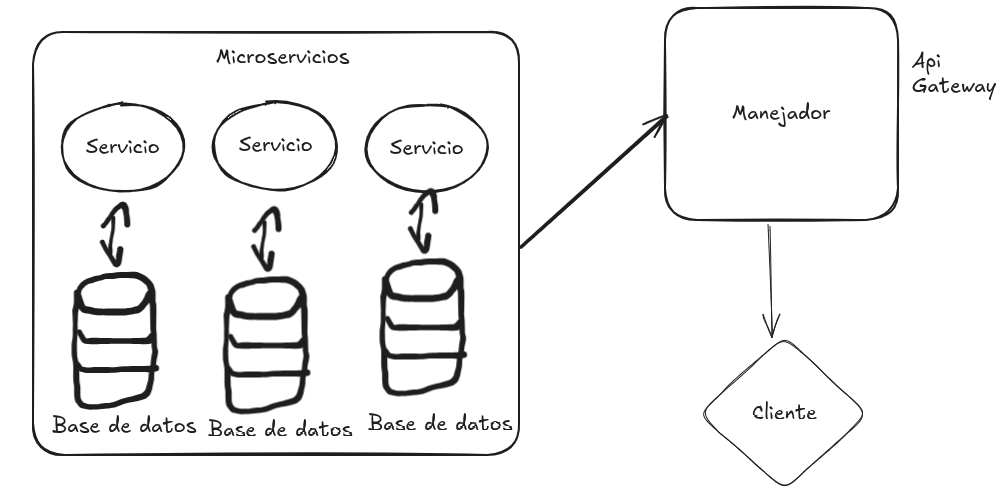
Articulos

# 1.Arquitectura de microservicios

La **arquit ectura de microservicios** surge como alternativa al estilo **monolítico**, donde las aplicaciones empresariales se construyen como una unión de tres partes: cliente, manejo de datos y servidor con lógica de dominio. Aunque este enfoque puede ser exitoso, los constantes cambios en el modelo de negocio hacen que la aplicación crezca, se complique su mantenimiento y su escalabilidad incremente la capacidad de toda la aplicación en lugar del segmento que lo requiere.

Para enfrentar este problema, la arquitectura de microservicios propone construir una aplicación como un conjunto de **servicios independientes**, incluso en diferentes lenguajes y equipos, lo que permite escalar solo el servicio que lo necesita. Ejemplos como Amazon, eBay y Netflix adoptaron este enfoque. Cada servicio tiene su propia **arquitectura hexagonal** y base de datos independiente, lo que asegura bajo acoplamiento y diversidad tecnológica.



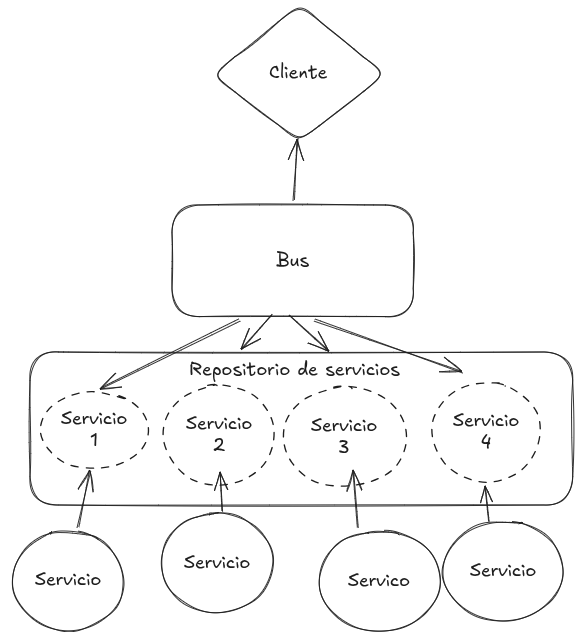
# Reflexión:

La arquitectura de microservicios no solo representa una solución efectiva para proyectos de gran escala, sino que, gracias a la independencia y flexibilidad entre servicios, permite reutilizar microservicios de un proyecto en otro, lo que ayuda a reducir el tiempo y los costos de desarrollo. Además, fomenta la creación de soluciones dinámicas y adaptables a los cambios en cualquier tipo de proyecto.

# Bibliografía:

Contreras, D. A. B. (2018). Arquitectura de microservicios. *Tecnología Investigación y Academia*, *6*(1), 36-46.

# 2.Arquitectura orientada a servicios

La arquitectura de software es una parte crucial de la ingeniería de software, encargada de estudiar, analizar y dibujar el mapa completo de un sistema. Se concentra mucho, en las conexiones entre sus piezas principales.  
En este ámbito, aparece la Arquitectura Orientada a Servicios (SOA) como un enfoque que une la TI con las exigencias de la empresa. SOA permite fusionar sistemas diversos, utilizando servicios independientes, estos se publican, encuentran y activan gracias a interfaces normales, sin importar el lenguaje o la plataforma empleada.  
El corazón de SOA late gracias a servicios autónomos que hablan a través de un bus de servicios y un repositorio. Facilita la reutilización, la integración y la gran flexibilidad. Hay servicios básicos, de negocio, de procesos o públicos, usando, frecuentemente, servicios web que usan estándares como XML, SOAP y WSDL.  
De esta manera, la productividad recibe un gran empujón, además de la seguridad y la flexibilidad organizacional frente a las transformaciones.

# Reflexion:

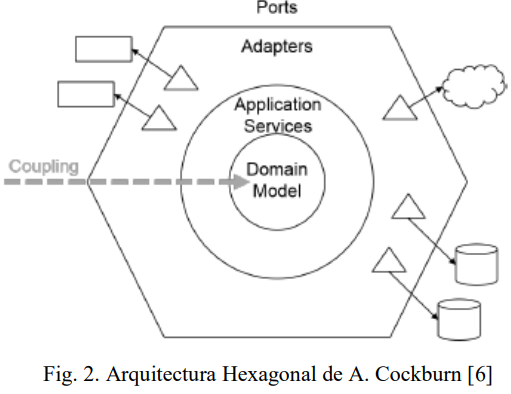
La arquitectura orientada a servicios puede reflejar un trabajo en equipo, cada persona representa un servicio el cual trabaja y funciona independientemente pero uniendo 2 o mas, se puede construir o hacer funcionar algo. El bus puede ser como un líder, el cual es el que solicita o requiere esos servicios para unificarlo o manejarlo para que se pueda usar.

# Bibliografía:

Martín, Y. E. (2012). Arquitectura de software. Arquitectura orientada a servicios. *Serie Científica de la Universidad de las Ciencias Informáticas*, *5*(1), 1-10.

# 3.Arquitectura hexagonal

La arquitectura hexagonal es un patrón estructural que establece entradas y salidas en los bordes del diseño, permitiendo intercambiar manejadores sin cambiar el código del núcleo. De esta forma, la aplicación puede ser manejada por usuarios, programas, pruebas automatizadas o scripts por lotes, y desarrollada de manera aislada de dispositivos y bases de datos en tiempo de ejecución.  
A diferencia de la arquitectura en capas, la arquitectura hexagonal permite que la interfaz de usuario también se pueda intercambiar. El modelo resultante se compone del modelo de dominio, los servicios de aplicación, los puertos y los adaptadores. Cada lado del hexágono representa un puerto concreto, aunque en la práctica puede haber más puertos con su correspondiente adaptador.  
El modelo de dominio, como núcleo, es independiente de los servicios de aplicación y adaptadores. Esto facilita cambiar frameworks o herramientas escribiendo un nuevo adaptador. El interior del hexágono se mantiene libre de cuestiones tecnológicas, y la comunicación se da mediante interfaces llamadas puertos.



Reflexion:

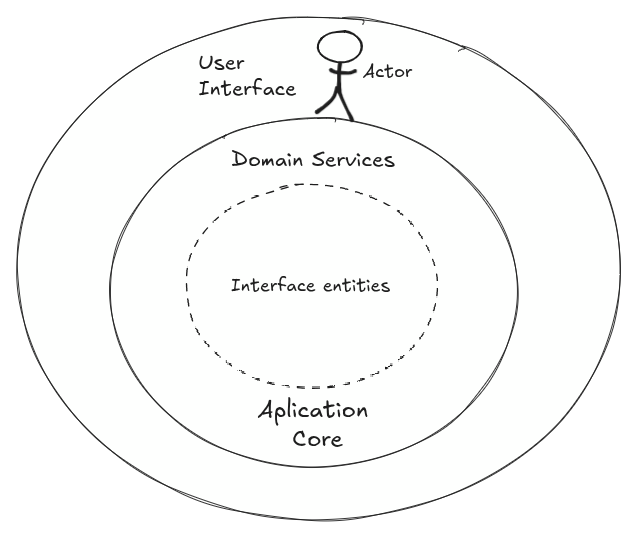
La arquitectura hexagonal propone que en lugar de construir sistemas dependientes de frameworjs, base de datos o interfaces graficas, planeta que el “corazon debe ser el dominio”, esto quiere decir las reglas y logica de negocio que realmente dan valor,

Bibliografia:

Cambarieri, M., Difabio, F., & García Martínez, N. (2020). Implementación de una Arquitectura de Software guiada por el Dominio. In *XXI Simposio Argentino de Ingeniería de Software (ASSE 2020)-JAIIO 49 (Modalidad virtual)*.

4.Arquitectura onion

La arquitectura onion combina las capas de Domain Driven Desing (DDD), que permite colocar en el centro la aplicación. En ese sentido, cada microservicio diseñado consta de este principio, donde la capada de interfaz de usuario se encarga de la presentación de la información al usuario, lo que permite la menor exposición del usurario. En la capa de dominio, se establece la lógica general del desarrollo de la plataforma web, permitiendo establecer la capa que contiene los datos y lógica de manipulación de los mismo. En la capa de aplicación, se establecen los casos de uso, que permiten la activación por medio de la application core mediante interfaz de usuario. La capa de datos queda la interacción de las bases de datos empeladas en el desarrollo de la plataforma web.

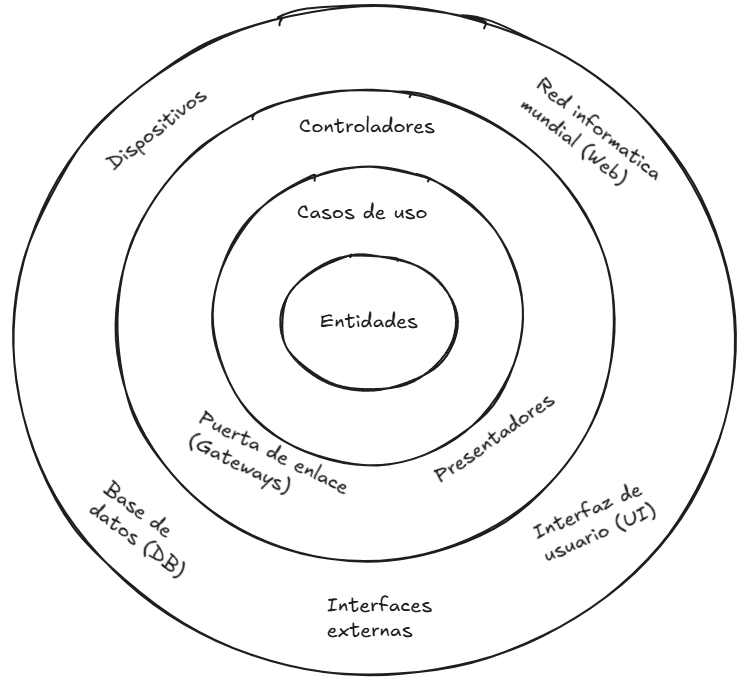


Reflexión:

Bibliografía:

Velásquez Meriño, H. D., & Viloria, J. (2025). Plataforma web con inteligencia artificial para la evaluación de habilidades de pensamiento crítico en estudiantes de la Corporación Universitaria del Caribe CECAR utilizando los TEST PC-B y TEST PC-A.

5. Arquitectura limpia

La arquitectura limpia se puede entender como la combinación de diferentes arquitecturas que buscan separar las preocupaciones a través de una capa de negocio y otra de interfaces. Esta estructura permite que las aplicaciones evolucionen de manera continua y se modifiquen sin afectar otras partes del sistema.  
  
Su funcionamiento se organiza en capas: la capa de presentación, que se encarga de interactuar con la interfaz de usuario utilizando el patrón de diseño MVP; la capa de dominio, que alberga la lógica de negocio y los casos de uso; y la capa de datos, que se ocupa del acceso y almacenamiento de la información. Cada capa se comunica hacia adentro, según la regla de la dependencia.  
  
Así, la lógica de negocio se mantiene independiente de detalles externos como frameworks, servicios o bases de datos. Si se necesita un cambio de tecnología, simplemente se pueden reemplazar los componentes externos sin tocar el núcleo de la aplicación.

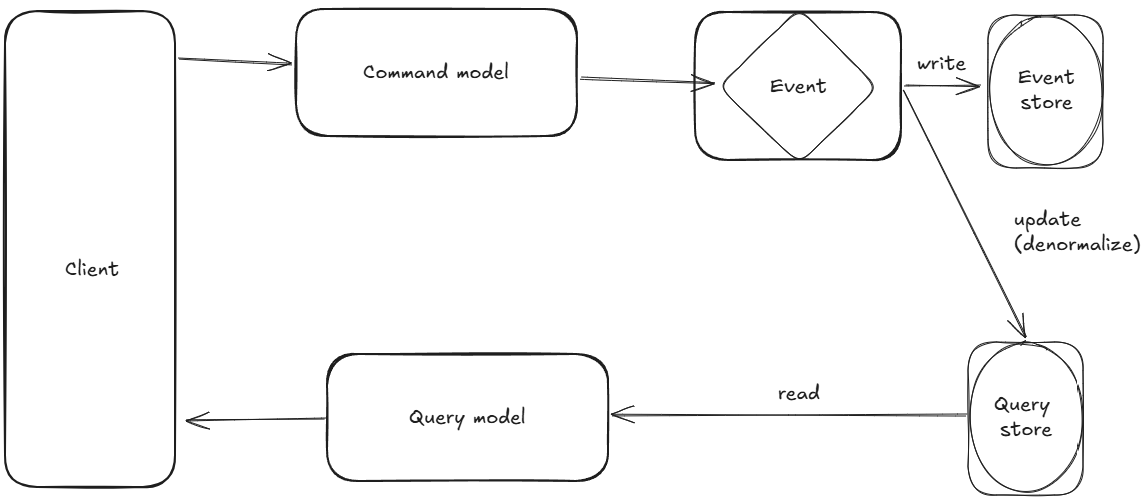
Reflexión:

Bibliografía:

Arias-Orezano, J. F., Reyna-Barreto, B. D., & Mamani-Apaza, G. (2021). Repercusión de arquitectura limpia y la norma ISO/IEC 25010 en la mantenibilidad de aplicativos Android. TecnoLógicas, 24(52), 226-241

6. Arquitectura CQRS

La arquitectura CQRS se fundamenta en el principio de Command Query Separation, donde las operaciones se dividen en dos categorías: los comandos, que modifican el estado de la aplicación, y las consultas, que recuperan información. En este modelo, los comandos y las consultas se implementan como servicios separados dentro de una arquitectura de microservicios y la comunicación entre ellos se realiza a través de mensajes en un entorno orientado a eventos.  
  
El funcionamiento se organiza alrededor de tres tipos de mensajes: comandos, eventos y consultas. Los comandos expresan la intención de cambiar el estado y son manejados por command handlers que publican eventos. Los eventos notifican que una acción ocurrió y son capturados por event handlers y event sourcing handlers, los cuales actualizan el estado de los agregados y las proyecciones. Finalmente, las consultas solicitan datos y se gestionan con query handlers que acceden a los modelos de lectura.  
  
De esta manera, el núcleo del sistema se organiza en agregados, que representan entidades funcionales procesadas en un límite consistente. Los cambios se registran como eventos que mantienen sincronizados el almacén de escritura y el de lectura.



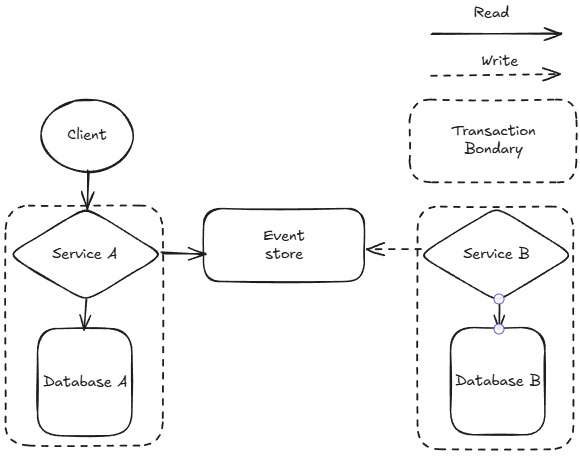
Reflexion:

Bibliografia:

Kindson, M., & Péter, M. (2023). A simplified approach to distributed message handling in a CQRS architecture. *Acta Polytechnica Hungarica*, *20*(4).

7.Arquitectura Event Sourcing

Los comandos son objetos que representan las tareas o acciones básicas de nuestro sistema. Cuando se lanza un comando, este es capturado por un CommandHandler, el cual a su vez lanza el evento o los eventos derivados de esta acción.  
Estos eventos son objetos que codifican la información del evento producido por la acción que acaba de ocurrir. En el subsistema de escritura, los eventos son capturados en el propio agregado por los EventSourcingHandlers los cuales se encargan de persistir dichos eventos en el EventStore.  
Los agregados o aggregates son entidades que representan el estado de la información ante cada evento que las modifique. Cada evento que provoca un cambio sobre un agregado crea una especie de snapshot del estado del agregado en ese momento, por lo que podemos ver todos los eventos en orden cronológico que han modificado el objeto desde su creación hasta el momento actual.



Reflexion:

Bibliografia:

Ropero Morales, F. J. (2021). Aplicación de los patrones CQRS y Event/Sourcing en el desarrollo de un portal inmobiliario.

8.Arquitectura reactiva

La arquitectura reactiva se basa en cuatro principios fundamentales: Responsividad, Resiliencia, Elasticidad y Orientación a Mensajes, los cuales son esenciales para garantizar sistemas que se adapten a demandas de alta carga y baja latencia. Estos principios permiten que un sistema sea reactivo, respondiendo eficientemente a eventos y ajustándose a condiciones cambiantes, como la sobrecarga o la pérdida de disponibilidad.

En una arquitectura reactiva, el flujo de datos se maneja a través de flujos reactivos (como *Mono* y *Flux* en el caso de Java con Reactor), lo que permite que las operaciones se realicen de manera asíncrona, no bloqueante, y sin la necesidad de esperar respuestas de procesos anteriores. Esto mejora la resiliencia y el rendimiento general del sistema, al permitir que los recursos se gestionen de forma más eficiente.

Además, los sistemas reactivos están diseñados para ser distribuidos, con un enfoque de comunicación asíncrona y basada en mensajes, donde los componentes interactúan mediante eventos y comandos. Esto reduce el acoplamiento entre componentes y permite que el sistema sea más flexible y escalable, además de mejorar su tolerancia a fallos mediante el manejo adecuado del flujo de datos y el control de carga (backpressure).