Sesión 02

Dominio & Gobierno de datos

Instructor:

ERICK ARÓSTEGUI

earostegui@galaxy.edu.pe

GALAXY
TRAINING

ÍNDICE

GALAXY
TRAINING

01

Características de una arquitectura de Microservicios

GALAXY
TRAINING

Arquitectura de Microservicios

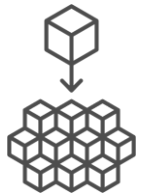
El estilo arquitectónico de microservicio es un enfoque para **desarrollar una sola aplicación como un conjunto de servicios pequeños**, cada uno que se ejecuta en su propio proceso y se comunica con mecanismos ligeros, a menudo una API de recursos HTTP.

Estos **servicios se basan en las capacidades del negocio** y se pueden implementar de forma independiente mediante maquinaria de implementación totalmente automatizada.

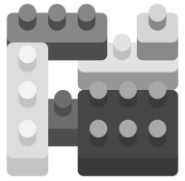
Hay un mínimo de **gestión centralizada de estos servicios**, que puede escribirse en diferentes lenguajes de programación y utilizar diferentes tecnologías de almacenamiento de datos.

Características de una arquitectura de Microservicios

Características de diseño

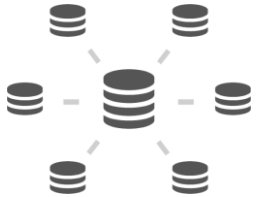


Componente a través de Servicios

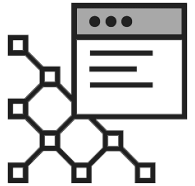


Organizado en torno a las capacidades de negocio.

End Points inteligentes y Pipes tontos



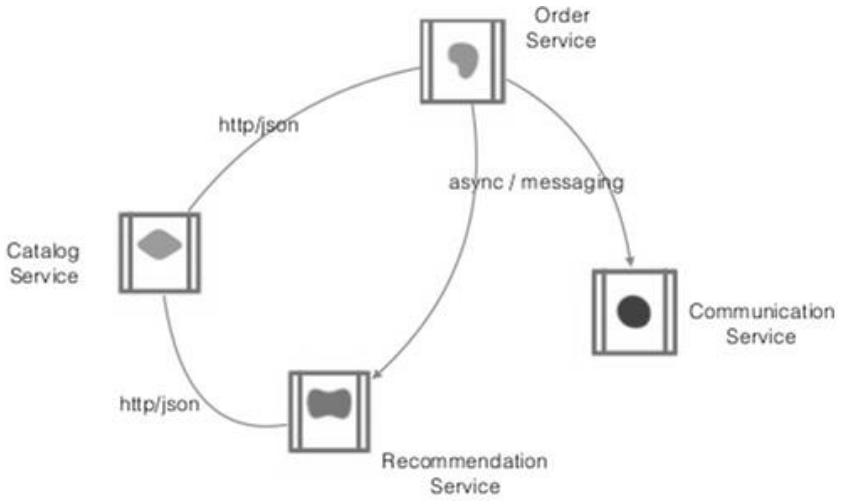
Gestión descentralizada de datos



Automatización de infraestructuras

Diseño para fallos

Componente a través de Servicios

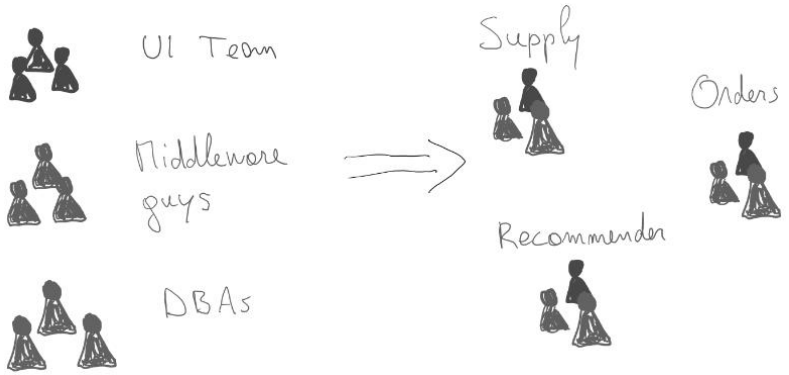


Es mejor utilizar algún paradigma del mundo real, como cómo un usuario se refiere a un “Sistema estéreo". Lo mueven, lo fijan en otro lugar. Mueva, quite y conecte diferentes altavoces.

Básicamente **un componente es algo que es actualizable y reemplazable de forma independiente.**

La diferencia es simplemente que los servicios no se incorporan directamente a su código base, sino que se llaman a través de llamadas remotas.

Organizado en torno a las capacidades de negocio.

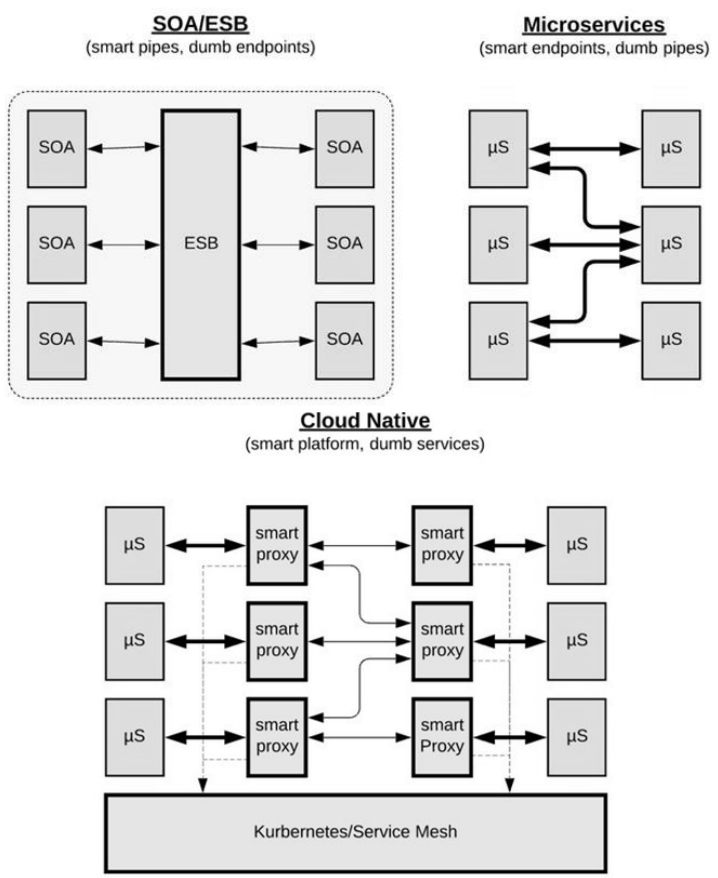


Muchas organizaciones estructuran sus equipos en torno a la tecnología: especialistas en interfaz de usuario, equipo de middleware, DBAs.

Más bien, con los microservicios, las personas deben organizarse en torno a las capacidades empresariales dentro de los equipos multifuncionales: como el "equipo de envío", el "equipo de pedidos“

Microservicios se trata mucho más de la organización del equipo que de la arquitectura de software. La arquitectura y la organización del equipo siempre están muy unidas.

End Points inteligentes y Pipes tontos



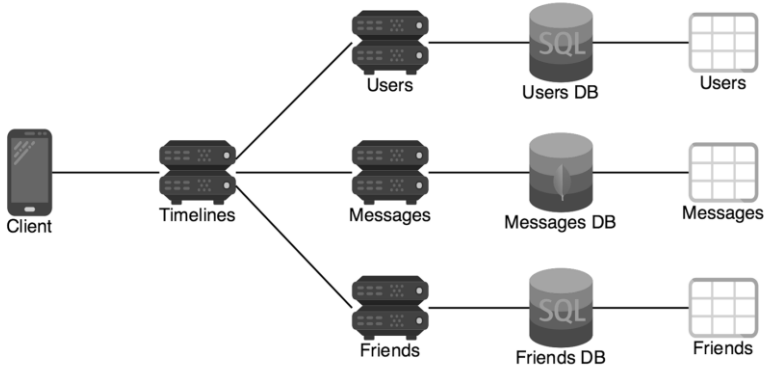
Es una práctica común utilizar la infraestructura de red inteligente como **ESB que contienen lógica** sobre cómo tratar ciertos mensajes de red y cómo enrutarlos.

En su lugar, los microservicios facilitan los pipes tontos y los **Endpoints/Aplicaciones** inteligentes.

El problema es que los pipes inteligentes (es decir, ESB) conducen a problemas con la entrega continua, ya que no se pueden controlar o integrar fácilmente en un pipe grande.

Además, crea dependencias con la propia aplicación, lo que significa que cuando decide actualizar el endpoint o servicio, a menudo también tiene que realizar algún trabajo en el ESB.

Gestión descentralizada de datos

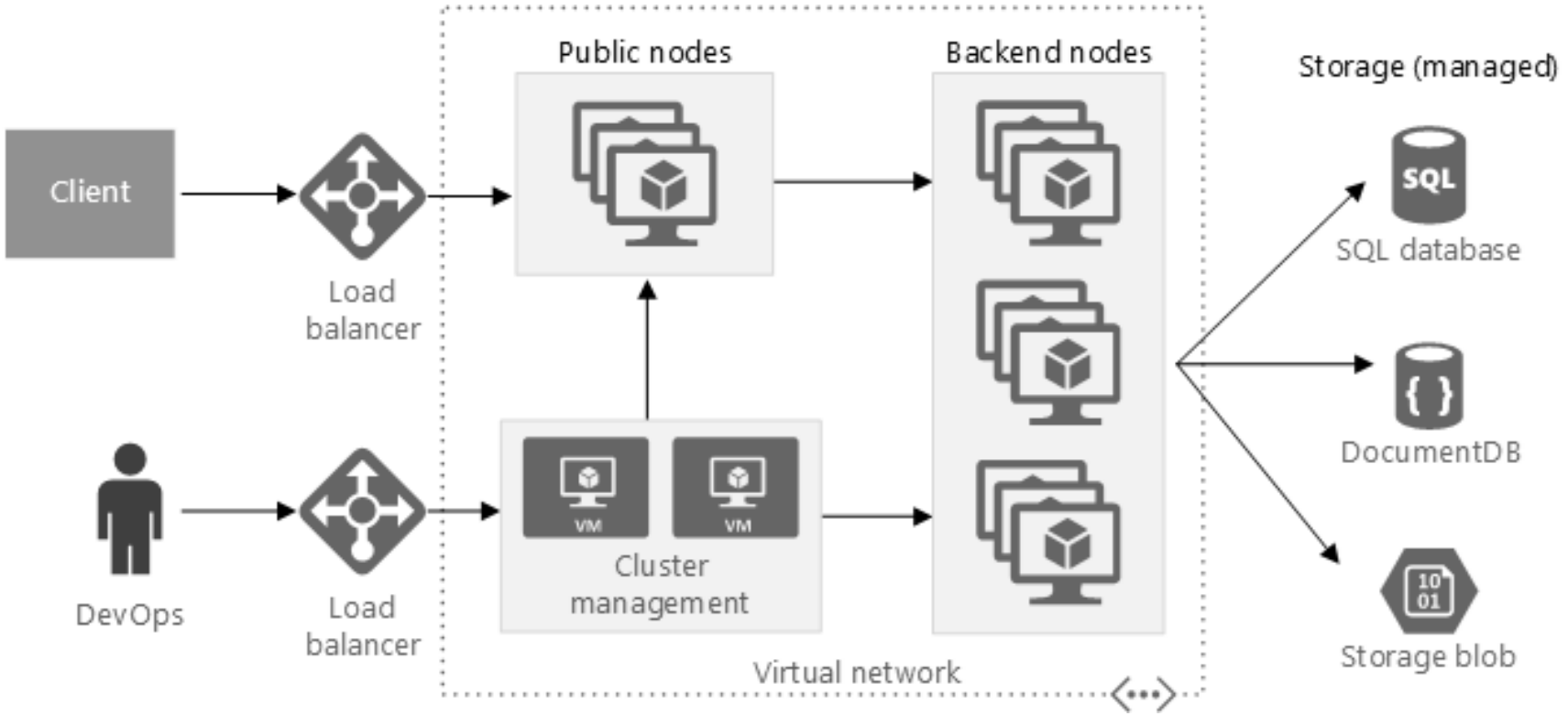


Normalmente, en un **sistema monolito hay una enorme base de datos** donde se almacenan todos los datos. A menudo incluso hay varios monolitos que se conectan a la misma base de datos.

En un enfoque orientado a servicios, **cada servicio obtiene su propia base de datos y los datos no se comparten directamente con otros**. Compartir tiene que pasar por el servicio que envuelve los datos. Esto conduce a una gran **flexibilidad** beneficiosa por parte del servicio, ya que **puede decidir qué tecnología adoptar**, qué sistema DBMS, etc.

GALAXY TRAINING

Automatización de infraestructuras



La entrega continua es una necesidad, así como mecanismos automatizados para el aprovisionamiento de máquinas, para la implementación, pruebas, etc.

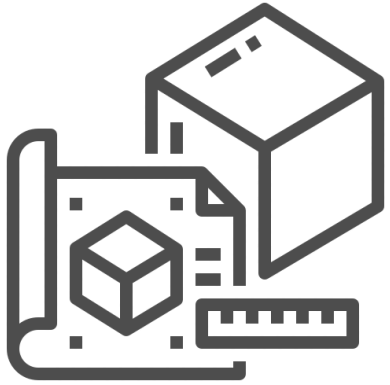
Diseño para fallos

Inevitablemente se tiene que diseñar para el error, ya que **los microservicios fallarán, incluso con frecuencia**.

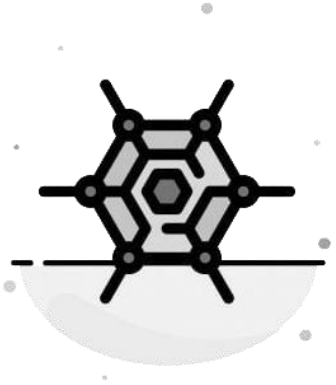
Netflix es famoso por llevar esto al extremo. Tienen un "**Chaos Monkey**" que se ejecuta sobre su sistema de producción durante el día y cierra al azar los servicios.

Aunque muchas personas tienden a abstraer y ocultar llamadas remotas, no puede esperar que funcionen como llamadas normales. **Espere que fracasen en lugar de tener éxito.**

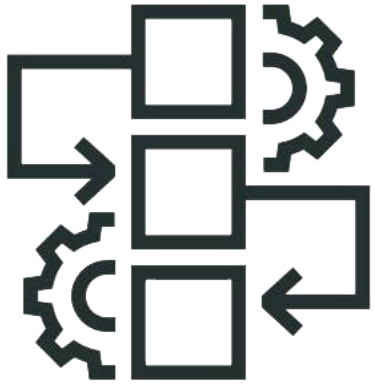
Características de gestión



Productos no Proyectos



Gobernanza descentralizada



Diseño evolutivo

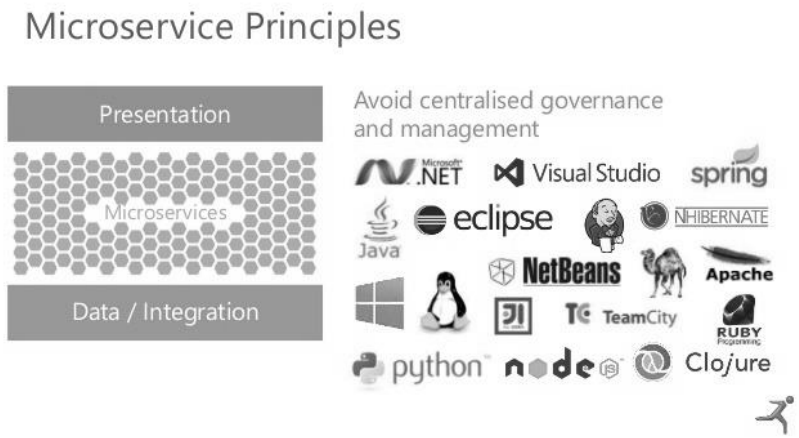
Productos no Proyectos

La mayoría de los esfuerzos de desarrollo de aplicaciones que vemos utilizan un modelo de proyecto: donde el objetivo es entregar algún software que luego se considera completado. Al finalizar, el software se entrega a una organización de mantenimiento y el equipo del proyecto que lo creó se disuelve.

Los defensores de microservicios tienden a evitar este modelo, prefiriendo en su lugar la noción de que **un equipo debe poseer un producto a lo largo de toda su vida útil.**

Una inspiración común para esto es la noción de Amazon de **"usted construye, usted lo ejecuta"** donde un equipo de desarrollo asume toda la responsabilidad por el software en producción. Esto lleva a los desarrolladores al contacto diario con el modo en que su software se comporta en producción y aumenta el contacto con sus usuarios, ya que tienen que asumir al menos parte de la carga de soporte.

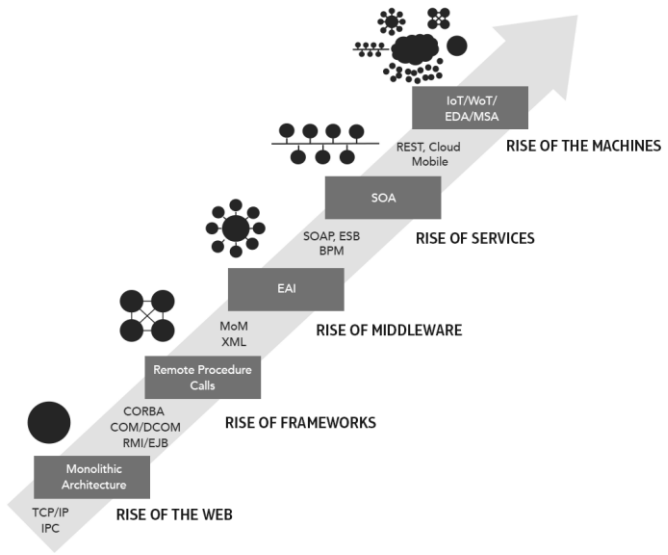
Gobernanza descentralizada



Una de las consecuencias de la **gobernanza centralizada es la tendencia a estandarizarse en plataformas tecnológicas únicas**. La experiencia demuestra que este enfoque es **restrictivo - no todos los problemas son un clavo y no todas las soluciones un martillo**.

En lugar de utilizar un conjunto de estándares definidos escritos en algún lugar en papel prefieren la idea de **producir herramientas útiles que otros desarrolladores pueden utilizar para resolver problemas similares** a los que están enfrentando.

Diseño evolutivo



Cada vez que intenta dividir un sistema de software en componentes, se enfrenta a la decisión de cómo dividir las piezas - ¿cuáles son los principios sobre los que decidimos cortar nuestra aplicación? **La propiedad clave de un componente es la noción de reemplazo independiente y capacidad de actualización**

Este énfasis en la reemplazabilidad es un caso especial de un principio más general del **diseño modular**. Si se encuentra cambiando repetidamente dos servicios juntos, eso es una señal de que deben combinarse.

02

GALAXY
TRAINING

Introducción al patrón Bounded Context

GALAXY TRAINING

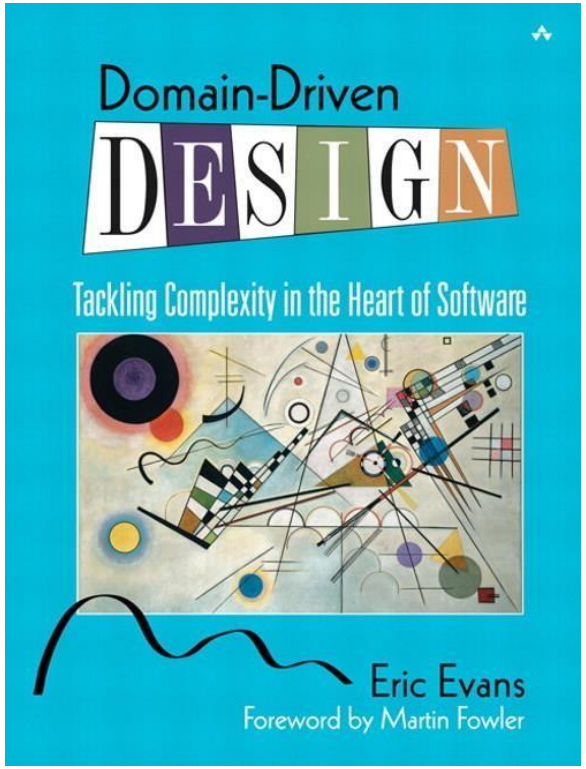
¿Que enfoque utilizar para la descomposición?

TRANSACTION SCRIPT	TABLE MODULE ACTIVE RECORD	DOMAIN DRIVEN DESIGN
Procedural	Orientado a los datos	Orientado al negocio

¿Que enfoque utilizar para la descomposición?

TRANSACTION SCRIPT	TABLE MODULE ACTIVE RECORD	DOMAIN DRIVEN DESIGN
Procedural	Orientado a los datos	Orientado al negocio

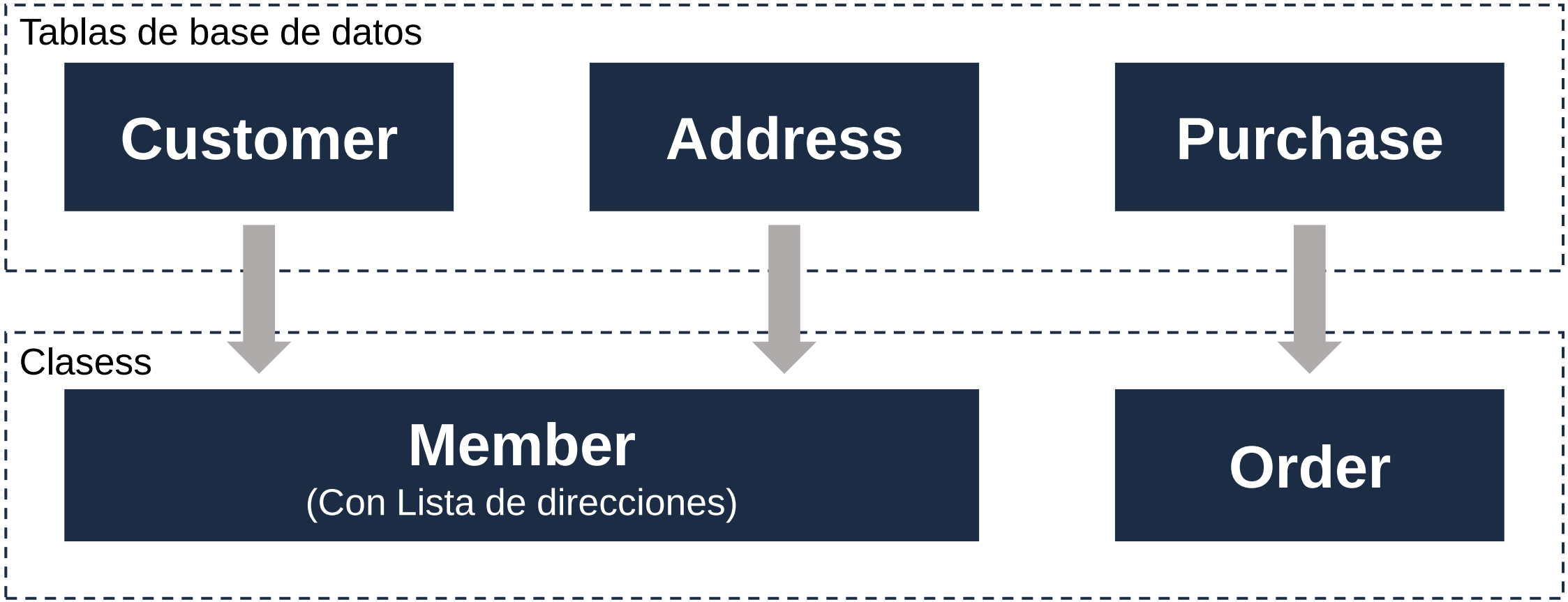
Modelo de Dominio



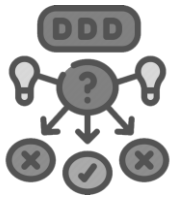
Introducción al patrón Bounded Context

Modelo de Dominio

Modelo de Dominio



Domain Driven Design (DDD)

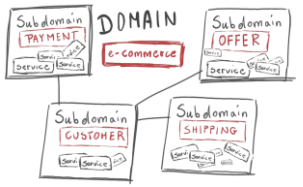


Software que modela dominios del mundo real

(Expertos en dominios y desarrolladores de software)

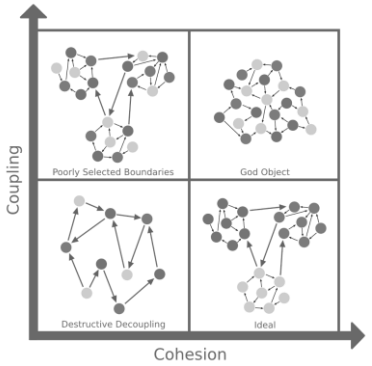
Muchas herramientas y técnicas

Concept of bounded contexts



El dominio consta de múltiples contextos delimitados

(Cada BC representa una función de dominio)



El Bounded context fomenta

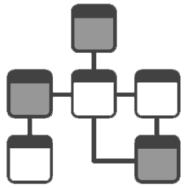
(Bajo acoplamiento, Alta cohesión)

Modelo de Dominio - Ventajas

Gestionar complejidad

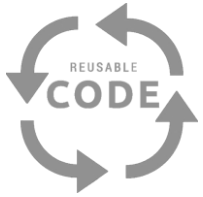
Aprovecha los patrones de diseño

Hablar el lenguaje del negocio



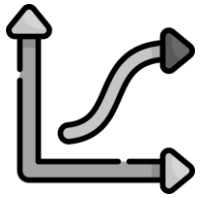
Abstrae el esquema de DB poco bonitas (nombres feos)

Facilitación para equipos grandes

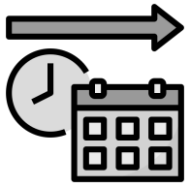


Reutilizable

Modelo de Dominio - Desventajas

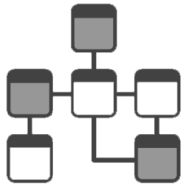


Curva de aprendizaje



Compromiso a largo plazo

Diseño que consume mucho tiempo

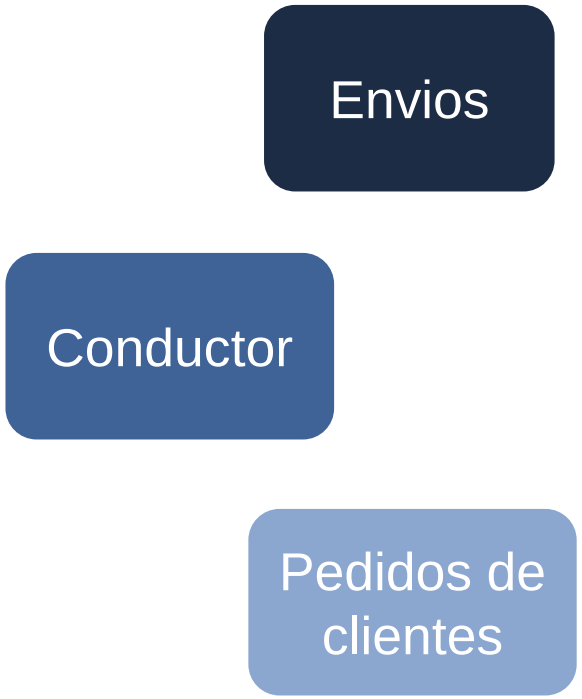


Gastos generales de mapeo de DB

Bounded Context

Una responsabilidad específica impuesta por un límite explícito

Bounded Context

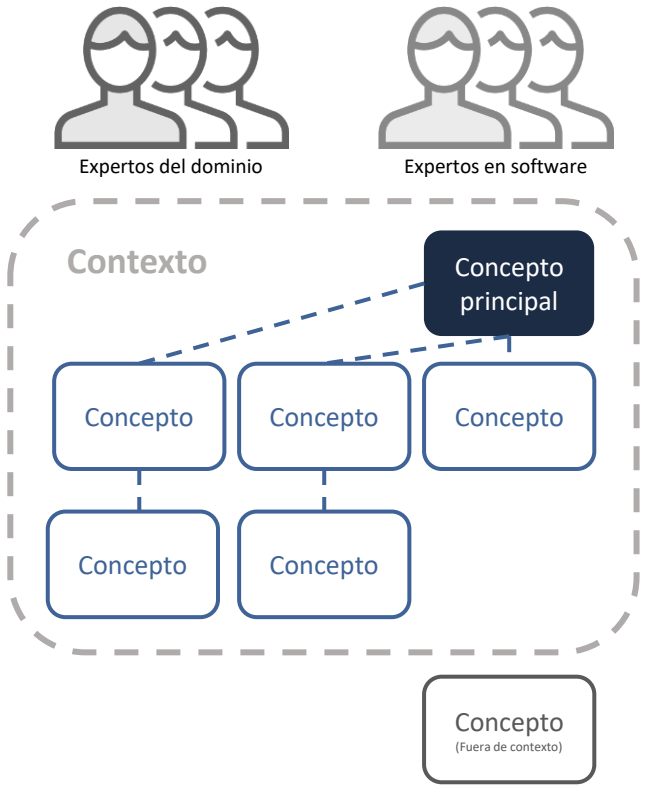


- Identificar el concepto de dominio principal
- Modelos internos (conceptos de apoyo)
- Cada BC tiene una interfaz explícita (entradas y salidas)
- Modelos compartidos para la comunicación
- Microservicio = Bounded Context
  - Pertenece a un equipo
  - Dueño de su propio almacén de datos.
  - Contratos (Interfaces)
- Bounded Context Lógico,
- Contextos externos (Fuera del contexto)

Lenguaje Ubicuo

Lenguaje que pertenece a una función de dominio específica (Bound Context). También utilizado por todos los miembros del equipo para conectar todas las actividades del equipo con el software.

Bounded Context y lenguaje ubicuo



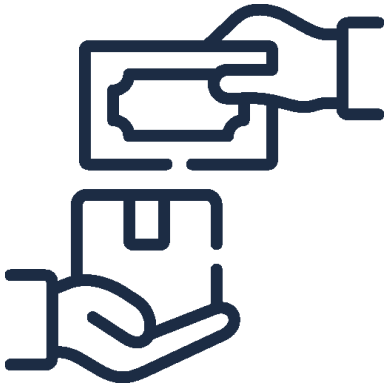
- El concepto principal define el lenguaje
- Lenguaje ubicuo
- Lenguaje principal natural
- Se usa como filtro de Bounded Context
  - Conceptos en contexto
  - Conceptos contexto externos (fuera del contexto).
- Como definir el lenguaje
  - Expertos en dominios
  - Desarrolladores de software

DEMO

GALAXY TRAINING

Contextos sin limites
Enfoque usualmente utilizado en el desarrollo tradicional

Conceptos/aspectos principales del dominio

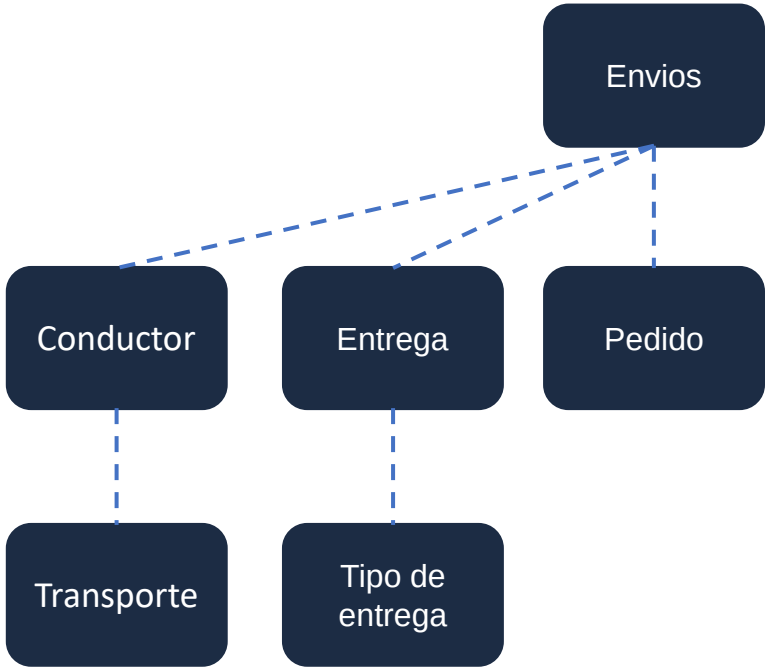


Envíos (Delivery)
Conductores
Pedidos
entregas

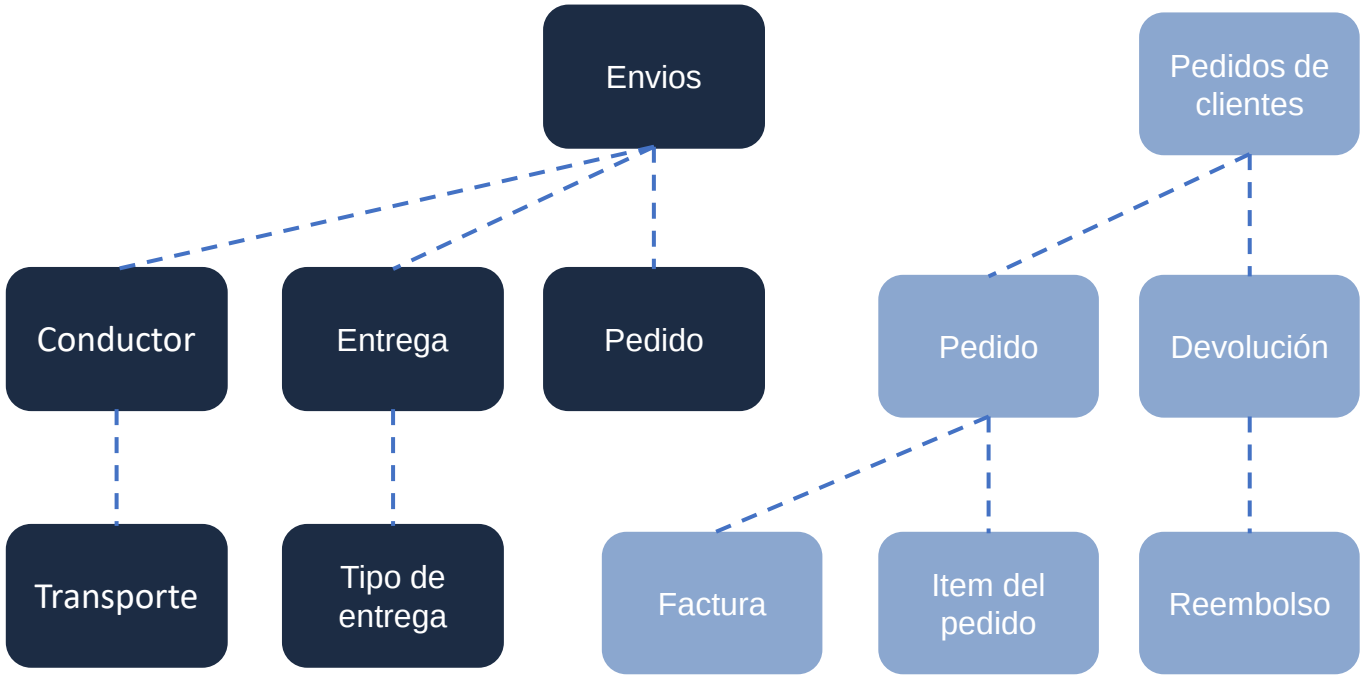
Pedidos de clientes (Customer Orders)
Pedidos
Devoluciones

Conductor (Driver)
Transporte
Turnos
Vacaciones anuales

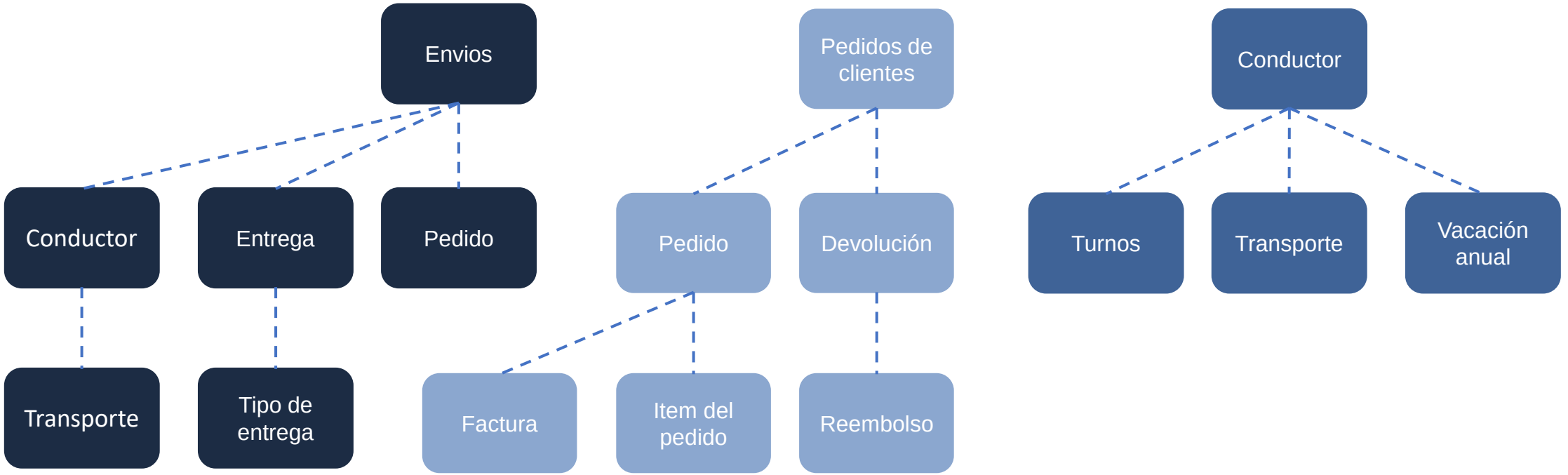
Conceptos principales y conceptos de soporte



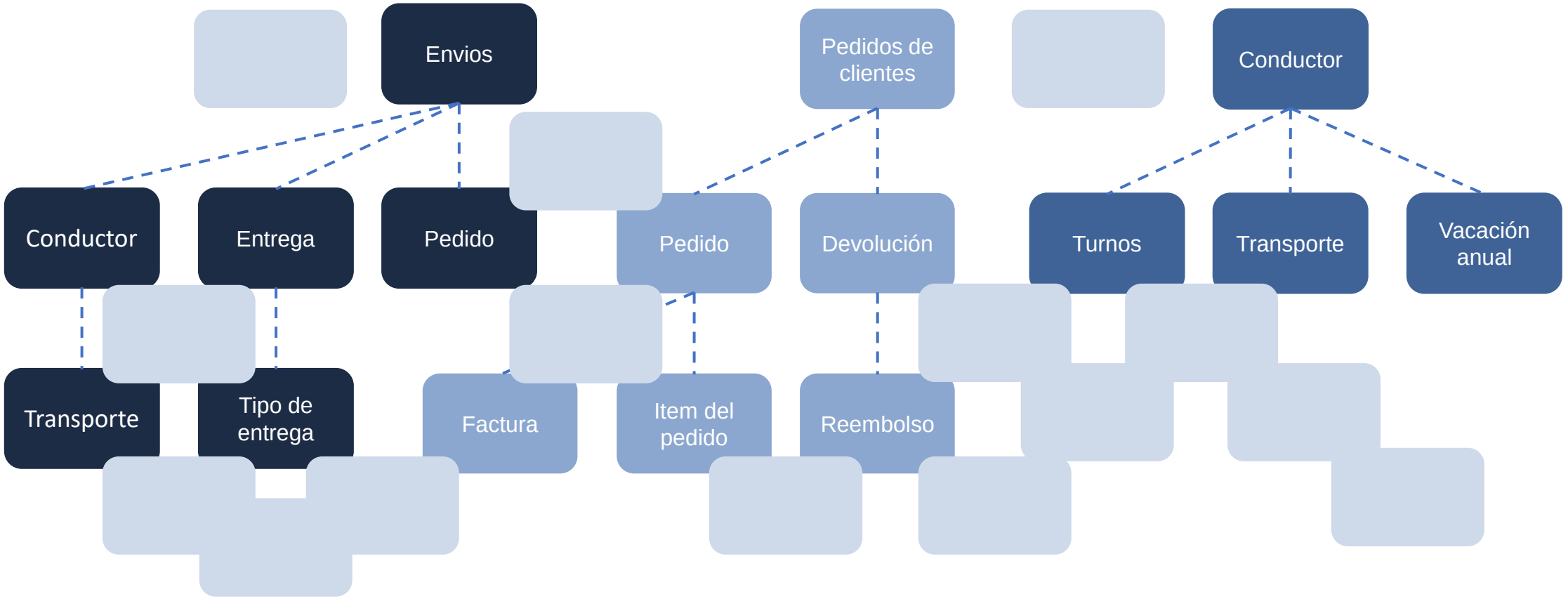
Conceptos principales y conceptos de soporte



Conceptos principales y conceptos de soporte



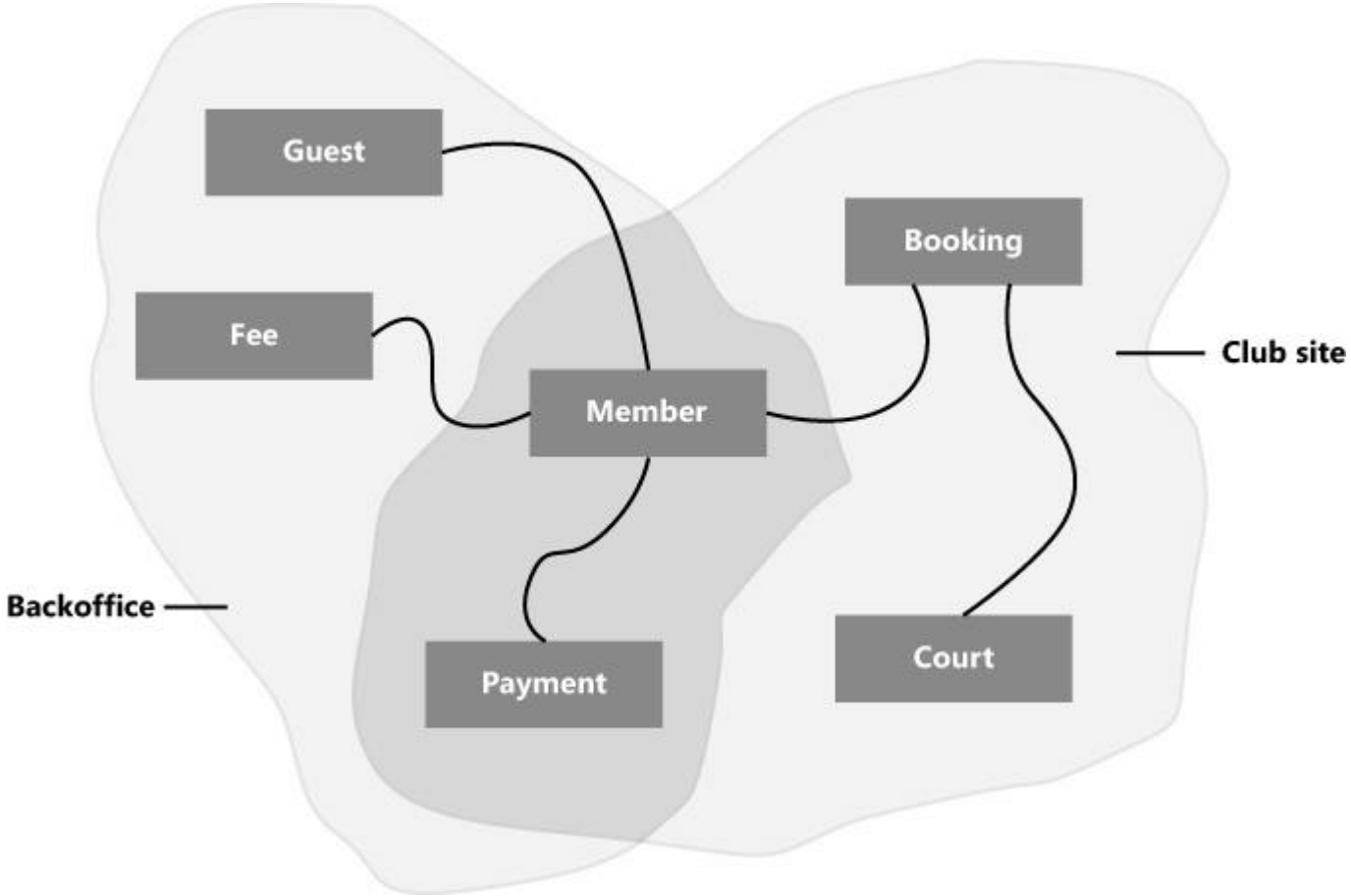
Conceptos principales y conceptos de soporte



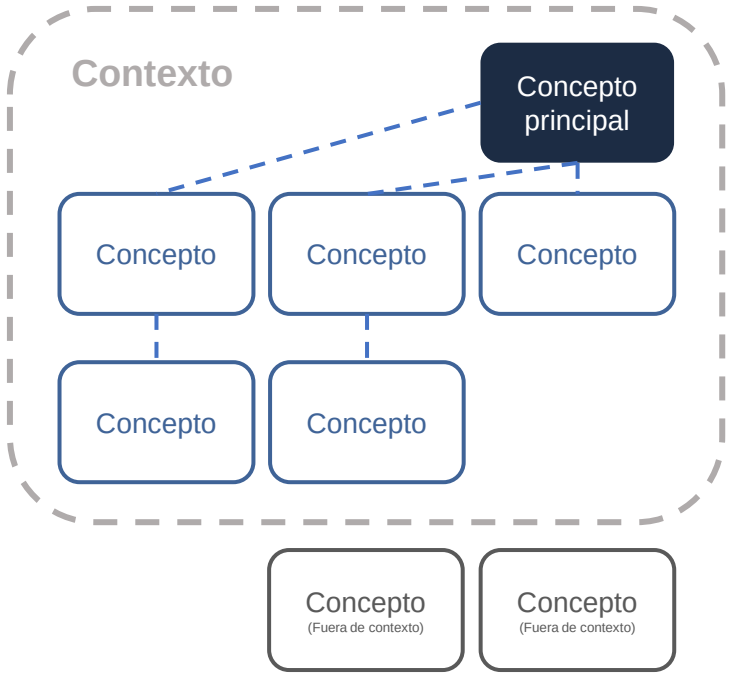
DEMO

GALAXY TRAINING

Uso de Bounded Context para microservicios

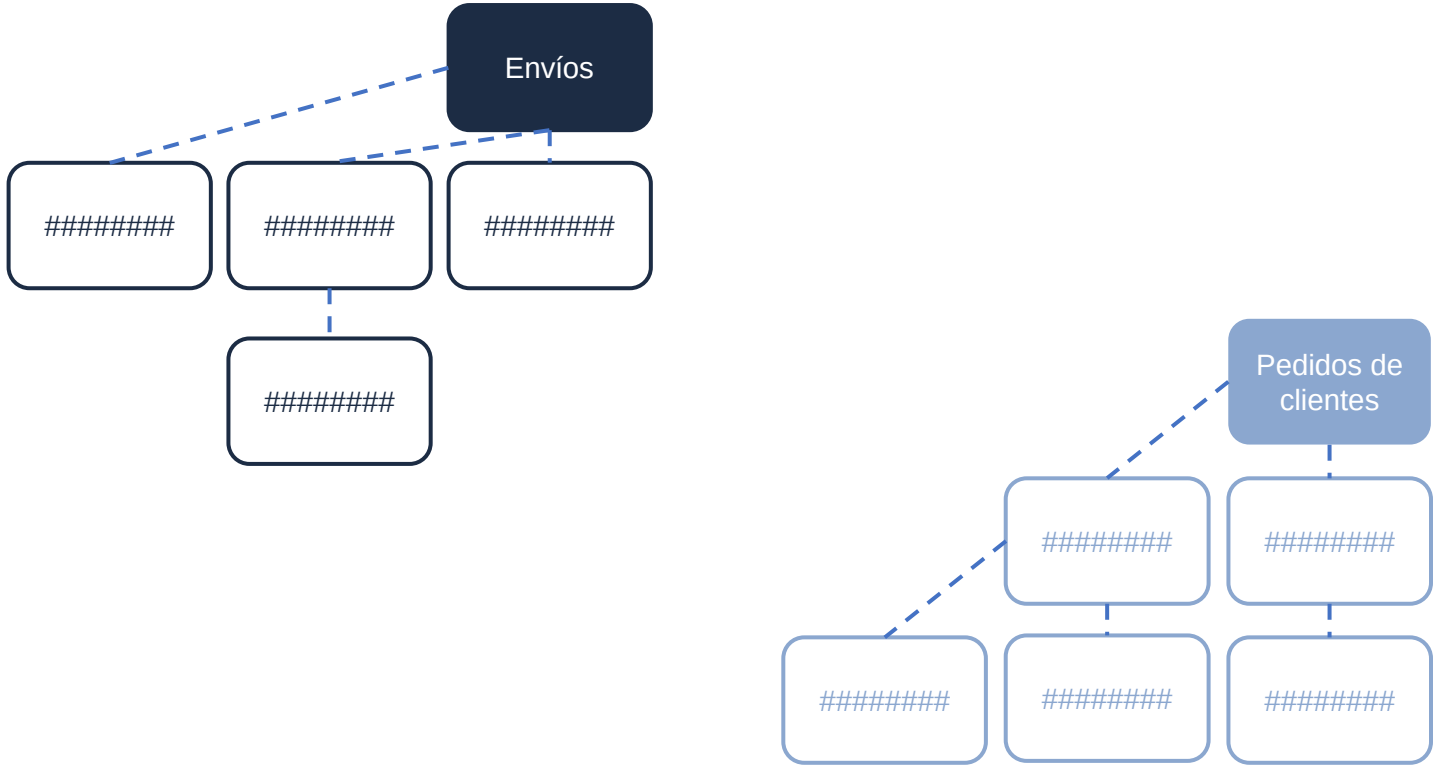


Bounded Context como técnica

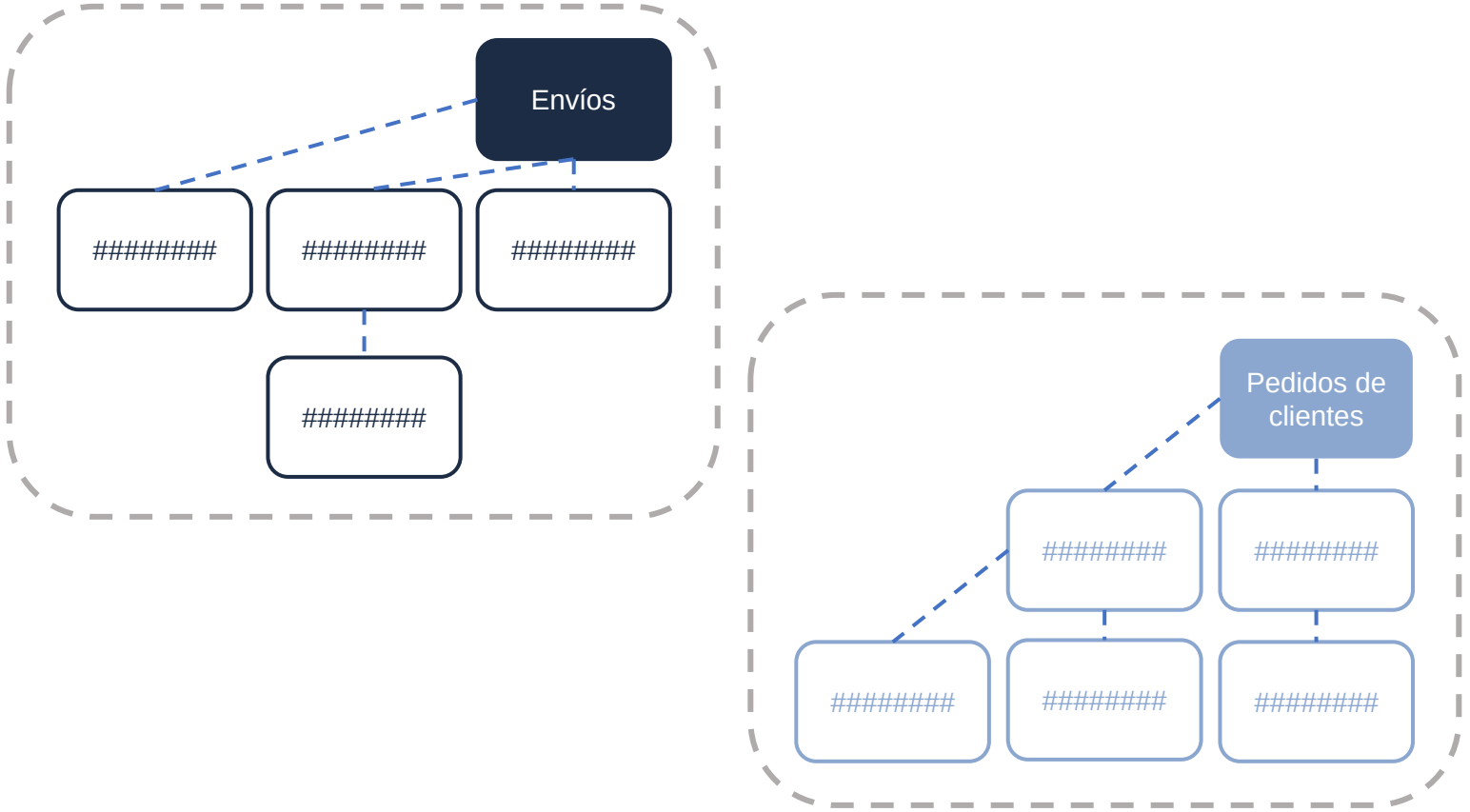


- Identificar los aspectos/conceptos clave del dominio
- Los conceptos forman los Bounded Contexts
- El concepto principal forma un lenguaje ubicuo
- Renombrar conceptos\modelos de soporte
- Renombrar los conceptos LU a naturales
- Mover a contextos externos (fuera del contexto) conceptos\modelos
- No es parte del lenguaje ubicuo
- Eliminar conceptos fuera de alcance
- Los conceptos individuales indican integración
- Integración con otros Bounded Context .

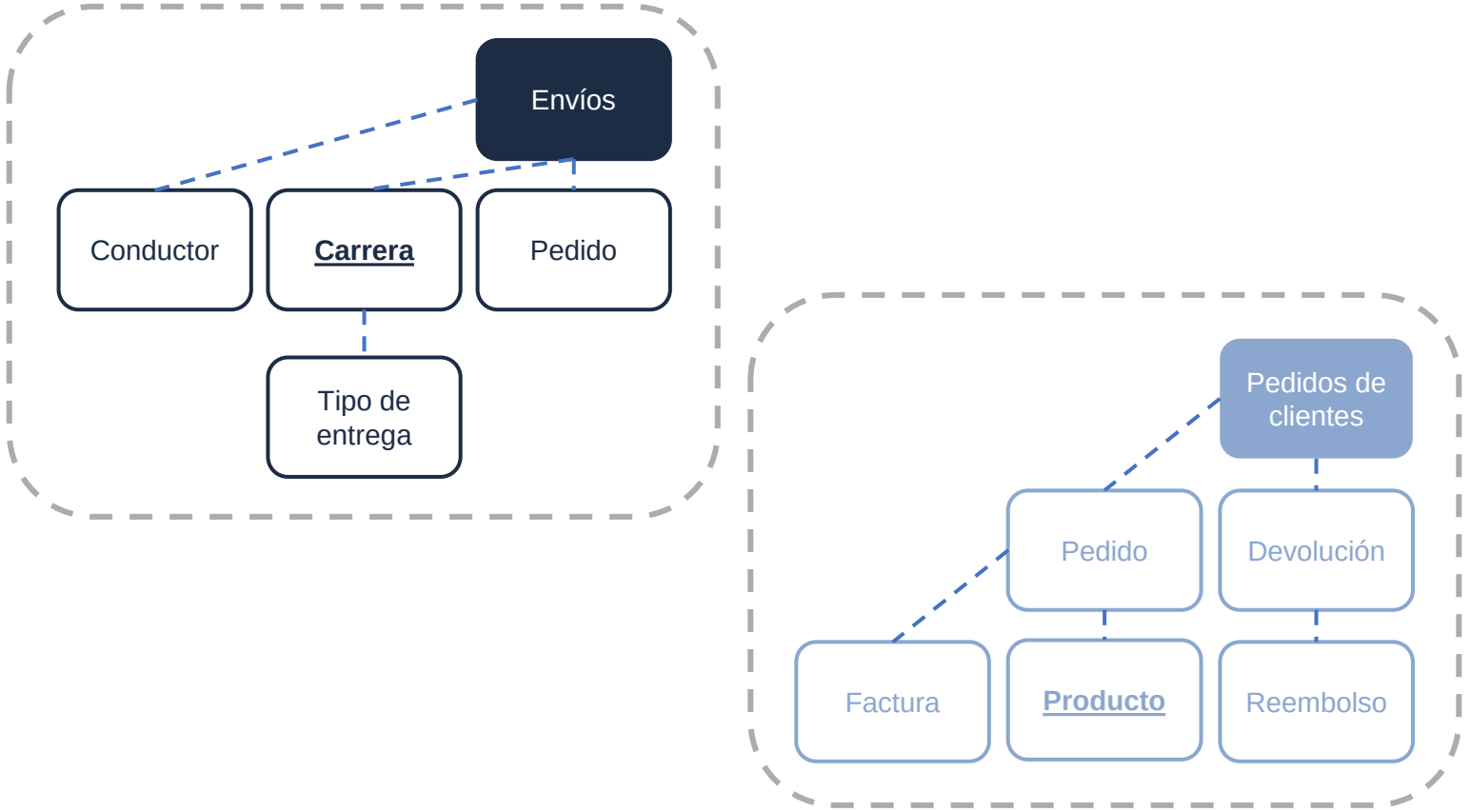
¿Identificando el core (conceptos principales)?



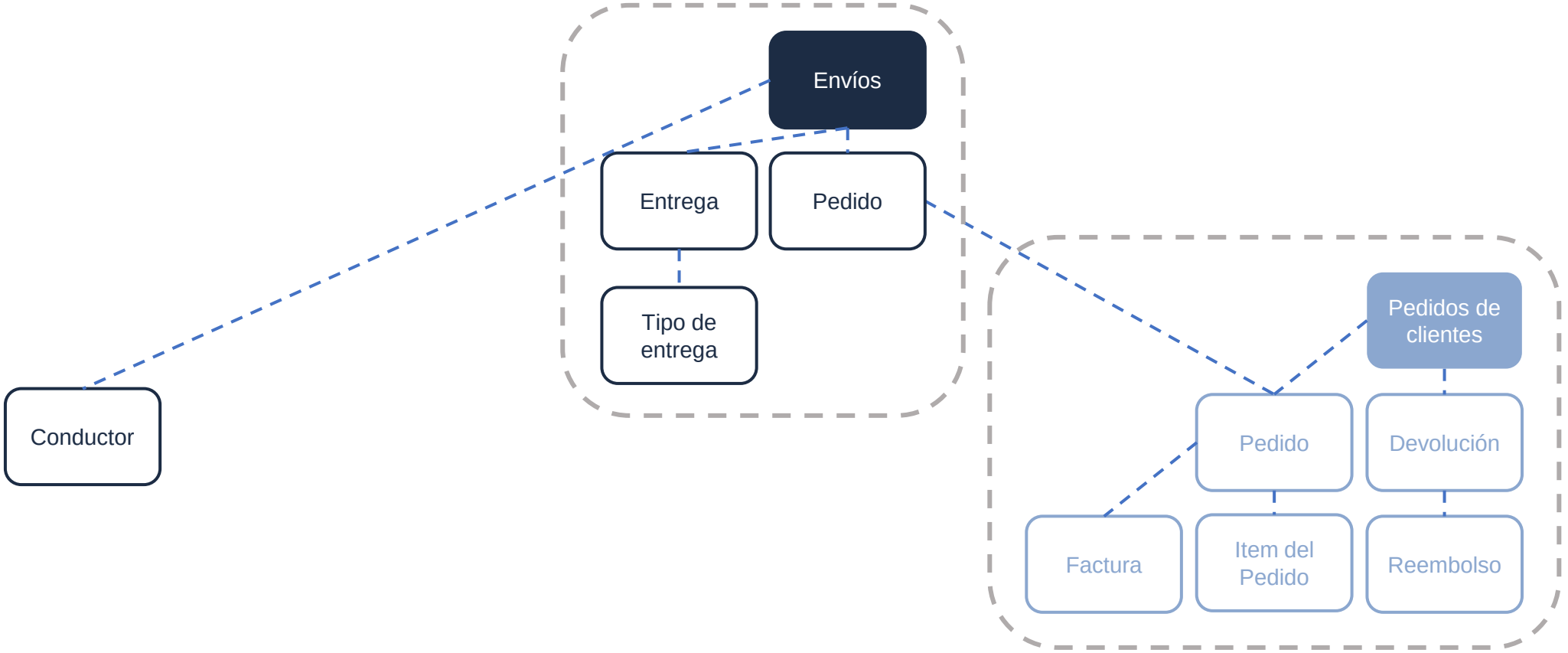
¿Identificando el core (conceptos principales)?



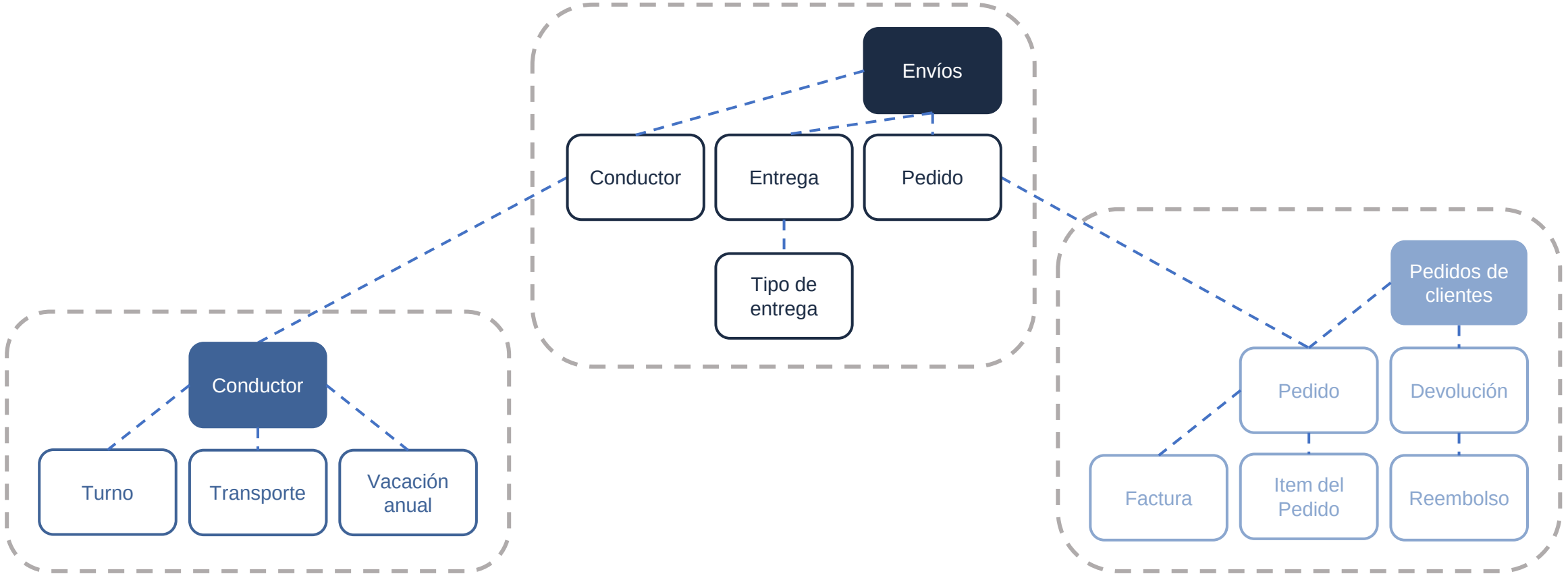
Renombrar conceptos



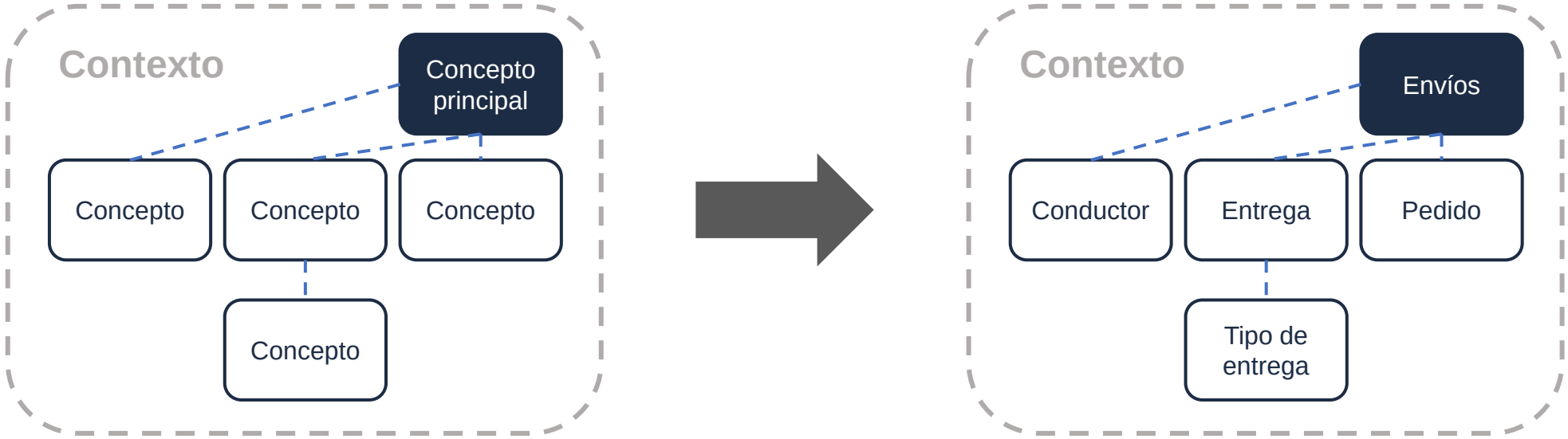
Conceptos fuera de contexto



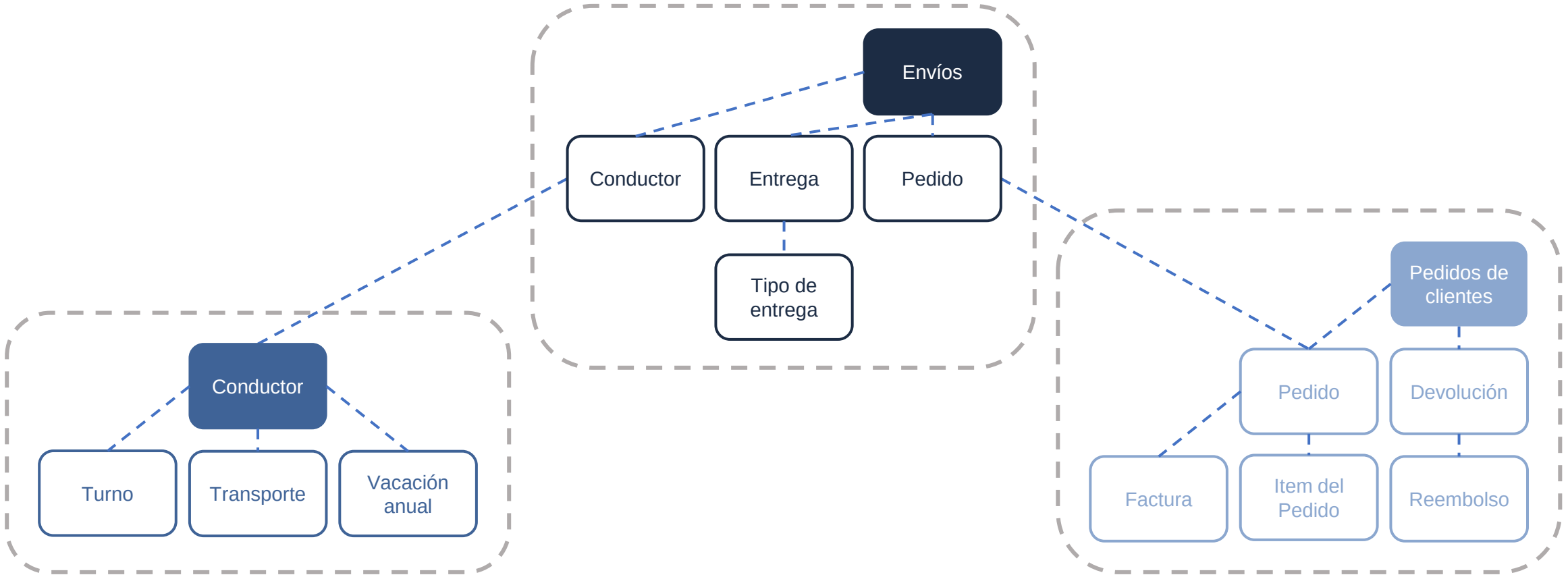
Conceptos únicos indican integración



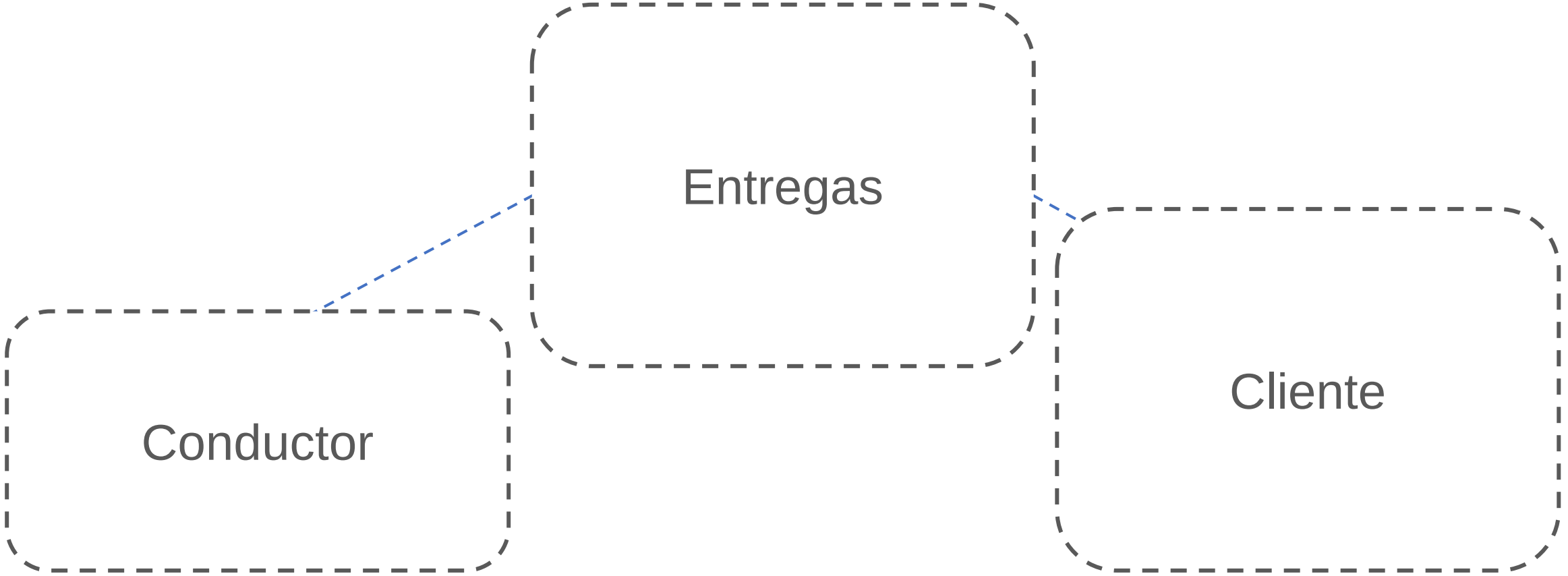
Bounded Context a Microservicios



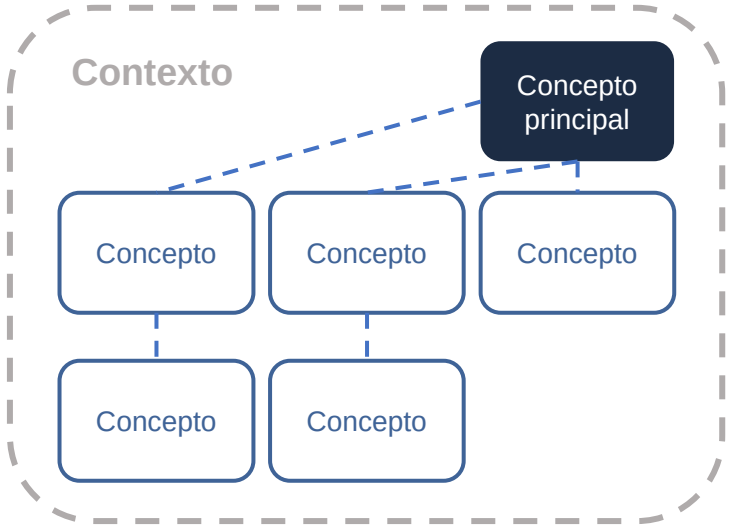
Los Bounded contexts se convierten en microservicios



Los Bounded contexts se convierten en microservicios



¿Cuándo componer o realizar agregación de servicios?



- Servicios combinados
- Uso de descomposición
  - Método de Bounded Context
- Razones para la agregación
  - Informes
  - Funcionalidad mejorada
  - Usabilidad para clientes
  - Performance

Granularidad de servicios

Realización de las funciones de un proceso mediante la implementación de una gran cantidad de servicios detallados conduce a la reducción de los esfuerzos de desarrollo y mantenimiento. Además, un mayor potencial de reutilización de los servicios se puede lograr.

Sin embargo, al mismo tiempo los costes de composición de (muchos) servicios aumentan. Cuanto más fina sea la granularidad para realizar las funciones de un proceso, mayor es el número de servicios y más esfuerzo tiene que ser gastado en componerlos. Por el contrario, los servicios de grano muy grueso tienen las desventajas de mayores costos de implementación y menor potencial de reutilización.

https://core.ac.uk/download/pdf/11552376.pdf

Granularidad de servicios – Teraservices

Los Teraservices son lo opuesto a los microservicios.

El diseño de teraservices implica una especie de servicio monolítico. Los Teraservices requieren dos terabytes de memoria o más. Estos servicios se pueden utilizar cuando los servicios solo se requieren en la memoria y tienen un uso elevado.

Estos servicios son bastante costosos en entornos de nube debido a la memoria necesaria, pero el costo adicional se puede compensar cambiando de servidores de cuatro núcleos a servidores de doble núcleo.

Granularidad de servicios – Microservices

En los Microservices, los componentes de diseño no se agrupan y tienen acoplamientos flexibles.

Cada servicio tiene sus propias capas y base de datos, y se agrupan en un archivo independiente para todos los demás.

Todos estos servicios implementados proporcionan sus API específicas, como Clientes o Reservas. Estas API están listas para consumirse. Incluso la interfaz de usuario también se implementa por separado y se diseña mediante el uso de los servicios de la interfaz de usuario.

Por esta razón, los microservicios proporcionan varias ventajas sobre su contraparte monolítica.

Granularidad de servicios – Nanoservices

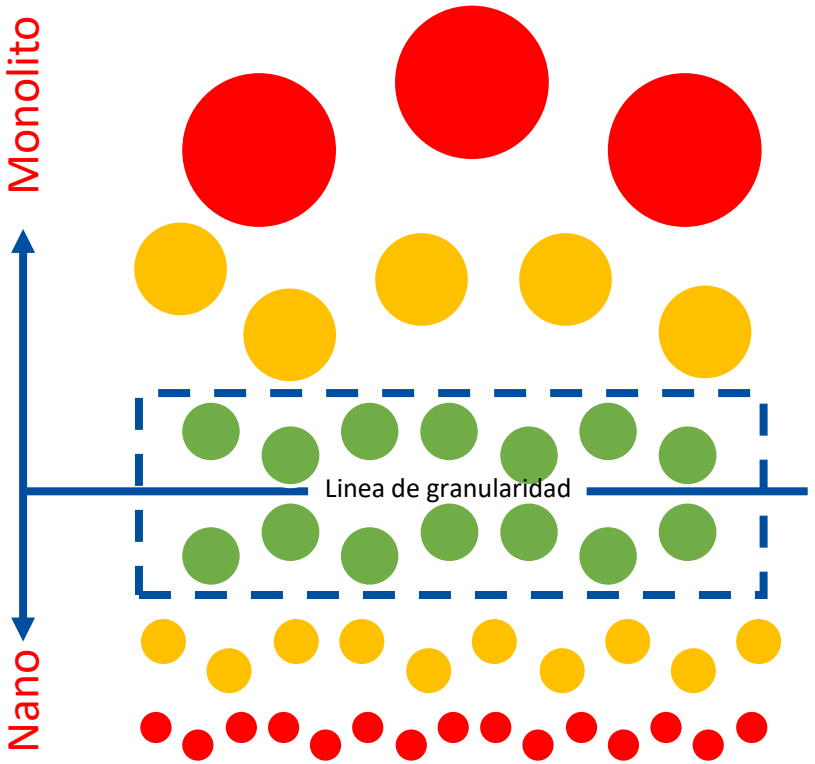
Los microservicios que son especialmente pequeños o detallados se denominan **nanoservicios**.

Un patrón de nanoservicios es realmente un **anti-patrón**.

En el caso de los nanoservicios, las sobrecargas, como las actividades de comunicación y mantenimiento, superan su utilidad.

Se deben evitar los nanoservicios. Un ejemplo de un patrón de nanoservicios (anti) sería crear un servicio independiente para cada tabla de base de datos y exponer su operación CRUD mediante eventos o una API REST.

Granularidad de servicios

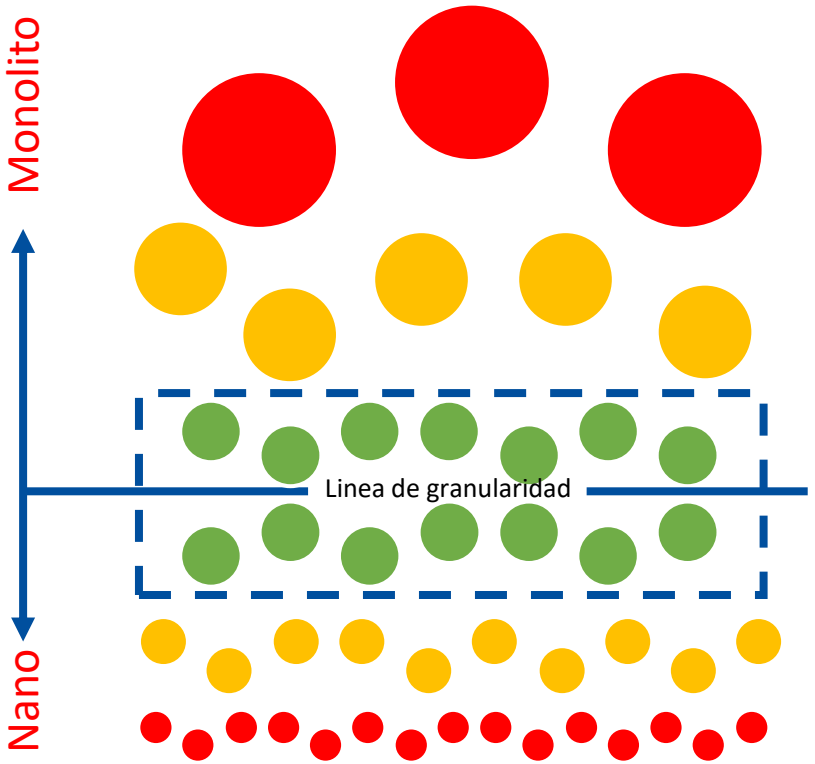


Supongamos que hay una línea horizontal imaginaria (eje x), que representa la línea de granularidad; el nivel adecuado determinado para MSA.

Los servicios que están más cerca, o alrededor de esta línea (servicios verdes dentro de la caja punteada), son buenos microservicios; aquellos que están muy por encima de esta tendencia de línea hacia la exhibición de características de los monolitos, y los que caen muy por debajo de la tendencia de línea hacia la exhibición de características de nanoservicios.

Muchos de los servicios amarillos y todos los servicios rojos caen en los problemas de :

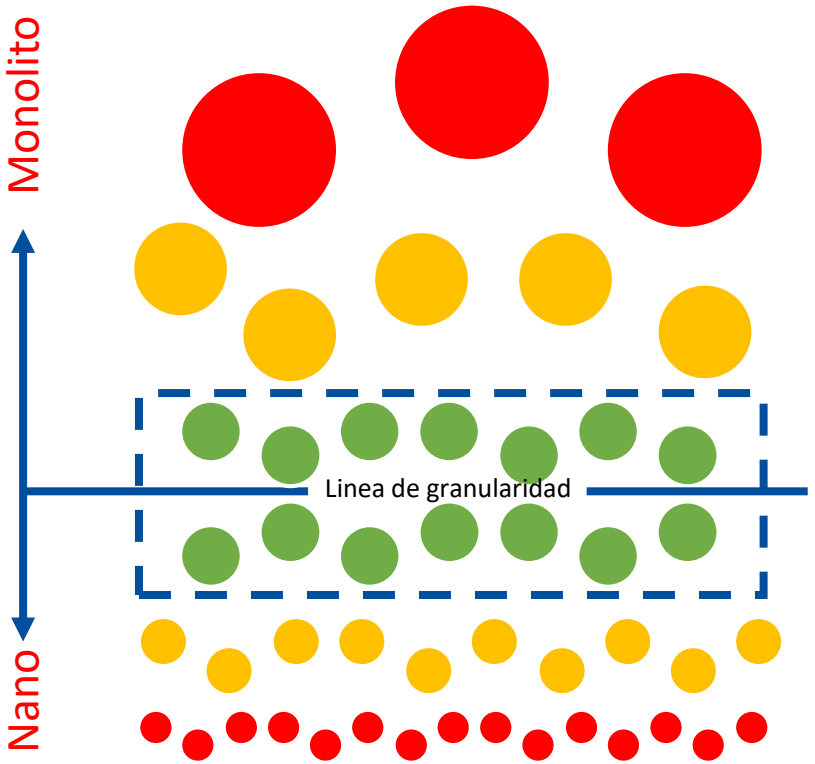
Granularidad de servicios



Los problemas con los monolitos incluyen:

- Incluso los cambios pequeños y menores requieren la reconstrucción de toda la base de código y la re-implementación de la nueva compilación.
- Los ciclos de cambio (para varias funciones y características) tendrán que estar unidos entre sí, lo que provocará una dependencia no deseada.
- Lograr una estructura modular dentro de un monolito es difícil de aplicar.
- El escalado se logra replicando toda la aplicación (aunque funciones específicas pueden tener diferentes requisitos de escalabilidad).

Granularidad de servicios

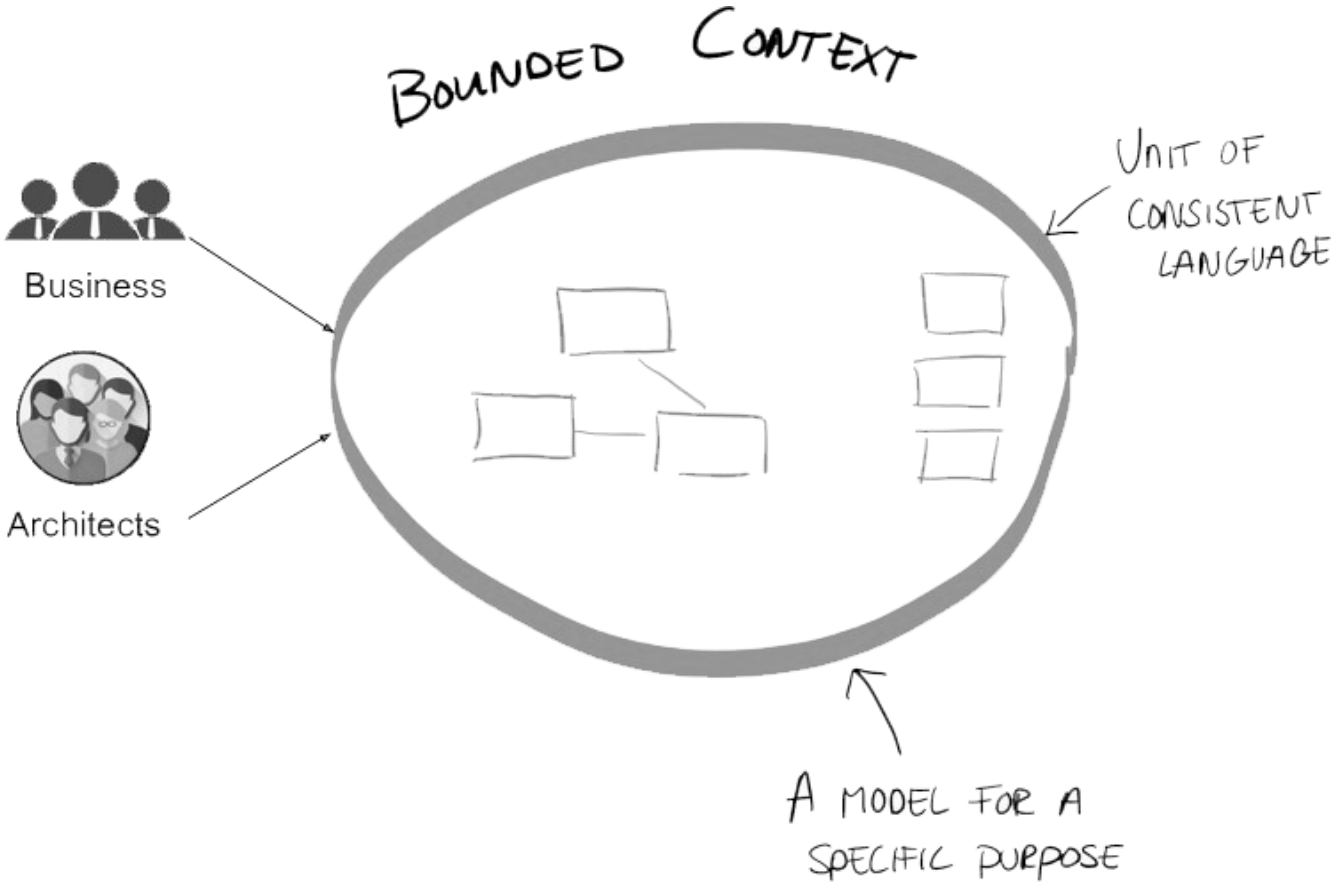


Los problemas con los nanoservicios incluyen:

- Las llamadas remotas son costosas (desde una perspectiva de rendimiento).
- La comunicación entre los servicios se vuelve habladora, lo que resulta en un sistema subóptimo.
- La explosión inmanejable de los servicios puede dar lugar a la proliferación de servicios, a una gobernanza desafiante.

DEMO

Aplicando el patrón DDD a un microservicio.



GALAXY TRAINING

03

Aplicando el patrón Clean Architecture a un microservicio

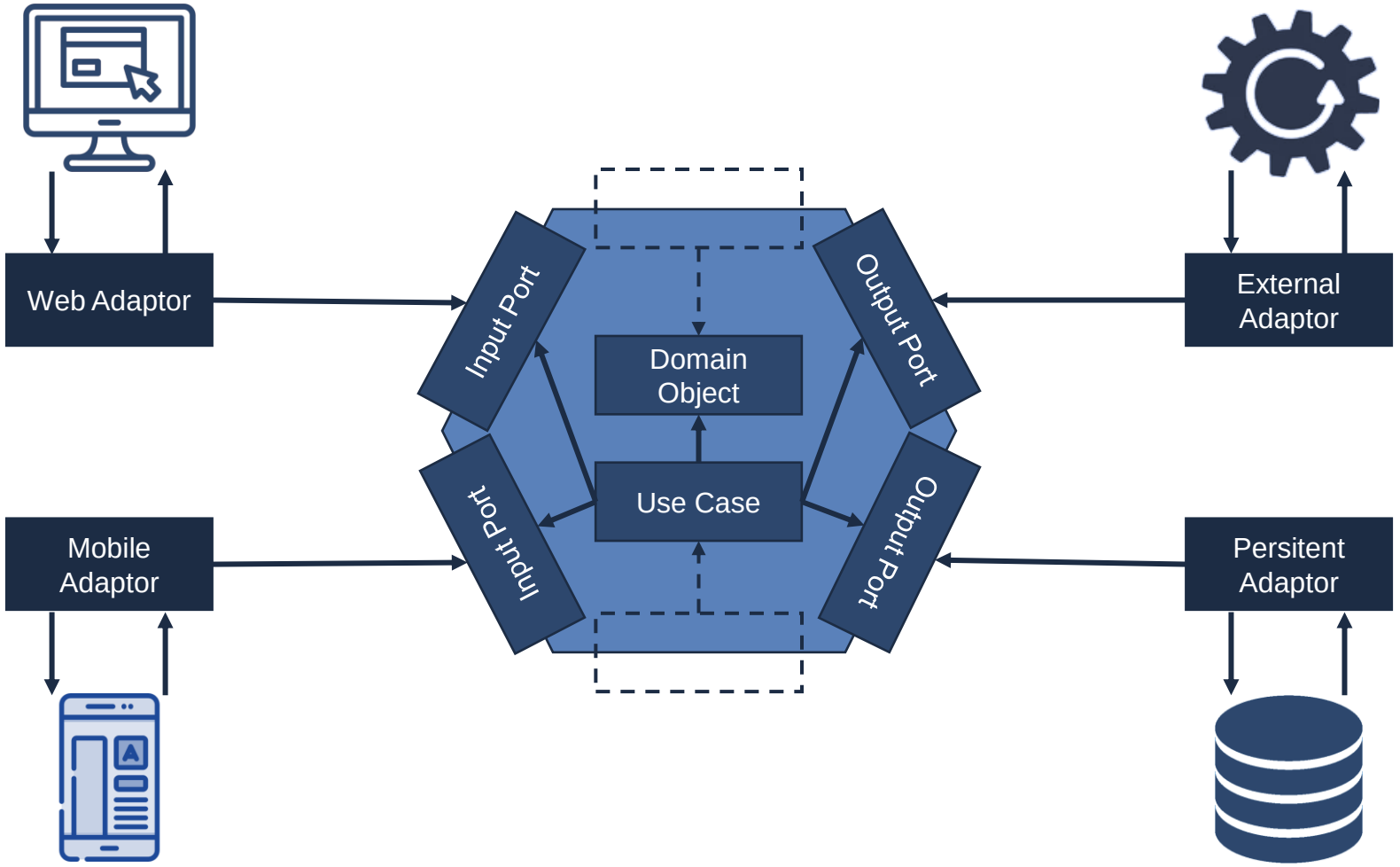
GALAXY
TRAINING

Arquitectura Hexagonal

La arquitectura hexagonal, también conocida como **patrón de puertos y adaptadores** intentan mantener toda la lógica y los modelos de negocios en un solo lugar y todo lo demás depende de esto en lugar de lo contrario.

Si miramos desde cualquier perspectiva de capas de aplicación, **las dependencias serían hacia el dominio** desde el controlador/interfaz y la persistencia en lugar de la arquitectura en capas que solemos hacer en nuestros proyectos. De esta manera hacemos que estos hexágonos sean independientes y puedan relacionarse con el contexto acotado en términos DDD. Al usar la analogía anterior, arquitectura se llamaría **'Plugin-Architecture'**; Su modelo base permanece intacto y puede ampliar aún más las funcionalidades colocando más implementaciones de plugin que se adhieran a las mismas interfaces.

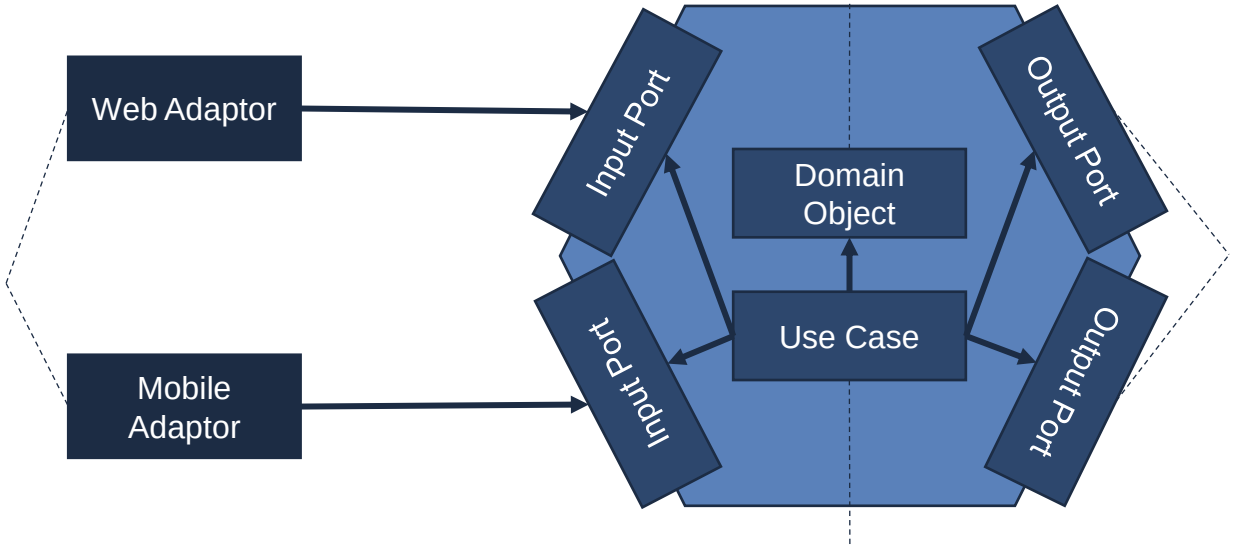
Arquitectura Hexagonal



Arquitectura Hexagonal

Objeto de dominio
Sujeto que representa el mundo real.

Adaptadores
Estas son las implementaciones concretas de la interfaz lógica central que llama al dominio través de la capa de aplicación.



Puertos
Los puertos pueden considerarse como una interfaz simple para su lógica central. Podría haber puertos entrantes y salientes correspondientes a las llamadas entrantes y salientes.

Puerto de entrada

Exponer la lógica central a los adaptadores de consumo.

Puerto de salida

Consumir servicio externo por la lógica central.

Caso de uso
Cualquier función de negocio, accesible e implementación de una interfaz de puerto.

Clean Architecture

En los últimos años hemos visto una amplia gama de ideas con respecto a la arquitectura de los sistemas. Estos incluyen:

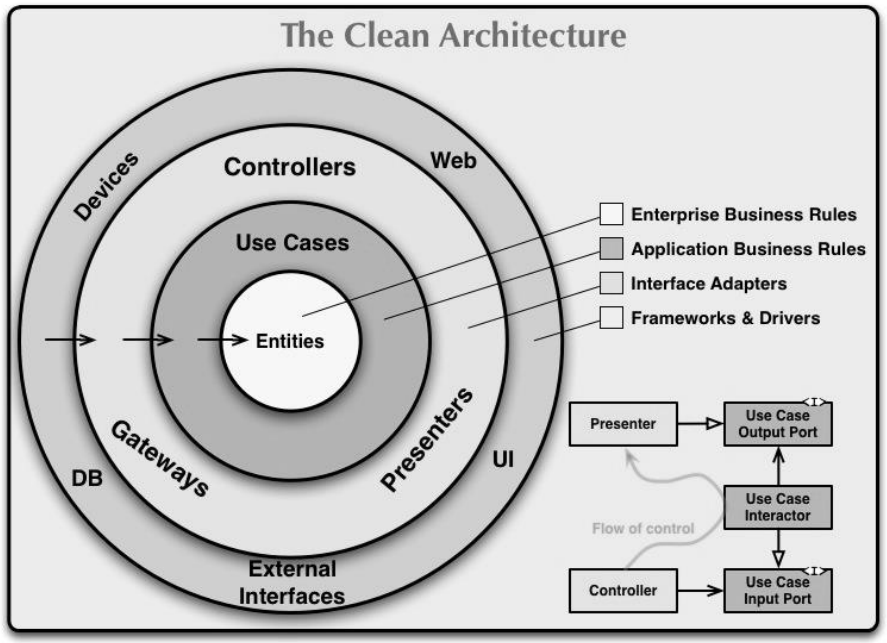
Hexagonal Architecture (a.k.a. Ports and Adapters) por Alistair Cockburn y adoptado por Steve Freeman, y Nat Pryce en su libro Growing Object Oriented Software
Arquitectura de cebolla por Jeffrey Palermo
Screaming Architecture de un blog de Robert C. Martin
DCI de James Coplien y Trygve Reenskaug.
BCE por Ivar Jacobson de su libro Object Oriented Software Engineering: A Use-Case Driven Approach

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

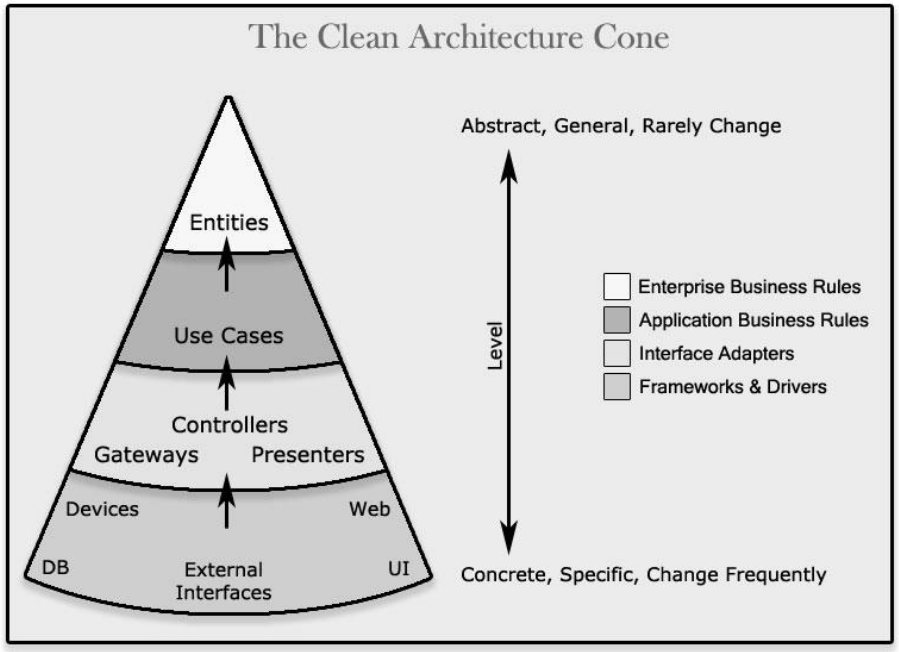
Clean Architecture

El objetivo principal de Clean Architecture es **permitir que el negocio se adapte a la tecnología e interfaces cambiantes**. Si bien Internet puede pasar del escritorio al móvil, o del móvil al asistente virtual, el negocio principal sigue siendo el mismo. Particularmente en el acelerado mundo de los marcos de Javascript y las bibliotecas front-end, el uso de una arquitectura limpia puede salvarlo del patrón común de acoplar estrechamente la lógica empresarial a la capa de presentación o al marco.

Clean Architecture

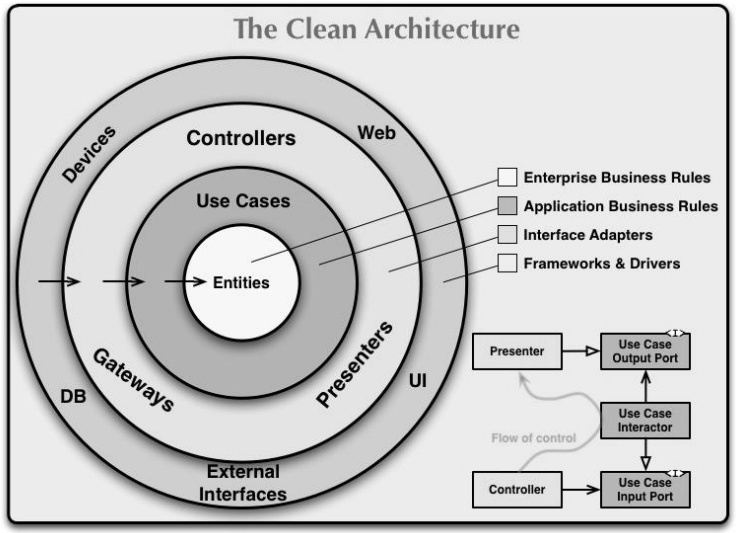


Nuestra reimaginación de la arquitectura a un cono



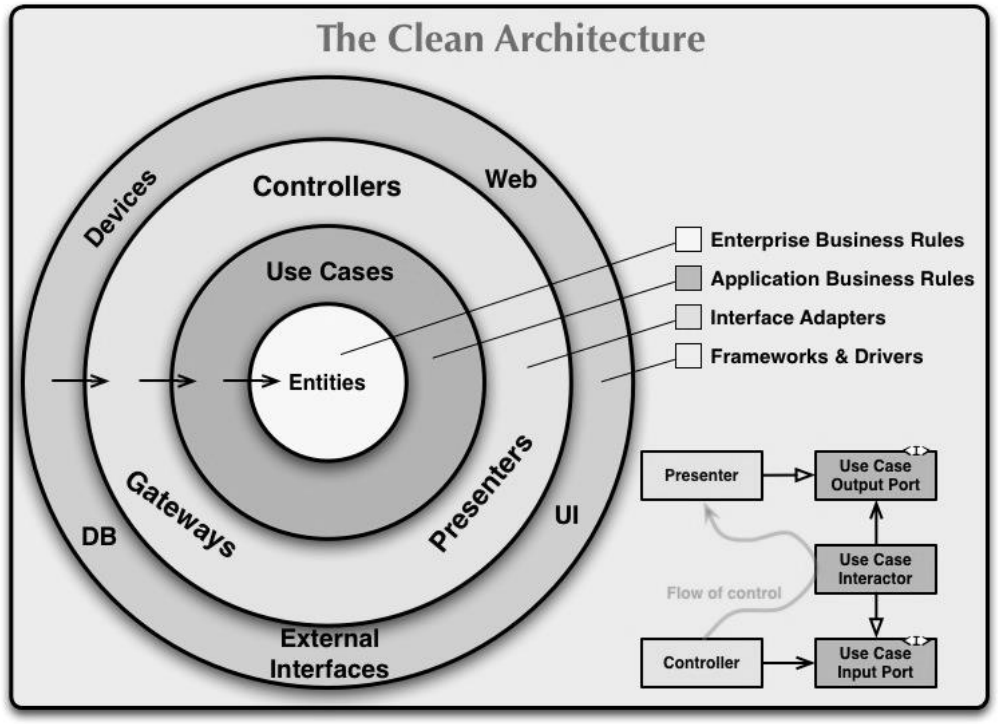
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Características de Clean Architecture



- **Se trata de la separación de responsabilidades**
  - **Dividir el software en capas**
- Debe ser independiente de los Framework's
- Deben ser testeable
- Deben ser independiente de la interfaz de usuario
- Deben ser independiente de la base de datos
- Deben ser independiente de las dependencias de 3rd party
- Avanzando hacia adentro, el nivel de abstracción y el aumento de políticas
- El círculo más interno es el más general/nivel más alto
- Los círculos internos son políticas
- Los círculos exteriores son mecanismos
- **Los círculos internos no pueden depender de los círculos externos**
- **Los círculos exteriores no pueden influir en los círculos internos**

Componentes de Clean Architecture



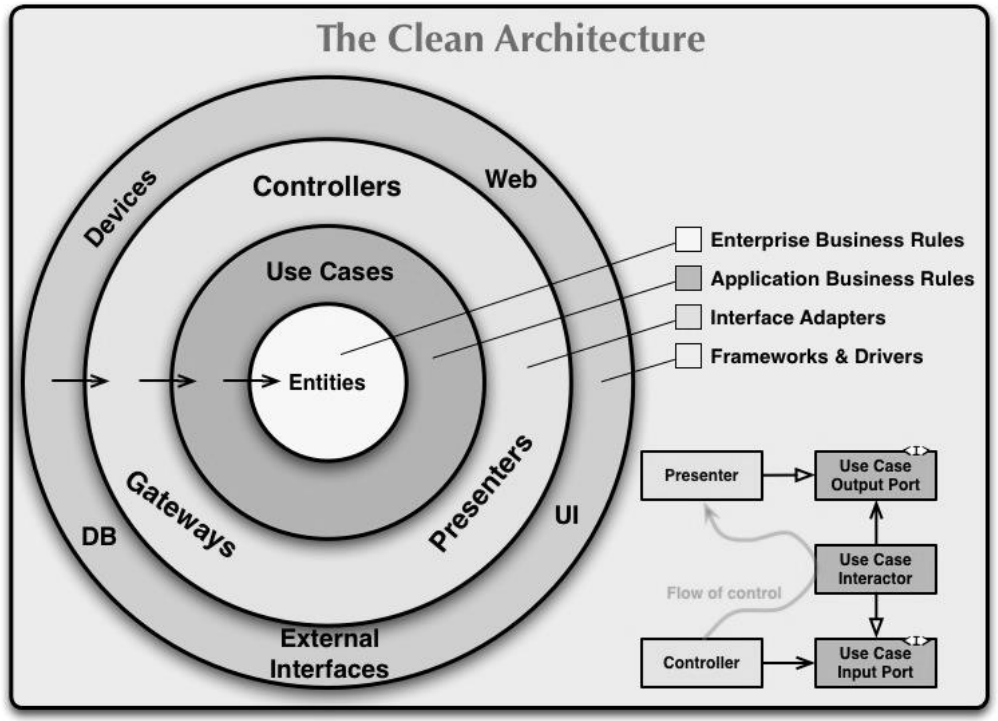
Entidades

- Las entidades deben ser utilizables por muchas aplicaciones (**reglas de negocio críticas**) y no deben verse afectadas por nada más que un cambio en la regla de negocio crítica en sí.
- **Encapsulan las reglas más generales/de alto nivel**.

Casos de uso

- Los casos de uso son **reglas de negocio específicas** de la aplicación
  - Los cambios **no deben afectar a las Entidades**
  - Los cambios **no deben verse afectados por la infraestructura**, como una base de datos.
- Los casos de uso orquestan el flujo de datos que entran / salen de las entidades y dirigen a las entidades a usar sus reglas de negocio críticas para lograr el caso de uso.

Componentes de Clean Architecture



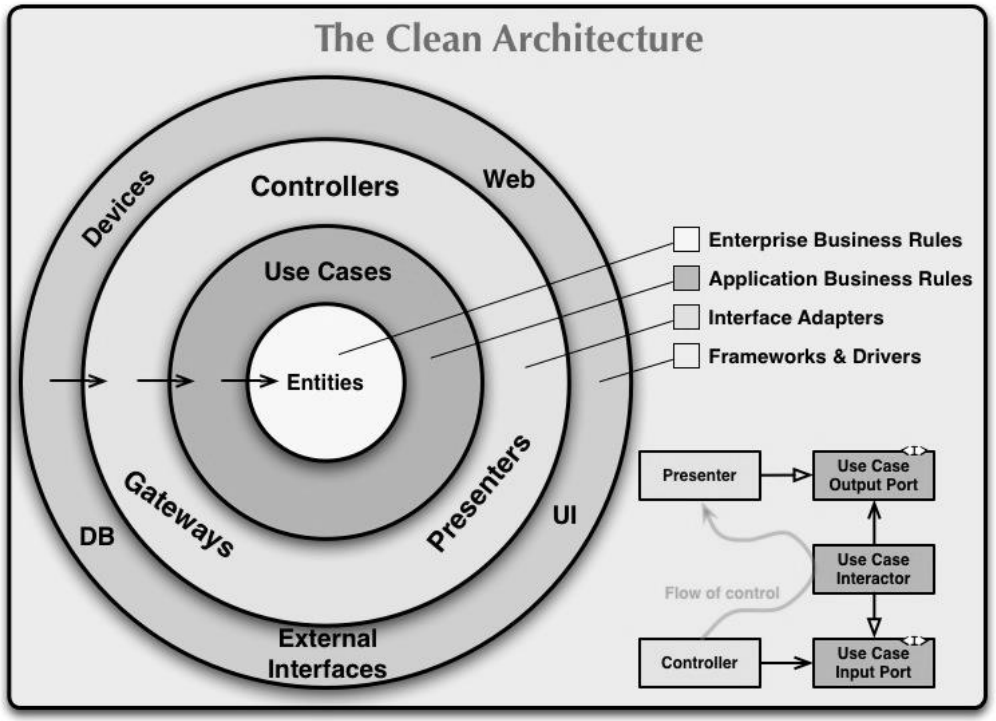
Adaptadores de interfaz

- Convierte datos de capas de datos en capas de casos de uso o capas de entidad
    - Los presentadores, las vistas y los controladores pertenecen aquí
- Ningún código más avanzado en (casos de uso, entidades) debe tener ningún conocimiento de la base de datos

Frameworks & Drivers

- Estos son el pegamento que engancha las diversas capas hacia arriba
- **Los detalles de la infraestructura viven aquí**
- No estás escribiendo mucho de este código, es decir, usamos SQL Server, pero no lo escribimos.

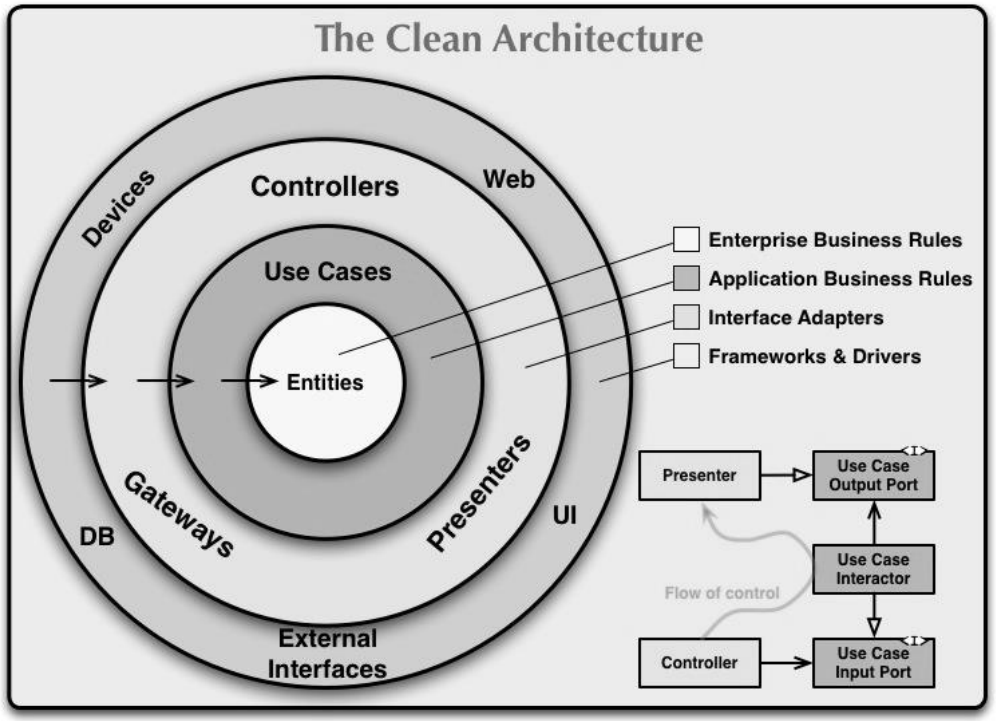
Interacción de Clean Architecture



Cruzando límites

- El flujo de control pasó del controlador, a través del caso de uso de la aplicación, luego al presentador
- Las dependencias del código fuente apuntan hacia los casos de uso
- Principio de inversión de dependencias
    - El caso de uso necesita llamar a un presentador, hacerlo violaría la regla de dependencia: los círculos internos no pueden llamar (o conocer) círculos externos....
        - El caso de uso necesitaría llamar a una interfaz
            - La implementación de esa interfaz sería proporcionada por la capa de adaptador de interfaz: así es como se invierte la dependencia.
                - Este mismo tipo de inversión de control se utiliza en toda la arquitectura para invertir el flujo de control.

Interacción de Clean Architecture

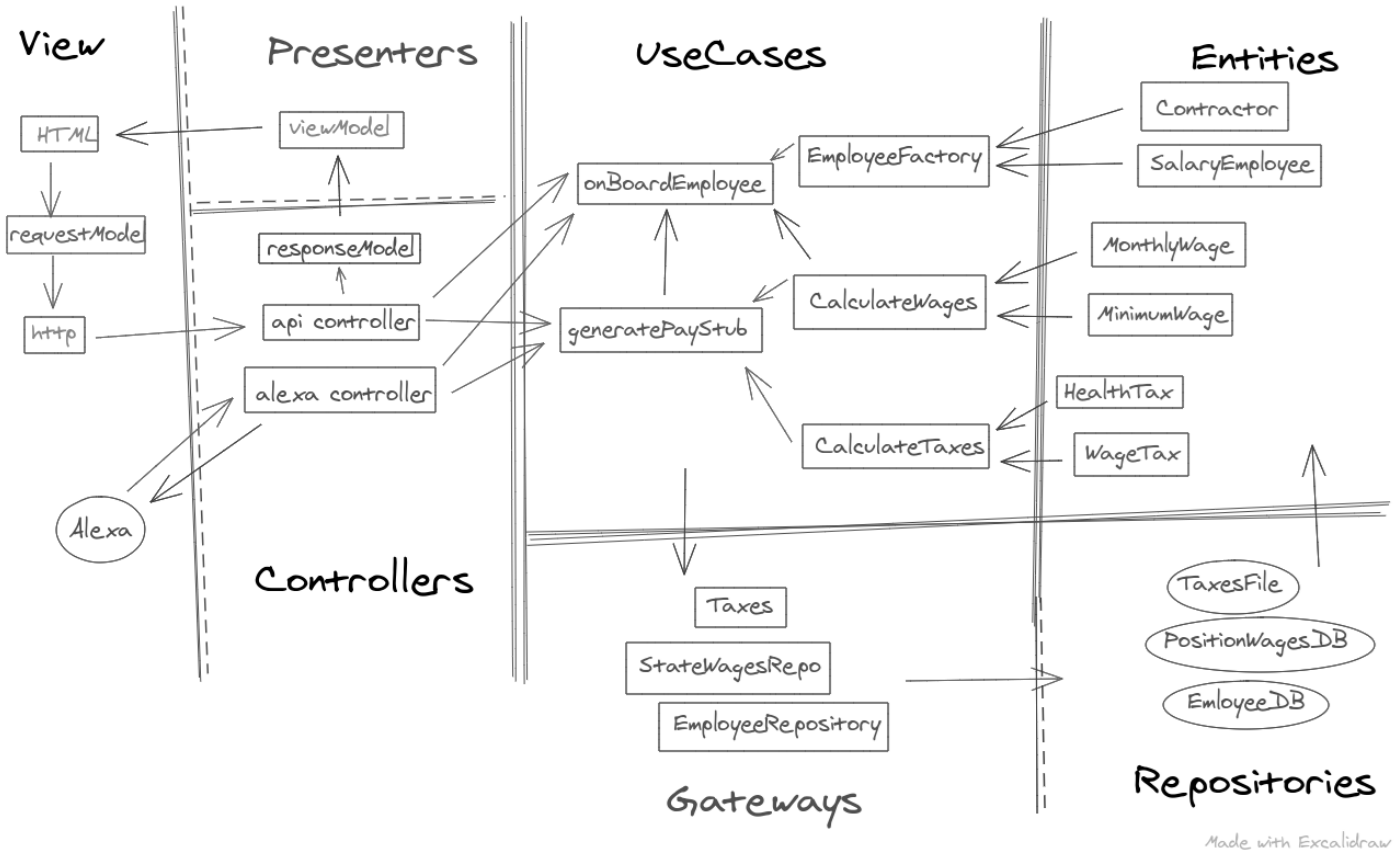


Datos cruzando fronteras

- Normalmente, los datos que cruzan los límites consisten en estructuras de datos simples
- **¡NO PASE OBJETOS DE ENTIDAD NI FILAS DE DATOS!**
  - **Esto violaría las reglas de dependencia**
- Los datos se pasan en el formato que sea más conveniente para el círculo / capa interna
- Estas son estructuras de datos aisladas y simples.
  - Lo que significa que nuestros DTO necesarios para cruzar los límites deben pertenecer al círculo interno, o al menos a su definición (interfaz, clase abstracta)

Aplicando el patrón Clean Architecture a un microservicio

DEMO



GALAXY TRAINING

04

Gestión y gobierno de datos

GALAXY
TRAINING

Gestión descentralizada de datos

Cada microservicio tiene sus propios datos, por lo que la integridad y la coherencia de los datos deben considerarse con mucho cuidado.

El principio fundamental de la gestión de datos de microservicios es que **cada microservicio debe administrar sus propios datos**, podemos llamar a este principio como una **base de datos por servicio**. Eso significa que los microservicios no deben compartir bases de datos entre sí. En lugar de eso**, cada servicio debe responsable de su propia base de datos**, a la que otros servicios no pueden acceder directamente.

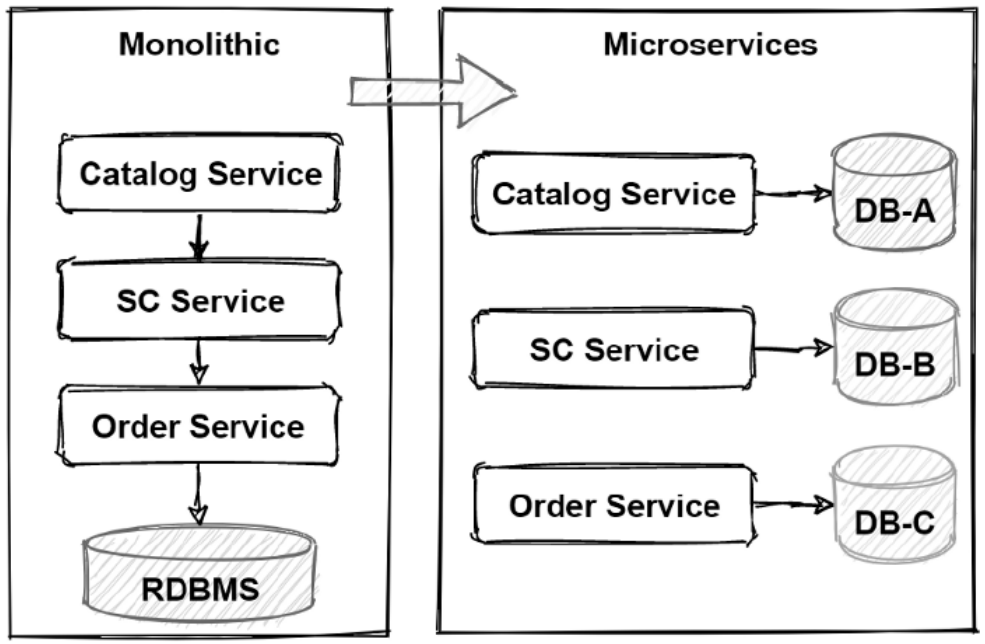
Debe compartir datos a través de los microservicios de la aplicación con el uso de API de REST.

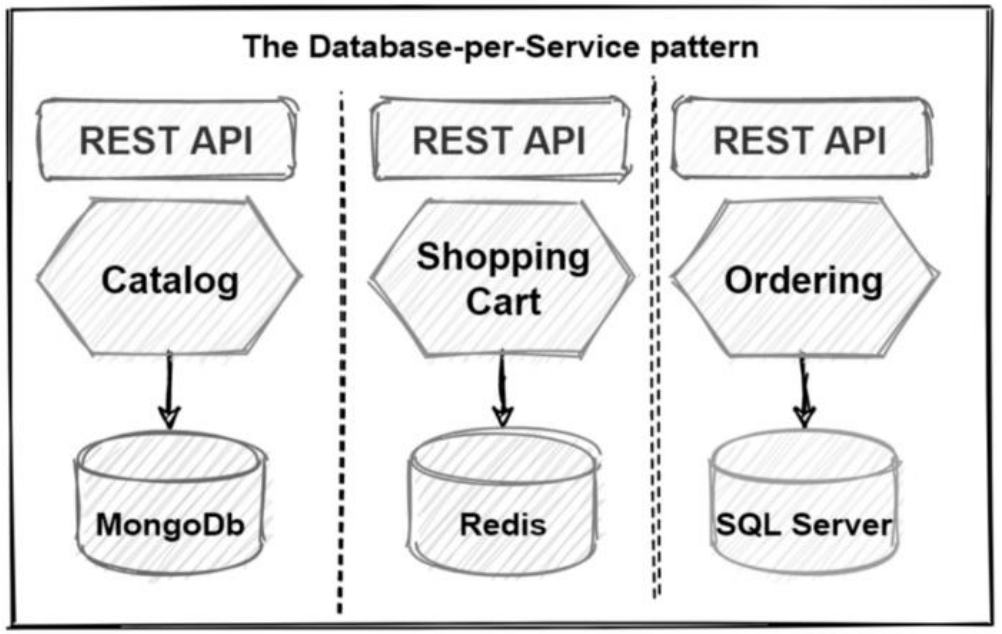
Gestión descentralizada de datos

La razón de este aislamiento es **romper las dependencias innecesarias entre microservicios**. Si hay una actualización en el esquema de base de datos de 1 microservicios, **la actualización no debe afectar directamente a otros microservicios**, incluso otros microservicios no deben conocer los cambios en la base de datos. Al aislar las bases de datos de cada servicio, podemos limitar el ámbito de los cambios en los microservicios cuando se producen cambios en el esquema de la base de datos.

También de esta manera, podemos elegir diferentes bases de datos según los microservicios qué base de datos puede elegir la mejor opción. Llamamos a esto principio de **"persistencia políglota".**

Patrón de base de datos por servicio





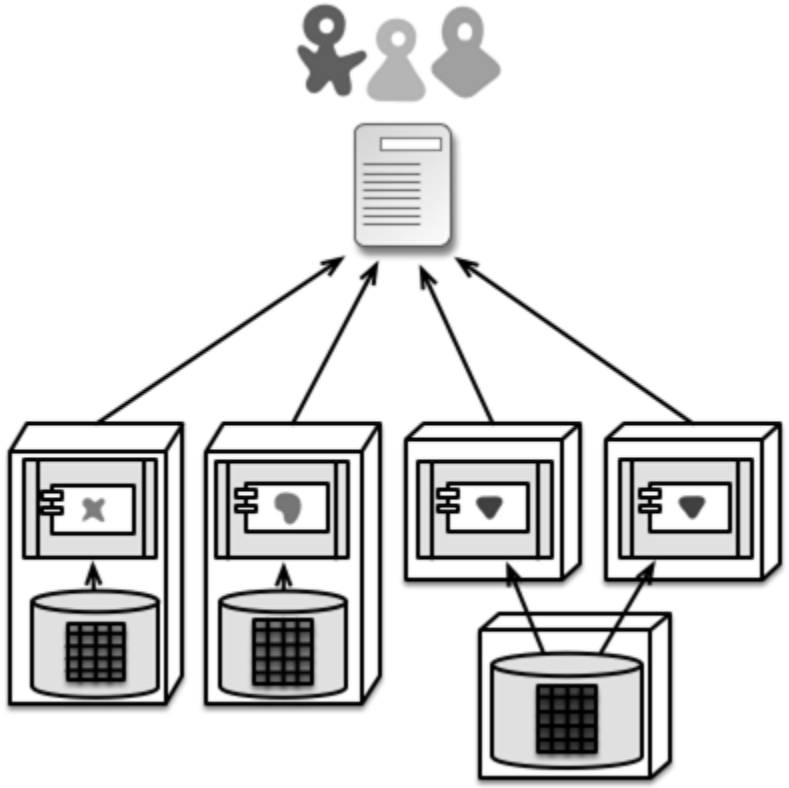
Arquitectura lógica vs Arquitectura física

La creación de microservicios no requiere el uso de ninguna tecnología específica. Por ejemplo, los contenedores de Docker no son obligatorios para crear una arquitectura basada en microservicios. Esos microservicios también se pueden ejecutar como procesos sin formato. **Los microservicios son una arquitectura lógica.**

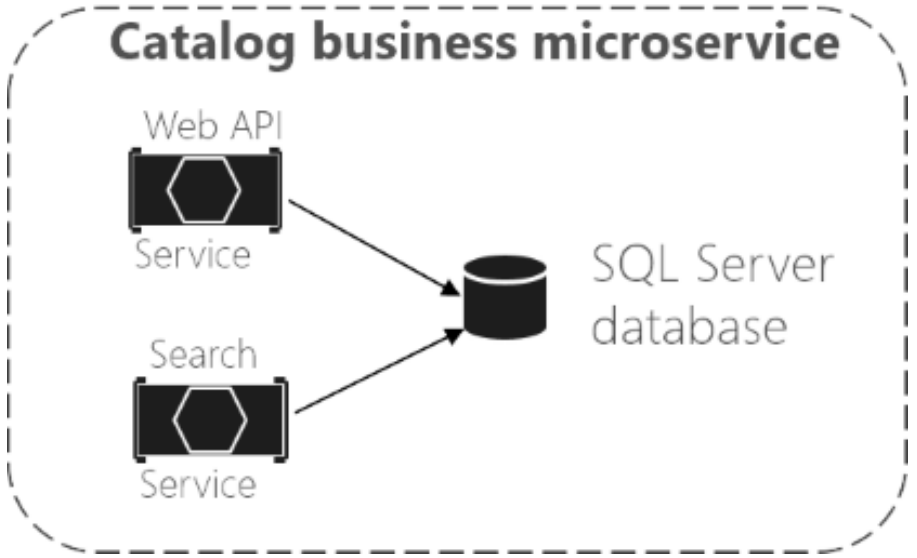
La arquitectura lógica y **los límites lógicos de un sistema no se asignan necesariamente uno a uno a la arquitectura física** o de implementación. Esto puede suceder, pero a menudo no es así.

Por tanto, un microservicio o contexto limitado empresarial es **una arquitectura lógica que podría coincidir (o no) con la arquitectura física**. Lo importante es que **un microservicio** o contexto limitado empresarial **debe ser autónomo** y permitir que el código y el estado se versionen, implementen y escalen de forma independiente.

Arquitectura lógica vs Arquitectura física



microservices - application databases



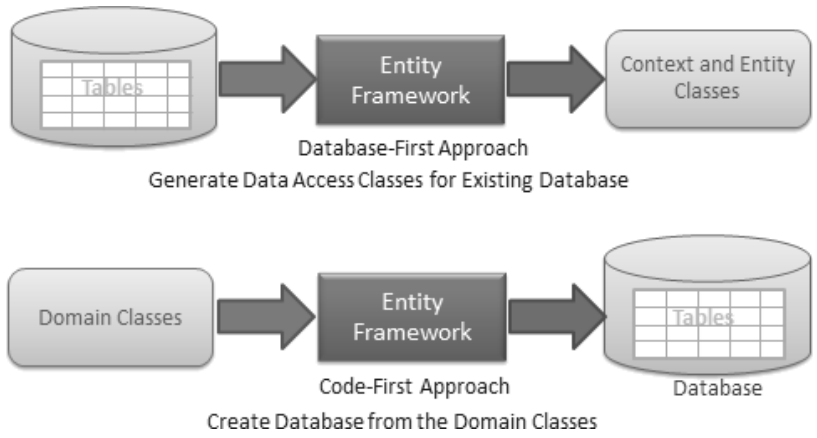
05

Infraestructura de persistencia en NET 6 y MS SQL Server 2022

Entity Framework Core

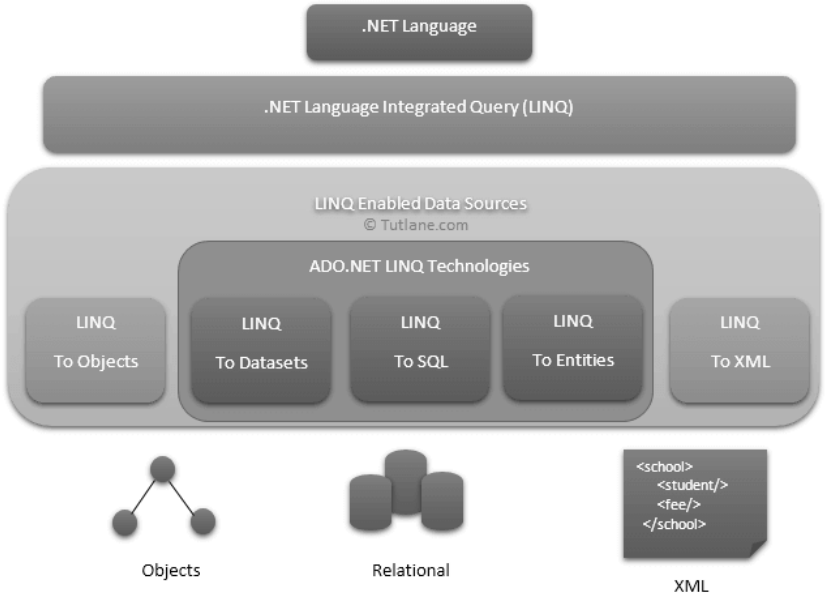
- EF Core es una versión ligera, extensible, de **código abierto y multiplataforma** de la popular tecnología de acceso a datos Entity Framework.
- **EF Core puede servir como asignador relacional de objetos (O/RM),** lo que permite a los desarrolladores de .NET trabajar con una base de datos mediante objetos .NET y eliminar la mayoría del código de acceso a los datos que normalmente deben escribir.
- EF Core es **compatible con muchos motores de base de datos**

Entity Framework Core - Modelo



- Con EF Core, **el acceso a datos se realiza mediante un modelo**.
- Un modelo se compone de clases de entidad y un objeto de contexto que representa una sesión con la base de datos, lo que permite consultar y guardar los datos.
- Puede generar un modelo a partir de una base de datos existente, codificar manualmente un modelo para que coincida con la base de datos o usar migraciones de EF para crear una base de datos a partir del modelo y que evolucione a medida que cambia el modelo.

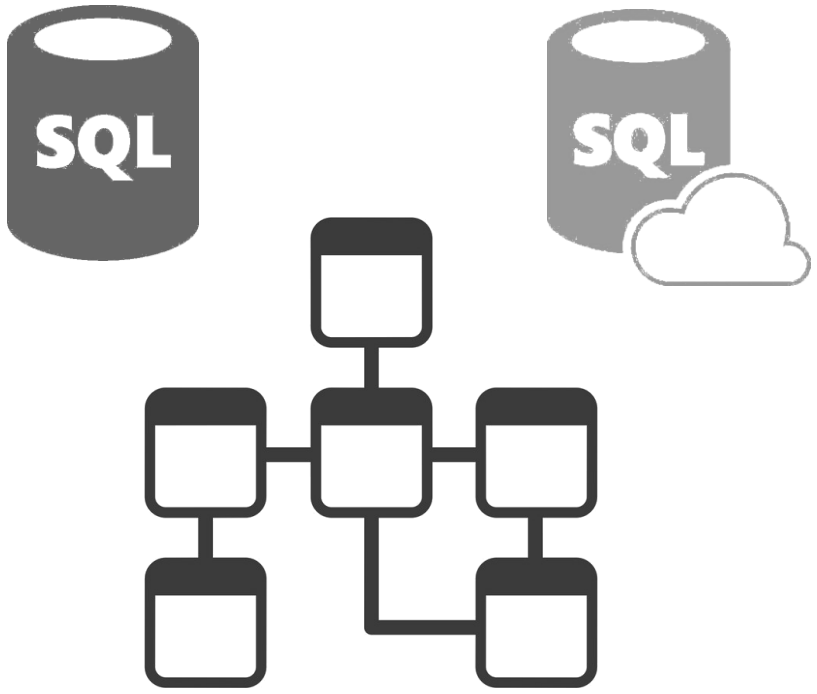
Entity Framework Core - Consultas



- EF Core usa **Language Integrated Query (LINQ) para consultar datos de la base de datos**. LINQ permite usar C# (o el lenguaje .NET que prefiera) para escribir consultas fuertemente tipadas.

- Usa el contexto derivado y las clases de entidad para hacer referencia a los objetos de base de datos. EF Core **pasa una representación de la consulta LINQ al proveedor de la base de datos**. A su vez, **los proveedores de la base de datos la traducen al lenguaje de la consulta específico** para la base de datos (por ejemplo, SQL para una base de datos relacional).

SQL Server / Azure SQL Database



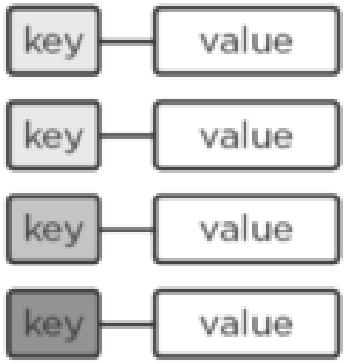
- Una base de datos de SQL Server consta de una colección de tablas en las que se almacena un **conjunto específico de datos estructurados**.
- **Una tabla contiene una colección de filas**, también denominadas tuplas o registros, y columnas, también denominadas atributos.
- **Cada columna de la tabla se ha diseñado para almacenar un determinado tipo** de información; por ejemplo, fechas, nombres, importes en moneda o números.

06

Infraestructura de persistencia – NoSQL (CosmoDB)

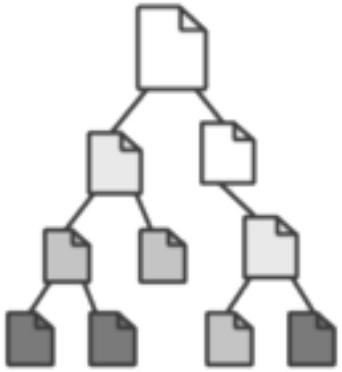
GALAXY
TRAINING

Tipos de Base de Datos NoSQL



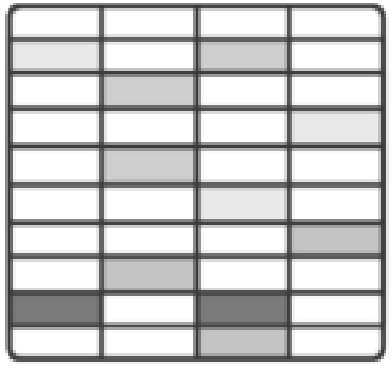
Clave-Valor

Vincula una clave dada a un registro de cualquier tipo



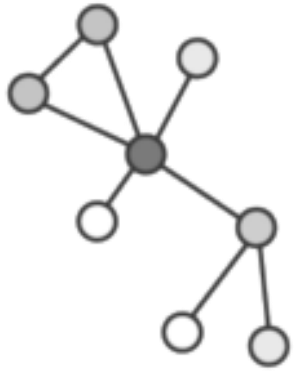
Documental

Se basan en el concepto detrás de clave-valor, extendiéndolo para admitir objetos complejos de varias capas denominados documentos



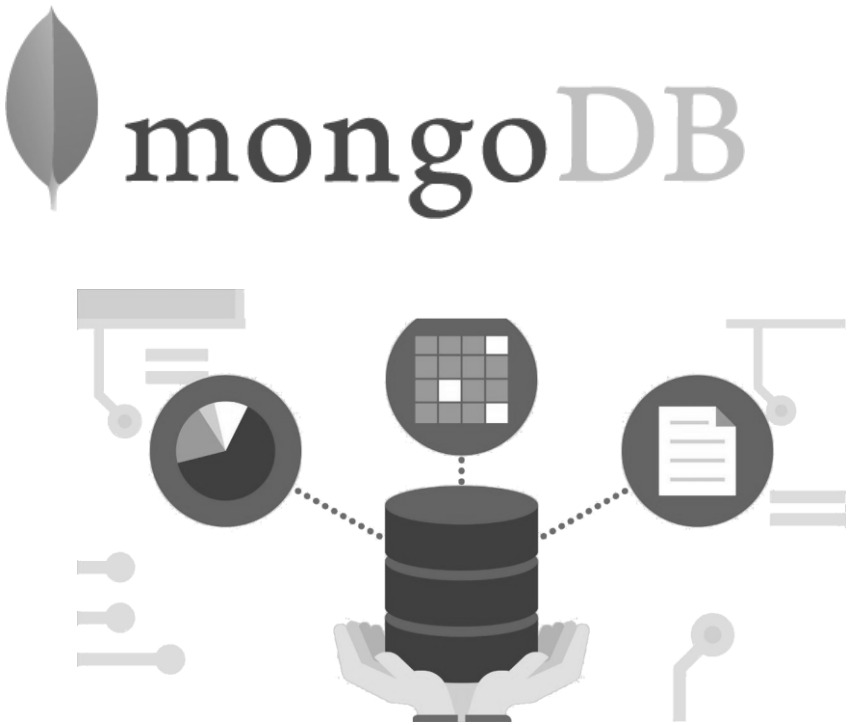
Columnar

Puede pensar en las filas como claves en un almacén clave-valor y las columnas como el valor



Grafos

Enfoque hacia la relación entre entidades. Las entidades, como los usuarios, están representadas por nodos, mientras que las conexiones entre entidades dictan cómo se relacionan



MongoDB

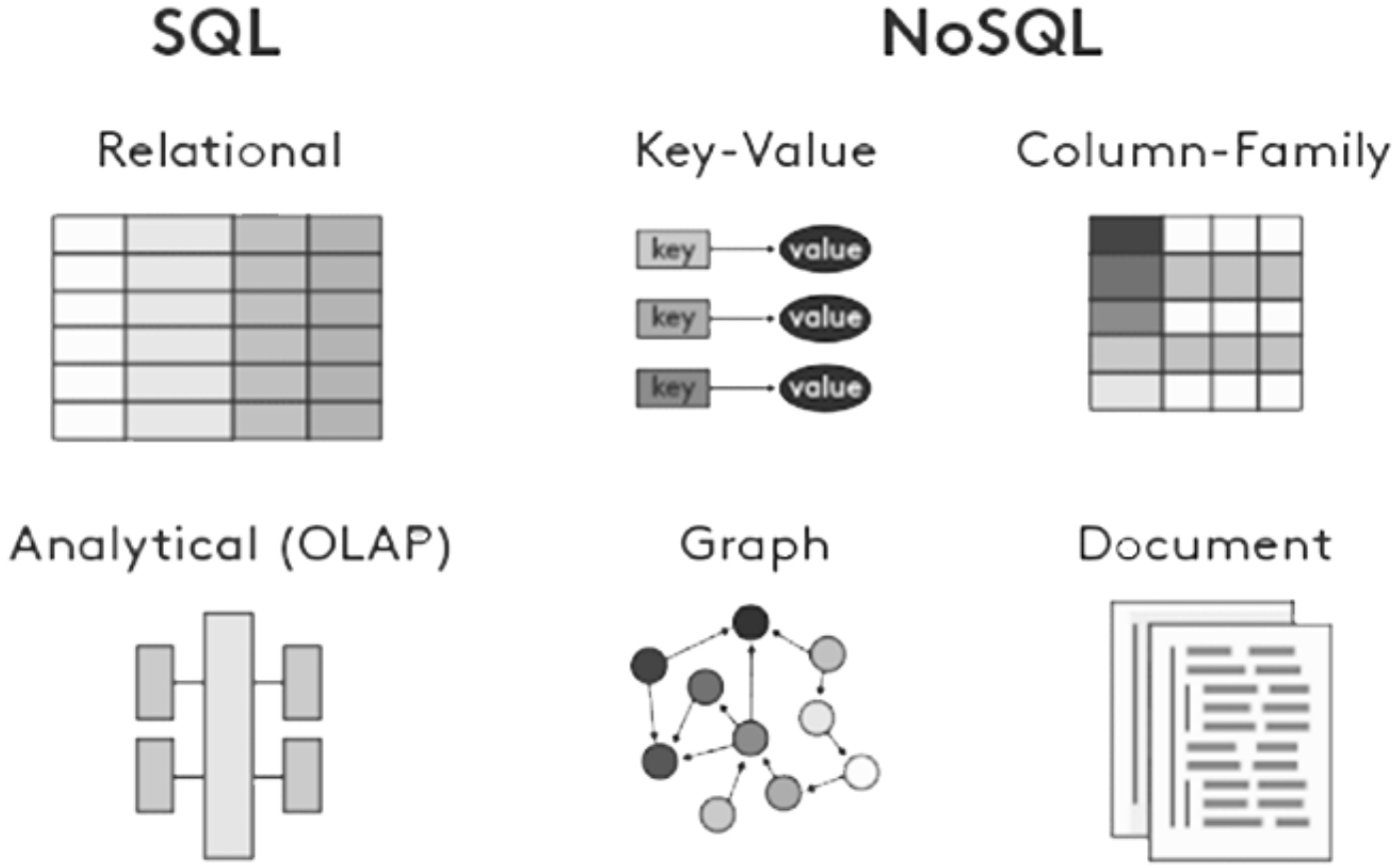
- MongoDB es una **base de datos de documentos** que ofrece una gran escalabilidad y flexibilidad, y un modelo de consultas e indexación avanzado.
- MongoDB almacena datos en **documentos flexibles similares a JSON**, por lo que **los campos pueden variar entre documentos** y la estructura de datos puede cambiarse con el tiempo
- El modelo de documento se asigna a los objetos en el código de su aplicación para facilitar el trabajo con los datos
- MongoDB es una base de datos distribuida en su núcleo

Cosmo DB

- Azure Cosmos DB **es un servicio de base de datos con varios modelos distribuido** de forma global de Microsoft.
- **Puede escalar de forma elástica el rendimiento y almacenamiento**, y sacar provecho del rápido acceso a datos.
- Base de datos relacional: PostgreSQL
- Base de datos NoSQL: MongoDB, Cassandra, Tables o Gremlin

Infraestructura de persistencia - EF Core, MSSQL, NoSQL (CosmoDB)

DEMO



GALAXY TRAINING

GRACIAS

POR SU PREFERENCIA

GALAXY
TRAINING