Universidad Católica Boliviana “San Pablo”
Facultad de Ingeniería
La Paz - Bolivia

ARQUITECTURA DE SOFTWARE
SIS-311

INVESTIGACIÓN DE ARQUITECTURA DE SOFTWARE

Realizado por:

-Johan Marco Quispe Rodríguez

SEMESTRE 2-2024

ARQUITECTURA DE SOFTWARE

1. INTRODUCCIÓN

   La arquitectura de software no es solo un diseño técnico, es denominada como la base que establece cómo un sistema debería organizarse para cumplir con sus objetivos y enfrentar los desafíos del mundo real. Desde la gestión de la complejidad hasta la garantía de escalabilidad y mantenimiento, la arquitectura enlaza las necesidades del negocio con las soluciones tecnológicas. Según Bass, Clements y Kazman en “Software Architecture in Practice”, es crucial para construir sistemas exitosos, tanto técnica como estratégicamente. Este documento se encargará de investigar conceptos fundamentales, desde modelos arquitectónicos hasta patrones actuales como microservicios, para diseñar sistemas más flexibles y sólidos.

2. OBJETIVO

   El objetivo del presente documento es comprender cómo los principios de arquitectura de software ayudan a diseñar sistemas seguros, flexibles, escalables y fáciles de mantener. Este análisis busca explicar técnicas y patrones arquitectónicos que faciliten la descomposición de sistemas complejos en soluciones más manejables. Esto permitirá al lector visualizar y aplicar conceptos como el diseño basado en el dominio o la arquitectura hexagonal en problemas reales.

   Así mismo se busca plantear un esquema de arquitectura basado en micro-servicios.

3. ARQUITECTURA DE SOFTWARE

   La arquitectura de software define la estructura organizacional de un sistema, incluyendo sus componentes, relaciones y principios de diseño. Según Bass, es el arte de tomar decisiones estratégicas que afectan el éxito del software a largo plazo. Este apartado introduce el tema como la combinación de decisiones técnicas y de negocio, esenciales para crear sistemas que respondan al cambio continuo.

   Para comprender más a profundidad, la arquitectura de software llega a presentar diferentes modelos o tipos de esquemas, los cuales cada uno llega a tener su ventaja ante casos específicos.

   En vista general una arquitectura de software tiene objetivos a abarcar, como:

   - Ofrecer una vista general más integral del sistema al que se está implementando
   - Tener un flujo de negocios mejor establecido, ayudando a tener un mejor control de la funcionalidad de los requerimientos a resolver.

- Facilitar el proceso, modularizando de mejor manera los procesos, asegurando así la escalabilidad.

3.1. PROCESO DE ARQUITECTURA DE SOFTWARE

Diseñar una arquitectura no significa simplemente elegir tecnologías: es un proceso iterativo que establece un puente entre los requisitos del negocio y las posibles soluciones técnicas. En “Software Architecture in Practice”, se revela que implica establecer prioridades, sopesar alternativas e incorporar información para asegurarse de que la solución sea de alta calidad y sostenible. Esta entrada se trata de cómo dirigir sistemáticamente ese proceso.

Otro punto importante es que el proceso arquitectónico implica analizar y considerar múltiples alternativas. No siempre hay una única respuesta correcta o una forma única de diseñar un sistema, sino un conjunto de decisiones independientes. Aquí entra en juego la experiencia y conocimiento del arquitecto para considerar factores como costos, tiempos de implementación y el impacto futuro de cada elección. A menudo, estas decisiones requieren compromisos, donde priorizar un aspecto puede implicar sacrificar otro en mayor o menor medida.

Además, incorporar información implica no solo consultar a los stakeholders del negocio pero también al equipo técnico. Los desarrolladores abundan los detalles que pueden convertir una solución en otra. Por eso, no es un ejercicio técnico, secamente puesto, dirigir este proceso. Eso es, también, la capacidad de reunir un equipo y construir un compromiso entre necesidades y limitaciones.

3.2. MODELOS Y VISTAS ARQUITECTURALES

Para entender un sistema complejo, necesitamos perspectivas claras. Los modelos y vistas arquitecturales son herramientas esenciales para representar diferentes ángulos de un sistema: desde lo técnico hasta lo funcional. Aquí exploramos cómo estas representaciones ayudan a comunicar ideas y detectar problemas antes de que aparezcan.

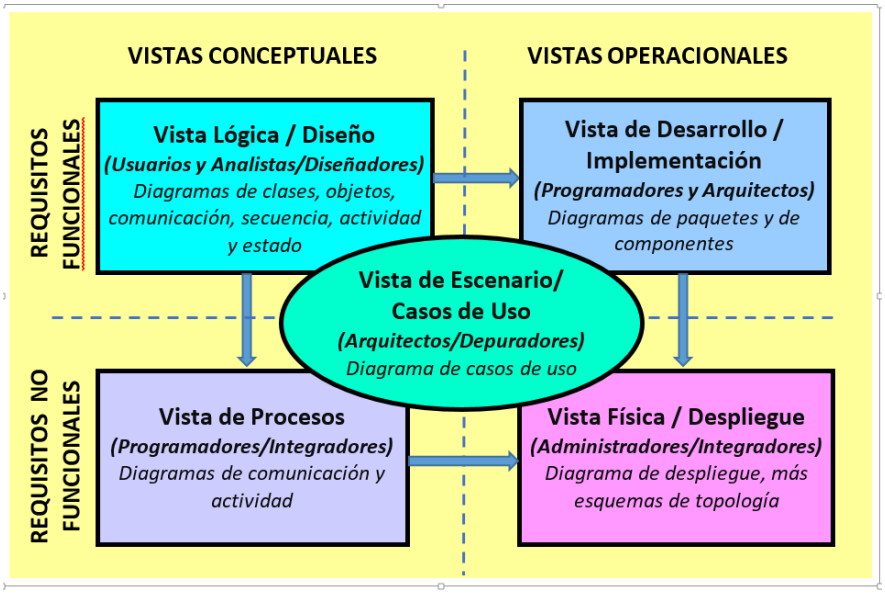


Figura 1.- Tabla de vistas

3.3. DISEÑO BASADO EN EL DOMINIO (DDD)

Eric Evans, en “Domain-Driven Design” o Diseño Basado en el Dominio, afirma que para resolver problemas técnicos de manera efectiva, debemos priorizar el entendimiento a fondo del negocio. DDD significa enfocar el diseño de software en el “lenguaje” y las necesidades del dominio, integrar a los equipos técnicos y no técnicos en una visión compartida. Es especialmente importante cuando tratamos con sistemas complejos y diversos.

El valor del DDD aparece cuando enfrentamos sistemas complejos, donde las reglas del negocio pueden ser confusas o ser modificadas rápidamente. Al dividir el sistema en "subdominios" claros y definidos nos ayuda a desarrollar modelos que representen estas áreas, podemos gestionar mejor la complejidad y construir un software que evolucione junto con el negocio. Este enfoque, como menciona Evans, no solo mejora la comunicación, sino también el diseño general del sistema al mantener cada parte enfocada, comprensible y a la vez comunicada con las otras.

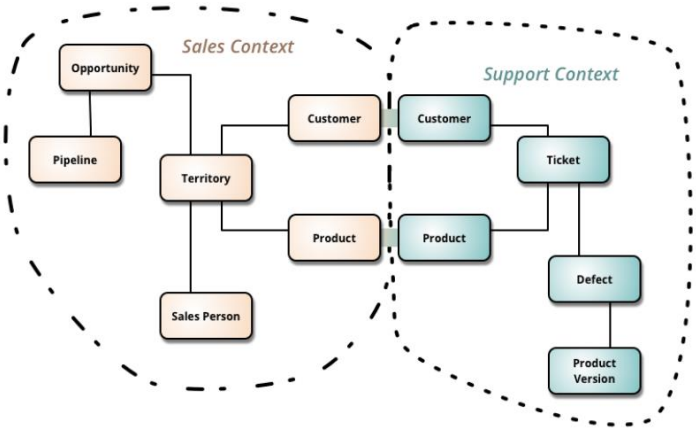


Figura 2.- Bounded Context

Por ejemplo en la imagen se observar cómo 2 áreas diferentes pueden interactuar a pesar de tener modelos y enfoques que no son iguales, por lo que nos ayuda a comprender cómo el DDD llega a facilitar el trabajo conjunto, brindando escalabilidad y mantenimiento a largo plazo.

3.4. ARQUITECTURA HEXAGONAL

La arquitectura hexagonal, introducida por Alistair Cockburn y popularizada por autores como Robert C. Martin, propone una estructura que conecta el núcleo del sistema (lógica de negocio) con el exterior a través de interfaces claras. En *Clean Architecture*, se explica cómo este enfoque facilita pruebas y reduce dependencias, creando sistemas más modulares y flexibles.

La idea básica es construir una capa interna que represente la lógica de negocio pura: las reglas, cálculos y decisiones que realmente definen lo que el software debe hacer. Luego, esta capa se conecta con el "mundo exterior" (interfaces de usuario, servicios externos, almacenamiento) a través de adaptadores bien definidos. Estos adaptadores funcionan como traductores, asegurando que la lógica del negocio nunca tenga que preocuparse por los detalles técnicos de cómo se implementan esas conexiones.

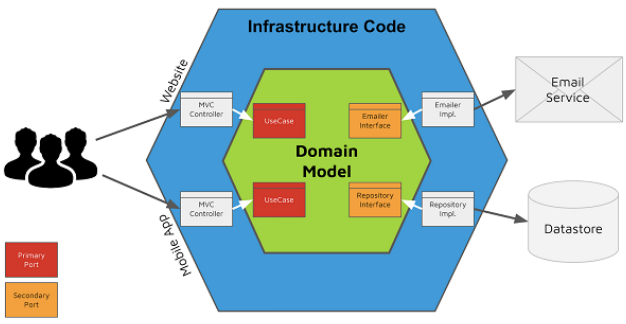


Figura 3.- Arquitectura Hexagonal

Una de las grandes ventajas de este enfoque es que facilita muchísimo las pruebas. Al poder aislar el núcleo del sistema, puedes probar la lógica de negocio sin tener que preocuparte por si la base de datos funciona o si hay conexión a internet. Además, reduce el acoplamiento entre las partes del sistema, lo que hace más fácil reemplazar o actualizar componentes externos sin romper todo el sistema.

3.5. MICROSERVICIOS

Los microservicios no son solo una moda; representan una evolución en cómo pensamos y diseñamos software. Sam Newman, en *Building Microservices*, los describe como pequeñas aplicaciones independientes que trabajan juntas como un sistema más grande. Este patrón permite a los equipos moverse rápido, escalar componentes de forma independiente y adaptarse al cambio con mayor agilidad.
Los microservicios no son solamente una moda; son una revolución en cómo pensamos y manejamos sistemas de software complejos. Este paradigma surge en respuesta directa al problema generado por las aplicaciones monolíticas, convirtiendo todas las funcionalidades en un solo código base.

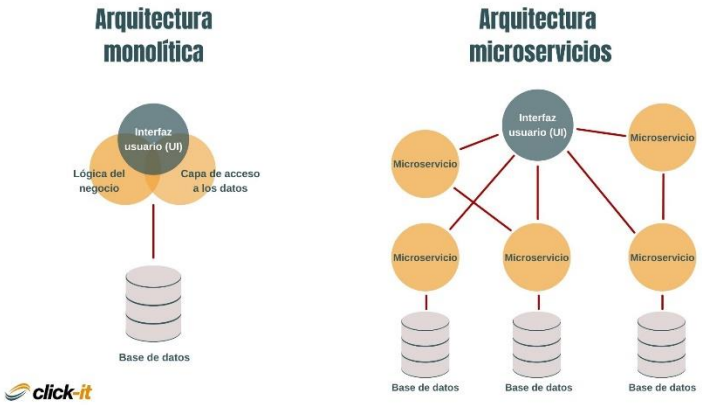


Figura 4.- Diferencia entre una arquitectura monolítica y una de microservicios

Como lo describe Sam Newman en “Building Microservices”, los microservicios delimitan un sistema en múltiples aplicaciones independientes que tienen como objetivo resolver al mismo tiempo específicamente lo que el dominio pide. Desarrolla “piezas” autónomas que se comunican entre sí para manejarse como masaje de sistema. Uno de los beneficios de este modelo es la independencia de los componentes. Porque separando las funcionalidades las desarrollas, desplaza y como puede escalarse. Por ejemplo, si alguna del funcionalidades en el sistema empieza a recibir más usuarios de los esperados, como sería el caso del servicio de procesamiento de pagos, solo su escalado puedes realizar y ahorrar recursos no

necesarios para sus demás procesos. Esto mejora la eficiencia de uso de recursos, reduce costos y simplifica la planificación de interfaz.

3.6. ESTRATEGIAS DE DESCOMPOSICIÓN

Migrar a microservicios no es un proceso automático; requiere estrategias inteligentes para dividir una aplicación monolítica en servicios manejables.

Se pueden dividir por diferentes tipos de estrategias, y es cuestión del arquitecto de software qué estrategia es más conveniente.

3.6.1. Descomposición por capacidades empresariales

En esta estrategia de descomposición, lo que haces es dividir tu aplicación monolítica en microservicios autónomos basándote en las capacidades empresariales que has identificado. Esta descomposición se basa en conceptos de modelado de la arquitectura empresarial, donde una capacidad empresarial es lo que tu empresa hace para generar valor. Por ejemplo, en una aplicación de tarjetas de crédito, puedes tener capacidades empresariales como el marketing de productos de tarjetas de crédito, la apertura de cuentas de tarjetas de crédito, la activación de tarjetas y la generación de estados de cuenta.

3.6.2. Descomposición por subdominios

La descomposición por subdominios está estrechamente vinculada con los principios del Diseño Basado en el Dominio (DDD). En este enfoque, el sistema se divide según las diferentes áreas funcionales del negocio, conocidas como subdominios. Cada subdominio representa una parte específica del negocio que tiene reglas y procesos propios.

3.6.3.Descomposición por transacciones

La descomposición por transacciones se centra en identificar los procesos críticos que involucran múltiples pasos o interacciones dentro del sistema. Cada transacción representa una unidad de trabajo que tiene un principio y un fin claro, y se busca dividir la aplicación monolítica en servicios independientes que puedan gestionar partes específicas de esas transacciones.

3.7. TOPOLOGÍAS

La manera en que organizamos equipos y sistemas afecta directamente la efectividad de nuestra arquitectura. Newman explica cómo las topologías definen quién es responsable de qué, asegurando que equipos y servicios puedan escalar juntos. Aquí exploraremos las estructuras más comunes y sus beneficios.

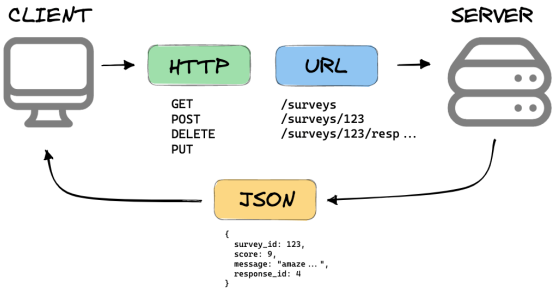
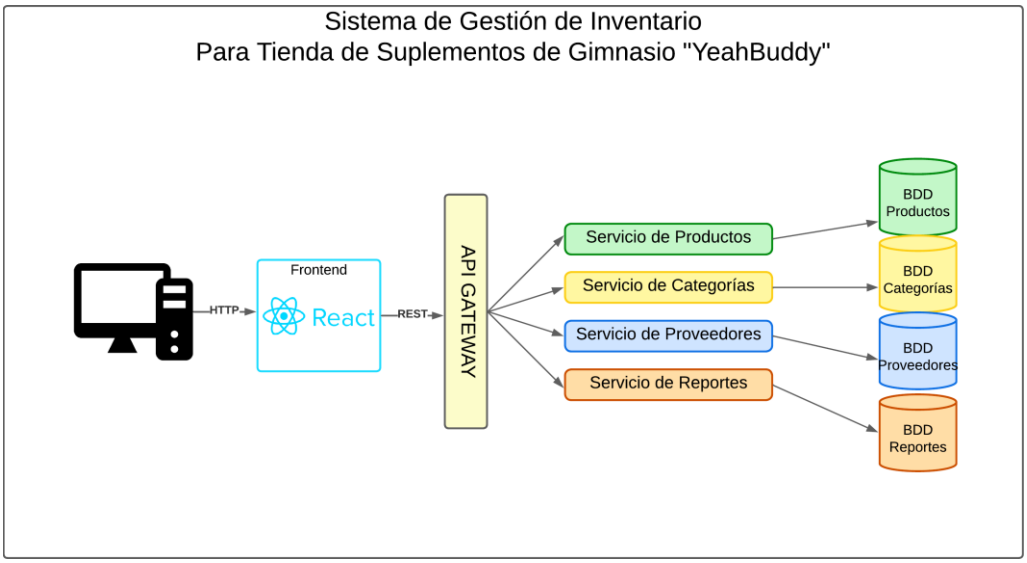


Figura 5.- Ejemplo de topología

4. EJEMPLO ESQUEMÁTICO DE UN SISTEMA BASADO EN MICROSERVICIOS

En este punto se muestra un breve ejemplo de cómo se puede plantear un diseño de arquitectura de micro servicios orientado a un Sistema de gestión de inventario para tienda de suplementos de gimnasio



5. CONCLUSIONES

La arquitectura de software revela que no se trata solo de tecnología, sino de decisiones estratégicas que impactan directamente en el éxito de un sistema. Reflexionaremos sobre cómo los conceptos aprendidos pueden ser aplicados para construir software más robusto, alineado con las necesidades del negocio y listo para los desafíos del futuro. Comprendiendo además que no hay una solución correcta si se trata de definir la arquitectura, se debe de poseer los conocimientos necesarios para decidir qué arquitectura será más óptima para el sistema en cuestión

6. BIBLIOGRAFÍA

- Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.
- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.
- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson.
- Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.
- Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional.
- Richardson, C. (2018). *Microservices Patterns: With Examples in Java*. Manning Publications.