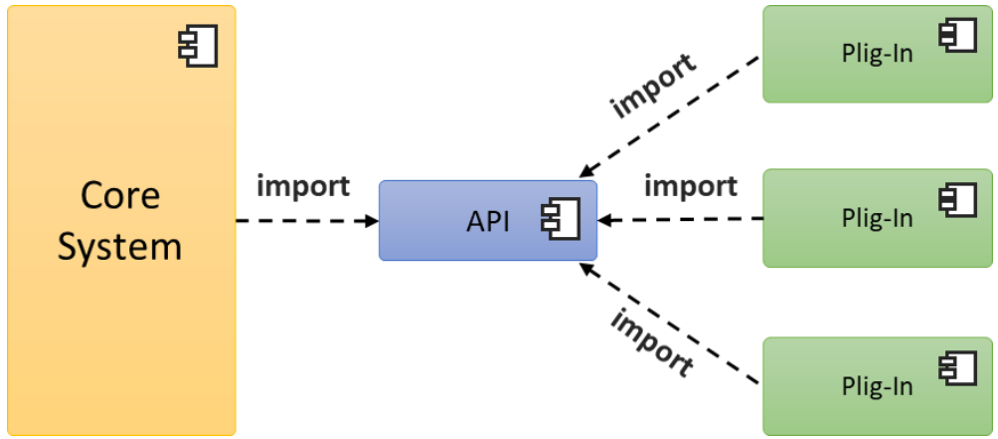
Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura de Microkernel - Como se estructura

Los sistemas que utilizan una arquitectura de Microkernel no son fáciles de desarrollar, pues necesitamos crear aplicaciones que son capaces de agrandar dinámicamente su funcionalidad a medida que nuevos plugins son instalados, al mismo tiempo que debemos de tener mucho cuidado de que los plugins no modifiquen o alteren la esencia de la aplicación.

El solo hecho de lograr que un sistema acepte plugins ya es complicado, sobre todo porque el sistema Core y los plugins son desarrollados por lo general por equipos separados, por lo que el sistema Core debe de dejar muy en claro como los plugins deben se desarrollados y deben de tener un archivo descriptor que le diga al sistema Core como debe de instalarse o debería de mostrar el plugin ante el usuario.

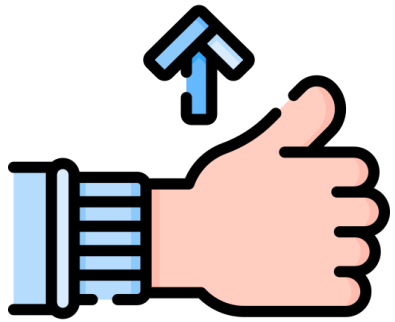
Para permitir la construcción de plugins, el sistema Core debe de proporcionar un API o una definición la cual el plugin debe de implementar, de esta forma tenemos el sistema Core y el API que provee para que los desarrolladores creen los plugins. El API es una serie de clases o interfaces que deben ser implementadas por el Plug-in, las cuales serán analizadas por el sistema Core al momento de la instalación.



API es una abreviatura de Application Programming Interfaces

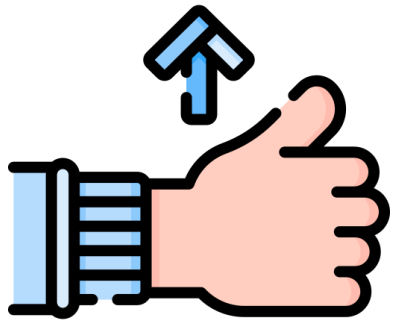
Arquitectura de Microkernel

Ventajas	• **Testabilidad:** Debido a que los Plugins y el sistema Core son desarrollados de forma por separado, es posible probarlos de forma aislada. • **Performance:** En cierta forma, muchas de las aplicaciones basadas en Microkernel trabajan de forma Monolítica una vez que el Plug-in es instalado, lo que hace que todo el procesamiento se haga en una sola unidad de software. • **Despliegue:** Debido a la naturaleza de Plugins es posible instalar fácilmente todas las características adicionales que sea necesarias, incluso, pueden ser agregar en tiempo de ejecución, lo que en muchos casos ni siquiera requiere de un reinicio del sistema.



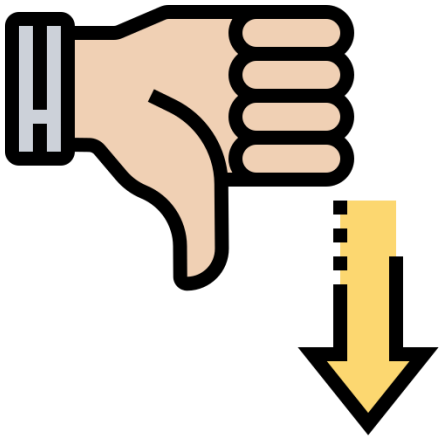
Arquitectura de Microkernel

Ventajas	• **Dinamismo:** Las aplicaciones basadas en Plugins pueden habilitar o deshabilitar características basadas en perfiles, lo que ayuda que solo los plugins necesarios sean activados. • **Construcción modular:** El sistema de Plugins permite que diferentes equipos puedan trabajar en paralelo para desarrollar los diferentes Plugins. • **Reutilización:** Debido a que los Plugins puede ser instalados en cualquier instancia del sistema Core, es posible reutilizar los módulos en varias instancias, incluso, es posible comercializarlas de forma independiente. Solo como ejemplo, existe empresas que se dedican exclusivamente a desarrollar Plugins para venderlos, como es el caso de los Plugins de Wordpress.



Arquitectura de Microkernel

Desventajas	• **Escalabilidad:** Las aplicaciones basadas en Microkernel son generalmente desarrolladas para ser ejecutadas en modo Standalone. • **Alta complejidad:** Las aplicaciones basadas en Microkernel son difíciles de desarrollar, no solo por la habilidad técnica para soportar la agregación de funcionalidad adicional por medio de Plugins, si no que requiere un análisis muy elaborado para identificar hasta qué punto puede ser extendida la aplicación sin afectar la esencia del sistema Core.



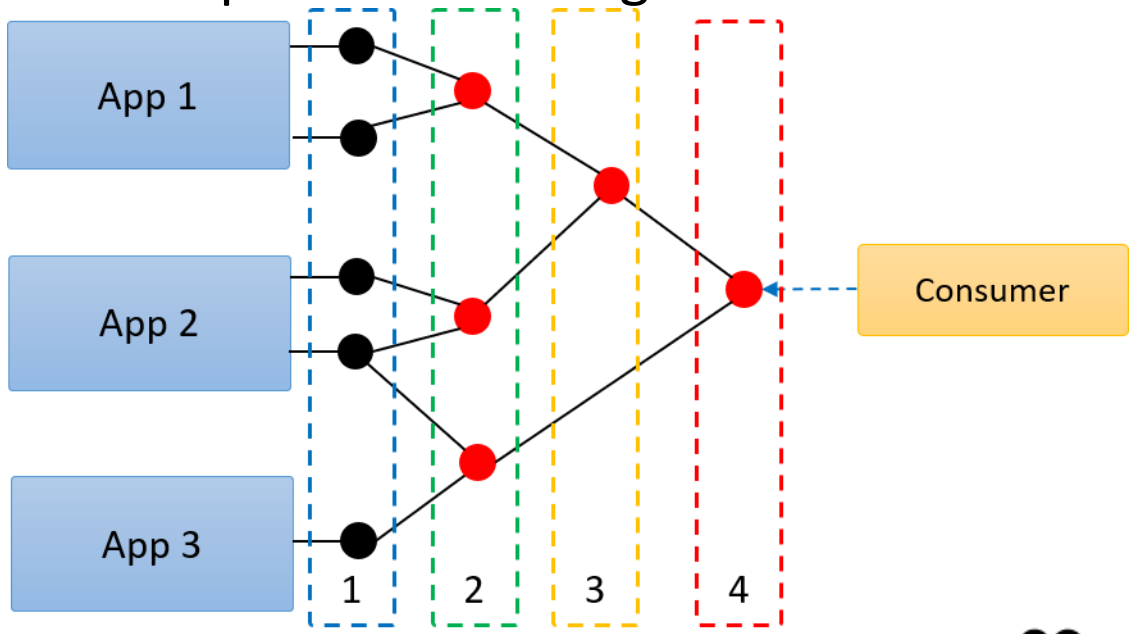
Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura de Microkernel

Conclusiones	El estilo arquitectónico de Microkernel permite extender la funcionalidad de sistema mediante la adición de Plugins, dichos plugins pueden ser desarrollados por terceros, lo que amplía las posibilidades de la aplicación Core si necesidad de costear los desarrollos, al mismo tiempo que permites que puedan personalizar al máximo el sistema. Por otra parte, podemos observar que un sistema basado en Microkernel permite tener múltiples equipos de desarrollo que construyen los módulos en paralelo sin interferir unos con otros, al mismo tiempo que permite que estos componentes sean probados de forma independientes, por lo que podemos decir que este estilo es fácil de probar. Sin embargo, hemos visto que una de las principales problemáticas de este estilo arquitectónico es su alta complejidad de desarrollar y su escalabilidad, lo cual puede ser una limitante para sistemas que deben ser diseñados para un alto escalamiento.

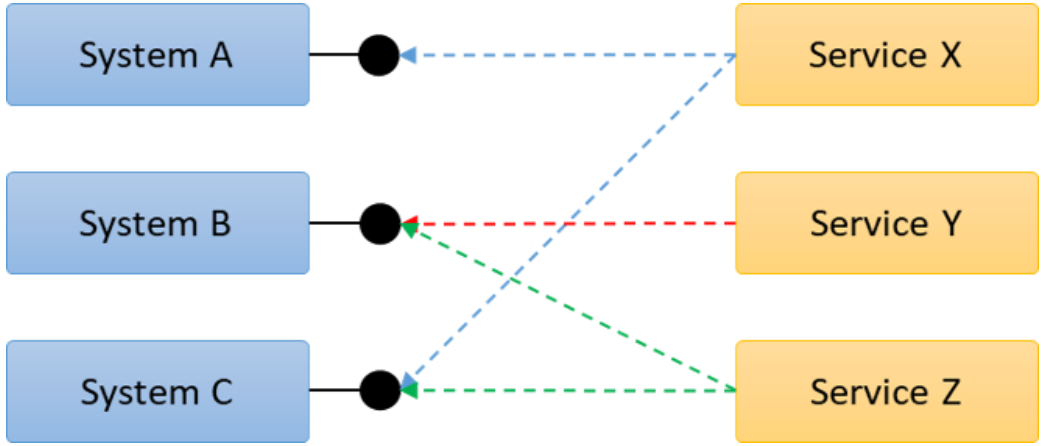
Service-Oriented Architecture (SOA)

La Arquitectura Orientada a Servicios (SOA, siglas del inglés Service Oriented Architecture) es un estilo de arquitectura de TI que se apoya en la orientación a servicios. La orientación a servicios es una forma de pensar en servicios, su construcción y sus resultados. Un servicio es una representación lógica de una actividad de negocio que tiene un resultado de negocio específico (ejemplo: comprobar el crédito de un cliente, obtener datos de clima, consolidar reportes de perforación)



Service-Oriented Architecture (SOA) - Como se estructura

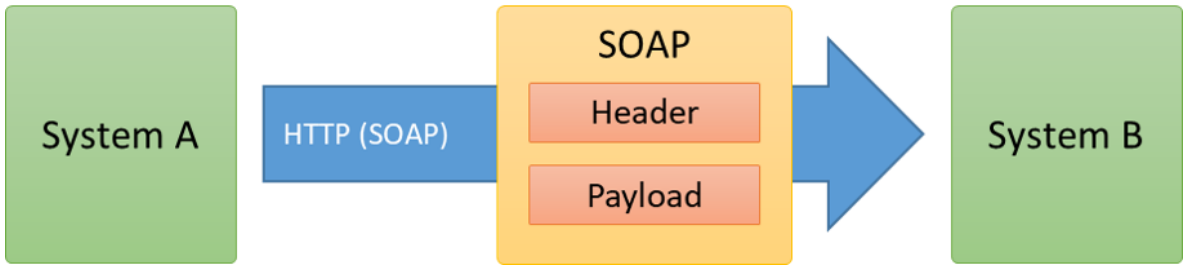
Lo primero que tenemos que entender al momento de implementar una arquitectura SOA es que, todas las aplicaciones que construimos están diseñadas para ser integradas con otras aplicaciones, por lo que deben de exponer como servicios todas las operaciones necesarias para que otras aplicaciones pueden interactuar con ella.
En la imagen podemos ver como las aplicaciones del lado izquierdo (A, B, C) exponen una serie de servicios que estarán disponibles por la red, los cuales son consumidos por las aplicaciones del lado derecho (X, Y, Z), aun que podrían ser consumidos por cualquier otra aplicación que este dentro de la misma red, o en su defecto, por internet.



Service-Oriented Architecture (SOA)

Independiente de la tecnología

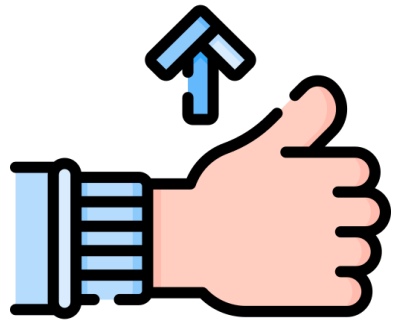
Lo primero que debemos de analizar es que, los servicios deben ser construidos con estándares abiertos, lo que quiere decir que es independiente del sistema operativo o del lenguaje de programación. Hoy en día conocemos a estos servicios como Web Services.
Para que un servicio sea considerado un Web Service, debe de publicar un WSDL (Web Services Description Language), el cual es un contrato que describe las operaciones que expone el servicio, los tipos de datos esperados en formato XSD (XML Schema).
Este contrato (WSDL) será utilizado por el consumidor para crear mensajes en formato XML para enviar la información, los cuales deberán de cumplir al pie de la letra la estructura definida en el contrato.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Service-Oriented Architecture (SOA)

Ventajas	• **Reduce el acoplamiento:** La comunicación basada en contratos permite desacoplar las aplicaciones, ya que no existe referencias a la aplicación, si no a un servicio que puede estar en cualquier parte. • **Testabilidad:** Las aplicaciones basadas en SOA son muy fáciles de probar, pues es posible probar de forma individual cada servicio. • **Reutilización:** Son desarrollados para hacer una sola tarea, por lo que facilita que otras aplicaciones reutilicen los servicios que existen. • **Agilidad:** Permite crear rápidamente nuevos servicios a partir de otros existentes. • **Escalabilidad:** Son muy fáciles de escalar, sobre todo porque permite agregar varias instancias de un servicio y balancear la carga desde el ESB. • **Interoperable:** La interoperabilidad es uno de los factores por lo que existe esta arquitectura y es que permite que aplicaciones heterogéneas se comuniquen de forma homogénea mediante el uso de estándares abiertos.

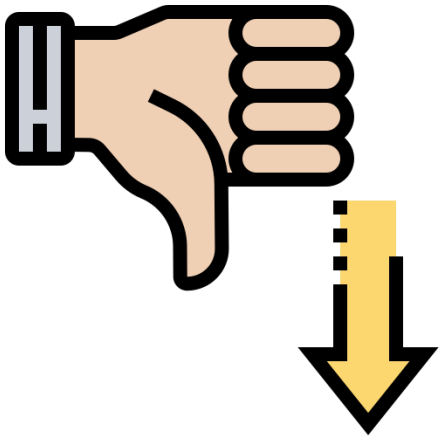


ESB (Enterprise Service Bus)

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Service-Oriented Architecture (SOA)

Desventajas	• **Performance:** No se caracteriza por el performance, pues la comunicación por la red y la distribución de los servicios agregar una latencia significativa en la comunicación. • **Volumetría:** SOA no se lleva con grandes volúmenes por petición, por lo que para procesar mucha información es recomendable utilizar otras tecnologías. • **Gobierno:** Para que SOA pueda ser implementado correctamente, es necesario crear un Gobierno que ponga orden sobre el uso y la creación de nuevos servicios, lo que puede representar una carga adicional a la empresa. • **Más puntos de falla:** Debido a que SOA es una arquitectura distribuida, es necesario implementar estrategias de falla, pues se incrementa el número de puntos de falla.



Service-Oriented Architecture (SOA)

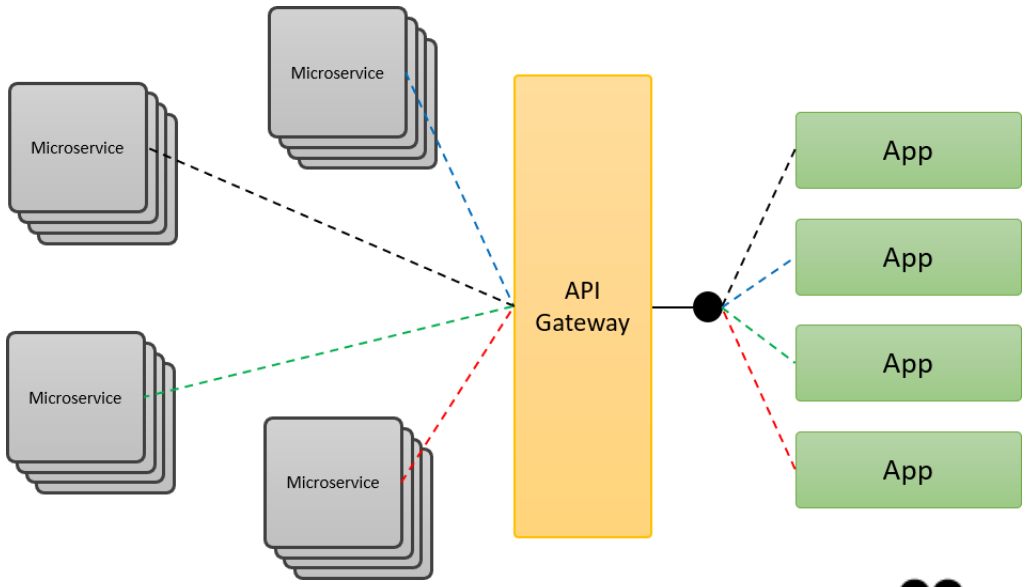
Conclusiones	SOA es una arquitectura muy flexible que permite la interoperabilidad entre las diferentes aplicaciones empresariales, ya que utiliza una serie de estándares abiertos que permite que sean independientes de la tecnología o la plataforma, sin embargo, SOA es uno de los estilos arquitectónicos más complejos por la cantidad de componentes que interviene, el gobierno y toda una serie de lineamientos que se deben de cumplir para implementarla correctamente.

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura de Microservicios

El estilo de Microservicios consiste en crear pequeños componentes de software que solo hacen una tarea, la hace bien y son totalmente autosuficientes, lo que les permite evolucionar de forma totalmente independiente del resto de componentes.

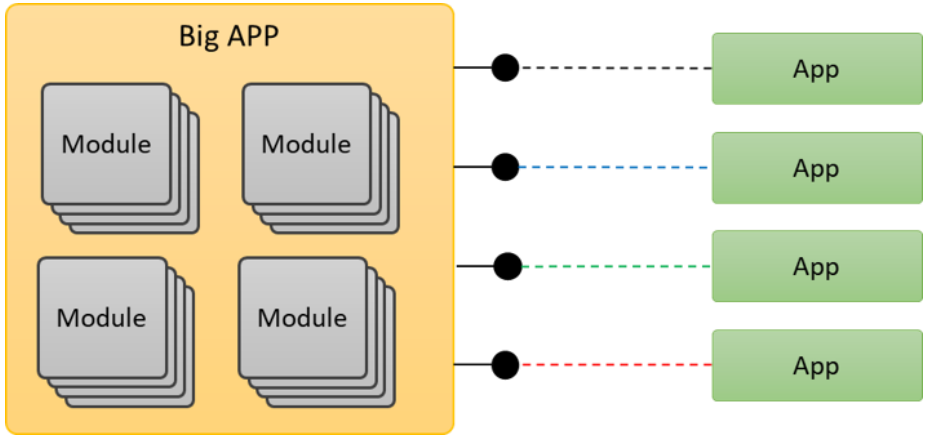
El nombre de “Microservicios” se puede mal interpretar pensando que se trata de un servicio reducido o pequeño, sin embargo, ese es un concepto equivocado, los Microservicios son en realidad pequeñas aplicaciones que brindan un servicio, y observa que he dicho “brinda un servicio” en lugar de “exponen un servicio”, la diferencia radica en que los componentes que implementan un estilo de Microservicios no tienen por qué ser necesariamente un Web Services o REST, y en su lugar, puede ser un proceso que se ejecuta de forma programada, procesos que mueva archivos de una carpeta a otra, componentes que responde a eventos, etc. En este sentido, un Microservicios no tiene porqué exponer necesariamente un servicio, sino más bien, proporciona un servicio para resolver una determinada tarea del negocio.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura de Microservicios - Como se estructura

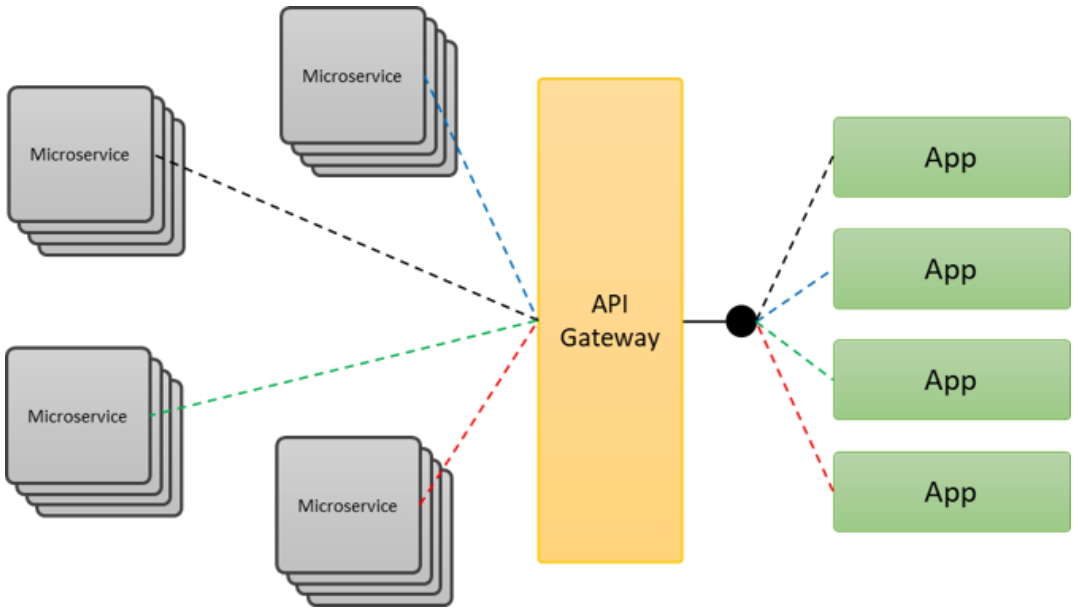
Para comprender los Microservicios es necesario regresar a la arquitectura Monolítica, donde una solo aplicación tiene toda la funcionalidad para realizar una tarea de punta a punta, además, una arquitectura Monolítica puede exponer servicios.
Una aplicación Monolítica tiene todos los módulos y funcionalidad necesario dentro de ella, lo que lo hace una aplicación muy grande y pesada, además, en una arquitectura Monolítica, es Todo o Nada, es decir, si la aplicación está arriba, tenemos toda la funcionalidad disponible, pero si está abajo, toda la funcionalidad está inoperable.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

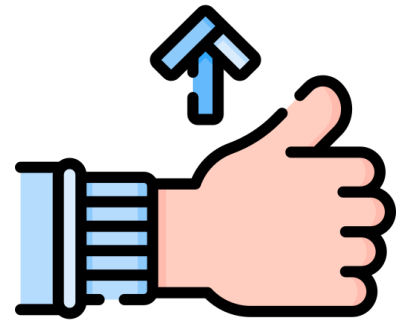
Arquitectura de Microservicios - Como se estructura

La arquitectura de Microservicios busca exactamente lo contrario, dividiendo toda la funcionalidad en pequeños componentes autosuficientes e independientes del resto de componentes, tal y como lo puedes ver en la imagen anterior.



Arquitectura de Microservicios

Ventajas	• **Alta escalabilidad:** Es un estilo arquitectónico diseñado para ser escalable, pues permite montar numerosas instancias del mismo componente y balancear la carga entre todas las instancias. • **Agilidad:** Debido a que cada Microservicios es un proyecto independiente, permite que el componente tenga ciclo de desarrollo diferente del resto, lo que permite que se puedan hacer despliegues rápidos a producción sin afectar al resto de componentes. • **Facilidad de despliegue:** Las aplicaciones desarrolladas como Microservicios encapsulan todo su entorno de ejecución, lo que les permite ser desplegadas sin necesidad de dependencias externas o requerimientos específicos de Hardware.

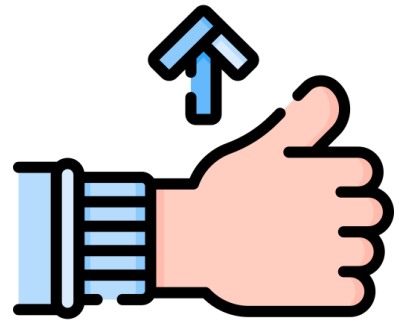


ESB (Enterprise Service Bus)

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura de Microservicios

Ventajas	• **Testabilidad:** Son especialmente fáciles de probar, pues su funcionalidad es tan reducida que no requiere mucho esfuerzo. • **Fácil de desarrollar:** Debido a que tienen un alcance muy corto, es fácil para un programador comprender el alcance del componente. • **Reusabilidad:** La reusabilidad es la médula espinal de la arquitectura de Microservicios, pues se basa en la creación de pequeños componentes que realice una única tarea, lo que hace que sea muy fácil de reutilizar por otras aplicaciones o Microservicios. • **Interoperabilidad:** Debido a que los Microservicios utilizan estándares abiertos y ligeros para comunicarse, hace que cualquier aplicación o componente pueda comunicarse con ellos, sin importar en que tecnología está desarrollado.

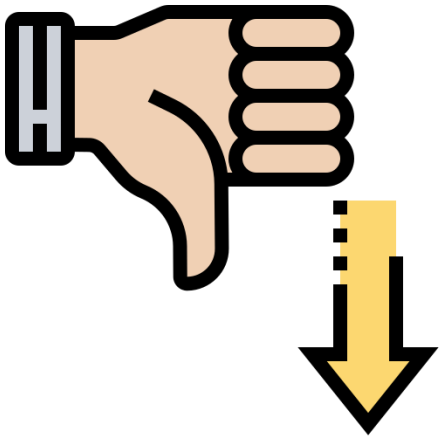


ESB (Enterprise Service Bus)

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura de Microservicios

Desventajas	• **Performance:** La naturaleza distribuida de los Microservicios agrega una latencia significativa que puede ser un impedimento para aplicaciones donde el performance es lo más importante, por otra parte, la comunicación por la red puede llegar a ser incluso más tardado que el proceso en sí. • **Múltiples puntos de falla:** La arquitectura distribuida de los Microservicios hace que los puntos de falla de una aplicación se multipliquen • **Trazabilidad:** La naturaleza distribuida de los Microservicios complica recuperar y realizar una traza completa de la ejecución de un proceso • **Madurez del equipo de desarrollo:** Debe ser implementada por un equipo maduro de desarrollo y con un tamaño adecuado, pues agregan muchos componentes que deben ser administrados, lo que puede ser muy complicado para equipos poco maduros.



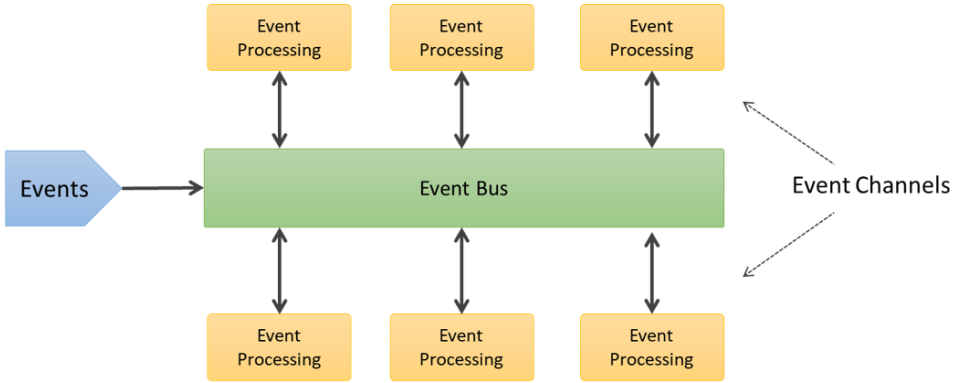
Arquitectura de Microservicios

Conclusiones	Los Microservicios son pequeños componentes que se especializan en realizar una sola tarea, lo hace bien y son totalmente desacoplados. Hemos analizado y comparado el estilo Monolítico con los Microservicios y hemos analizado las diferencias que existen entre estos dos estilos, sin embargo, el estilo de Microservicios es uno de los más complejos por la gran cantidad de componentes y patrones arquitectónicos necesario para implementarlo correctamente.

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Event-driven architecture (EDA)

La arquitectura dirigida por eventos[1] o simplemente EDA (por sus siglas en inglés) es una arquitectura asíncrona y distribuida, pensada para crear aplicaciones altamente escalables.
En una arquitectura EDA los componentes no se comunican de forma tradicional, en la cual se establece comunicación de forma síncrona, se obtiene una respuesta y se procede con el siguiente paso. En esta arquitectura, se espera que las aplicaciones lancen diversos “eventos” para que otros componentes puedan reaccionar a ellos, procesarlos y posiblemente generar nuevos eventos para que otros componentes continúen con el trabajo.
Todo inicia con la llegada de un nuevo evento a algo que llamaremos por ahora Event Bus, el cual es un componente que se encarga de administrar todos los eventos de la arquitectura. El Event Bus se encargará de recibir el evento y colocarlo en los Event Channels, las cuales son básicamente una serie de Colas (Queues) o Temas (Topics) sobre los cuales habrá ciertos componentes a la escucha de nuevos mensajes.



[1]Evento: Un evento lo podemos definir como un cambio significativo en el estado de la aplicación, ya sea por un dato que cambio o alguna acción concreta en el sistema que merece la pena ser observada para tomar acciones en consecuencia

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Event-driven architecture (EDA) (cont.)

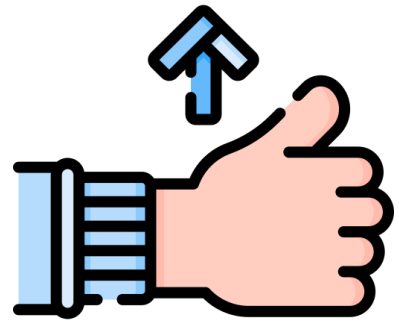
Por otra parte, los Event Processing son componentes de negocio que está a la escucha de nuevos eventos depositados en los Event Channels. Los Event Processing son los encargados de procesar los eventos, es decir, realizar acciones concretas sobre el sistema, como crear, actualizar o borrar algún registro, finalmente, el proceso puedo terminar de forma silenciosa, o puede generar un nuevo evento para que otro componente lo tome y continúe con el procesamiento.

Un punto importante a tomar en cuenta es que todos los Event Processing son componentes distribuidos, lo que quiere decir que están totalmente desacoplados del resto y que el funcionamiento de uno no afecta al resto. De esta forma, cada Event Processing administra sus propias transacciones, lo que hace difícil mantener la atomicidad de la transacción en una serie de eventos, por lo que es un punto importante a tomar en cuenta al momento de trabajar con esta arquitectura.

En esta arquitectura, cada Event Processing solo tiene conocimiento de la tarea que el realiza y no tiene conocimiento de la existencia de otros Event Processing, lo cual permite que trabajen de forma autónoma.

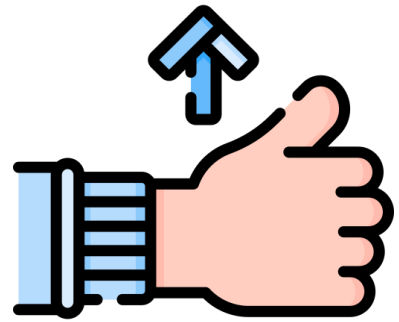
Arquitectura Event-driven architecture (EDA)

Ventajas	• **Escalabilidad:** La escalabilidad es uno de los puntos más fuertes de esta arquitectura, pues permite que cada consumidor (Event Consummer) puede escalar de forma independiente y reduce al máximo el acoplamiento entre los componentes. • **Performance:** Esta es que ventaja cuestionable, ya que al igual que en REST o SOA, EDA necesita pasar por una serie de pasos para completar una tarea, agregando retrasos en cada paso, desde colocar el Evento por parte del productor, esperar a que el consumidor lo tome, y generar nuevos Eventos, sin embargo, la naturaleza Asíncrona de EDA hace que esta desventaja se supere mediante el procesamiento en paralelo.



Arquitectura Event-driven architecture (EDA)

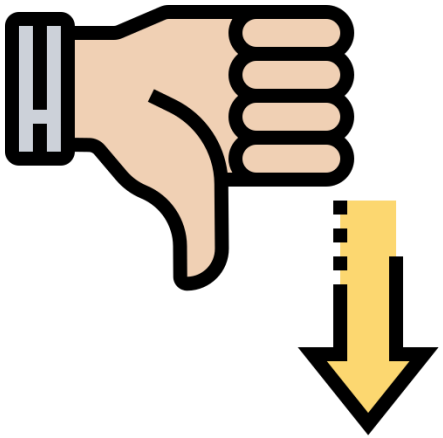
Ventajas	• **Despliegue:** Debido al bajo acoplamiento entre los componentes, es posible realizar el despliegue sin preocuparse por dependencias o precondiciones, al final, los componentes solamente se suscriben para recibir eventos y reaccionar ante ellos. • **Flexibilidad:** EDA permite responder rápidamente a un entorno cambiante, debido a que cada componente procesador de eventos tiene una sola responsabilidad y está completamente desacoplado de los demás, de esta forma, si ocurre un cambio, se puede aislar en un solo componente sin afectar al resto, además, si un nuevo requerimiento es requerido, solo es necesario regresar un nuevo tipo de procesador de eventos que escuche un determinado tipo de evento.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Arquitectura Event-driven architecture (EDA)

Desventajas	• **Testabilidad:** Una arquitectura distribuida y asíncrona agrega cierta complejidad a las pruebas, pues no es posible generar un evento y esperar un resultado para validar el resultado, en su lugar, es necesario crear pruebas más sofisticadas que creen los eventos y esperen la finalización del procesamiento del evento inicial más toda la serie de eventos que se podrían generar como consecuencia del evento inicial para validar el resultado final. • **Desarrollo:** Codificar soluciones asíncronas es complicado, pero aún más, es la necesidad de crear manejadores de errores más avanzados que permitan recuperarse en una arquitectura asíncrona, donde la falla en un procesador no significa la falla en el resto, lo que puede ocasionar la inconsistencia entre los sistemas.



Arquitectura Event-driven architecture (EDA)

Conclusiones	EDA es sin lugar a duda una de los estilos arquitectónicos más escalables que existe y que permite que empresa globales puedan atender a millones de solicitudes sin colgar sus sistemas, ya que permite dividir el trabajo en diversos procesadores de mensajes que trabajan de forma totalmente asiladas y desacopladas del resto, sin embargo, implementar EDA no es fácil, pues requiere un esfuerzo mucho mayor que una arquitectura síncrona, y requiere una mejor preparación de los ingenieros que hay detrás de la solución. Otra de las cosas a considerar al implementar EDA es que, debemos tener un conocimiento más profundo de la solución de negocio que vamos a desarrollar, pues necesitamos ese conocimiento para saber cómo dividir la aplicación en los diversos consumidores, ya que de lo contrario podemos crear módulos de software que no sean lo suficientemente desacoplados del resto.

Arquitectura Event-driven architecture (EDA)

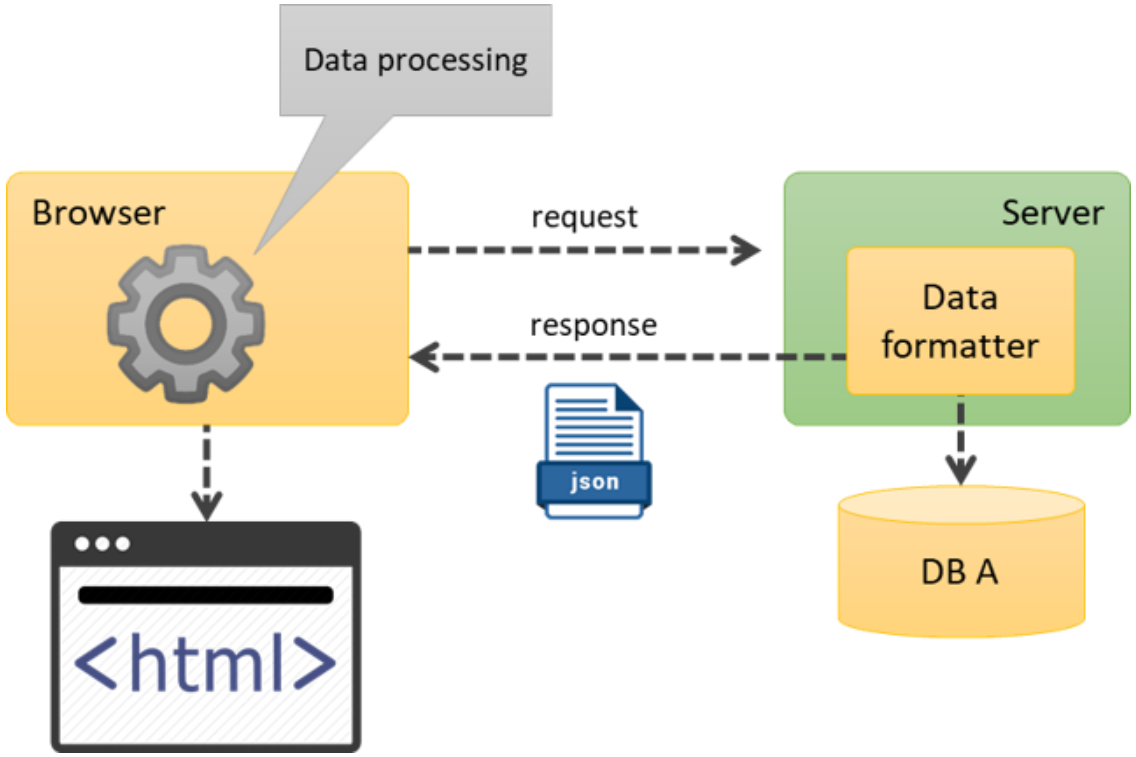
Conclusiones	Finalmente, debemos de entender que en una arquitectura EDA no existe el concepto de transacciones, pues cada consumidor manejará de forma independiente sus propias transacciones, por lo que si un paso de la cadena de procesamiento falla, no implicará el rollback en todos los consumidores, es por ello que debemos de tener manejadores de errores muy avanzados que permiten a un proceso recuperarse en caso de fallo o de plano que levante una serie de nuevos eventos para hacer el rollback en los otros consumidores. Y toma en cuenta que en EDA no hay garantía que un evento sea procesado, ya sea porque no hay consumidores para un determinado evento o por un error en el diseño.

Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Representational state transfer (REST)

REST[1] se ha convertido sin lugar a duda en uno de los estilos arquitectónicos más utilizados en la industria, ya que es común ver que casi todas las aplicaciones tienen un API REST que nos permite interactuar con ellas por medio de servicios.

Algo muy importante a tomar en cuenta es que REST ignora los detalles de implementación del componente y la sintaxis del protocolo para enfocarse en los roles de los componentes, las restricciones sobre su interacción con otros componentes y la representación de los datos. Por otro lado, abarca las restricciones fundamentales sobre los componentes, conectores y datos que definen la base de la arquitectura web, por lo tanto, podemos decir que la esencia de REST es comportarse como una aplicación basada en la red



[1]REST: proporciona un conjunto de restricciones arquitectónicas que, cuando se aplican como un todo, enfatizan la escalabilidad de las interacciones de los componentes, la generalidad de las interfaces, la implementación independiente de los componentes y los componentes intermedios para reducir la latencia de interacción, reforzar la seguridad y encapsular los sistemas heredados.

Representational state transfer (REST) - Como se estructura

REST describe 3 conceptos clave, que son: Datos, Conectores y Componentes, los cuales trataremos de definir a continuación.

1. Datos

Uno de los aspectos más importantes que propone REST es que los datos deben de ser transmitidos a donde serán procesados, en lugar de que los datos sean transmitidos ya procesados.

Representational state transfer (REST) - Como se estructura (cont.)

1. Datos (cont.)

En cualquier arquitectura distribuida, son los componentes de negocio quienes procesan la información y responde con la información que se produjo de tal procesamiento, por ejemplo, una aplicación web tradicional, donde la vista es construida en el backend y nos responde con la página web ya creada, con los elementos HTML que la componen y los datos incrustados. En este sentido, la aplicación web procesó los datos en el backend, y como resultado nos arrojó el resultado de dicho procesamiento, sin embargo, REST lo que propone es que los datos sean enviados en bruto para que sea el interesado de los datos el que los procese para generar la página (o cualquier otra cosa que tenga que generar), en este sentido, lo que REST propone no es generar la página web, si no que mande los datos en bruto para que el consumidor sea el responsable de generar la página a través de los datos que responde REST.

Representational state transfer (REST) - Como se estructura (cont.)

2. Conectores

Los conectores son los componentes que encapsulan la actividad de acceso a los recursos y transferencia, de esta forma, REST describe los conectores como interfaces abstractas que permite la comunicación entre compontes, mejorando la simplicidad al proporcionar una separación clara de las preocupaciones y ocultando la implementación subyacente de los recursos y los mecanismos de comunicación.

Por otro lado, REST describe que todas las solicitudes a los conectores sean sin estado, es decir, que deberán contener toda la información necesaria para que el conector comprenda la solicitud sin importar las llamadas que se hallan echo en el pasado, de esta forma, ante una misma solicitud deberíamos de obtener la misma respuesta.

Representational state transfer (REST) - Como se estructura (cont.)

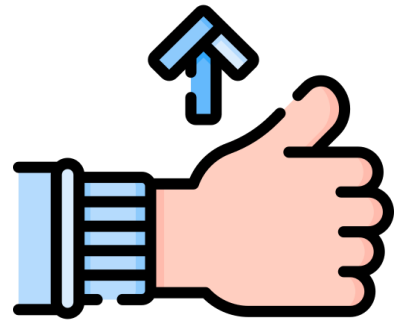
3. Componentes

Finalmente, los componentes con software concretos que utilizan los conectores para consultar los recursos o servirlos, ya sea una aplicación cliente como lo son los navegadores (Chrome, Firefox, etc) o Web Servers como Apache, IIS, etc. Pero también existen los componentes intermedios, que actúan como cliente y servidor al mismo tiempo, como es el caso de un proxy o túnel de red, los cuales actúan como servidor al aceptar solicitudes, pero también como cliente a redireccionar las peticiones a otro servidor.

Los componentes se pueden confundir con los conectores, sin embargo, un conector es una interface abstracta que describe la forma en que se deben de comunicar los componentes, mientras que un componente es una pieza de software concreta que utiliza los conectores para establecer la comunicación.

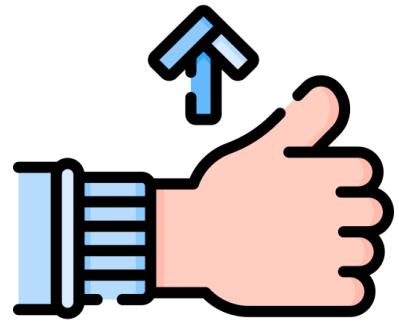
Representational state transfer (REST)

Ventajas	• **Interoperable:** REST no está casado con una tecnología, por lo que es posible implementarla en cualquier lenguaje de programación. • **Escalabilidad:** Debido a todas las peticiones son sin estado, es posible agregar nuevos nodos para distribuir la carga entre varios servidores. • **Reducción del acoplamiento:** Debido a que los conectores definen interfaces abstractas y que los recursos son accedidos por una URL, es posible mantener un bajo acoplamiento entre los componentes.



Representational state transfer (REST)

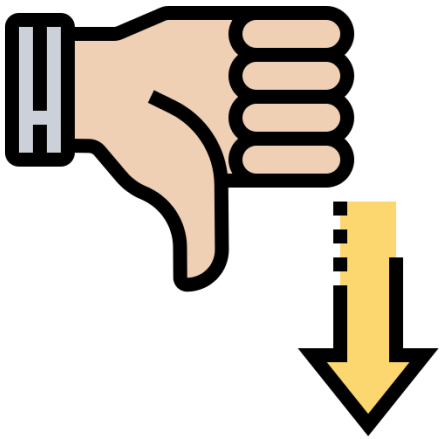
Ventajas	• **Testabilidad:** Debido a que todas las peticiones tienen toda la información y que son sin estado, es posible probar los recursos de forma individual y probar su funcionalidad por separado. • **Reutilización:** Uno de los principios de REST es llevar los datos sin procesar al cliente, para que sea el cliente quién decida cuál es la mejor forma de representar los datos al usuario, por este motivo, los recursos de REST pueden ser reutilizado por otros componentes y representar los datos de diferentes maneras.



Aprendemos: Arquitectura de Software - Estilos arquitectónicos

Representational state transfer (REST)

Desventajas	• **Performance:** Como en todas las arquitecturas distribuidas, el costo en el performance es una constante, ya que cada solicitud implica costos en latencia, conexión y autenticaciones. • **Más puntos de falla:** Nuevamente, las arquitecturas distribuidas traen con sigo ciertas problemáticas, como es el caso de los puntos de falla, ya que al haber múltiples componentes que conforman la arquitectura, multiplica los puntos de falla. • **Gobierno:** Hablando exclusivamente desde el punto de vista de servicios basados en REST, es necesario mantener un gobierno sobre los servicios implementados para llevar un orden, ya que es común que diferentes equipos creen servicios similares, provocando funcionalidad repetida que tiene que administrarse.



Arquitectura Event-driven architecture (EDA)

Conclusiones	REST no es como tal una arquitectura de software, sino más bien es un conjunto de restricciones que en conjunto hacen un estilo de arquitectura de software distribuido que se centra en la forma en la que transmitimos los datos y las diferentes representaciones que estos pueden tener. REST no especifica ningún protocolo, sin embargo, HTTP es el protocolo principal de transferencia y el más utilizado por REST, concretamente los servicios RESTfull.

Arquitectura Event-driven architecture (EDA)

Conclusiones	REST no es para nada una estilo de arquitectura nuevo, pues la misma WEB es un ejemplo de REST, pero sí que se ha popularizado enormemente debido a la creciente necesidad de integrar aplicaciones de una forma simple, pero sobre todo, se ha disparado su uso debido a los servicios RESTfull que llegaron para reemplazar los Webservices tradicionales (SOAP), los cuales si bien funciona bien, está limitado a mensajes SOAP en formato XML y que son en esencia mucho más difíciles de comprender por su sintaxis en XML y los namespace que un simple JSON. Por otra parte, los servicios RESTfull en combinación con un estilo de Microservicios está dominando el mundo, ya que hoy en día todas las empresas buscan migrar a Microservicios o ya lo están, lo que hace que REST sea una de las arquitecturas más de moda de la actualidad.

Verificamos lo aprendido:

Kahoot!

http://www.kahoot.it/

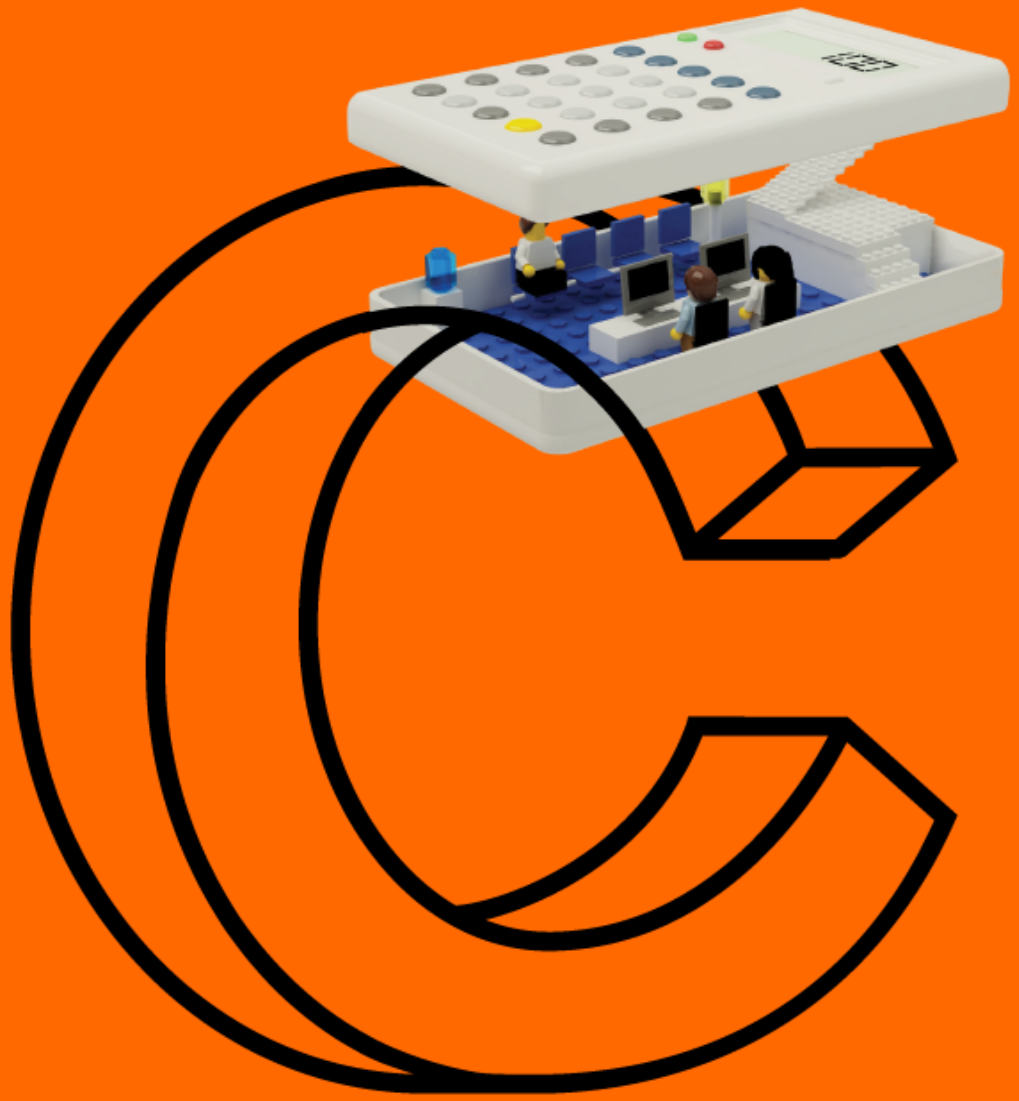
Verificamos lo aprendido:

- Absolución de Preguntas.
- Conclusiones y Recomendaciones.

Aplicamos lo aprendido: Asignación para la clase siguiente

- Comentar en el Foro

Gracias



idat