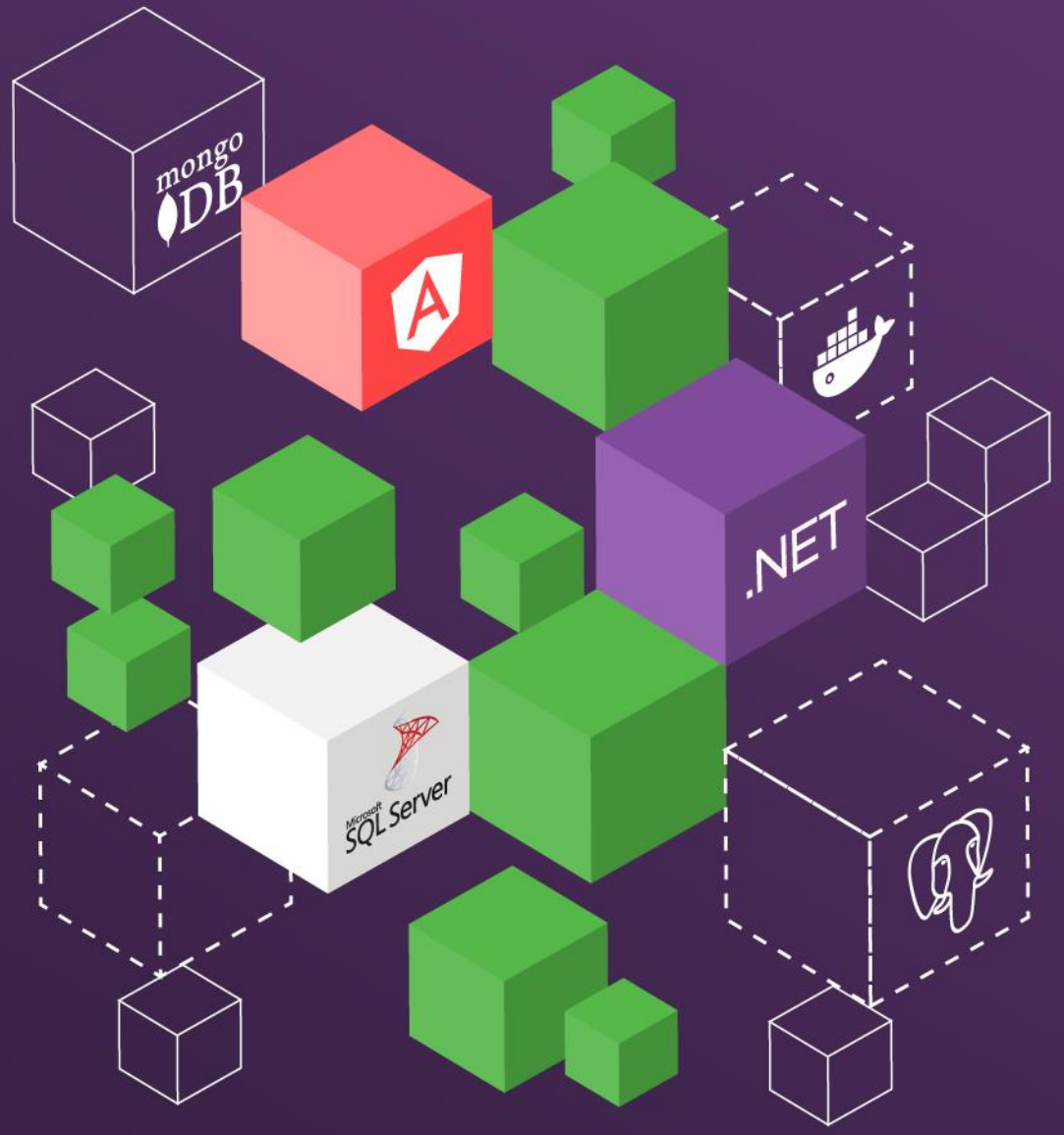


BIENVENIDOS
AL CURSO:

Arquitectura de Microservicios en Net

SESIÓN 03

GALAXY
TRAINING

ÍNDICE

Características de una arquitectura de microservicios

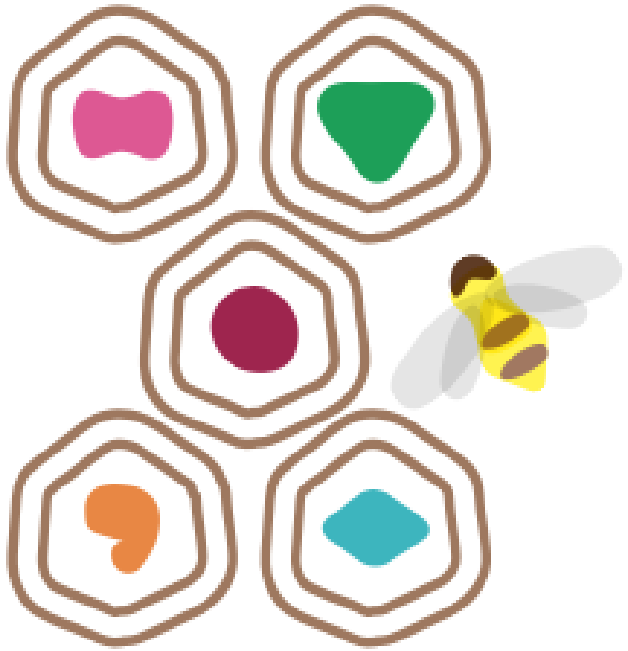
El estilo arquitectónico de microservicio es un enfoque para desarrollar una sola aplicación como un conjunto de servicios pequeños, cada uno que se ejecuta en su propio proceso y se comunica con mecanismos ligeros, a menudo una API de recursos HTTP.

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.

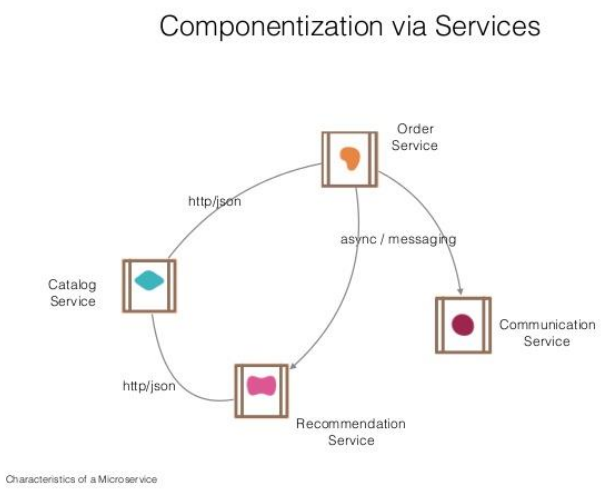
Características de una arquitectura de microservicios

Características de los microservicios



- 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



¿Qué es realmente un componente?

El término a menudo aparece como una alternativa al módulo o widget. Pero hay una definición vaga: **¿es una clase, una pieza ejecutable?**

Desde la perspectiva de Fowler 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.

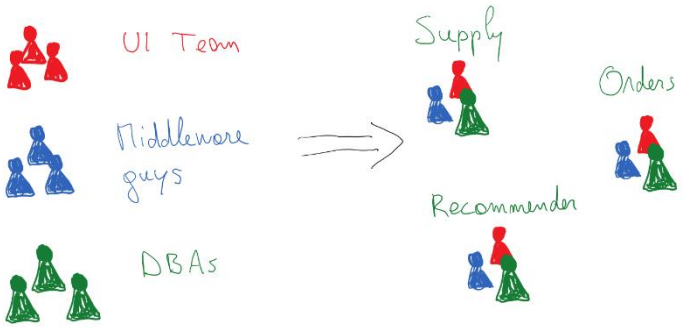
Básicamente un componente es algo que es actualizable y reemplazable de forma independiente.

Hay formas importantes sobre cómo se utilizan o se incorporan los componentes en un proyecto: a través de bibliotecas (jars, gemas, node_modules,...) o **a través de servicios**.

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

Características de una arquitectura de microservicios

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“

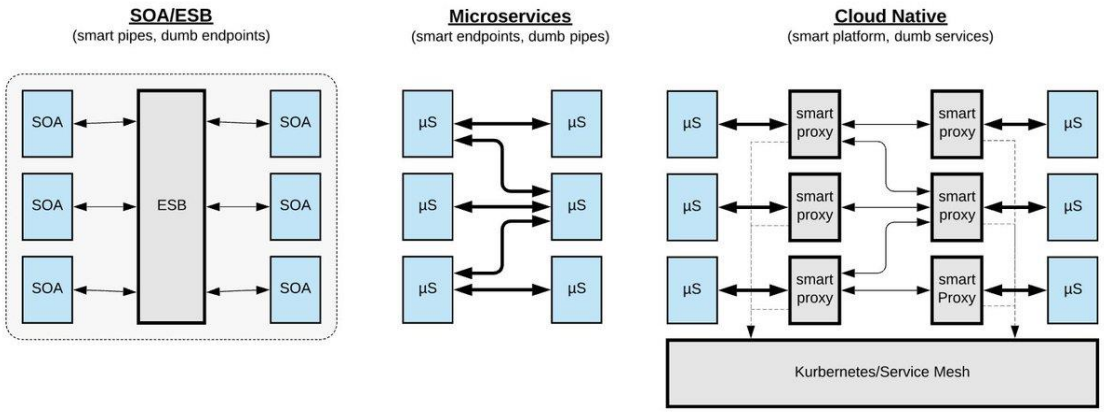
Según Amazon, estos equipos deberían ser tan grandes que somos capaces de alimentarlos con **2 pizzas (estadounidenses) (una docena de personas).**

Fowler destaca que un hecho importante es que estos equipos tienen una línea de **comunicación directa al usuario final o cliente** y obtienen comentarios de acuerdo sobre cómo se están utilizando las cosas que construyen y qué tan bien o no funciona.

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.

Características de una arquitectura de microservicios

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.

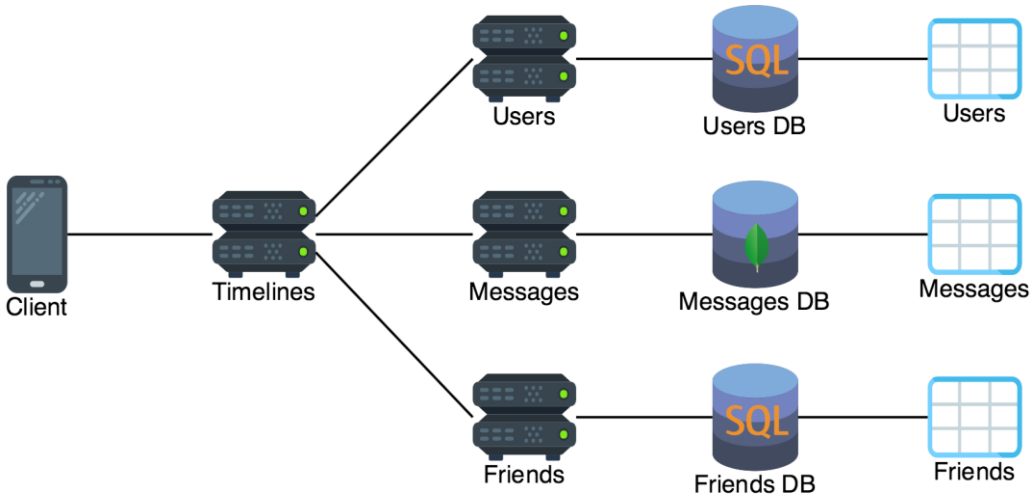
En su lugar, los microservicios facilitan los pipes tontos y los **Endpoints/Aplicaciones** inteligentes.

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.

Características de una arquitectura de microservicios

Gestión descentralizada de datos



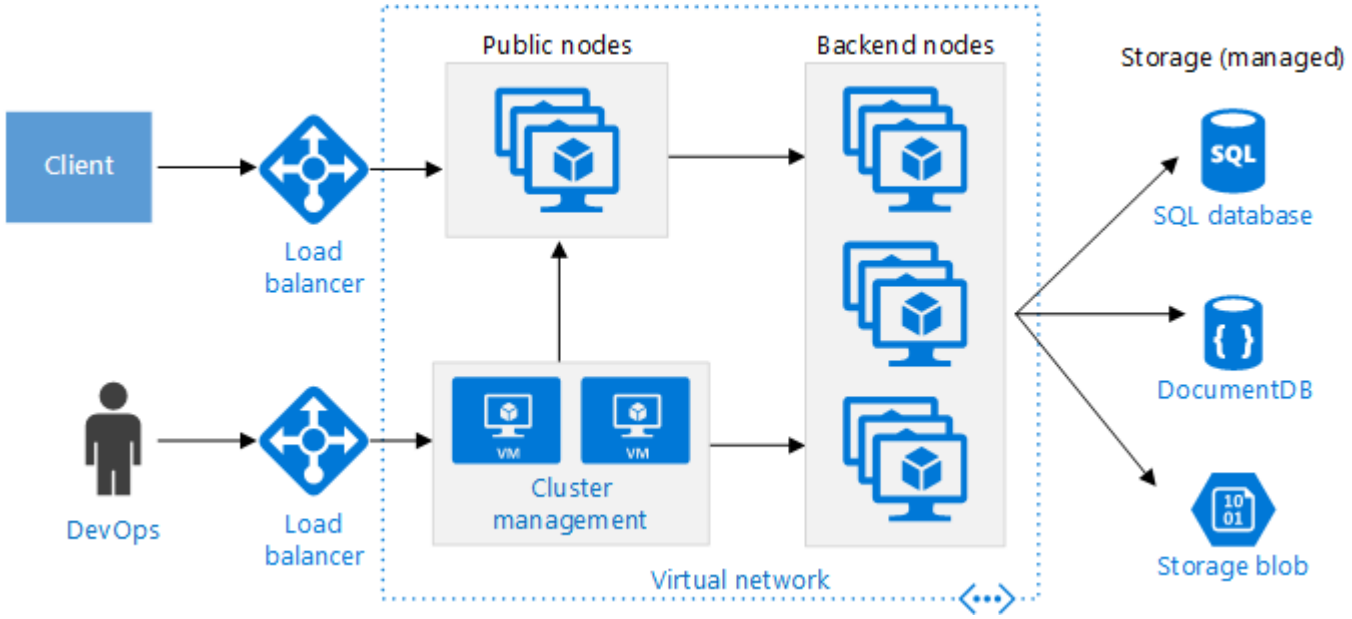
Normalmente, en un sistema monolito hay una enorme base de datos donde se almacenan todos los datos. A menudo incluso hay varios monolitos que se conectan a la misma base de datos.

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. De esta manera, se pueden utilizar diferentes lenguajes de programación y sistemas de bases de datos entre los servicios. Es la toma de decisiones descentralizada.

Características de una arquitectura de microservicios

Automatización de infraestructuras



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

Características de una arquitectura de microservicios

Diseño para fallos

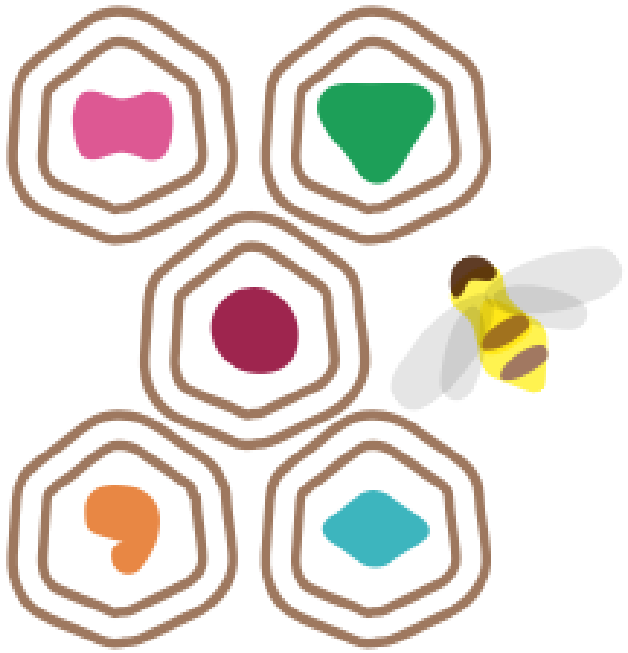
Inevitablemente se tiene que diseñar para el error, ya que los microservicios fallarán, incluso con frecuencia.

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.

Esto les da información valiosa sobre lo resistentes que son sus servicios, qué tan bien se recuperan de las interrupciones y cómo se ve afectado el sistema general.

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 los microservicios - Adicionales



- Productos no Proyectos
- Gobernanza descentralizada
- Diseño evolutivo

Productos no Proyectos

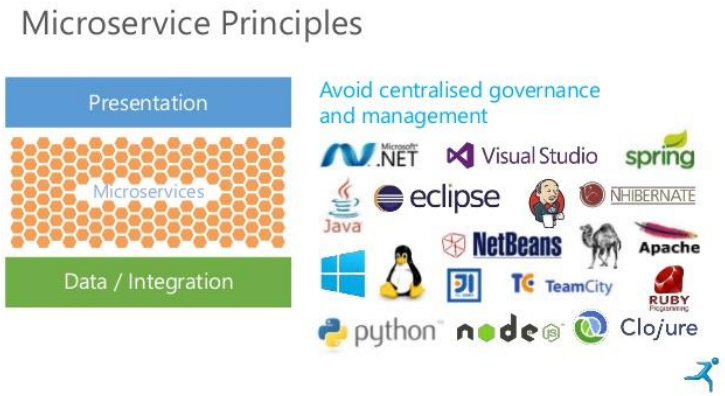
La mayoría de los esfuerzos de desarrollo de aplicaciones que vemos utilizan un modelo de proyecto: donde el objetivo es entregar algún software que luego se considera completado. Al finalizar, el software se entrega a una organización de mantenimiento y el equipo del proyecto que lo creó se disuelve.

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.

Características de una arquitectura de microservicios

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**.

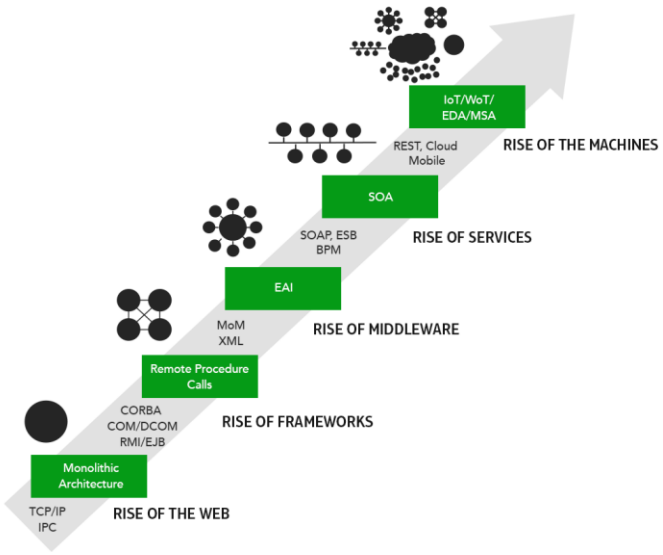
Preferimos utilizar la herramienta adecuada para el trabajo y aunque las aplicaciones monolíticas pueden aprovechar diferentes idiomas hasta cierto punto, no es tan común.

Los equipos que construyen microservicios también prefieren un enfoque diferente a los estándares.

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.

Características de una arquitectura de microservicios

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** - lo que implica que buscamos puntos donde podemos imaginar reescribir un componente sin afectar a sus colaboradores.

Este énfasis en la reemplazabilidad es un caso especial de un principio más general del **diseño modular**, que es impulsar la modularidad a través del patrón de cambio. Desea mantener las cosas que cambian al mismo tiempo en el mismo módulo. Partes de un sistema que cambian rara vez deben estar en diferentes servicios a los que actualmente están experimentando una gran cantidad de abandono.

Si se encuentra cambiando repetidamente dos servicios juntos, eso es una señal de que deben combinarse.

Alcance de un microservicio utilizando el patrón Bounded Context.

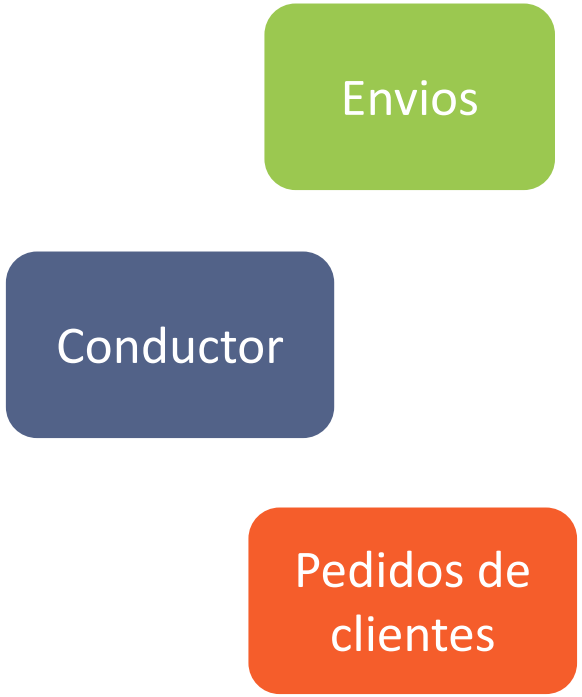
Domain Driven Design

- 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

Bounded Context

Una responsabilidad específica impuesta por un límite explícito

Bounded Context

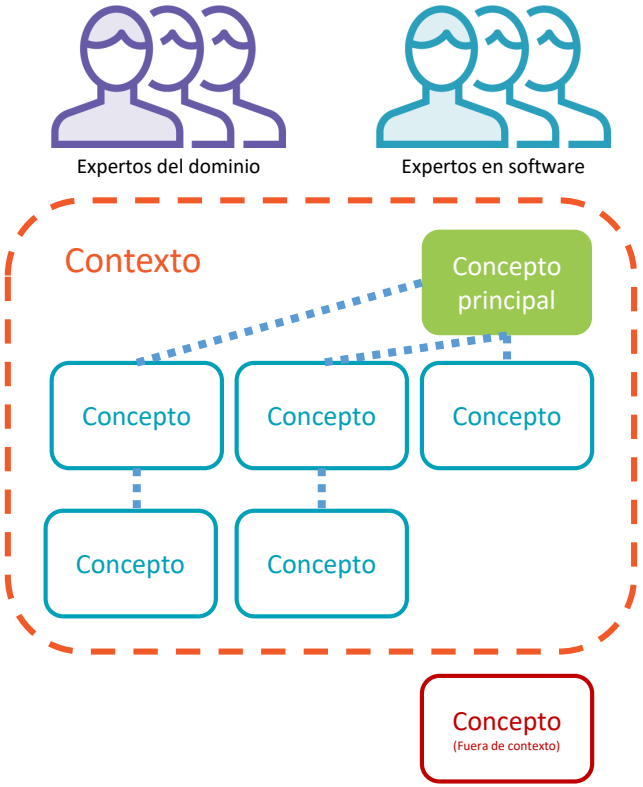


- Identificar el concepto de dominio principal
- Modelos internos (conceptos de apoyo)
- Cada BC tiene una interfaz explícita (entradas y salidas)
- Modelos compartidos para la comunicación
- Microservicio = Bounded Context
  - Pertenece a un equipo
  - Dueño de su propio almacén de datos.
  - Contratos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

Contextos sin limites
Enfoque usualmente utilizado
en el desarrollo tradicional

Conceptos\aspectos principales del dominio

Envíos (Delivery)

Conductores

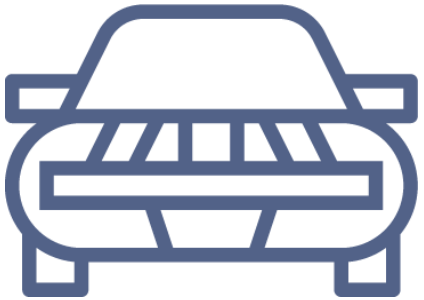
Pedidos

entregas

Pedidos de clientes (Customer Orders)

Pedidos

Devoluciones



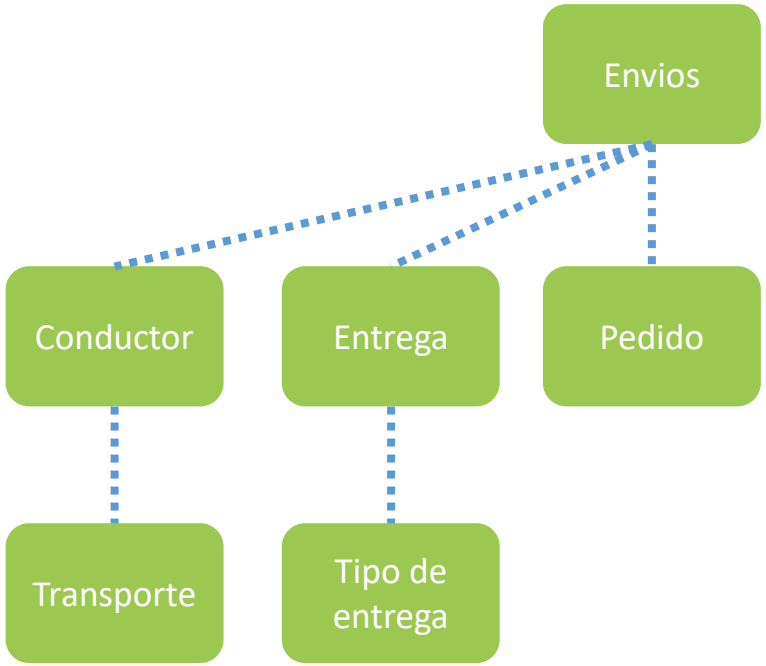
Conductor (Driver)

Transporte

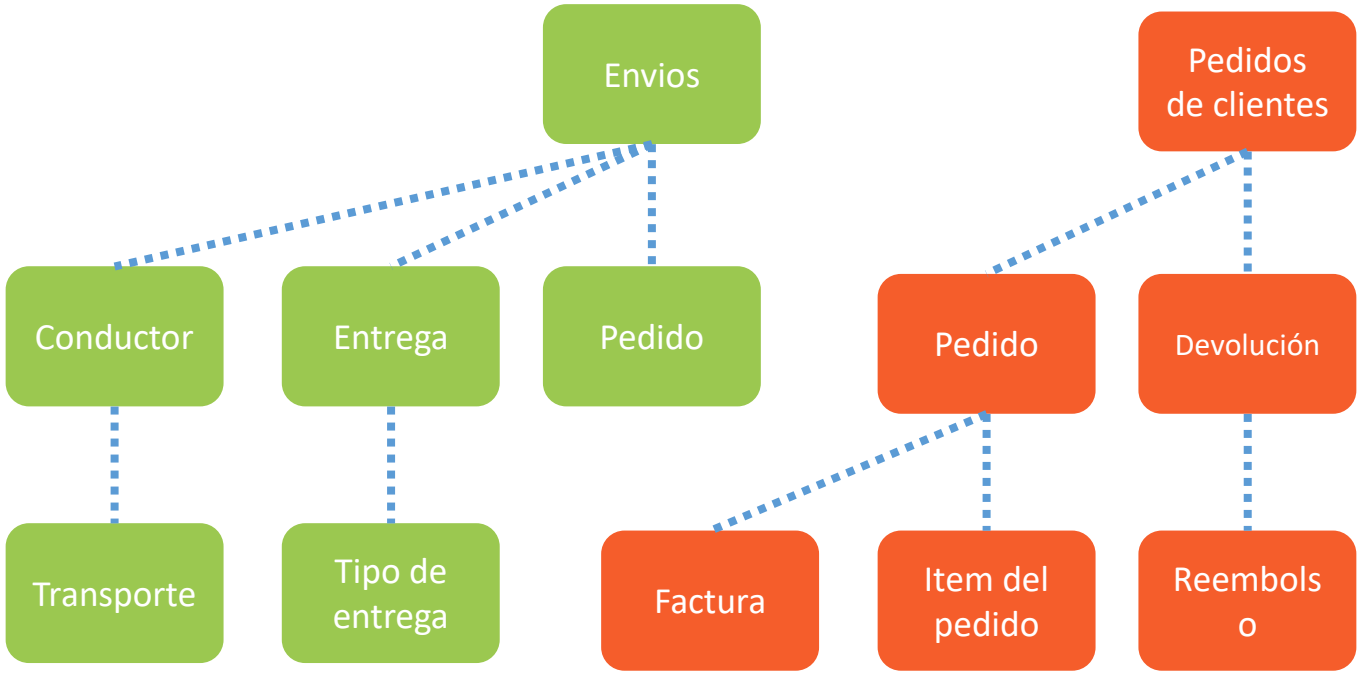
Turnos

Vacaciones anuales

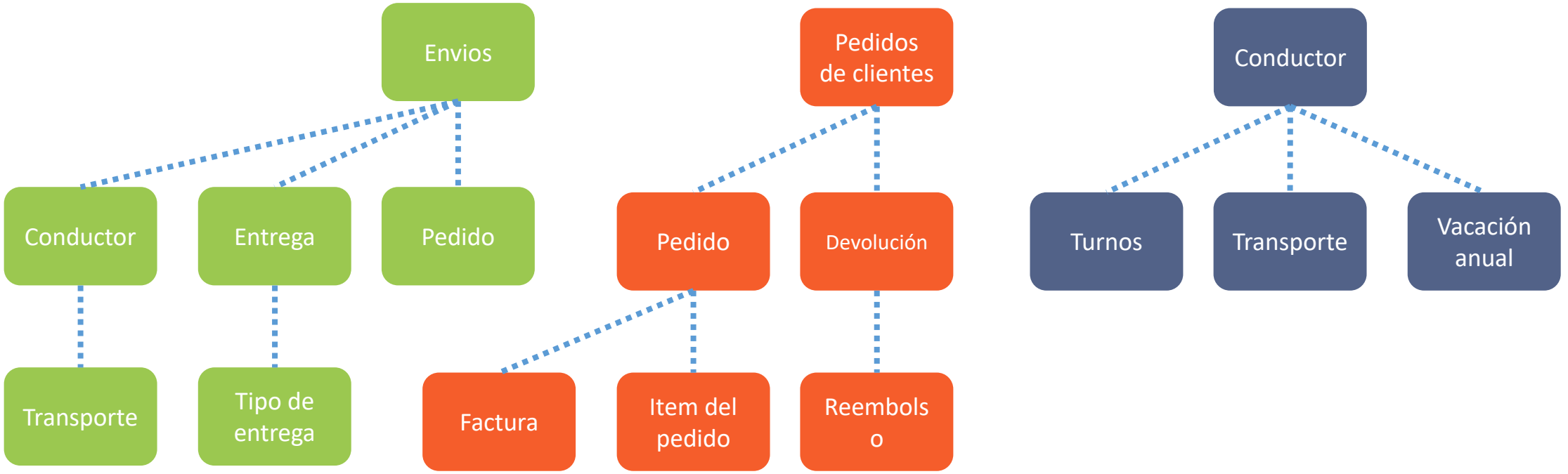
Conceptos principales y conceptos de soporte



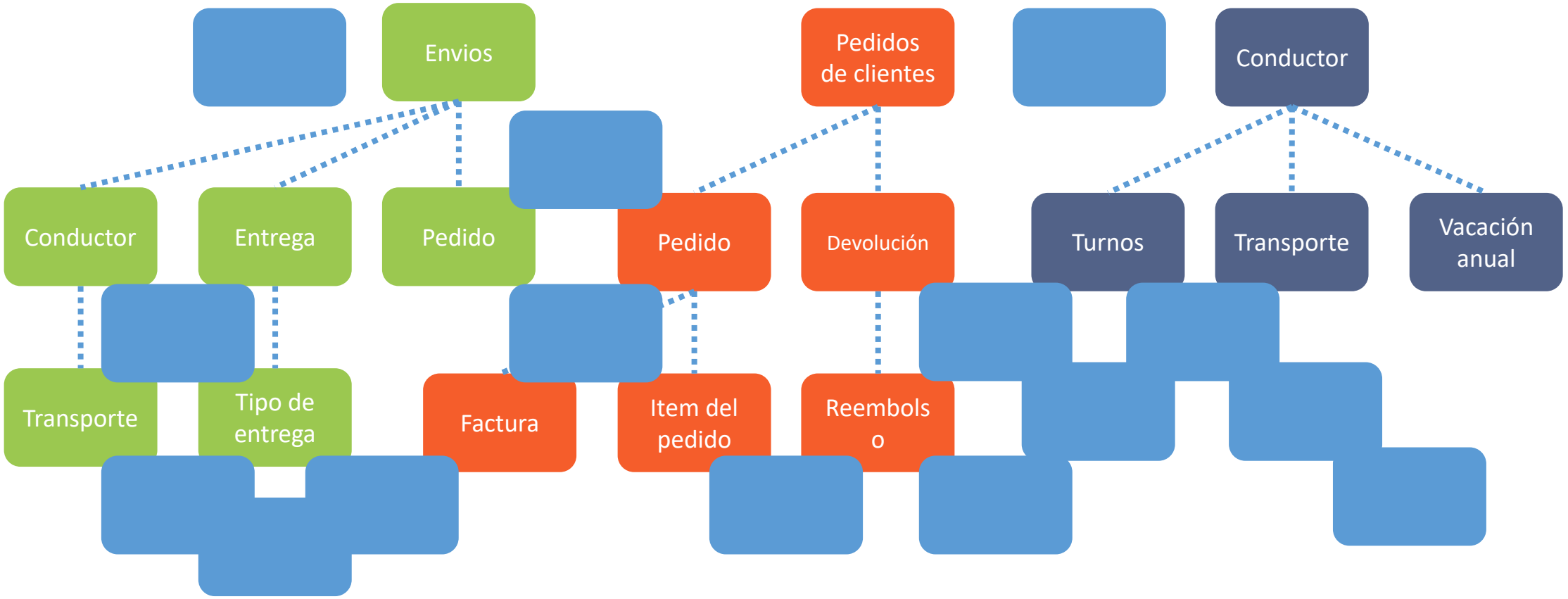
Conceptos principales y conceptos de soporte

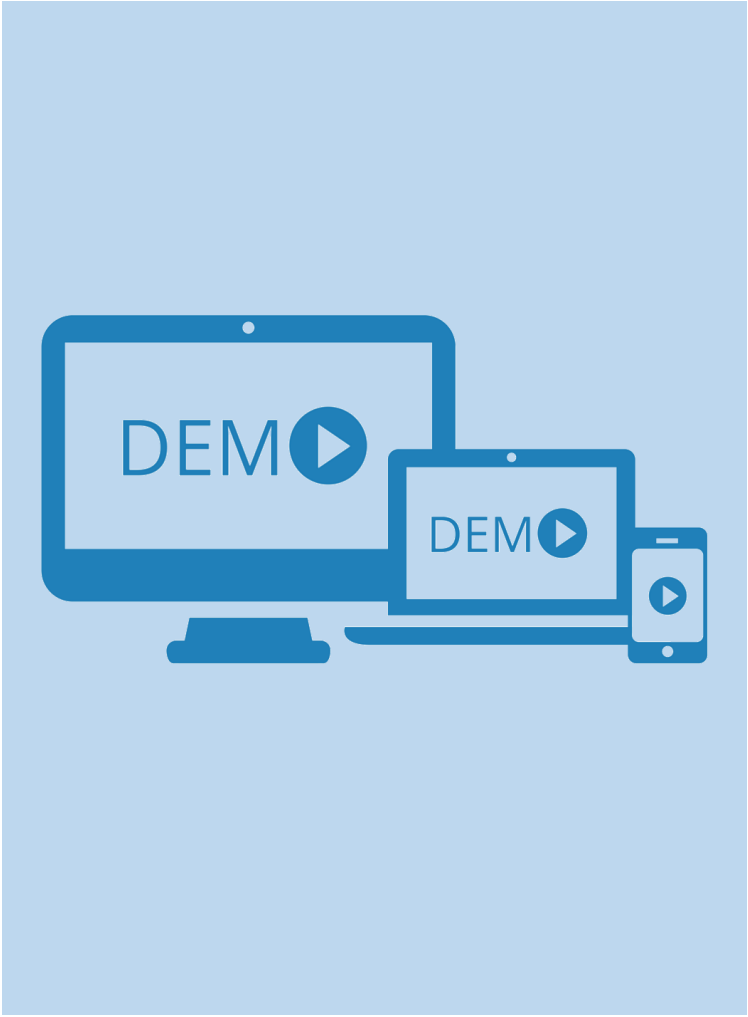


Conceptos principales y conceptos de soporte



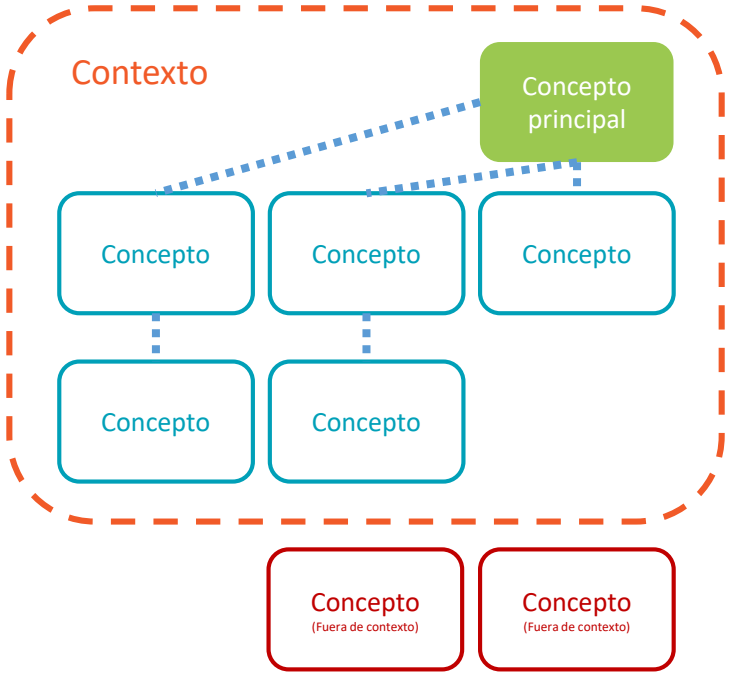
Contexto sin limites





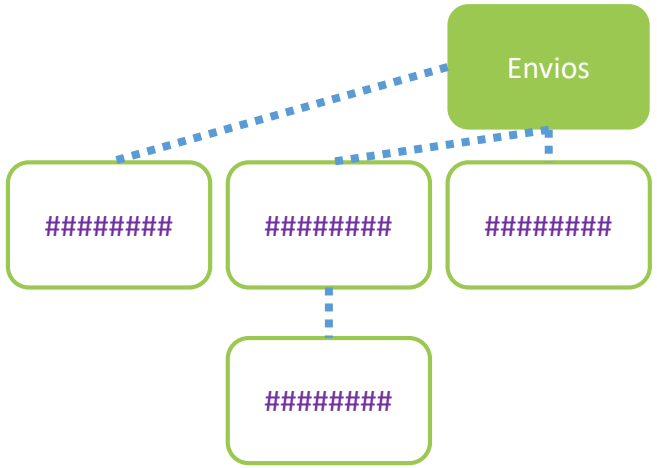
Uso de Bounded Context para microservicios

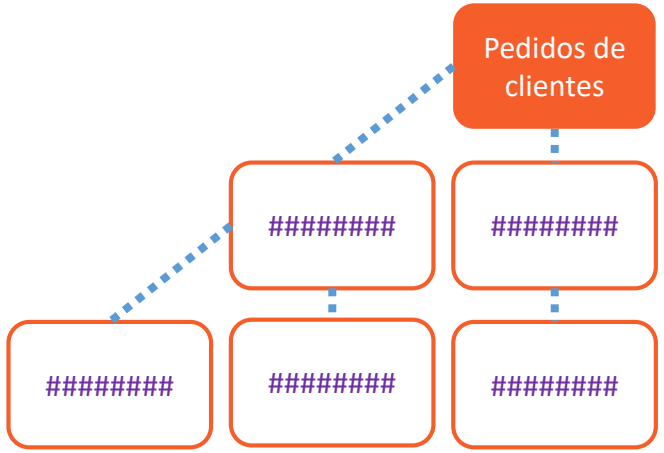
Bounded Context como técnica



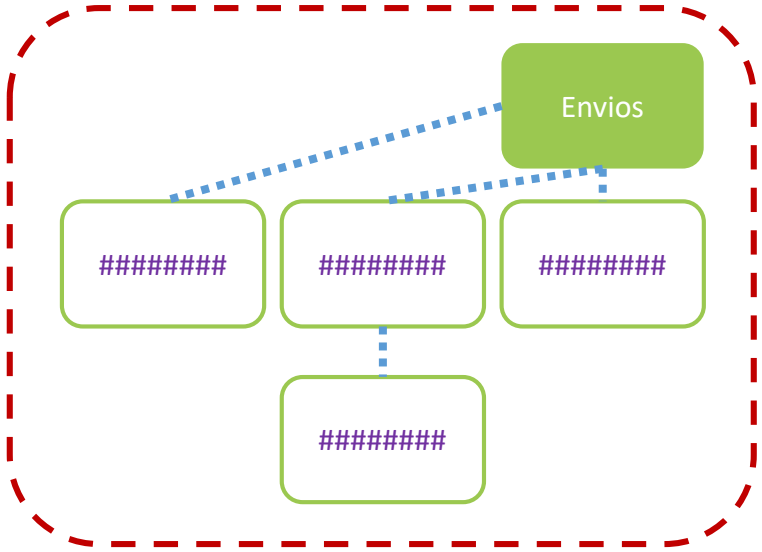
- Identificar los aspectos\conceptos clave del dominio
- Los conceptos forman los Bounded Contexts
- El concepto principal forma un lenguaje ubicuo
- Renombrar conceptos\modelos de soporte
  - Renombrar los conceptos LU a naturales
- Mover a contextos externos (fuera del contexto) conceptos\modelos
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

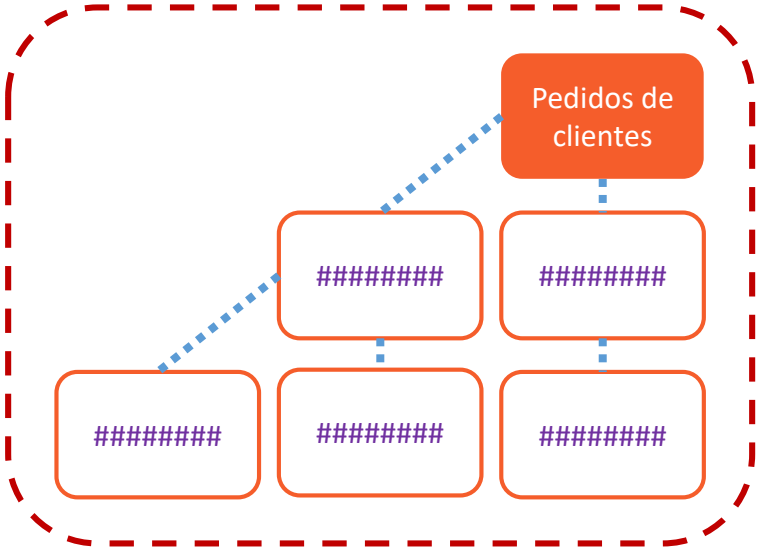
¿Identificando el core (conceptos principales)?



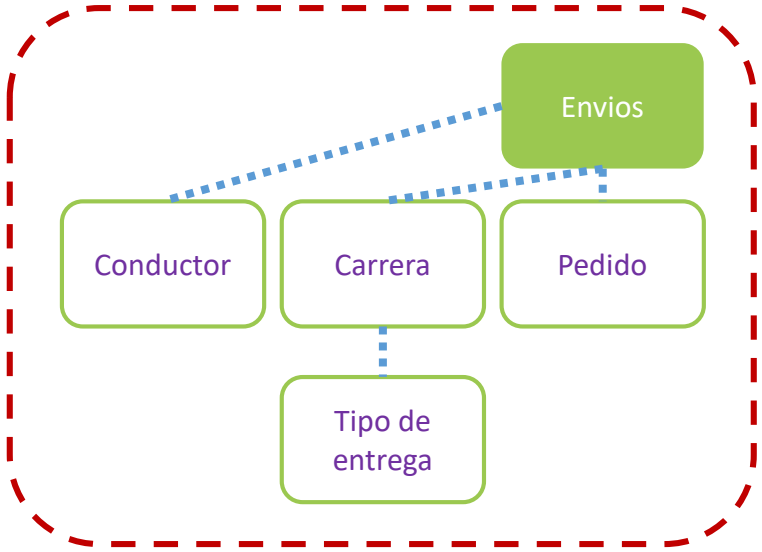


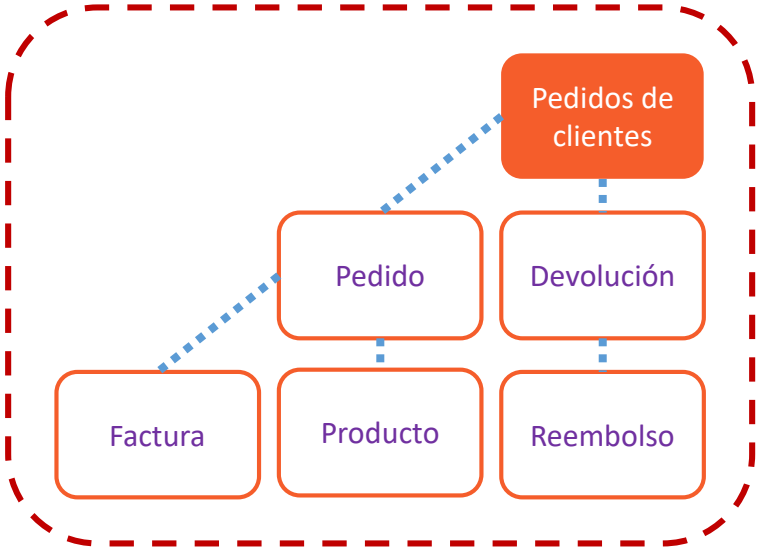
¿Identificando el core (conceptos principales)?



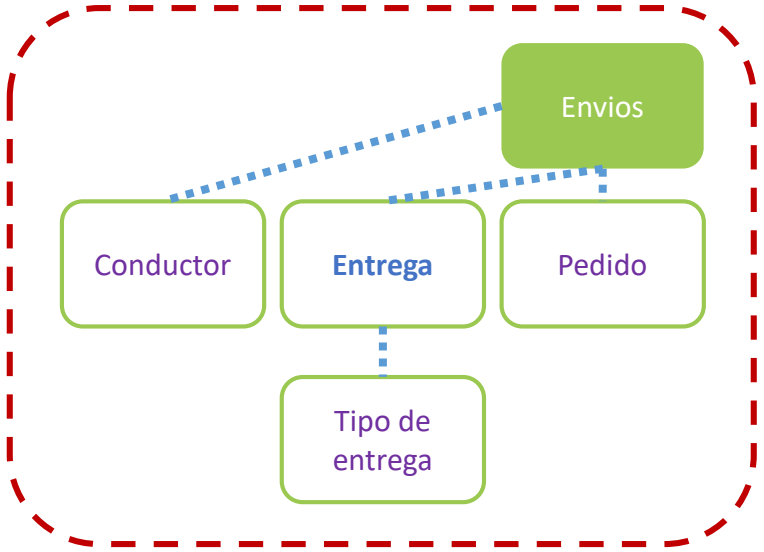


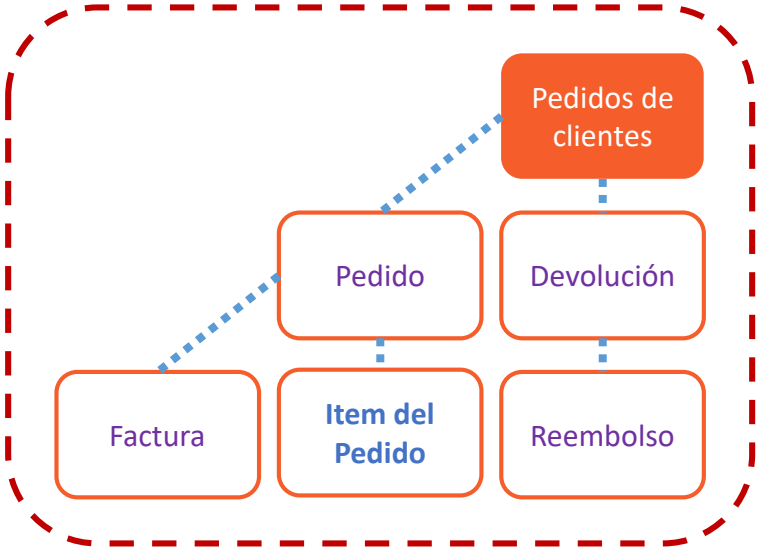
El concepto principal (Core) define el lenguaje ubicuo



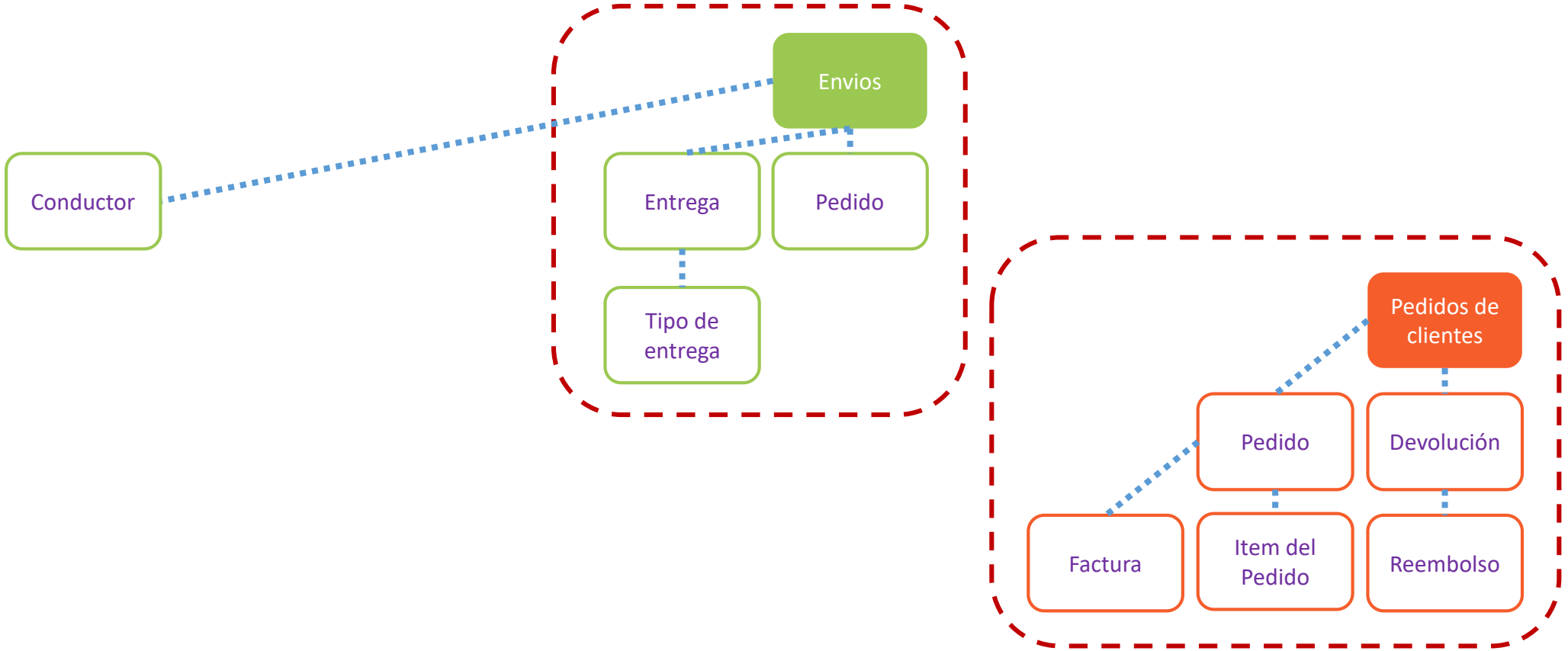


Renombrar conceptos

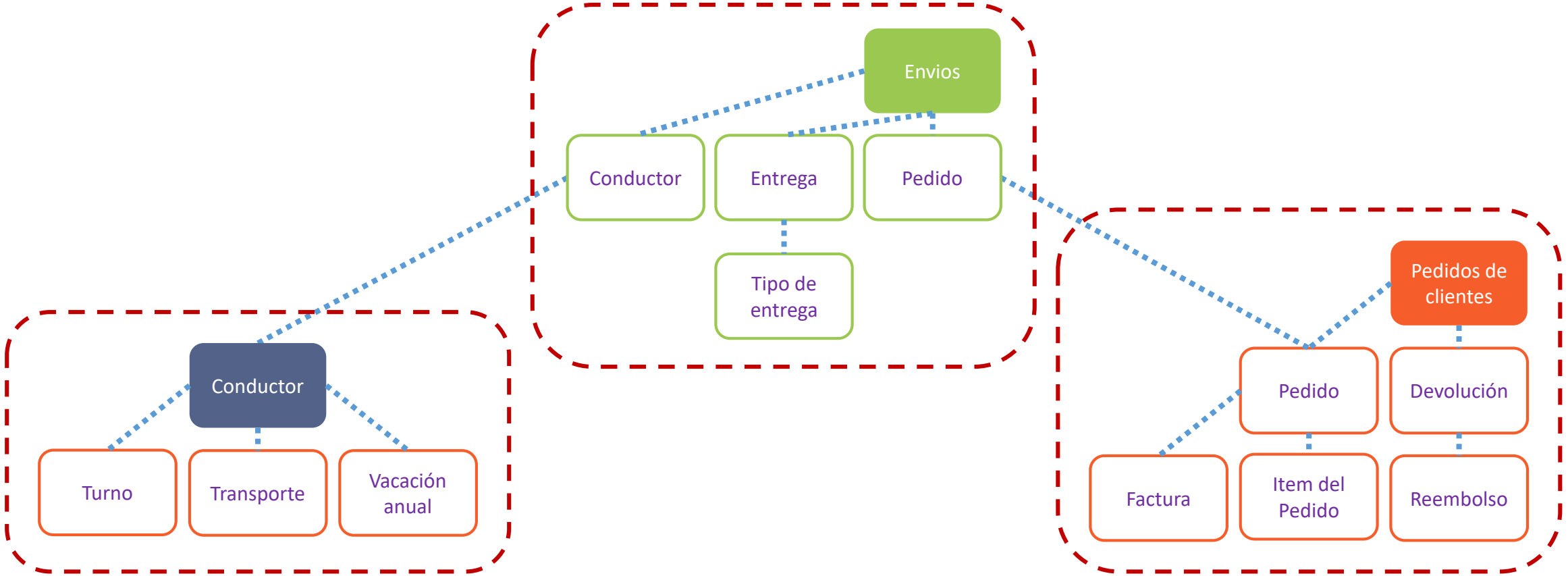




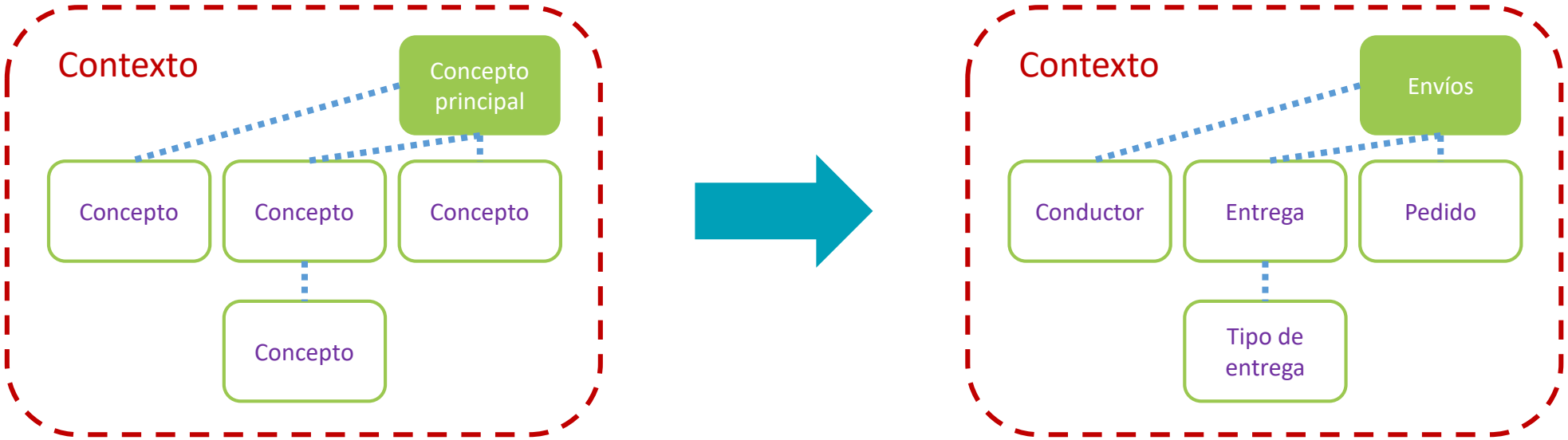
Conceptos fuera de contexto



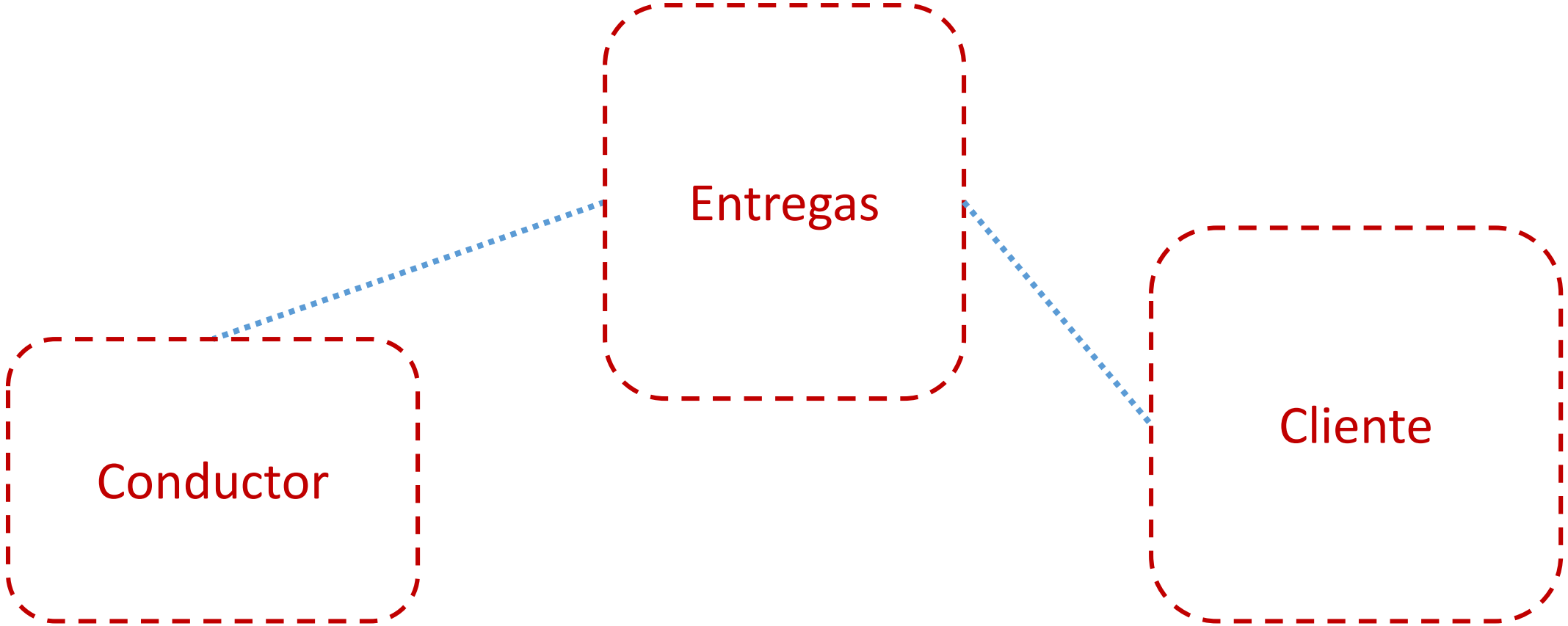
Conceptos únicos indican integración



Bounded Context a Microservicios

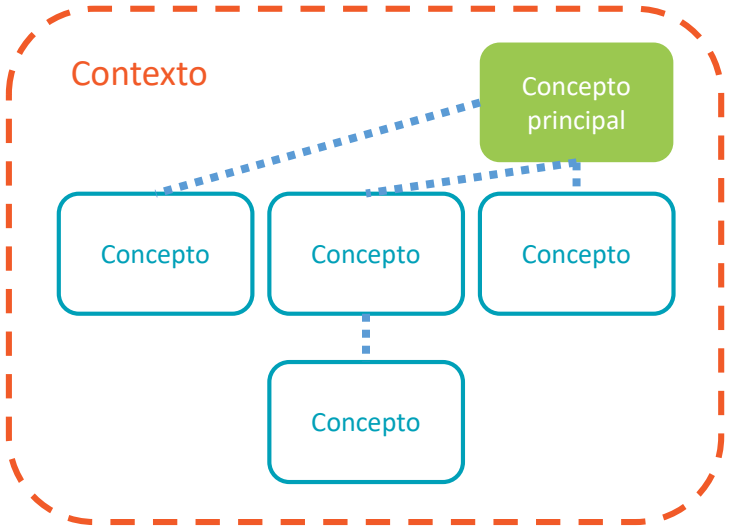


Los Bounded contexts se convierten en microservicios



Agregación

Agregación



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

¿Cuándo componer o realizar agregación de servicios?

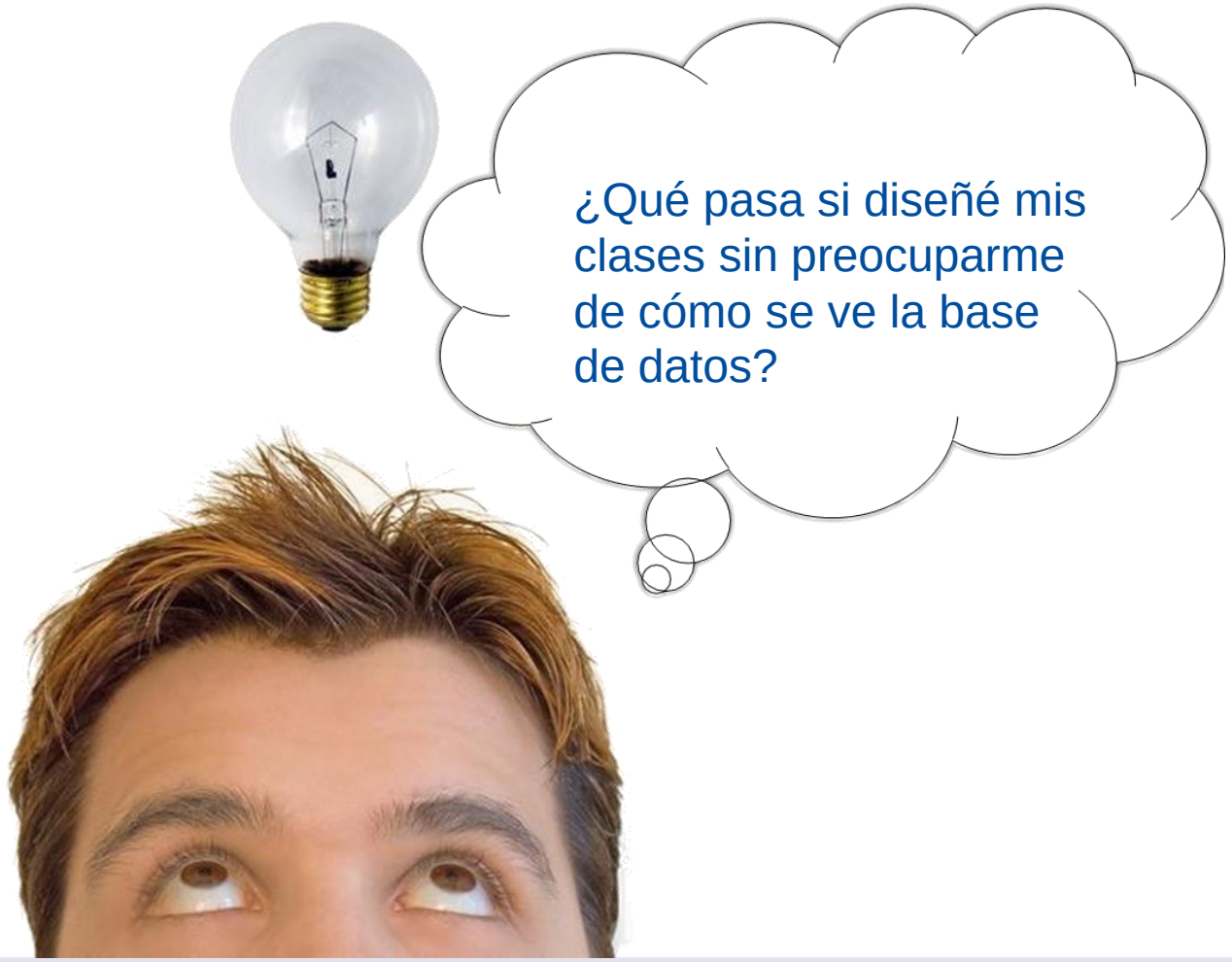
Aplicando el patrón DDD a un microservicio.

¿Que enfoque utilizar?

Procedural

Orientado a los Datos

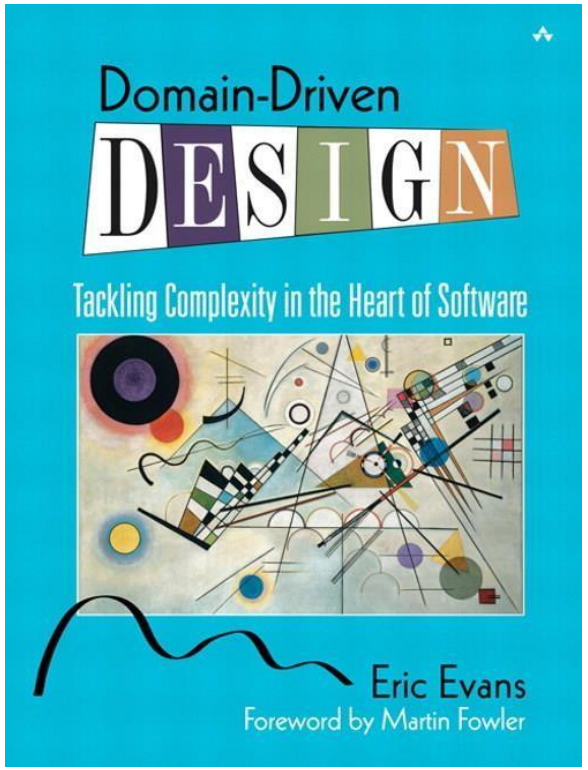
Orientado al negocio



ARQUITECTURA DE MICROSERVICIOS (DOMINIO Y GOBIERNO DE DATOS)

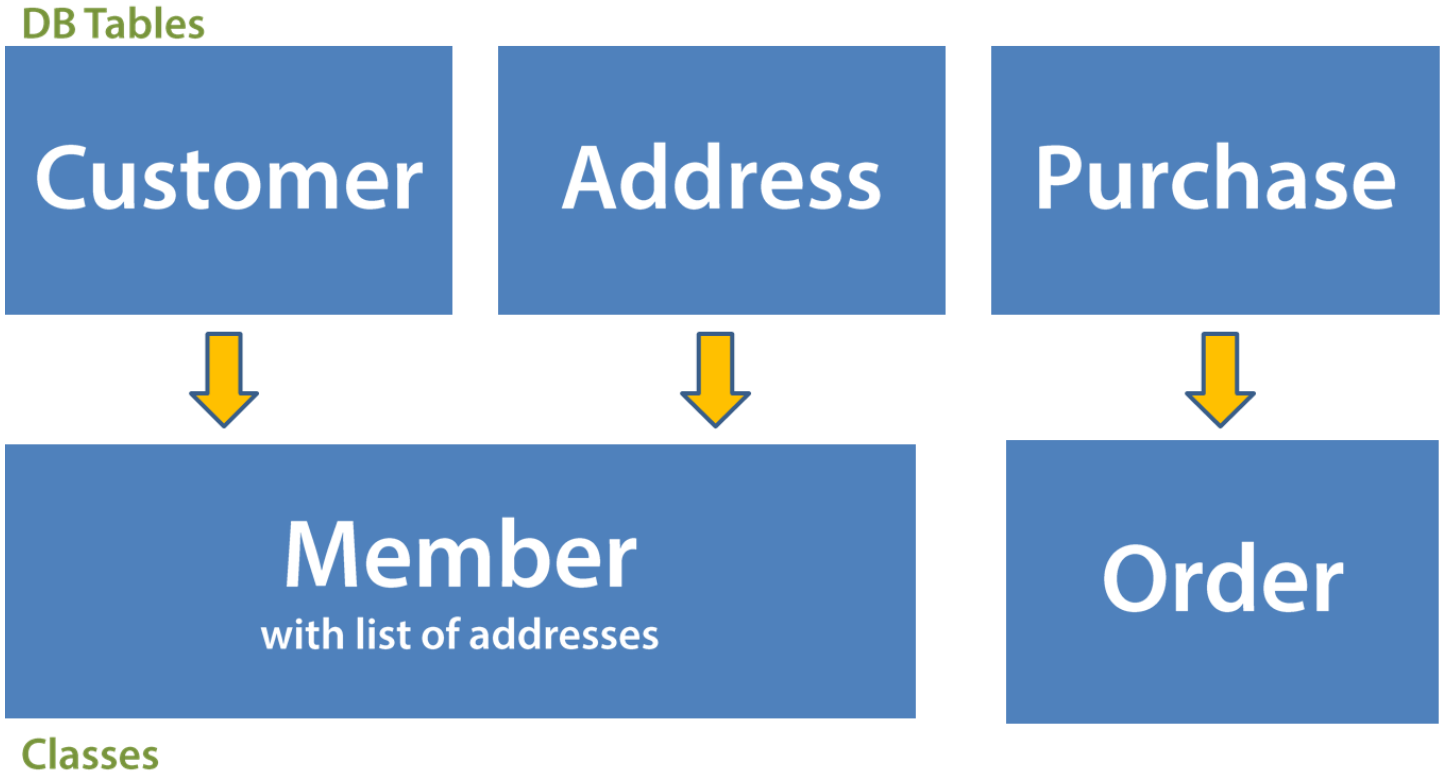
ARQUITECTURA DE MICROSERVICIOS (DOMINIO Y GOBIERNO DE DATOS)

Modelo de Dominio



Modelo de Dominio

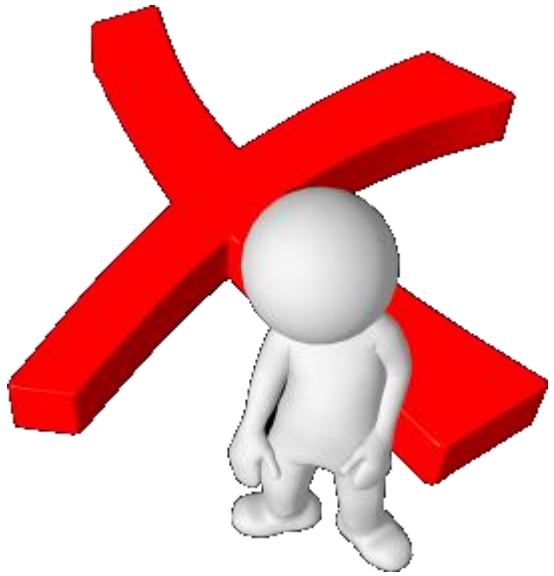
Modelo de Dominio



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
- Diseño que consume mucho tiempo
- Compromiso a largo plazo
- Gastos generales de mapeo de DB

Granularidad de servicios

¿Problema? ¿Por qué la "granularidad fina" es un problema de todos modos? ¿No se trata de SOA descomponer "silos" monolíticos en pequeños servicios reutilizables? Más aún, cuanto más fino es un servicio, menos contexto conlleva. Cuanto menos contexto tiene un servicio, más potencial de reutilización tiene, y la reutilización es uno de los santos griales de SOA, ¿no es así?

La reutilización es, de hecho, un objetivo, sin embargo, el culpable de los servicios detallados es la red.

Los servicios se consumen a través de redes, tanto locales (LAN) como remotas (extranets, WAN, etc.).

El resultado es que los servicios están sujetos a las limitaciones y costos incurridos por esas redes. Tratar de ignorar estos costos es exactamente lo que afectó a la mayoría, si no a todos, los enfoques de sistemas distribuidos RPC anteriores a SOA (Corba, DCOM, etc.)

Arnon Rotem-Gal-Oz el 18 de mayo de 2010 https://arnon.me/wp-content/uploads/2010/10/Nanoservices.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

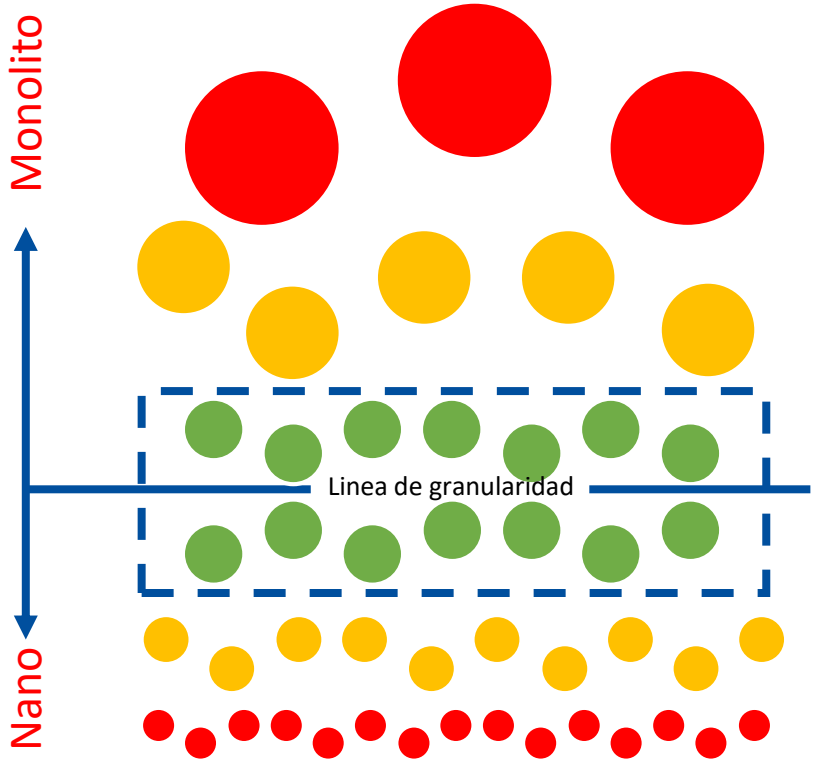
Los microservicios que son especialmente pequeños o detallados se denominan **nanoservicios**.

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

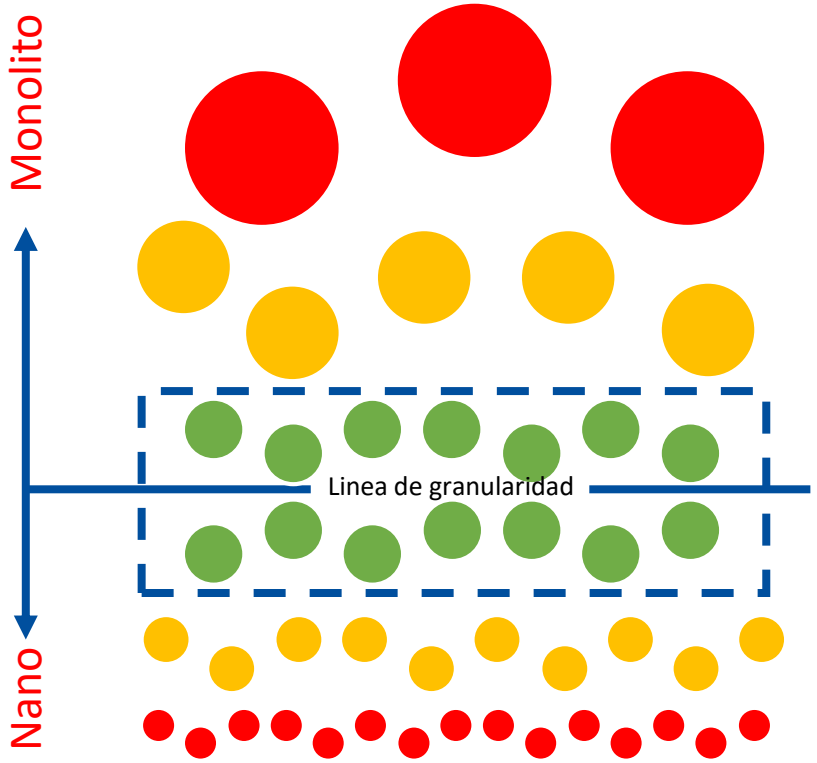


Supongamos que hay una línea horizontal imaginaria (eje x), que representa la línea de granularidad; el nivel adecuado determinado para MSA.

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 :

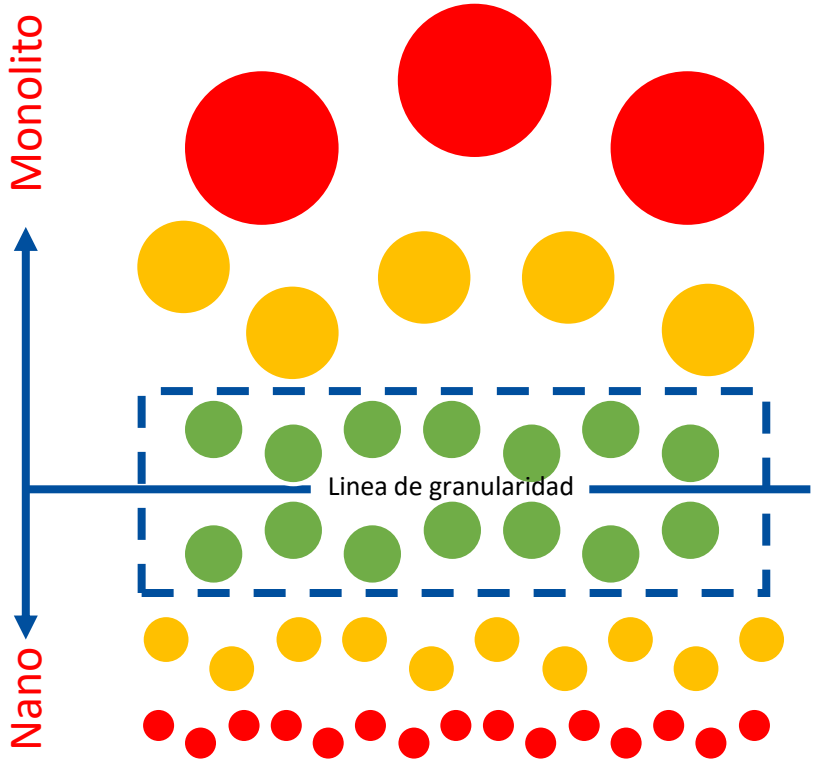
Granularidad de servicios



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.

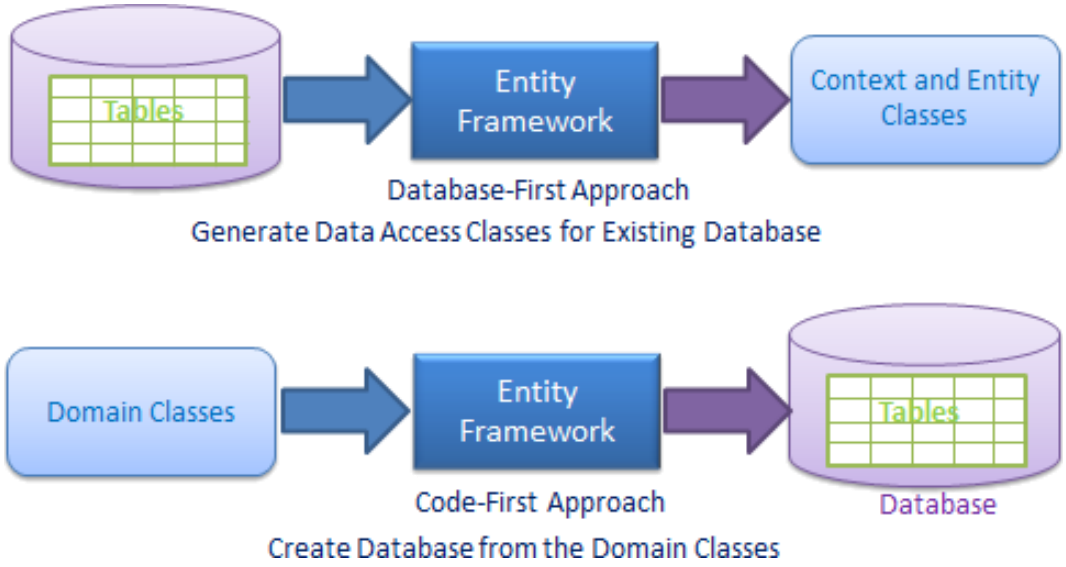
Aplicando el patrón DDD a un microservicio.

Infraestructura de persistencia - Entity Framework Core, MSSQL, Azure Data Base.

Entity Framework Core

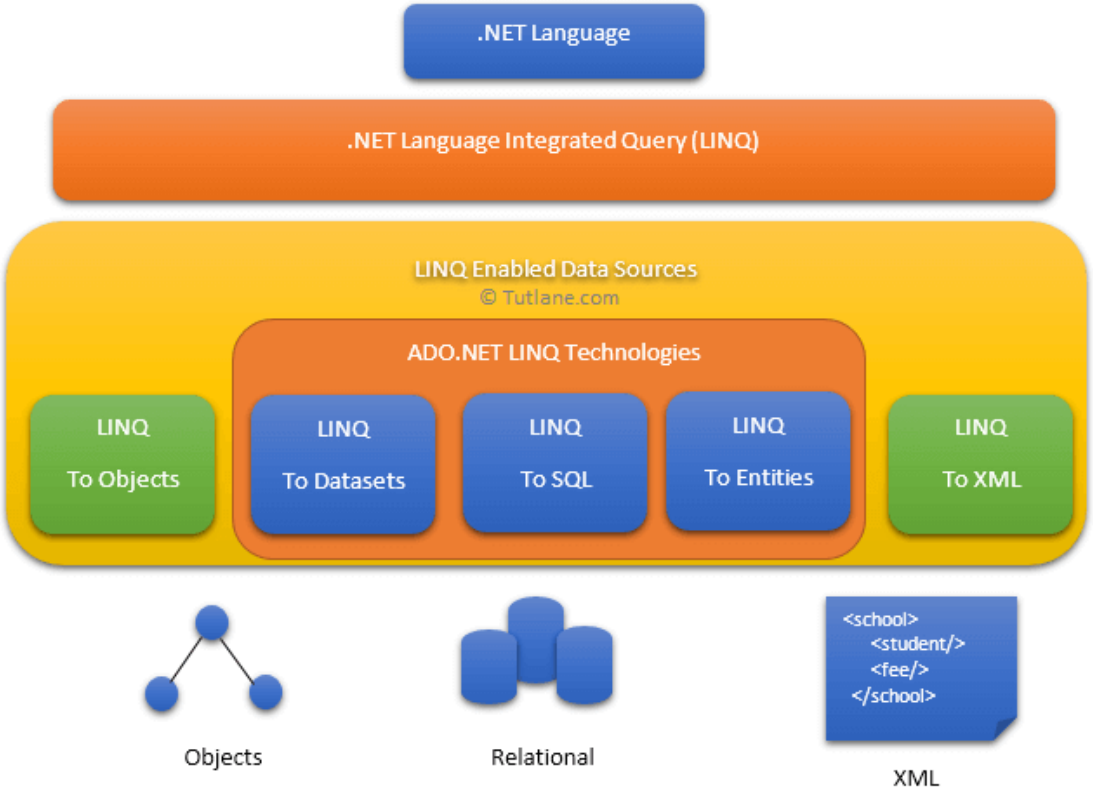
- Entity Framework (EF) Core es una versión ligera, extensible, de código abierto y multiplataforma de la popular tecnología de acceso a datos Entity Framework.
- EF Core puede servir como asignador relacional de objetos (O/RM), lo que permite a los desarrolladores de .NET trabajar con una base de datos 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



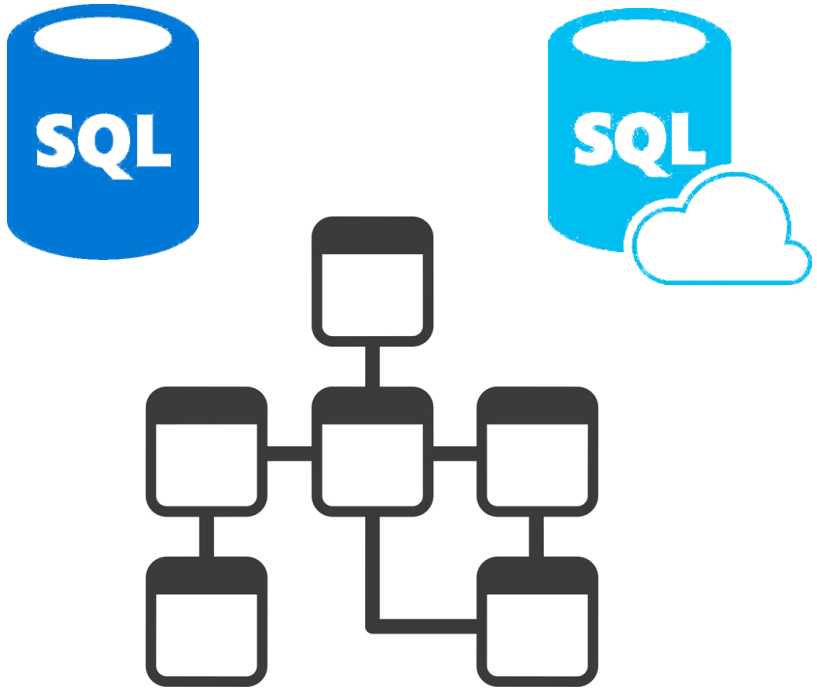
- Con EF Core, el acceso a datos se realiza mediante un modelo.
- Un modelo se compone de clases de entidad y un objeto de contexto que representa una sesión con la base de datos, lo que permite consultar y guardar los datos.
- Puede generar un modelo a partir de una base de datos existente, codificar 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



- Entity Framework 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.

GRACIAS

POR SU PREFERENCIA

GALAXY
TRAINING