

MICROSERVICIOS



Your Everything Innovation Partner

Confidential and Proprietary. © 2019 UST Global Inc





USTGlobal®

MICROSERVICIOS

Presentado por:
Daniel Nieto

AGENDA

01 FUNDAMENTOS MICROSERVICIOS

1. Arquitectura tradicional vs Microservicios
2. Herramientas de construcción
3. Beneficios y costos de la adopción de esta estrategia en una organización
4. Microservicios Beneficios y Anti patrones

03 MANEJO DE DATOS

1. Comunicación basada en eventos
2. Patrones de Gestión de datos
3. Patrón Event sourcing

04 ARQUITECTURA Y DISEÑO DE MICROSERVICIOS

1. Componentes de presentación
2. Lógica de dominio o de negocios
3. Lógica de acceso a la base de datos
4. Lógica de integración de aplicaciones

02 ARQUITECTURA DE MICROSERVICIOS

1. Construyendo bloques
2. Microservicios Restful
3. Comunicación a través de un puente API



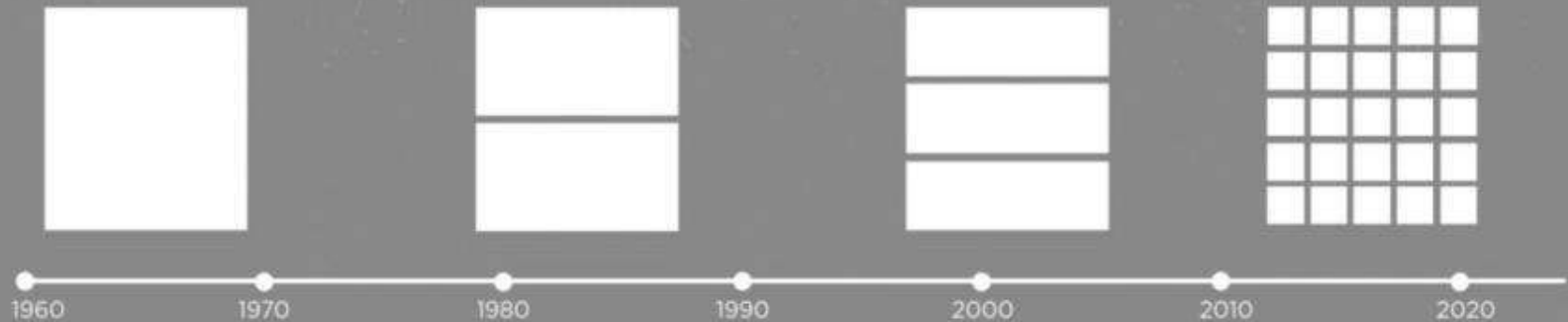
CONTEXTUALIZÁNDONOS

La evolución de las tecnologías de Nube trajo beneficios importantes como agilidad y **eficiencia** en el aprovisionamiento y **uso de recursos de cómputo**. El Cloud ha cambiado para siempre el mundo de la computación y ha favorecido la proliferación de las aplicaciones basadas en microservicios en detrimento de las arquitecturas monolíticas.



EVOLUCIÓN

ABSTRACTION IN THE ENTERPRISE



Era of
Mainframe



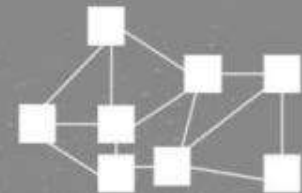
Era of Client
Server



Era of the
Web



Era of
the Cloud



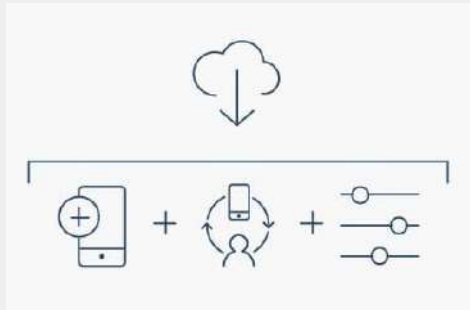
CONCEPTOS IMPORTANTES DE LA NUBE



IAAS

INFRAESTRUCTURA COMO SERVICIO

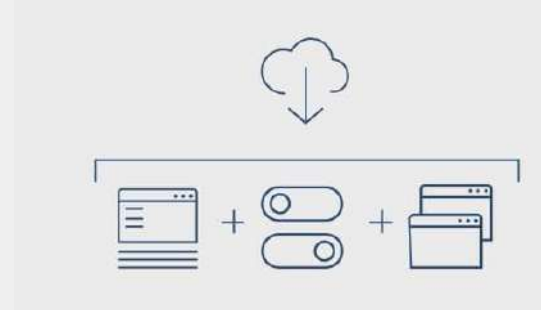
Un proveedor proporciona a los clientes acceso directo a almacenamiento, red, servidores y otros recursos de computación en la nube.



PAAS

PLATAFORMA COMO SERVICIO

Plataforma como servicio (PaaS) es un entorno de desarrollo e implementación completo en la nube, con recursos que permiten entregar todo

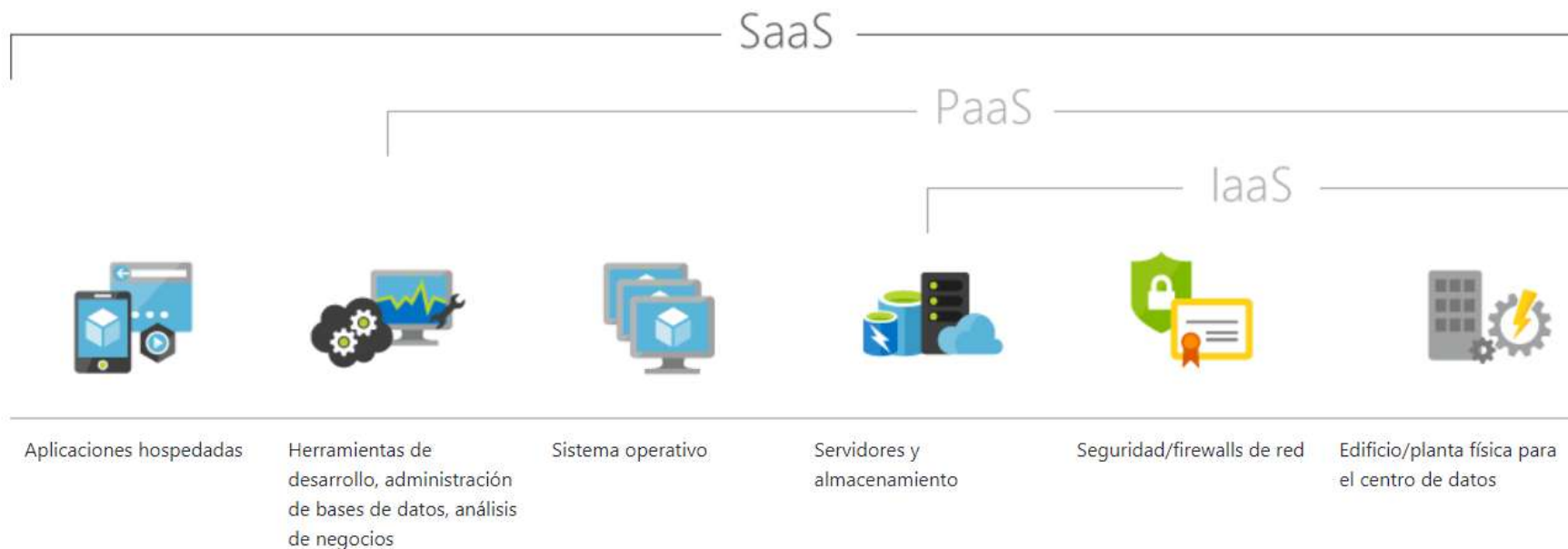


SAAS

SOFTWARE COMO SERVICIO

Es una aplicación para el usuario final donde paga un alquiler por el uso de software. No es necesario adquirir un software en propiedad

EN RESUMEN



¿QUÉ ES UNA ARQUITECTURA MONOLÍTICA ?



En ingeniería de software, una aplicación monolítica hace referencia a una aplicación software en la que la capa de interfaz de usuario y la capa de acceso a datos están **combinadas** en un **mismo programa** y sobre una misma plataforma. La filosofía de diseño es que la aplicación es **responsable no sólo de una única tarea**, si no que es capaz de realizar **todos** los pasos o tareas necesarias para completar una determinada función

VENTAJAS



Las aplicaciones monolíticas son fáciles de debuggear y probar, generalmente obtienen un desempeño superior dado que los componentes se encuentran dentro del mismo entorno de desarrollo, así mismo, la seguridad y el diseño se ven simplificados.

DESVENTAJAS



Este tipo de aplicaciones es muy dependiente del hardware. Esta gran dependencia se materializa en largos tiempos de inactividad a la hora de realizar actualizaciones del software o despliegues de nuevo hardware para añadir más capacidad.

¿QUE ES SOA - ARQUITECTURA ORIENTADA A SERVICIOS?

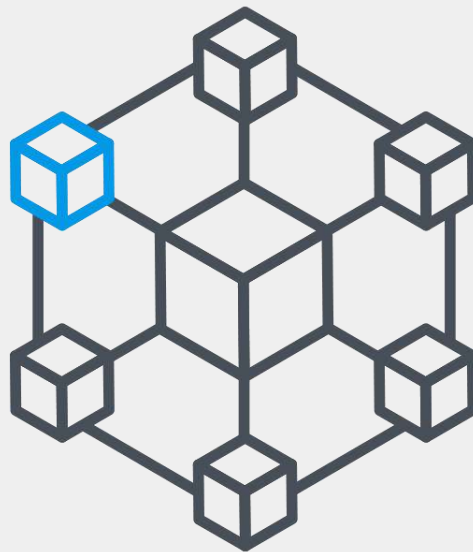


Frente a las limitaciones impuestas por los sistemas tradicionalmente monolíticos y a la evolución de las aplicaciones distribuidas surgieron nuevas arquitecturas enfocadas en aliviar este tipo de problemas.

RECORDEMOS.. ¿QUÉ ES UN SISTEMA DISTRIBUIDO?

Un sistema distribuido es un conjunto de equipos **independientes** que actúan de forma **transparente** actuando como un único equipo.

Su objetivo es **descentralizar** tanto el almacenamiento de la información como el procesamiento.





¿ENTONCES QUÉ ES SOA VS MICROSERVICIOS?

SOA

Service Oriented Architecture

Definición

Modelo arquitectónico para sistemas distribuidos, cuyos fines son mejorar la agilidad organizacional, mantener una alineación entre negocio y TI, y al mismo tiempo, implementar la Orientación a Servicios.

Bases

Se basa en **Servicios** como unidades fundamentales que engloban la lógica de negocio. Así, lo procesos de negocio son compuestos de dichas piezas.

Foco

Sigue una serie de principios de la Orientación a Servicios y patrones de diseño. Se enfoca en la composición y reutilización de Servicios y promueve que éstos sean agnósticos en su mayoría, y orquestaciones para procesos completos.

Tipos de Servicios

Servicios Web SOAP
Servicios REST
Componentes

Microservices

Martin Fowler



1

2

3

1

Enfoque para desarrollo de una sola aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso, y en constante comunicación usando mecanismos ligeros, a menudo un API HTTP.

2

Están contruidos alrededor de las capacidades de negocio y tienen independencia de despliegue, gracias a la maquinaria totalmente automatizada que los soporta.

3

Son altamente escalables y cada uno posee una firma asociada limitada de un módulo. Diferentes servicios pueden ser escritos en diferentes lenguajes de programación y también pueden ser administrados por diferentes equipos de trabajo.

<http://martinfowler.com/articles/microservices.html>

Microservices

SOA Patterns ORG



<http://soapatterns.org>

¿Cómo puede un servicio desplegarse de forma independiente para evitar las limitaciones impuestas por una implementación monolítica?

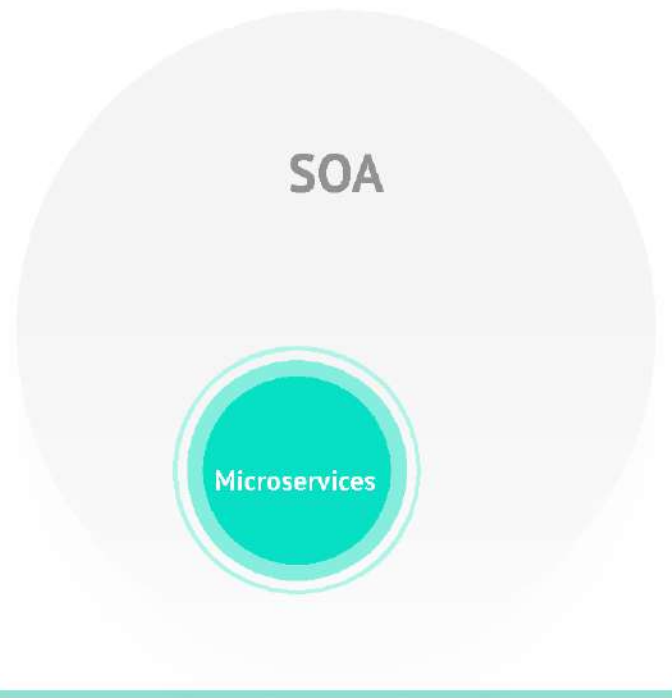
Cada servicio es tratado como un producto independiente, por ende se desarrolla, se empaqueta y se despliega de forma independiente. Dicho servicio puede ser calificado como un "Microservice".

Los Microservices están diseñados como servicios altamente autónomos que dependen principalmente de la comunicación asíncrona entre los distintos servicios.

El uso de contenedores de software (patrón Containerized Service Deployment Pattern) produce una variación del modelo. La tecnología de los contenedores se utiliza a menudo para empaquetar una aplicación completa junto con el servidor de aplicaciones y otra infraestructura necesaria para generar un único despliegue. Utilizando la tecnología de contenedores, cada Microservice puede ser "contenerizado" de forma independiente como si se tratara de una aplicación independiente.

SOA vs Microservices

Relación entre SOA, Microservices y la Orientación a Servicios

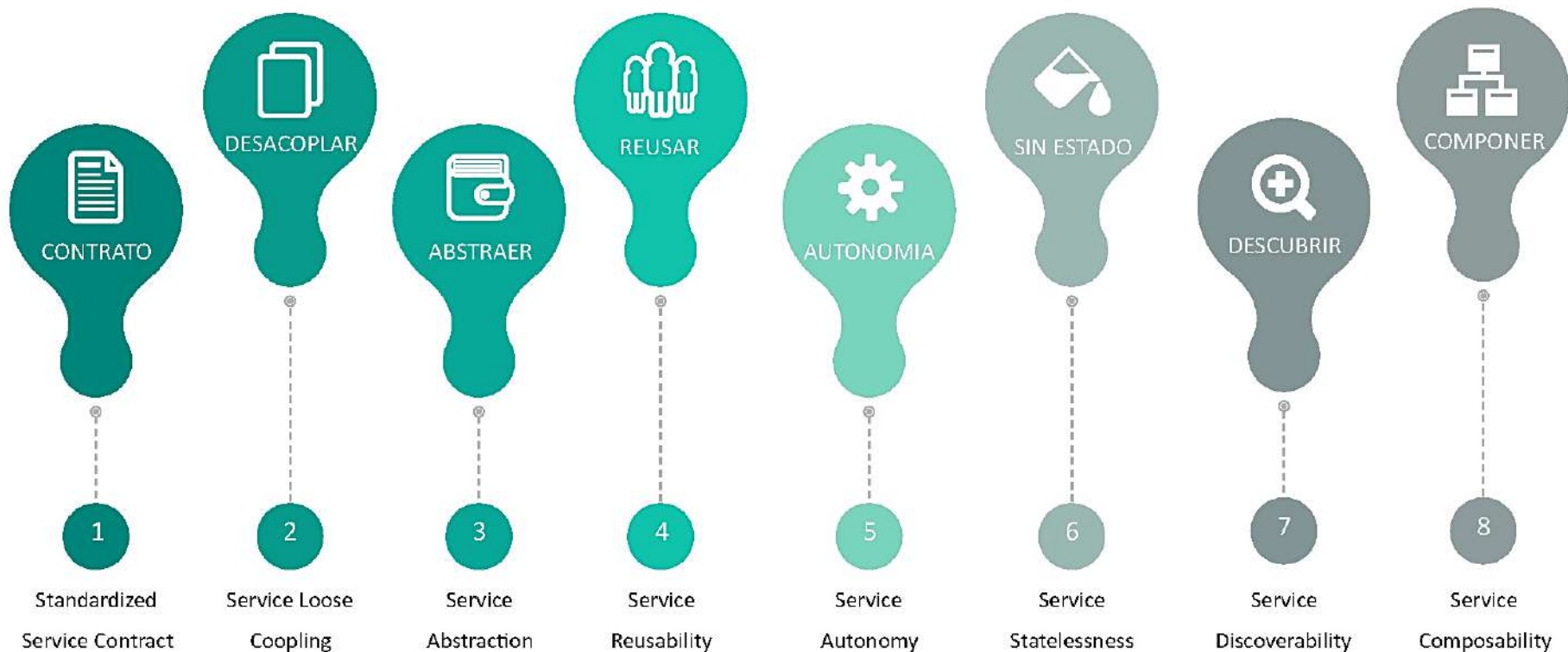


Los microservicios se **derivan** de SOA. Sin embargo, SOA es diferente de la arquitectura de microservicios ya que características como los orquestadores centralizados al nivel de la organización, y los buses de servicio empresariales (ESB - Enterprise service bus) son típicos en **SOA**. Pero en la mayoría de los casos, estos son anti-patrones para los microservicios.

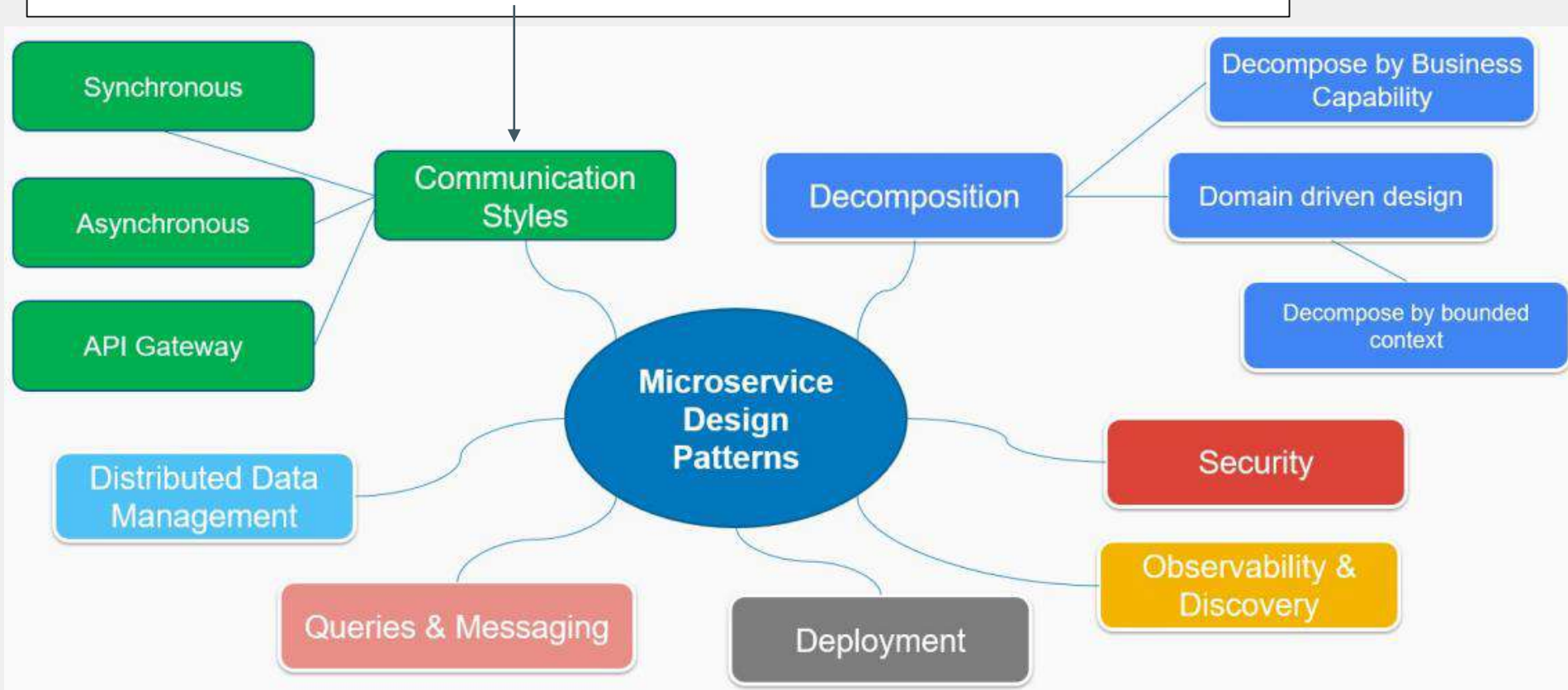
El enfoque SOA es menos "exigente" que los requisitos y técnicas utilizados en una arquitectura de microservicios. Si sabe cómo crear una aplicación basada en microservicios, también sabe cómo crear una aplicación más simple orientada al servicio

Orientación a Servicios

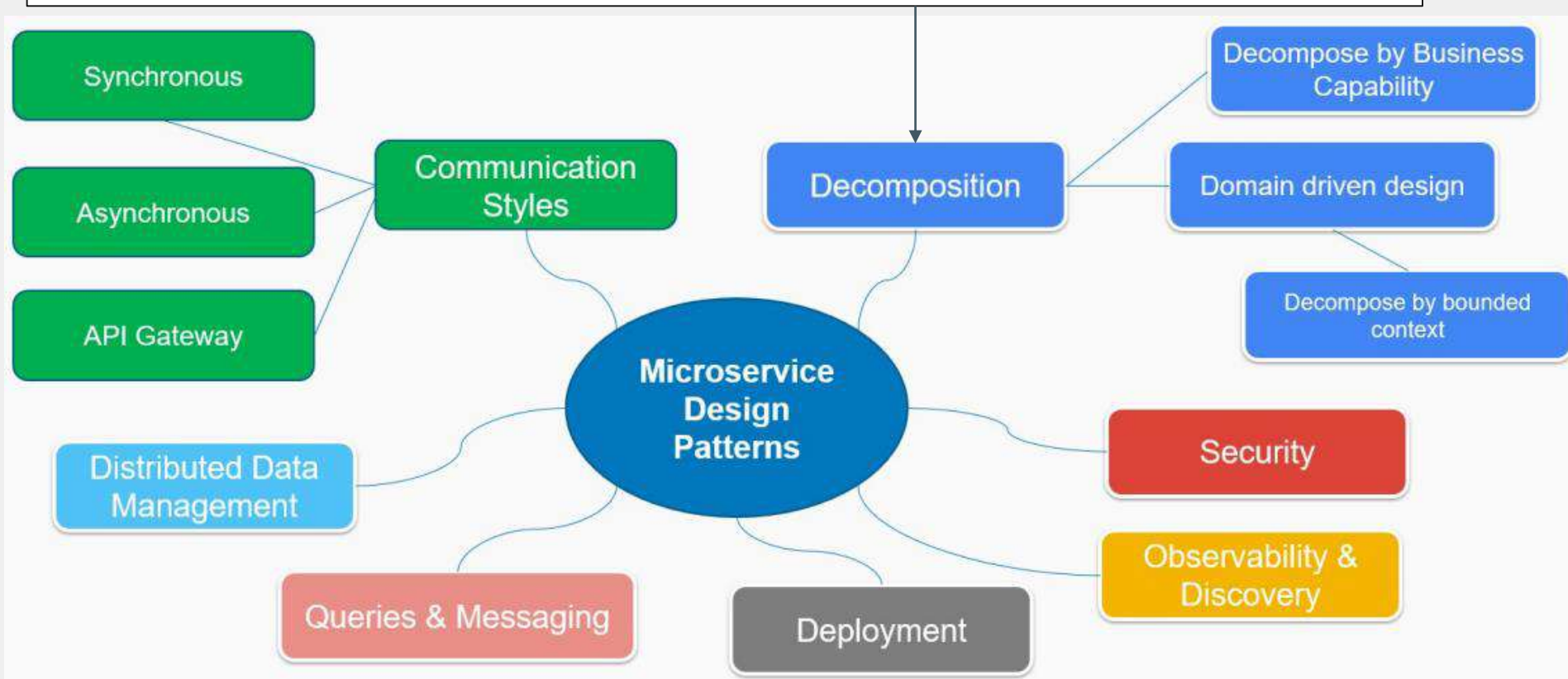
Principios Básicos

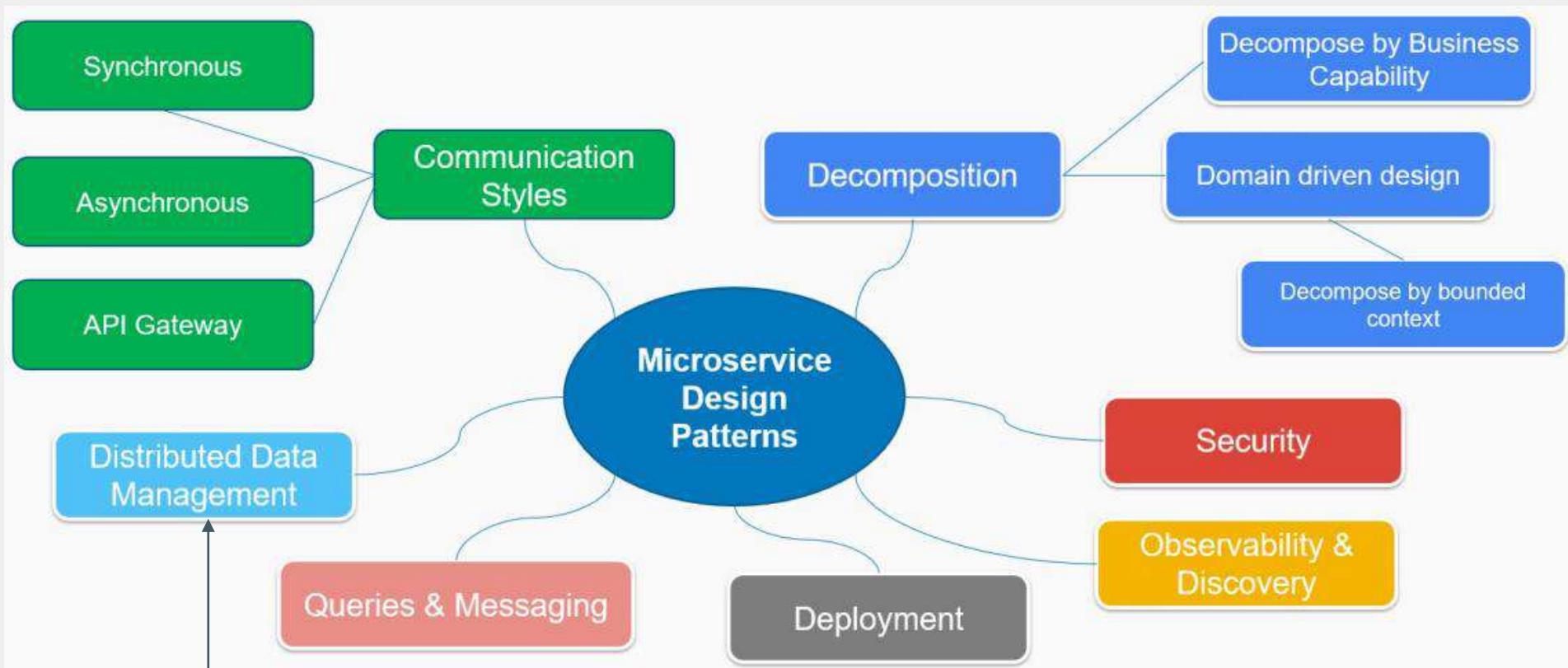


Estilos de Comunicación: Cómo se comunican los servicios entre si y externamente

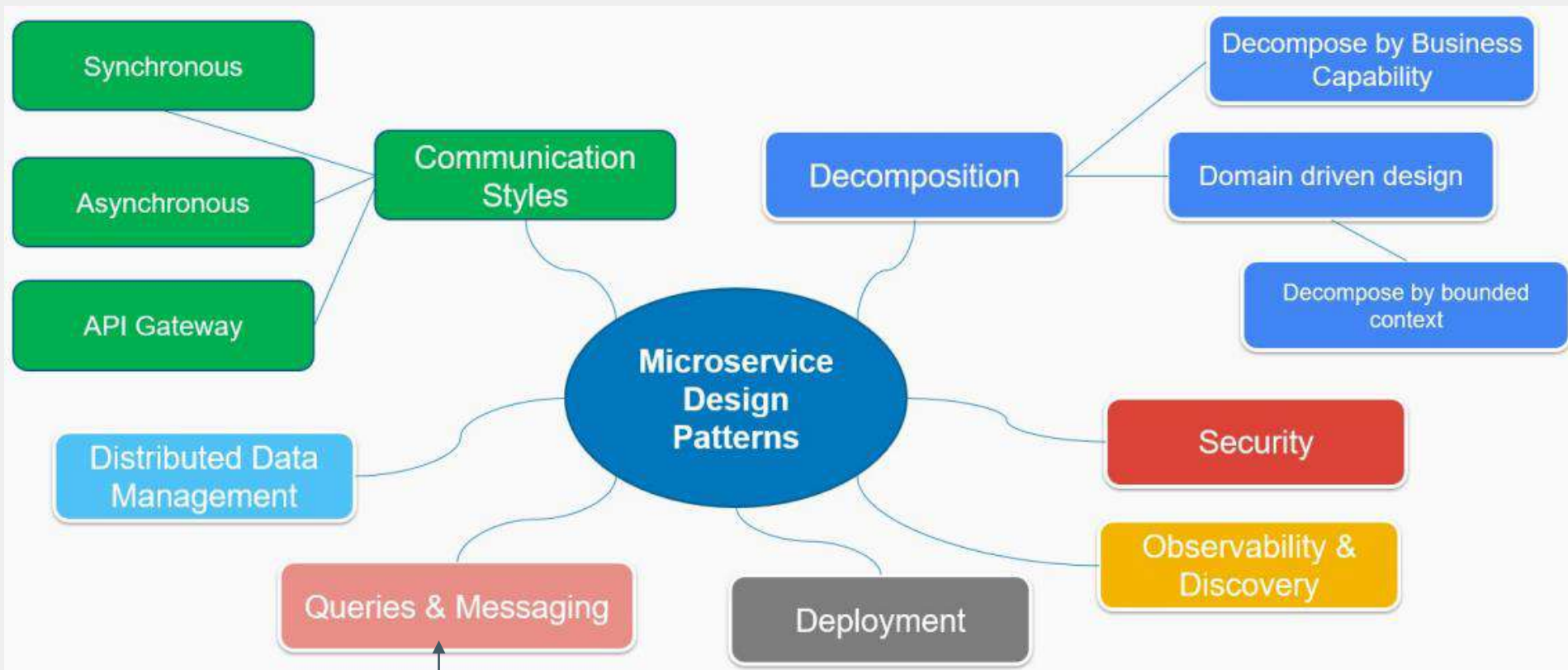


Patrón de Descomposición: Crear servicios de bajo acoplamiento por cada funcionalidad del negocio

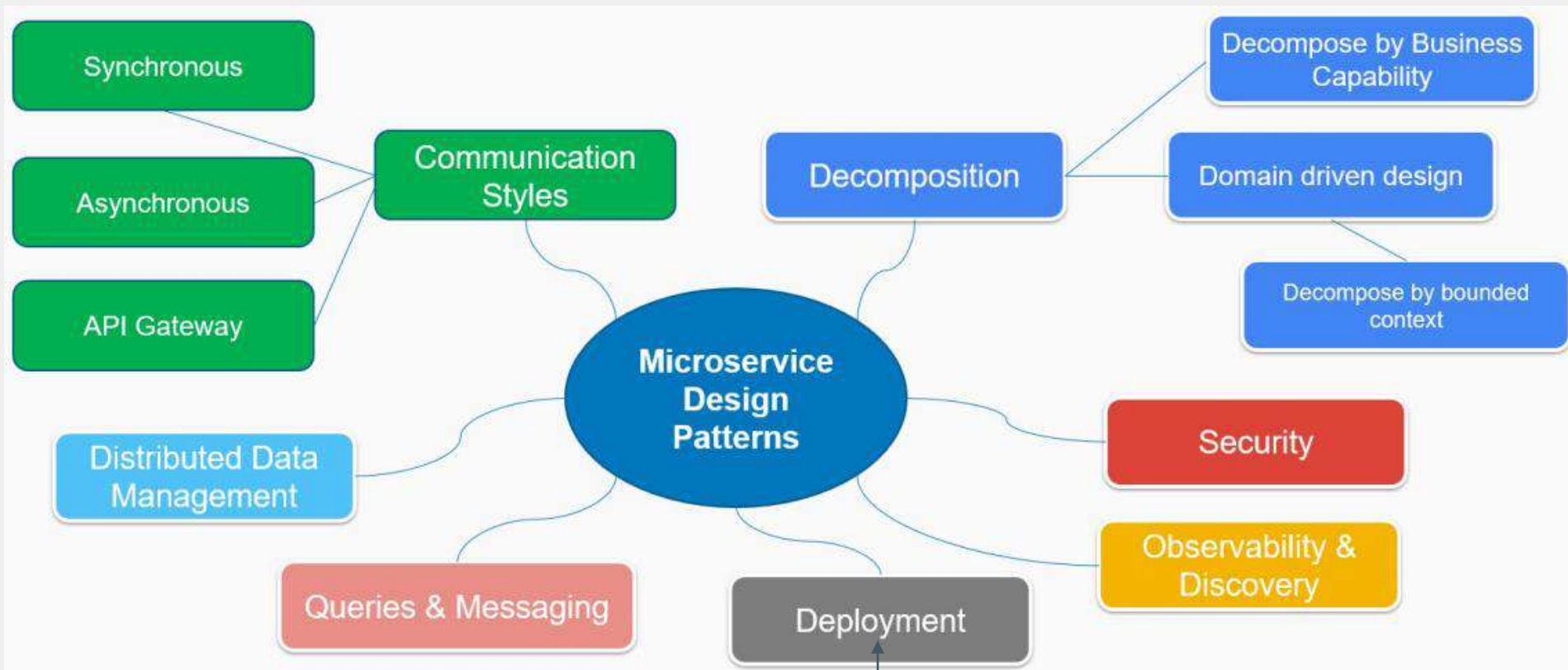




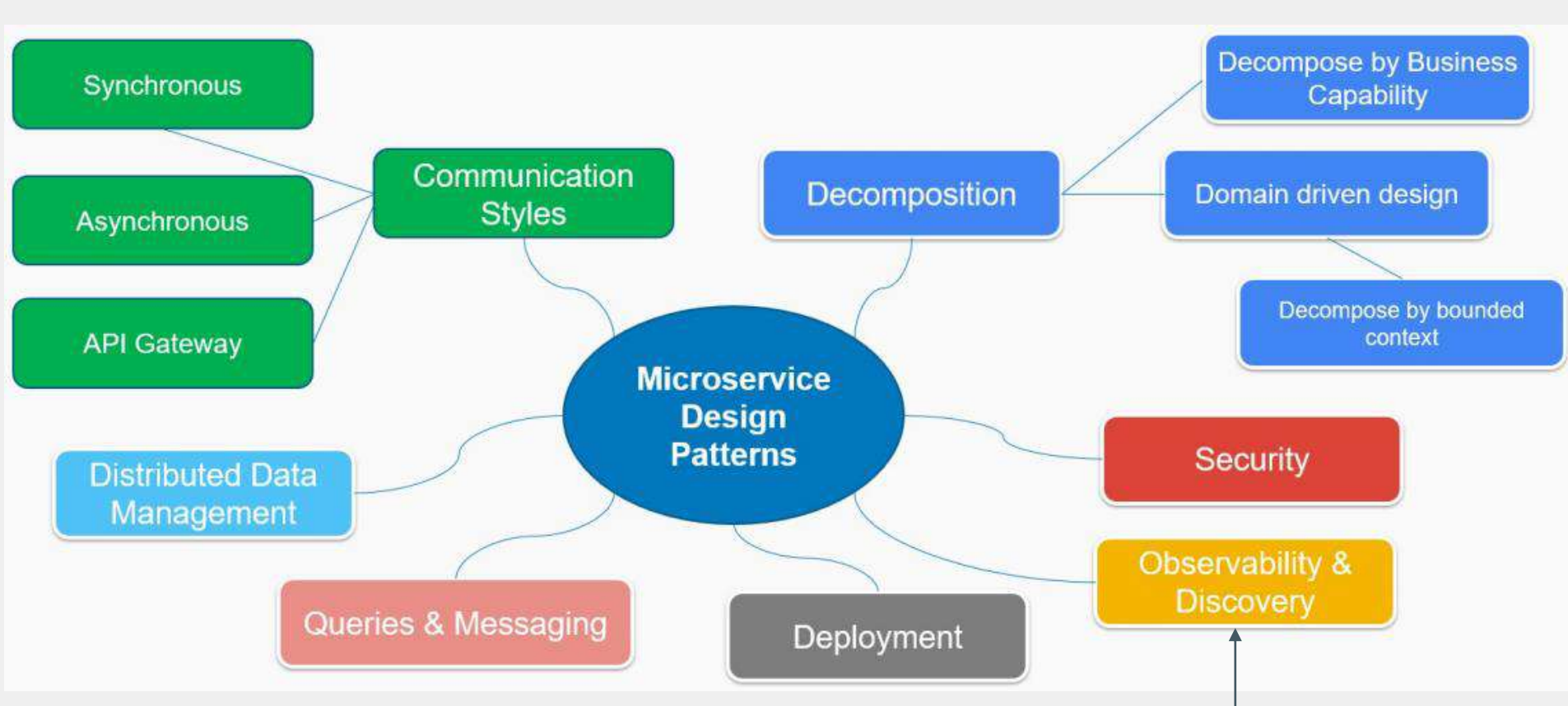
Manejo de datos: Como consumir las bases de datos locales y compartidas



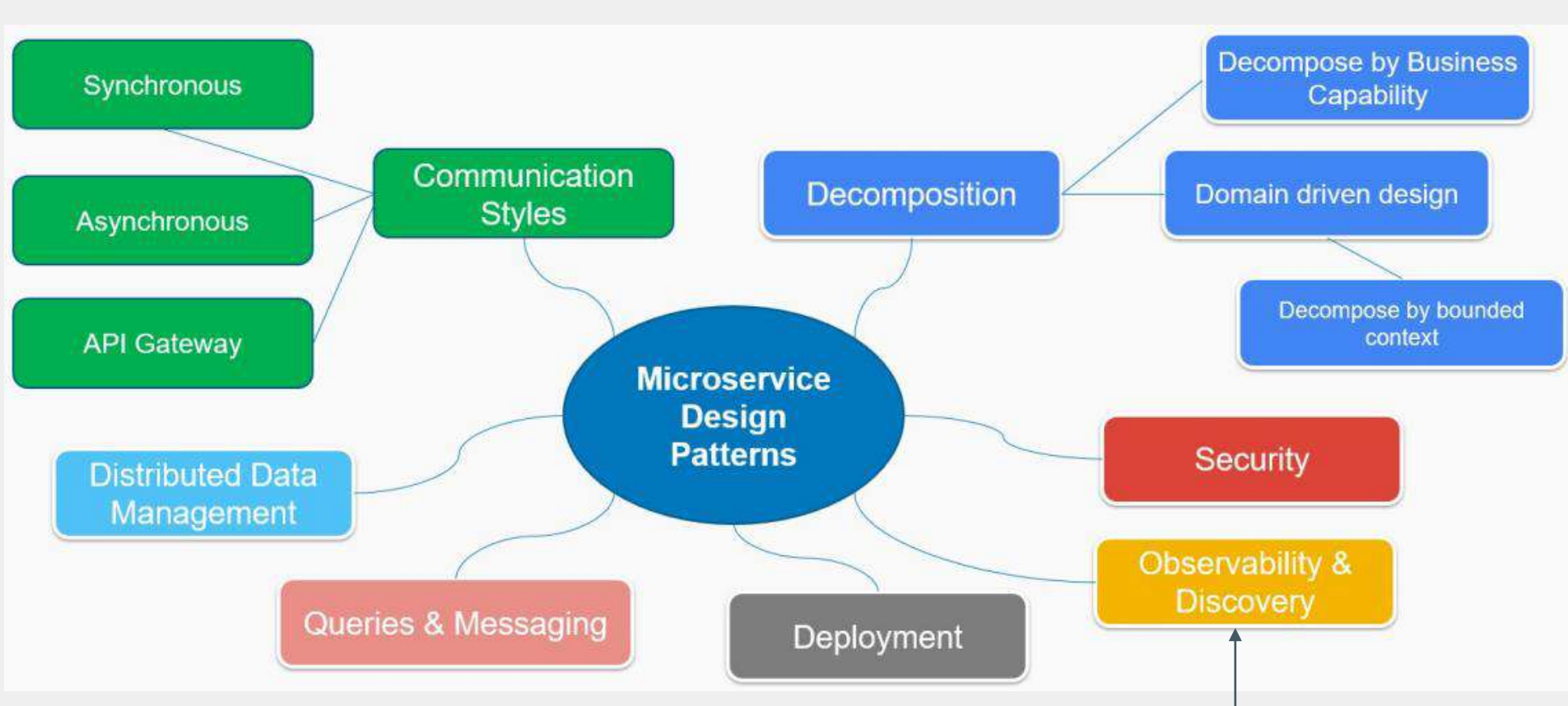
Queries y mensajería: Explora los eventos y mensajes que son enviados entre microservicios, y como los servicios son consumidos eficientemente.



Despliegue: Idealmente se quiere tener despliegues independientes y uniformes, evitando re-crear pipelines para cada microservicio.



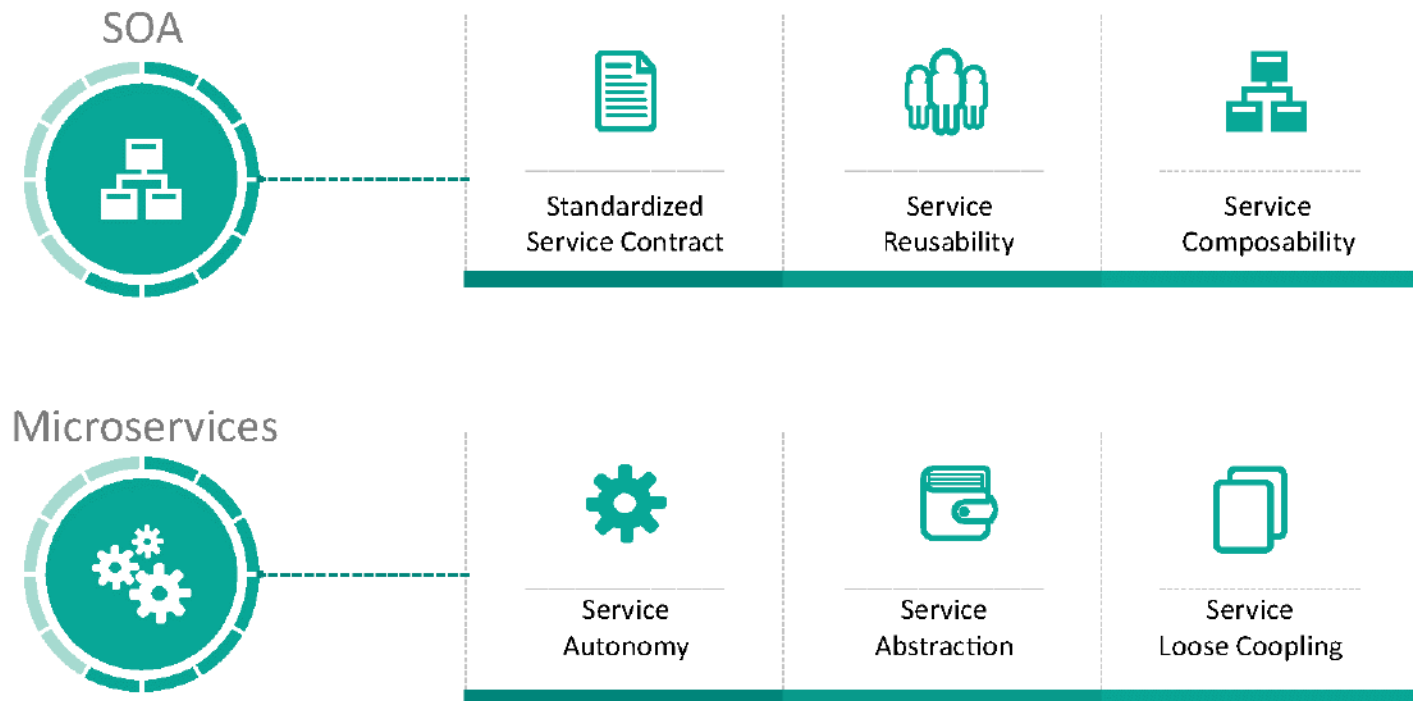
Monitoreo y descubrimiento: Capacidad para entender si un servicio está funcionando correctamente, monitoreando y llevando un log de las actividades de manera que permita investigar en caso de fallos. También conocer que se está ejecutando para asuntos de mantenimiento y costos.



Seguridad: Capacidad para entender si un servicio está funcionando correctamente, monitoreando y llevando un log de las actividades de manera que permita investigar en caso de fallos. También conocer que se está ejecutando para asuntos de mantenimiento y costos. Security: This is critical for compliance, data integrity, data availability, and potential financial damage. It's important to have

SOA vs Microservices

¿Realmente están peleados?



¿POR QUÉ USAR UN ENFOQUE DE MICROSERVICIOS PARA CREAR APLICACIONES?



Que los desarrolladores de software factorizan una aplicación en sus distintos componentes **no es nada nuevo**. Normalmente, se usa un enfoque con niveles con un almacén en el back-end, lógica de negocios en el nivel intermedio y una interfaz de usuario (IU) en el front-end. Lo que ha cambiado en los últimos años es que los desarrolladores ahora crean aplicaciones distribuidas para la nube, permitiendo :

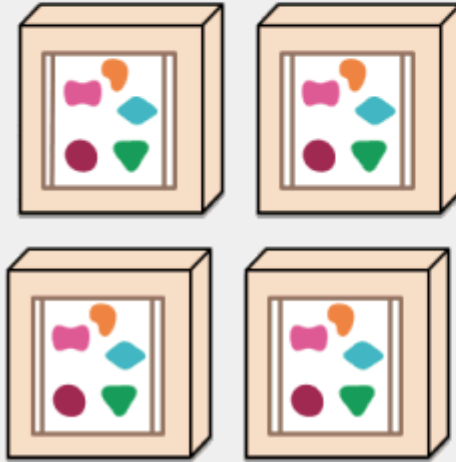
- Entrega más rápida de características y funcionalidades para poder responder a las demandas de los clientes de forma ágil.
- Mejora de la utilización de los recursos para reducir costos.

DE MONOLITO A MICROSERVICIOS

Una aplicación monolítica pone todos sus componentes en el mismo proceso



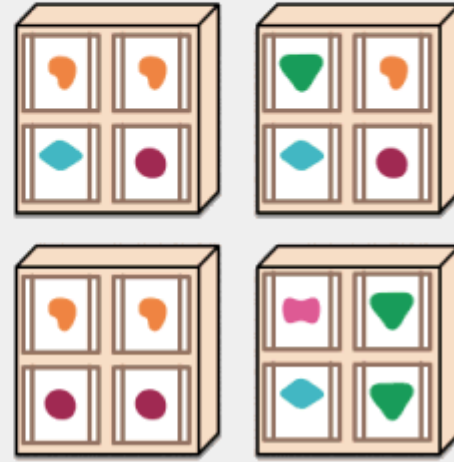
Y escala replicando el proceso a través de múltiples servidores



Una arquitectura basada en microservicios pone cada funcionalidad en un servicio



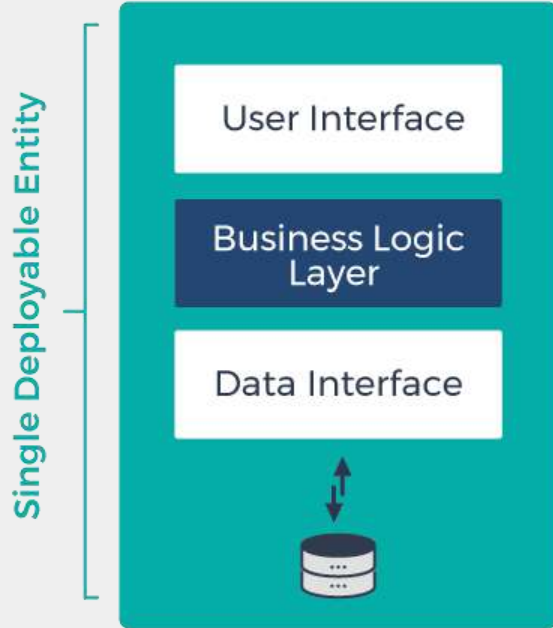
Y escala distribuyendo y replicando los servicios según sea necesario



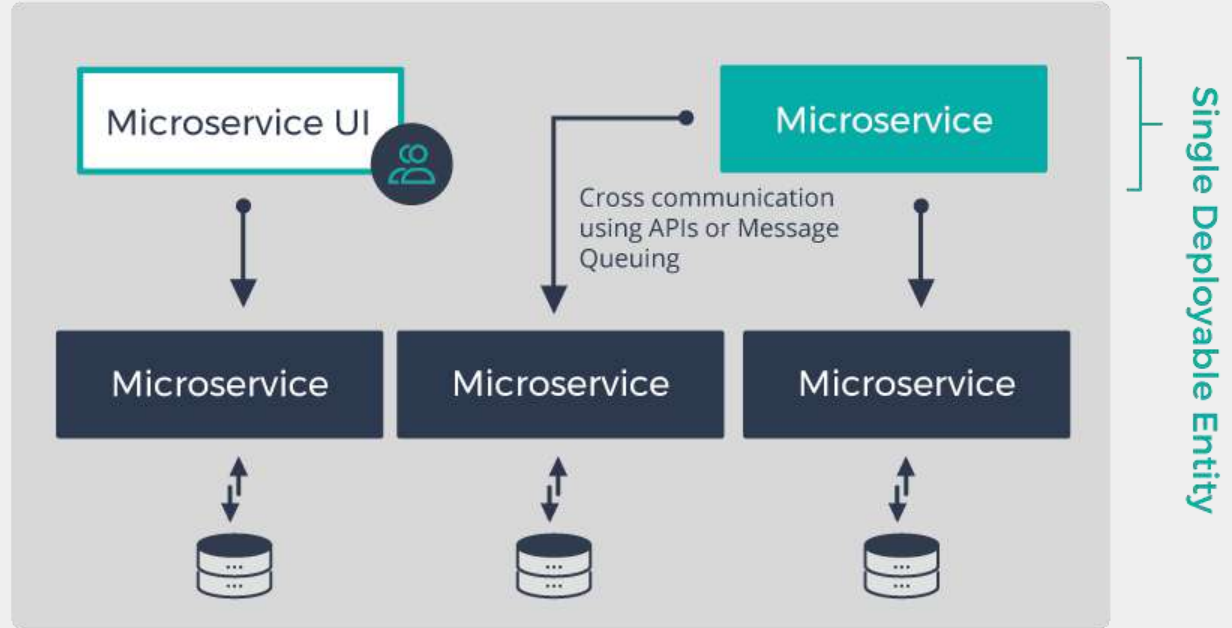
Los microservicios **son una manera** muy interesante de implementar una arquitectura orientada a servicios (SOA).

ARQUITECTURA MONOLÍTICA VS ARQUITECTURA DE MICROSERVICIOS

MONOLITHIC ARCHITECTURE



MICROSERVICE ARCHITECTURE



¿QUE DEBEMOS TENER EN CUENTA?



Las aplicaciones monolíticas contienen funciones específicas del dominio y normalmente se dividen en capas funcionales, como web, negocios y datos



Para escalar aplicaciones monolíticas, es preciso clonarlas en varios servidores, máquinas virtuales o contenedores.



Las aplicaciones de microservicios separan las funciones en servicios más pequeños independientes.



Este enfoque de microservicios se escala horizontalmente mediante la implementación de cada servicio de manera independiente, con la creación de instancias de estos servicios en servidores, máquinas virtuales y contenedores.



¿QUE SON LOS MICRO SERVICIOS?

Según James Lewis y Martin Fowler (los creadores del concepto) los microservicios son un enfoque para desarrollar una única aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros, a menudo una API a través de HTTP.

Estos servicios se desarrollan alrededor de las **necesidades** del negocio y se pueden **implementar de forma independiente y completamente automatizada**. Existe un mínimo de administración centralizada de estos servicios, que puede escribirse en diferentes lenguajes de programación y utilizando diferentes tecnologías de almacenamiento de datos.

Los microservicios se adaptan perfectamente a los requerimientos de agilidad, escalabilidad y confiabilidad de las aplicaciones modernas en la nube.



Características

Los Microservices:



Son totalmente autónomos y abstractos.



Están organizados entorno a las capacidades del negocio, por equipos de trabajo especializados.



Tienen un alcance funcional pequeño y limitado.



No dependen de invocaciones síncronas de otros servicios.



Son independientes de lenguajes concretos: cada microservicio puede ser desarrollados en un lenguaje distinto y apoyarse en diversas tecnologías.



No deben ser parte de composiciones complejas de servicios.

Características Cont.

Los **Microservices**:



Soportan interoperabilidad por medio de mecanismos de comunicación basados en mensajes.



Se enfocan en el planteamiento Smart Endpoints & Dumb Pipes



Son resilientes: en caso de error, un microservicio puede reiniciarse en otra maquina para seguir estando disponible, evitando la pérdida de datos y manteniendo su coherencia.



Poseen un gobierno descentralizado, sin ataduras a lenguajes o plataformas.



Usan infraestructura automatizada.



Manejan sus propio almacenamiento de datos (Persistencia Políglota)

ES IMPORTANTE



Cada microservicio tiene una **funcionalidad específica y bien definida** la cual se implementa en relativamente pocas líneas de código y en un lenguaje de programación particular que **no necesariamente tiene que ser el mismo** para todos los microservicios. **La clave está en el protocolo de comunicación y en la gestión de los mensajes** a través de los cuales los microservicios colaboran.

¿CÓMO SE COMUNICAN LOS MICROSERVICIOS?

Interactúan con otros microservicios por medio de protocolos e interfaces bien definidos. Por lo general, la comunicación en los servicios usa un enfoque de REST con los protocolos HTTP y TCP, y XML o JSON como formato de serialización. Desde la perspectiva de la interfaz, esto supone adoptar el enfoque de diseño web. No obstante, no hay nada que le impida usar protocolos binarios o sus propios formatos de datos.



VENTAJAS Y DESVENTAJAS DE USAR MICROSERVICIOS



VENTAJAS

Módulos concretos: Los microservicios refuerzan la estructura modular, lo cual es particularmente importante para equipos grandes



Implementación independiente: Los servicios simples son más fáciles de implementar, y dado que son autónomos, es menos probable que causen fallas en el sistema cuando se averían.

Diversidad Tecnológica: Con los microservicios se pueden mezclar múltiples lenguajes, frameworks etc.



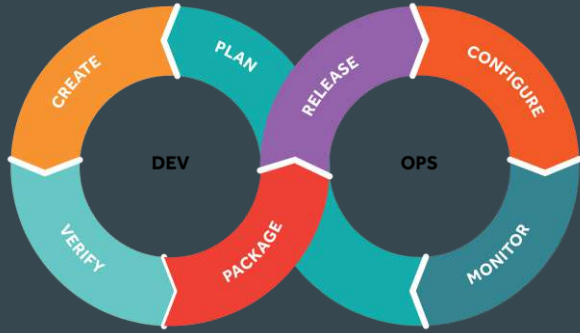
DESVENTAJAS

Distribución: Los sistemas distribuidos son más difíciles de programar a causa de los llamados remotos, los cuales son lentos y tienen un riesgo de fallo latente



Consistencia eventual: Mantener una consistencia fuerte es extremadamente difícil para un sistema distribuido, lo que significa que todos tienen que manejar consistencia eventual.

Complejidad operacional: Se requiere un equipo de operaciones experto para manejar muchos servicios que son re implementados frecuentemente



Las aplicaciones basadas en arquitecturas de microservicios facilitan escenarios de integración y entregas continuas, elementos claves dentro de la cultura DevOps

¿NECESITO UTILIZAR MICROSERVICIOS?



¿CUÁNDO DEBEMOS PENSAR EN MICROSERVICIOS?

A pesar que la arquitectura monolítica sigue siendo la norma para la mayor parte de las aplicaciones, no es lo mejor para sistemas complejos. Por tanto, en la medida en que un software **evoluciona** y se **desarrolla**, mayor se vuelve el **costo** de oportunidad de preservar una arquitectura monolítica y mayores son las ventajas de adoptar una arquitectura más flexible, que permita y promueva el crecimiento de su empresa.

MOTIVOS PARA USAR LOS MICROSERVICIOS



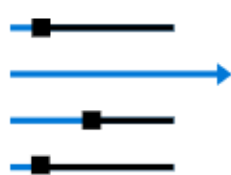
Compilar servicios de forma independiente

Las aplicaciones basadas en microservicios están compiladas como una colección de servicios altamente desacoplados que controlan una sola acción. Los equipos pueden compilar, comprobar, implementar y supervisar cada servicio de manera independiente.



Escalar servicios de forma autónoma

Los servicios independientes se pueden escalar según las respectivas exigencias, sin que ello afecte al rendimiento general, en lugar de escalar o reducir verticalmente la aplicación completa.



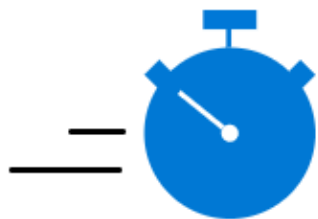
Usar el mejor enfoque

Los equipos de desarrollo obtienen flexibilidad para usar el enfoque de implementación, el lenguaje, la plataforma y el modelo de programación más recomendados para cada servicio.



Aislar puntos de error

Al aislar áreas problemáticas potenciales para servicios individuales, las arquitecturas de microservicios mejoran la seguridad y la confiabilidad. Los servicios se pueden reemplazar o retirar sin que ello afecte a la estructura general.



Entregar valor más rápido

Los equipos pueden implementar rápidamente pequeños módulos independientes. Varios equipos pueden trabajar en distintos servicios al mismo tiempo y poner nuevas características en producción más rápido.

¿ESTOY PREPARADO PARA UTILIZAR MICROSERVICIOS?



CONSIDERACIONES IMPORTANTES



De negocio: En comparación con el enfoque monolítico tradicional, una estrategia de microservicios implica inversiones financieras, en la cultura de empresa y en nuevos desarrollos .



De arquitectura y diseño: Los microservicios introducen nuevos enfoques y consideraciones para la arquitectura y el diseño; es necesario asegurarse de que se están utilizando los enfoques arquitectónicos adecuados.



De implementación: Hay que considerar varias opciones de implementación para una arquitectura de microservicios, incluidas las plataformas, los frameworks y los lenguajes de programación.



De resiliencia: Para crear una arquitectura de microservicios, es necesario tener en cuenta su tolerancia a fallos, lo que incluye alta disponibilidad, conmutación por error (failover), recuperación de desastres, interrupción de circuitos, aislamiento, etc.



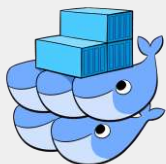
Operativas: Hay que asegurarse de que nuestro equipo pueda monitorear y administrar el ecosistema de microservicios.

GOBERNABILIDAD Y HERRAMIENTAS DE CONSTRUCCIÓN DE MICRO SERVICIOS



Gobierno Corporativo es un conjunto de personas, políticas e incluso procesos que **garantizarán** que su estrategia SOA sea implementada y resulte **exitosa**. Por tanto, tener una plataforma de microservicios que ofrezca funcionalidades como el descubrimiento, un directorio y la monitorización de los servicios se vuelve imprescindible.

DENTRO DE LAS PLATAFORMAS QUE SOPORTAN ESTA FUNCIONALIDAD SE TIENEN:



Docker Swarm y Docker Compose: La filosofía de contenedores Docker se alinea de forma natural con las arquitecturas de microservicios.



kubernetes

Kubernetes. Es un sistema de código abierto que automatiza la implementación, las operaciones y el escalado de las aplicaciones en contenedores. Originalmente desarrollado por Google, se basa en sus experiencias en servicios como Google Search y Gmail.



MESOSPHERE



Pivotal **Cloud Foundry**



OPENSIFT

CONTINUACIÓN:



MESOSPHERE

Mesosphere Datacenter Operating System (DCOS) es un administrador de clúster escalable que incluye Mesosphere's Marathon, una herramienta de orquestación de contenedores de nivel de producción



Pivotal **Cloud Foundry**

Pivotal Cloud Foundry. Permite arquitecturas de microservicios combinando flujo de trabajo y programación de contenedores desde Cloud Foundry con integraciones para patrones de microservicios



OPENSIFT

OpenShift. Es una plataforma como servicio (PaaS) que aprovecha el empaquetado basado en contenedores Docker para implementar orquestación de contenedores y capacidades de administración informática para Kubernetes.

ES IMPORTANTE RECORDAR QUE...



Es muy importante destacar que, aunque los contenedores y sus beneficios encajan a la perfección con los microservicios amplificando su potencial enormemente, **no es obligatorio** el uso de contenedores para hablar de microservicios.

También es importante aclarar que no es necesaria una arquitectura de microservicios para **contenizar** una aplicación por que una aplicación monolítica podría ser desplegada dentro de un contenedor.

MICROSERVICIOS EN LA NUBE



Los servicios en la nube se caracterizan por ofrecer **agilidad, flexibilidad, resiliencia** y capacidad de adaptación. Migrar a la nube es un paso importante del proceso de modernización de cualquier software, pero estos beneficios no se dan de manera automática por el simple hecho de estar en la nube.

Cumplir con la promesa de un servicio más eficiente requiere de un ajuste paralelo en la arquitectura de su aplicación. Las arquitecturas monolíticas se convierten fácilmente en un cuello de botella para sistemas complejos y a gran escala.

SOLUCIONES OFRECIDAS POR MICROSOFT



Service Fabric

Consiga el escalado automático, las actualizaciones graduales y la recuperación automática de errores con un marco de microservicios personalizado.



Azure Kubernetes Service (AKS)

Use un servicio de Kubernetes totalmente administrado para controlar el aprovisionamiento, la actualización y el escalado de recursos de clúster a petición.



API Management

Exponga y publique partes específicas de sus aplicaciones como una API, independientemente de dónde esté hospedada la implementación.

SOLUCIONES OFRECIDAS POR MICROSOFT



Azure Functions

Compile aplicaciones con funciones sin servidor simples que escalen a petición según el número de ejecuciones, sin administrar ninguna infraestructura.

RETOS AL IMPLEMENTAR MICROSERVICIOS



RETOS AL IMPLEMENTAR MICROSERVICIOS



Dividir una aplicación en servicios no es una tarea trivial.

Habilidades de aprovisionamiento rápido de recursos, ya sea en **cloud** o **on-premise**.

Esquemas de monitoreo para servicios distribuidos, incluyendo métricas de aplicación, redes, logs así como maneras de enlazar una operación o transacción de un cliente con los diferentes microservicios que utiliza.

Prácticas de desarrollo ágil y equipos de desarrollo con enfoque de producto.

¿CÓMO MIGRAR A MICROSERVICIOS?

Inicialmente **buscar partes de la lógica de negocio** que sea posible separar. Adicionalmente, identificar **partes del código que consuman muchos recursos**, pero que puedan ser aisladas.



Si un microservicio tiene que estar constantemente interconectado para realizar cualquier operación puede ser un indicativo de que quizás debería estar contenido dentro de él

Casos de Exito

Soluciones con **Microservices**



Amazon



Netflix



ebay



Twitter



google



soundCloud



Nike



Hailo

ANTIPATRONES DE LOS MICROSERVICIOS

PROVISIONAMIENTO LENTO

Se debe poder iniciar un nuevo servidor en cuestión de horas. Naturalmente esto encaja con computación en la nube, pero es algo que puede ser hecho sin tal servicio. Para llegar a este punto se necesita una gran cantidad de automatización - tal vez no sea necesario estar a tope de eficiencia al iniciar, pero para manejar microservicios serios es obligatorio.



IMPLEMENTACIÓN LENTA

Con muchos servicios para administrar, estos deben poderse implementar rápidamente, tanto para entornos de prueba como para producción. Por lo general, esto implicará una Línea de implementación que se pueda ejecutar en no más de un par de horas. Algunas intervenciones manuales están bien en las primeras etapas, pero pronto buscará automatizarlas por completo.



MONITOREO BÁSICO

Con muchos servicios poco acoplados que colaboran en la producción, las cosas van a salir mal en formas que son difíciles de detectar en entornos de prueba. Como resultado, es esencial que exista un régimen de monitoreo para detectar problemas graves rápidamente. Lo principal es detectar problemas técnicos (conteo de errores, disponibilidad del servicio, etc.) pero también vale la pena monitorear los problemas comerciales (como detectar una caída en los pedidos). Si aparece un problema repentino, debe asegurarse de que se puede hacer rollback



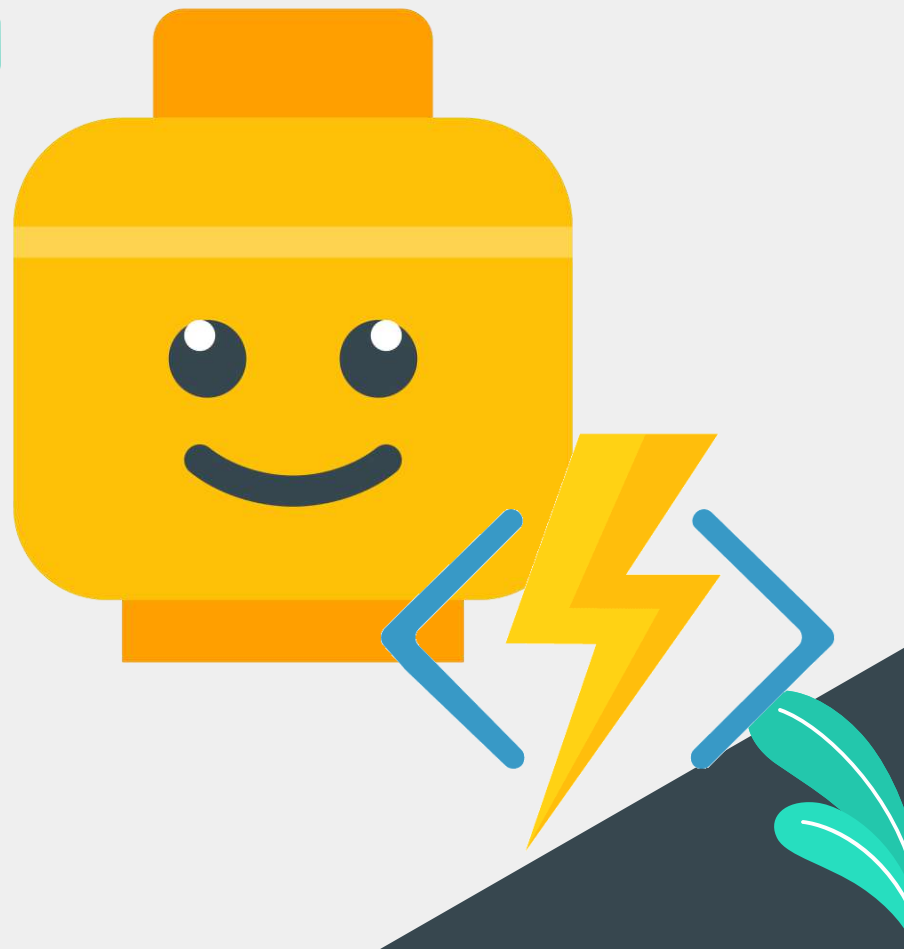
CONSTRUYENDO BLOQUES

Una de las maneras más sencillas de implementar Microservicios en la nube es a través de **AZURE FUNCTIONS**, ya que con esta forma de implementación :

- No piensas en los Servidores ni en toda su configuración o mantenimiento.

- Escalan automáticamente de acuerdo a tus necesidades.

- Brindan conectividad e integración con otros servicios



ANTES DE CONSTRUIR RECODEMOS QUE...

Una regla importante en las arquitecturas de microservicios es que cada microservicio debe ser el dueño de su propia lógica por tal motivo cada microservicio tendrá un **Bounded Context (BC) – Contexto Acotado**.

Análisis de
dominio



Definición de
contextos
acotados



Definición de
entidades,
relaciones y
servicios



Identificación de
microservicios

PASOS PARA IDENTIFICAR MICROSERVICIOS

1. Primero se **analiza** el dominio del negocio para entender los requerimientos funcionales de nuestra aplicación.
2. Se definen los **BC –Contextos Acotados** de nuestro dominio, es decir un subdominio atómico de nuestra aplicación.
3. Definimos las entidades y relaciones entre ellas.
4. El resultado del paso anterior nos permite identificar los microservicios de nuestra aplicación.

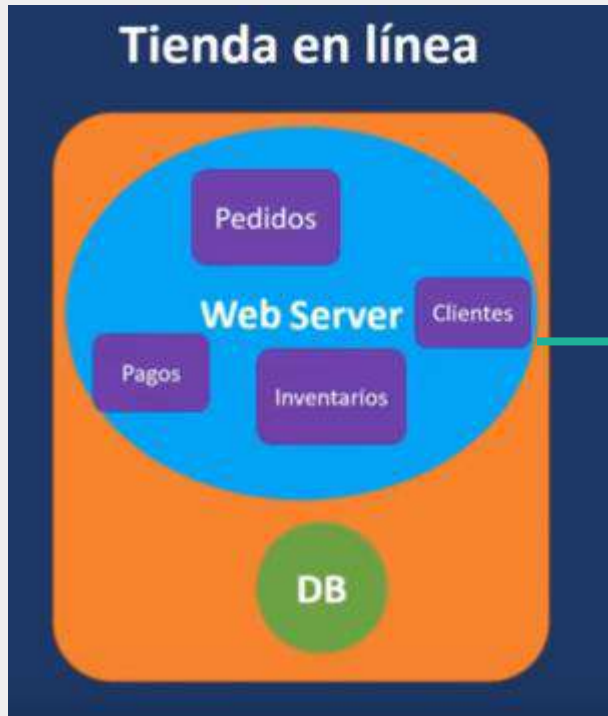
EJEMPLO PRÁCTICO



Doña Pepita, es dueña de un local del comidas, y desea implementar la creación de una tienda Online, como su negocio no es tan grande decide implementar una arquitectura monolítica.



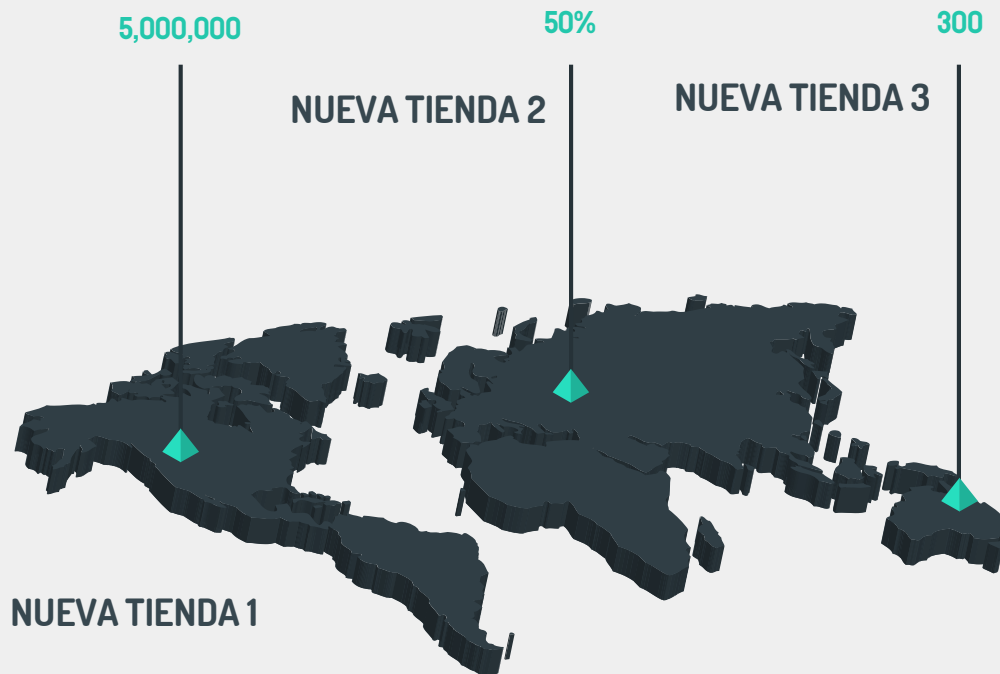
EJEMPLO PRÁCTICO



Los servicios que deben ser implementados son:

- I. **Pedidos:** Se deben gestionar todos los pedidos de las personas.
- II. **Pagos:** Se deben registrar todos los pagos y los medios con los cuales se realizan.
- III. **Clientes:** Se deben gestionar toda la información de los clientes.
- IV. **Inventario:** Se deben gestionar los stocks de todos los productos ofrecidos.

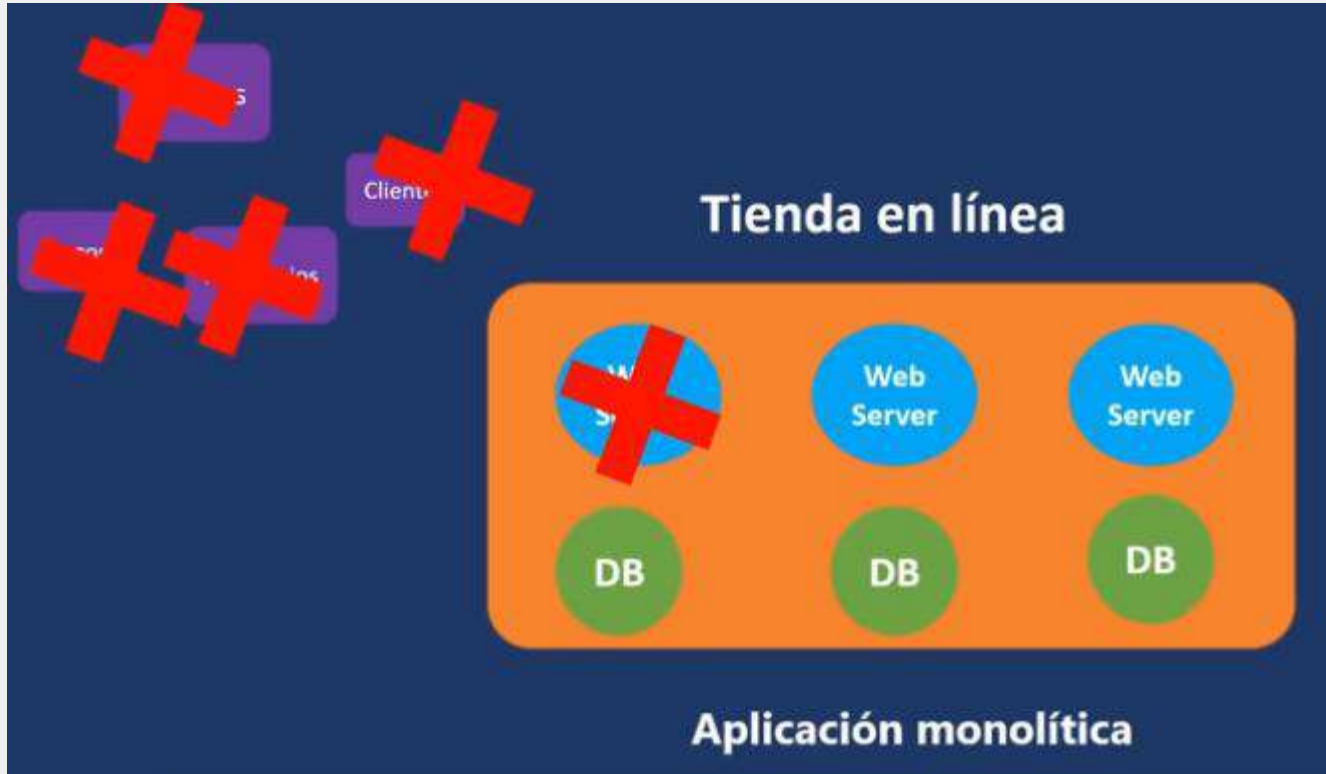
EJEMPLO PRÁCTICO



A pesar de que la aplicación ha funcionado correctamente, la tienda necesita abrir nuevos puntos de venta a internacional y por tal motivo un ingeniero decidió plantear el siguiente modelo de escalamiento:

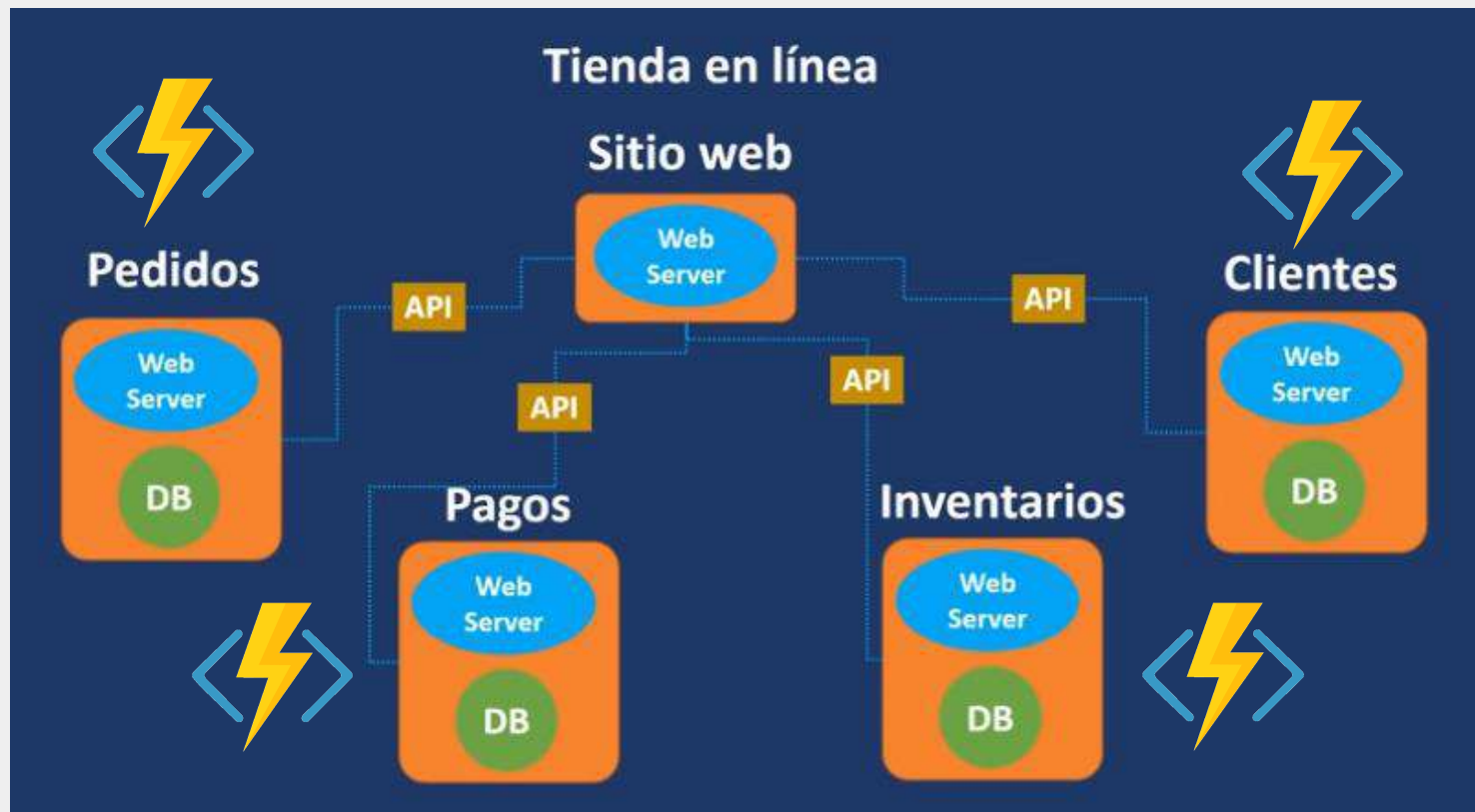


OOPS!!



Sin embargo se ha presentado un fallo en uno de los **Web Server**, ocasionando que una tienda estuviese fuera de línea ocasionando una perdida de 120.000.000 millones de Dólares, por tal motivo es necesario cambiar la arquitectura.

ARQUITECTURA POR MICRO SERVICIOS



CONSIDERACIONES IMPORTANTES

Azure Functions: Actualmente hay disponibles dos versiones del tiempo de ejecución de para la compilación de Azure functions que soportan los siguientes lenguajes:



PowerShell



TypeScript



SESIÓN 2



AGENDA



01 FUNDAMENTOS MICROSERVICIOS



02 ARQUITECTURA DE MICROSERVICIOS

1. Construyendo bloques
2. Microservicios Restful
3. Comunicación a través de un puente API

03 MANEJO DE DATOS

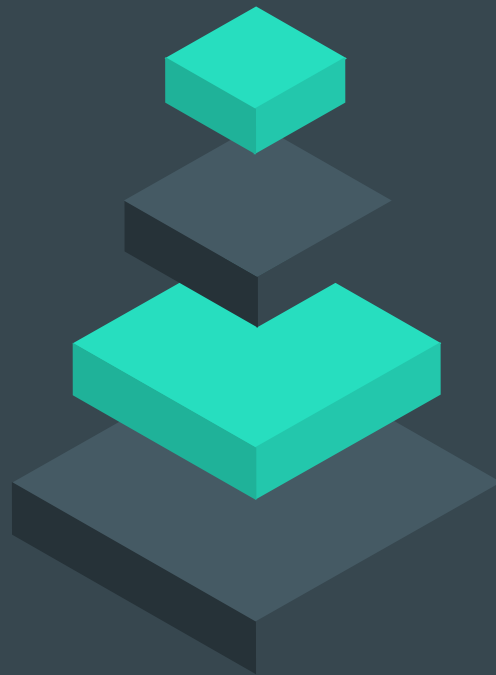
1. Comunicación basada en eventos
2. Patrones de Gestión de datos
3. Patrón Event sourcing

04 ARQUITECTURA Y DISEÑO DE MICROSERVICIOS

1. Componentes de presentación
2. Lógica de dominio o de negocios
3. Lógica de acceso a la base de datos
4. Lógica de integración de aplicaciones

MATERIALIZANDO LOS MICROSERVICIOS

PARTE I



Si se empieza desde un modelo de dominio cuidadosamente diseñado,
es mucho más fácil razonar acerca de los microservicios.

¿CÓMO CONSTRUIR MICROSERVICIOS?

Uno de los mayores desafíos de los microservicios es definir los **límites** de los servicios individuales. La regla general es que un servicio debe hacer "algo"; sin embargo, llevarla a la práctica requiere una **minuciosa** reflexión. No hay ningún proceso mecánico que cree el diseño "correcto". Debe meditar **detenidamente** sobre el dominio de su empresa, los requisitos y los objetivos. En caso contrario, puede terminar con un diseño incoherente que exhiba algunas características no deseables, como son dependencias ocultas entre servicios, un acoplamiento rígido o interfaces mal diseñadas.

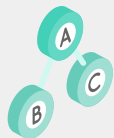
DOMAIN DRIVEN DESIGN

El diseño basado en dominios (DDD) proporciona una plataforma que puede ayudar en gran medida a diseñar bien los microservicios.

Tiene **dos fases** distintas, tácticas y estratégicas, las cuales son **iterativas y continuas**



El **estratégico**, se define la estructura a gran escala del sistema. Ayuda a garantizar que la arquitectura permanece centrada en las funcionalidades del negocio.



El **táctico** proporciona un conjunto de modelos de diseño que puede usar para crear el modelo de dominio. Los modelos tácticos ayudarán a diseñar microservicios coherentes y con acoplamiento flexible.

RECURSOS IMPORTANTES

1. Eric Evans: Domain-Driven Design (Diseño basado en dominios)

2. Vaughn Vernon: Implementing Domain-Driven Design (Implementación del diseño basado en dominios)

PASOS A SEGUIR PARA LA IDENTIFICACIÓN



- 1, Analizar el **dominio de la empresa** para conocer los **requisitos funcionales** de la aplicación. El resultado de este paso es una descripción informal del dominio.
- 2, Definir los **contextos delimitados** del dominio. Cada contexto delimitado contiene un modelo de dominio que representa un **subdominio** concreto de la aplicación mayor.
- 3, Dentro de un contexto delimitado, se aplican los modelos tácticos de diseño basado en dominios para **definir las entidades, los agregados y los servicios de dominio**.
- 4, Usar los resultados del paso anterior para **identificar los microservicios** de la aplicación.

EJEMPLO

DRONE DELIVERY



ESCENARIO

Fabrikam, Inc. está iniciando un servicio de entrega con drones. La empresa administra una flota de drones. Las empresas se registran en el servicio y los usuarios pueden solicitar que un dron recoja los bienes para la entrega. Cuando un cliente programa una recogida, un sistema back-end asigna un dron y notifica al usuario con un tiempo de entrega estimado. Con la entrega en curso, el cliente puede realizar el seguimiento del dron, con una fecha estimada que se actualiza constantemente.

ANÁLISIS DEL MODELO DE DOMINIO

Antes de escribir ningún código, se necesita una **visión general del sistema** que se va a crear. Con el diseño basado en dominios, se empieza por modelar el dominio **empresarial** y se crea un ***modelo de dominio***. El modelo de dominio es un modelo abstracto del ámbito empresarial.

Análisis de
dominio



Definición de
contextos
acotados



Definición de
entidades,
relaciones y
servicios

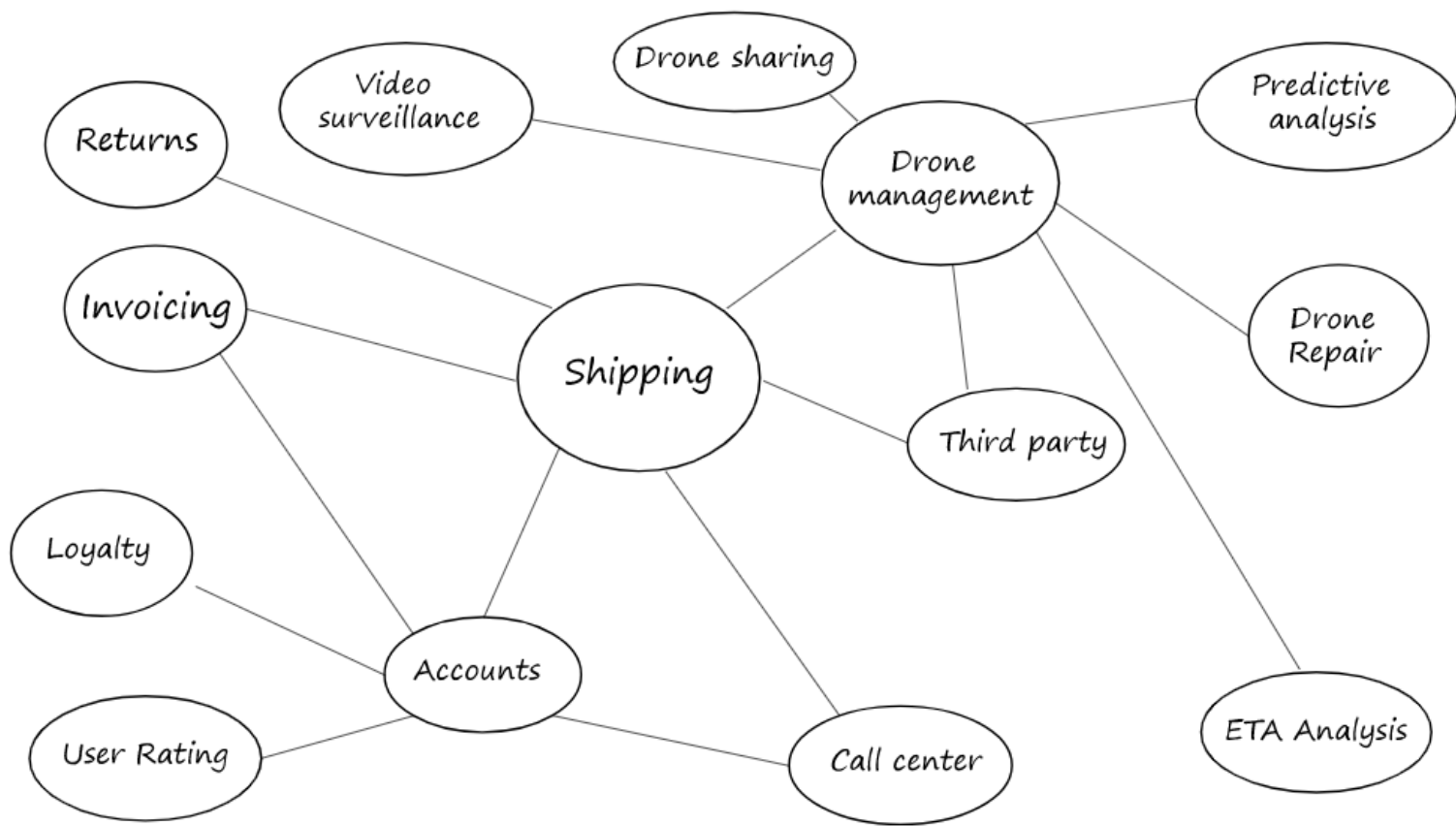


Identificación de
microservicios

¿QUÉ DEBE RESPONDER?

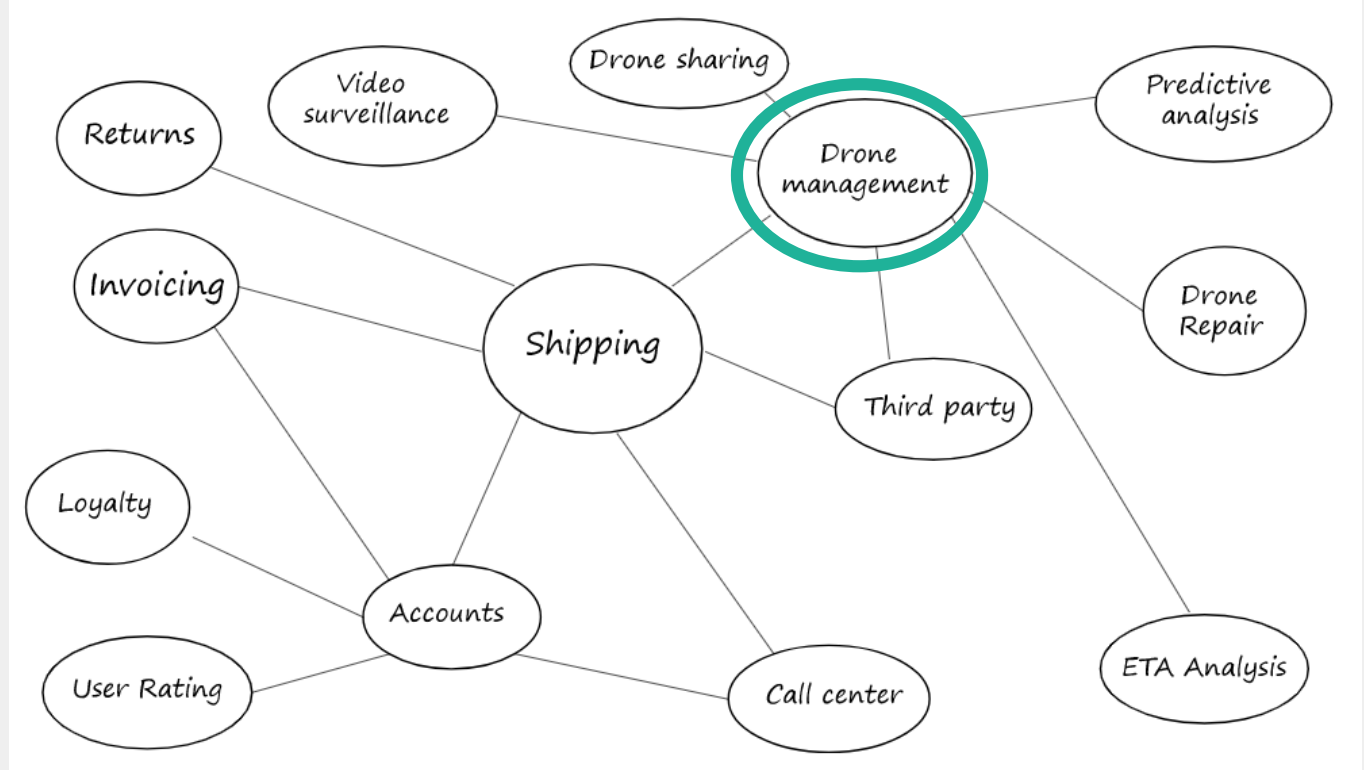
- ¿Cuáles son las funciones que tiene el negocio?
- ¿Cuáles son fundamentales para la empresa y cuáles proporcionan servicios auxiliares?
- ¿Qué funciones están estrechamente relacionadas?







El elemento **Shipping** (envío) se coloca en el centro del diagrama, ya que es importante para el negocio. Todo lo demás dentro del diagrama existe para habilitar esta funcionalidad.



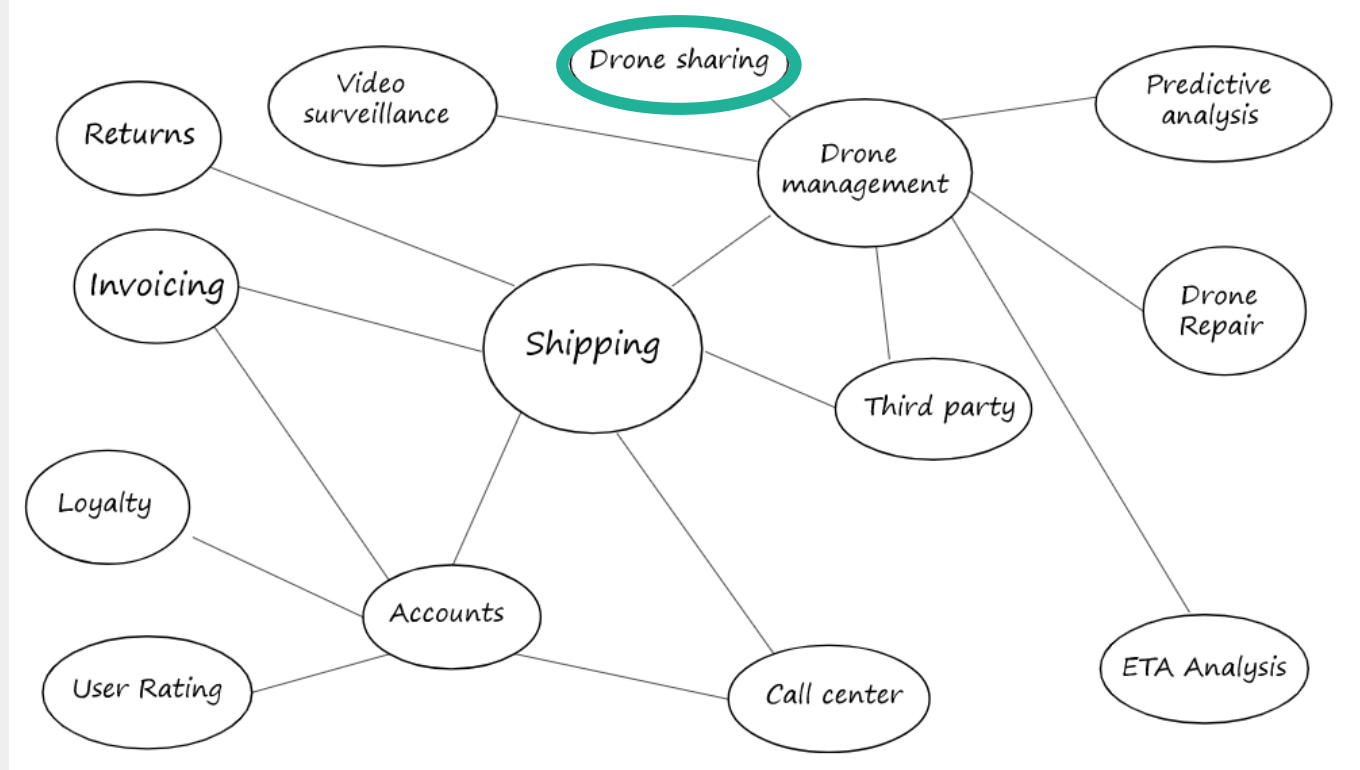
El elemento **Drone management** (administración de drones) también es esencial para la empresa. La funcionalidad que está **estrechamente relacionada** con la anterior es Drone repair (reparación de drones) y el uso de predictive analysis (análisis predictivos) que permiten predecir el momento en que los drones tienen que someterse a tareas de mantenimiento.



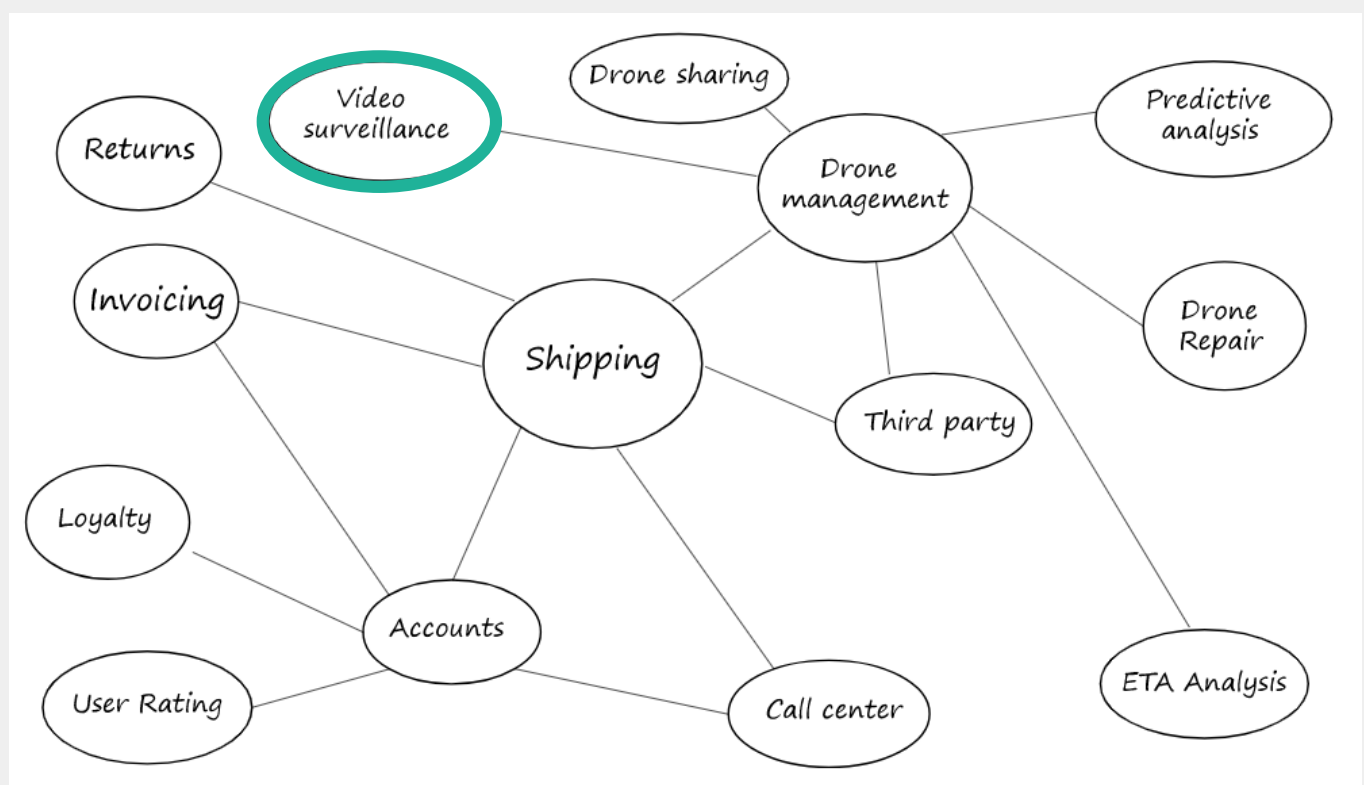
ETA analysis (análisis del tiempo estimado de llegada) proporciona estimaciones de tiempo para la recogida y la entrega.



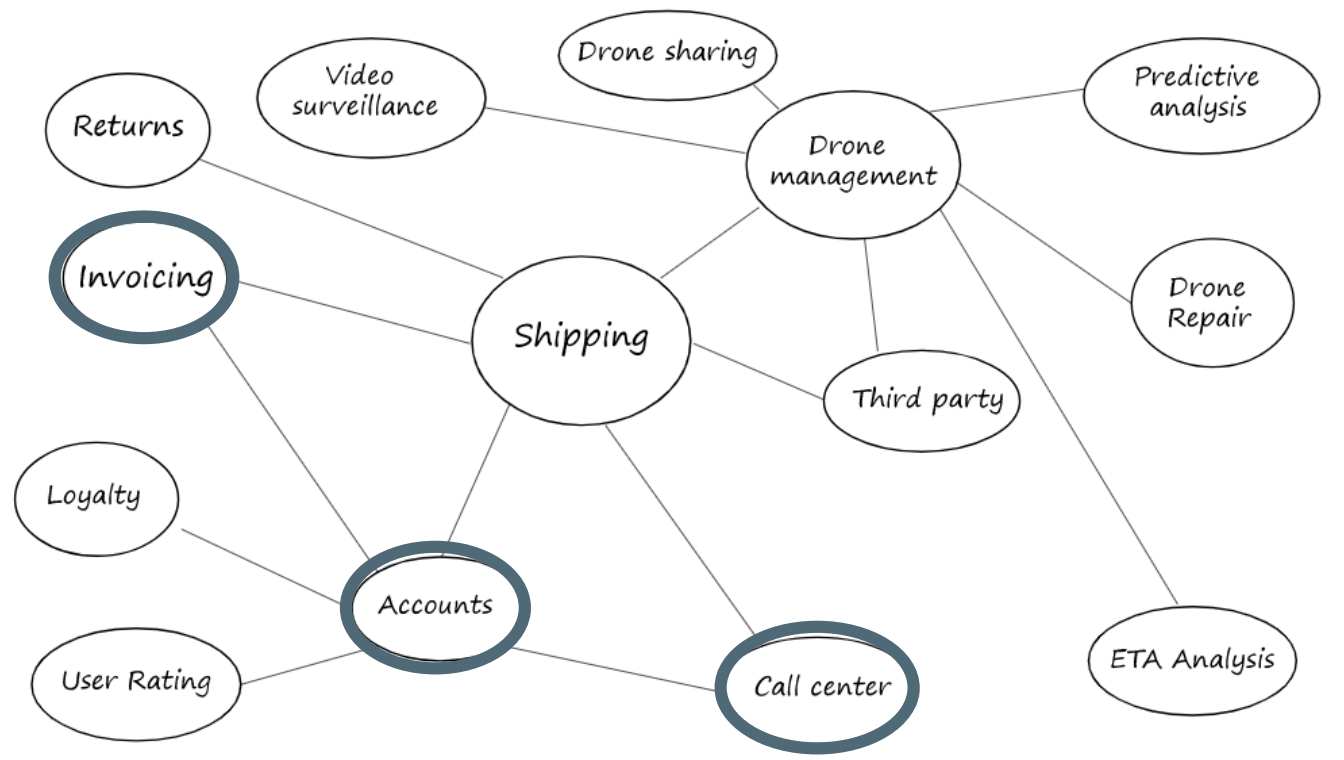
La funcionalidad **Third-party transportation** (transporte de terceros) permitirá que la aplicación programe métodos de transporte alternativo, si un dron no puede entregar un paquete completo..



La funcionalidad **Drone sharing** (uso compartido de drones) es una posible extensión del negocio principal. La empresa puede tener drones de sobra durante ciertas horas y alquilar el excedente que, de otro modo, permanecería inactivo. Esta característica no estará en la versión inicial.



La funcionalidad **Video surveillance** (vigilancia por vídeo) es otra área que la empresa podría expandir en versiones posteriores.



User accounts (cuentas de usuario), **Invoicing** (facturación) y **Call center** (centro de llamadas) son subdominios que contribuyen al negocio principal..



Tenga en cuenta que, en este punto del proceso, **no** hemos tomado decisiones sobre la **implementación** o las **tecnologías**.

Es posible que una **misma funcionalidad** tenga que ser vista desde **contextos diferentes**, a menudo es mejor diseñar modelos **independientes** que representen la misma entidad del mundo real (en este caso, un dron) en dos contextos diferentes. Cada modelo contiene solo las características y los atributos que sean pertinentes en su contexto determinado.

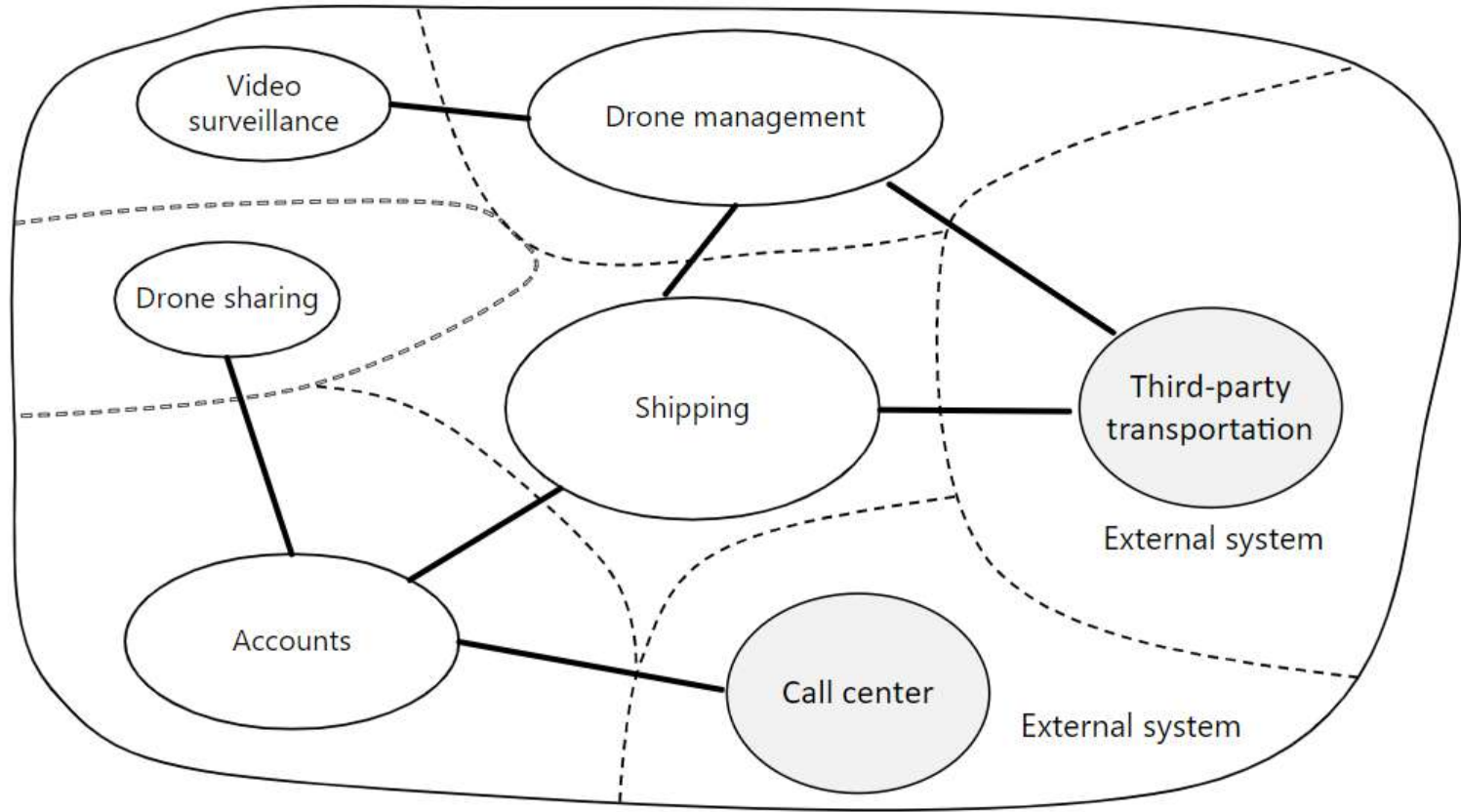
LOS CONTEXTOS ACOTADOS

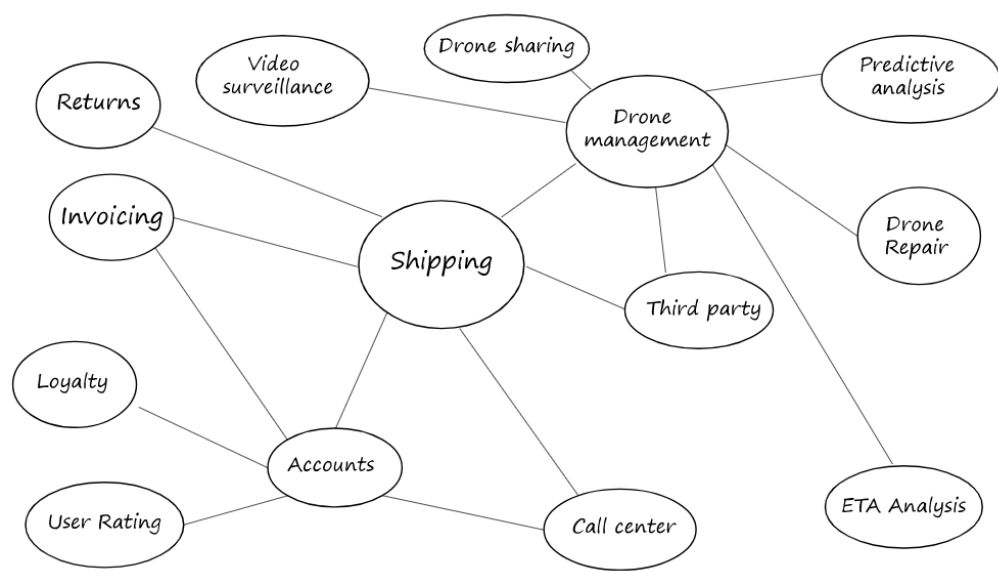
Aquí es donde entra en juego el concepto del diseño basado en dominios referente a los contextos acotados. Un contexto acotado y/o delimitado es simplemente el **límite dentro de un dominio**.

Si examinamos el diagrama anterior, podemos agrupar la funcionalidad teniendo en cuenta si varias funciones compartirán un único modelo de dominio.

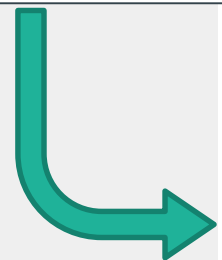


Diagrama de Contextos Limitados

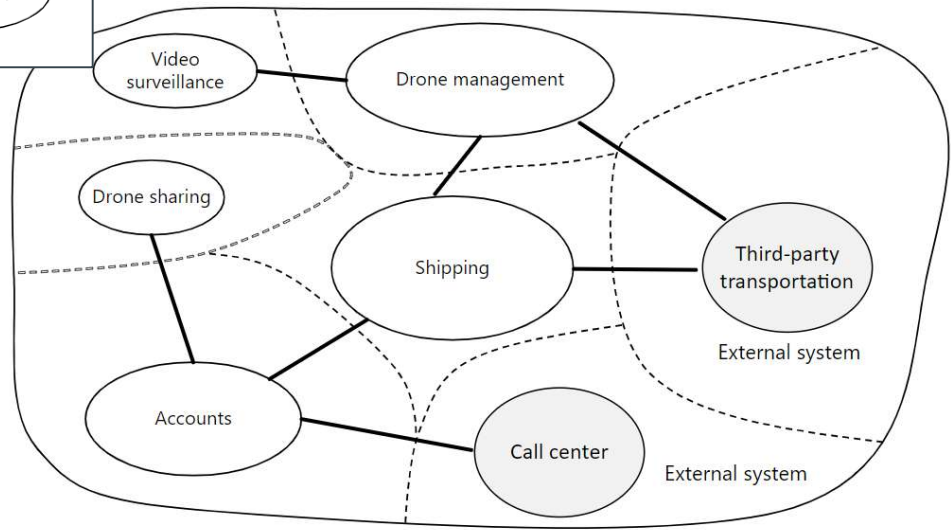




Análisis inicial del dominio



Definición de Contextos Limitados



IMPORTANTE



Los contextos delimitados no están **necesariamente aislados** entre sí. En este diagrama, las líneas continuas que conectan los contextos delimitados representan los lugares donde dos de ellos **interactúan**. Por ejemplo, el envío depende de las cuentas de usuario (Accounts) para obtener **información** sobre los clientes y de la administración de drones (Drone management) para programar los de la flota.

DEBEMOS SABER..

En el libro **Domain Driven Design** de Eric Evans, se describen varios patrones para mantener la integridad de un modelo de dominio cuando interactúa con otro contexto delimitado. Uno de los principios fundamentales de los microservicios es que los servicios se comunican a través de API bien definidas.

Este método se corresponde con dos patrones que Evans **llama Open Host Service** (servicio de host abierto) y **Published Language** (lenguaje publicado).

ANÁLISIS DEL MODELO TÁCTICO

Consiste en definir los modelos de dominio con más **precisión**. Los patrones tácticos se aplican dentro de un **único contexto delimitado**.

En una arquitectura de microservicios, interesan especialmente los **patrones de agregados** y **entidades**. Aplicar estos patrones nos ayudará a identificar los límites naturales de los servicios en nuestra aplicación

Análisis de
dominio



Definición de
contextos
acotados



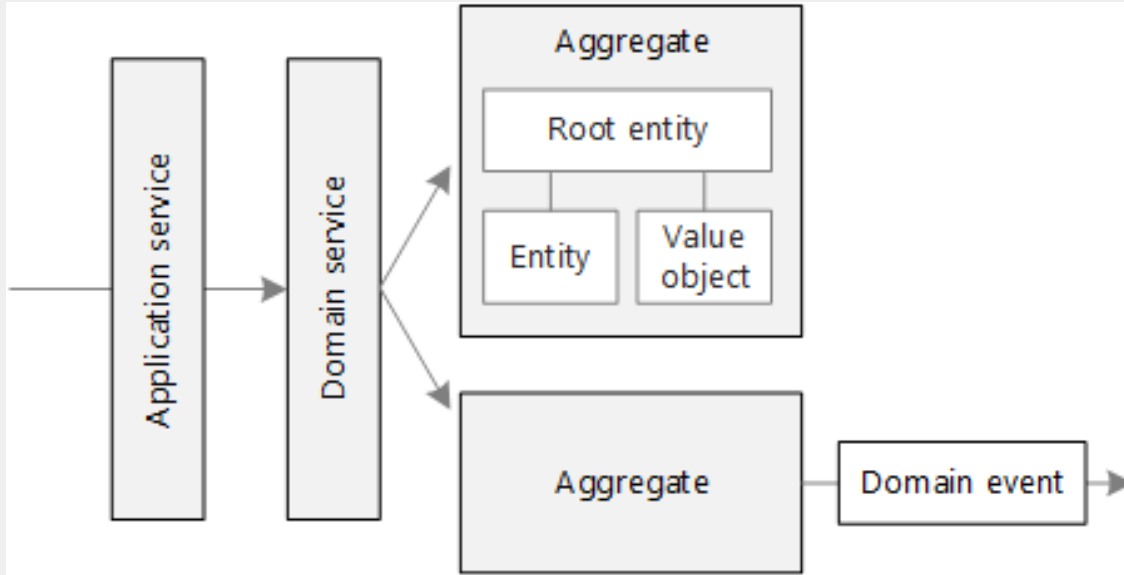
Definición de
entidades,
relaciones y
servicios



Identificación de
microservicios



REPRESENTACIÓN DEL MODELO TÁCTICO



ENTIDADES - ENTITIES



Una entidad es un objeto con una **identidad única** que **persiste en el tiempo**. Por ejemplo, en una aplicación bancaria, las cuentas y los clientes serían entidades.

Características:

- Una entidad tiene un **identificador único** en el sistema, que se puede usar para buscar la entidad o para recuperarla.
- Una identidad puede **abarcар varios contextos delimitados** y puede durar más que la aplicación. Ej: los números de cuentas bancarias o las cédulas no están asociadas a la duración de una aplicación en particular.

ENTIDADES - ENTITIES



Una entidad es un objeto con una **identidad única** que **persiste en el tiempo**. Por ejemplo, en una aplicación bancaria, las cuentas y los clientes serían entidades.

Características (cont):

- Los atributos de una entidad **pueden cambiar con el tiempo**. Por ejemplo, el nombre o la dirección de una persona pueden variar, pero ella sigue siendo la misma.
- Una entidad puede contener referencias a otras entidades.

OBJETOS DE VALOR - VALUE OBJECTS



Un objeto de valor **no tiene identidad**. Se define únicamente mediante los **valores de sus atributos**. Los objetos de valor también son **inmutables**. Para actualizar un objeto de valor, siempre hay que crear una nueva instancia que reemplace a la anterior. Los objetos de valor pueden tener métodos que encapsulen la lógica del dominio, pero esos métodos **no deben afectar al estado del objeto**.

Ejemplos típicos de objetos de valor son los colores, las fechas y horas, los valores de divisa, edad, ubicación.

OBJETOS DE VALOR — PARA QUÉ?



A menudo es útil representar las cosas como un **compuesto**.

- Una coordenada 2D consta de un valor X , y un valor Y .
- Una cantidad de dinero consiste en un número y una moneda.
- Un rango de fechas consta de fechas de inicio y finalización, que pueden ser compuestos de año, mes y día.

Si tengo dos objetos que representan las coordenadas cartesianas de (2,3), tiene sentido tratarlos como iguales. Sin embargo, algunos lenguajes de programación no permiten comparar objetos por los valores que contienen

OBJETOS DE VALOR — PROBLEMAS DE ALIASING



```
Date fechaRetiro= new Date(Date.parse("Martes 1 Nov 2016"));
```

```
// Necesitamos una fiesta de despedida
```

```
Date fechaFiesta= fechaRetiro;
```

```
// Pero es un martes, mejor lo cuadramos para el fds
```

```
fechaFiesta.setDate(5);
```

```
//Al revisar si la fecha del 5 de noviembre es igual a nuestra fecha de retiro, da true  
assertEquals(new Date(Date.parse("Sabado 5 Nov 2016")), retirementDate);
```

OBJETOS DE VALOR - ALIASING

Ejemplo: Un empleado se retira el 1 de noviembre del 2016



```
Date retirementDate = new Date(Date.parse("Tue 1 Nov 2016"));
```

retirementDate

variable

2016 Nov 1

date

*creates a variable
referencing a date object*

Lo que significa que necesitamos una fiesta de despedida

```
Date partyDate = retirementDate;
```

retirementDate

partyDate

2016 Nov 1

*creates a second variable
referencing the same
date object*

OBJETOS DE VALOR - ALIASING



```
partyDate.setDate(5);
```



*changing the date
through one reference
updates the single date
used by both references*

¿CÓMO LO SOLUCIONO?

```
class Point {  
  constructor(x, y) {  
    this._data = {x: x, y: y};  
  }  
  get x() {return this._data.x;}  
  get y() {return this._data.y;}  
  equals (other) {  
    return this.x === other.x && this.y === other.y;  
  }  
}
```

- Es inmutable y no hay setters definidos;
- Refleja la semántica del dominio;
- Muestra cómo fluye la información y se transforma durante el tiempo de ejecución;
- Se puede comparar con otros objetos de valor de la misma clase leyendo directamente propiedades privadas.

VENTAJAS DE LOS OBJETOS DE VALOR



- **Compartir sin problemas:** Al ser inmutables, no se corre riesgo de modificaciones indeseadas
- **Semánticas mejoradas:** Incrementa considerablemente la legibilidad del código y el modelo
- **Comparación entre objetos:** Se comparan viendo los valores en los atributos
- **Auto-Validación:** Los if de la regla de negocio van en el constructor

¿COMO MANIPULO LOS OBJETOS DE VALOR?

Por lo general, así es como se manipula un objeto de valor:

- Crea una nueva instancia a través de un constructor o método estático
- Crea otro objeto de valor a partir de él
- Extrae datos internos y los convierte a otro tipo.

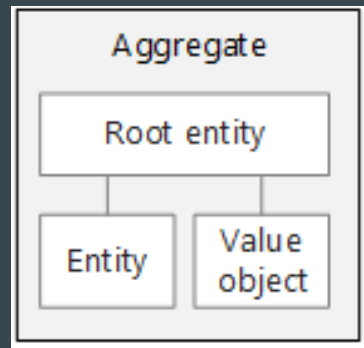


AGREGADOS - AGGREGATES



Un agregado es un grupo de clases que mantienen un invariante de forma conjunta. La raíz de ese agregado es el objeto «padre» que nos permite interactuar con el grupo de clases.

Un ejemplo sería una factura con sus líneas, donde la raíz del agregado sería la factura y las líneas, la información de los productos, los datos del cliente o la dirección de envío serían entidades o value objects (dependiendo del diseño) que forman parte del agregado.



AGREGADOS - AGGREGATES



Si trabajamos con agregados, deberíamos seguir ciertas reglas (más recomendaciones que leyes innegociables):

- Todas las interacciones con clases que forman parte de un agregado deberíamos hacerlas a través del **aggregate root**. Esto es necesario para que podamos garantizar los invariantes.
- Sólo podemos recuperar de la base de datos (a través de repositorios o como más te guste) aggregate roots. Nunca recuperaremos **entidades internas** al agregado.
- Si necesitamos mantener una relaciones entre entidades de **diferentes agregados**, siempre será hacia el aggregate root, nunca hacia clases internas del agrega

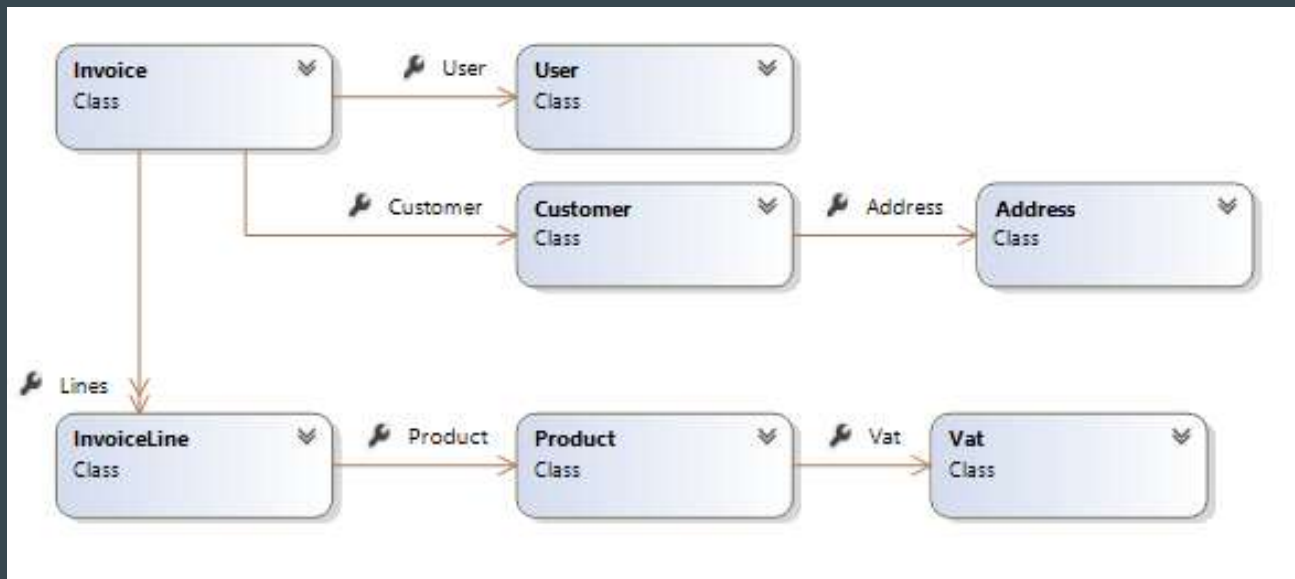
IMPORTANTE

Un agregado puede constar de **una sola entidad**, sin entidades secundarias. Lo que lo convierte en agregado es el **límite transaccional**.

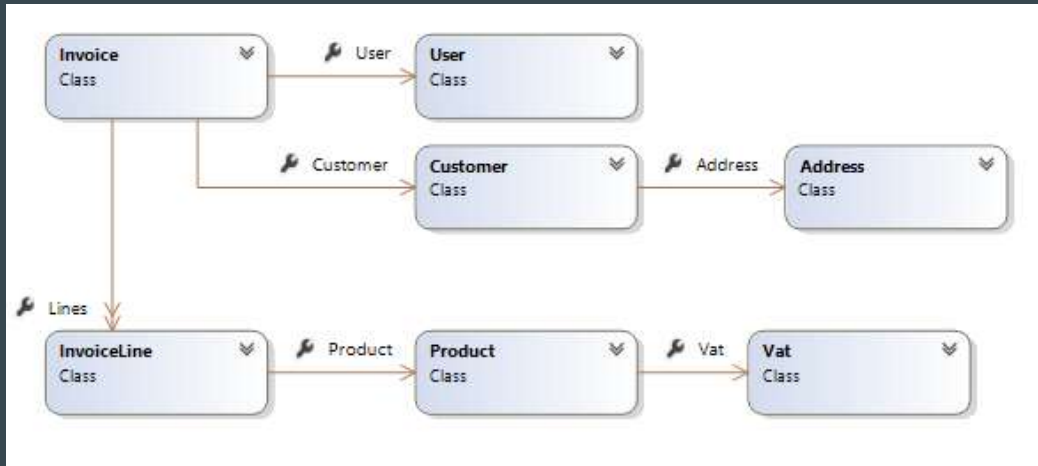


AGREGADOS — COMO ELEGIRLOS?

La clave es pensar en los **invariantes**. Vamos al modelo de la factura y sus líneas. Simplificando mucho podemos tener un modelo inicial parecido a éste:



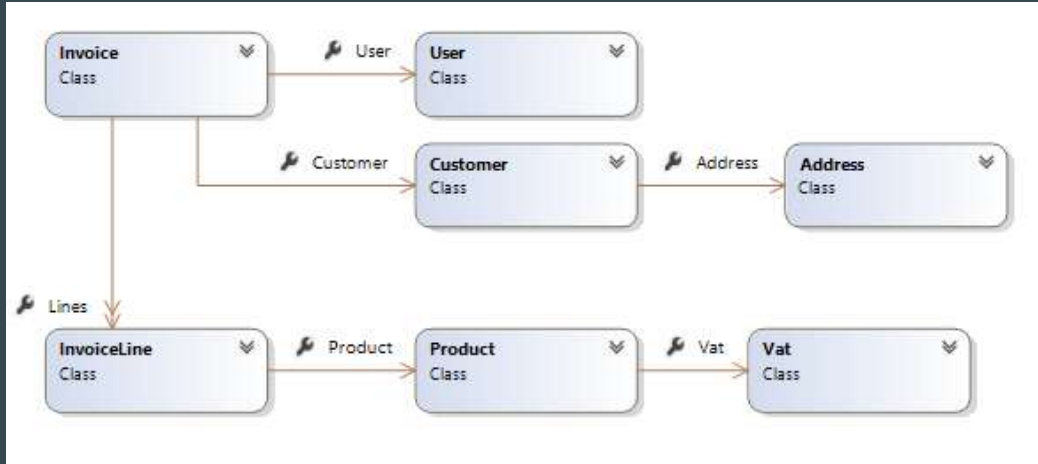
AGREGADOS — COMO ELEGIRLOS?



A simple vista, siguiendo la dirección de las dependencias, podríamos pensar que tenemos un único agregado cuya raíz es la clase Invoice, pero si nos centramos en invariantes la cosa cambia.

Por ejemplo, ¿podemos cambiar la dirección de email del usuario que la ha emitida sin que afecte a la factura? Probablemente sí. Sin embargo, ¿podemos modificar las líneas de una factura una vez que ésta ha sido cerrada? Claramente no.

AGREGADOS — COMO ELEGIRLOS?



Si lo vemos desde el punto de vista transaccional/operacional, pasa algo parecido. Tendremos operaciones para añadir líneas a una factura mientras la estamos construyendo, y tendremos operaciones para modificar usuario, pero sería extraño tener una operación que a la vez modifique la factura y el usuario que la creó.

AGREGADOS — COMO ELEGIRLOS?

Eso nos va a ir dando pistas de qué forma parte realmente del agregado y qué no. La factura y sus líneas deben mantener invariantes que afectan a ambas, pero la factura y el usuario no, por lo que seguramente formen parte de diferentes agregados, uno que contenga la factura y sus líneas y otro que contenga el usuario.

¿Qué pasa con el cliente o con los productos? Aquí entra en escena otro factor que a veces pasamos por alto, y es que algunos conceptos que son entidades en determinados contextos, en otros son solamente value objects.

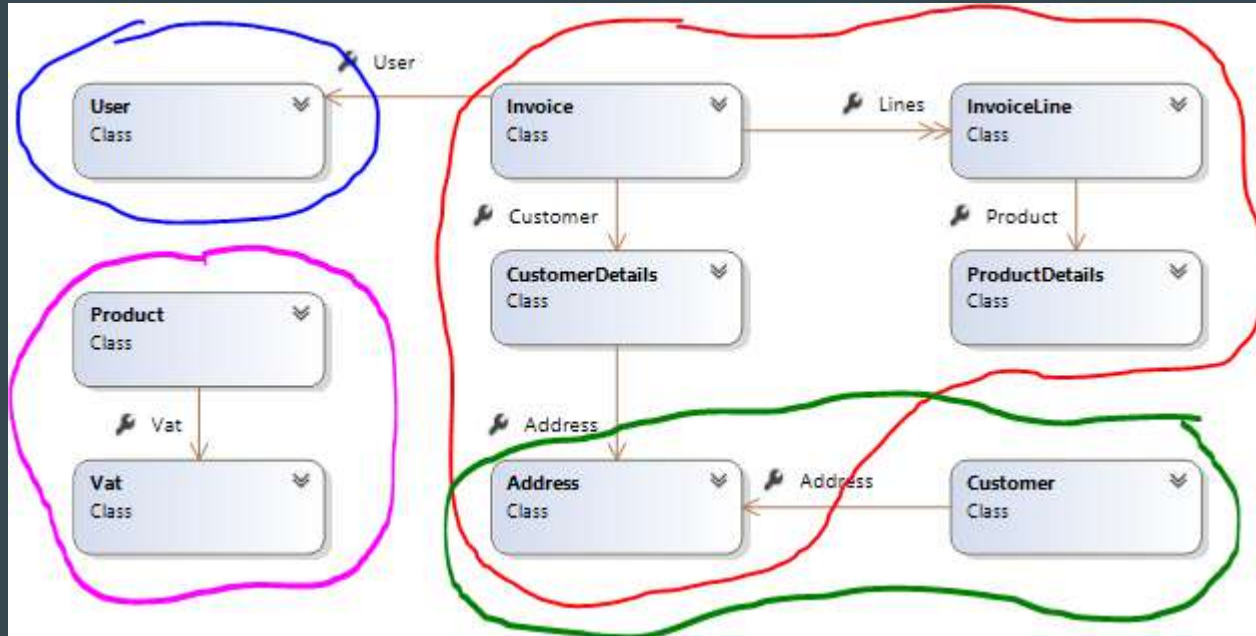


AGREGADOS — COMO ELEGIRLOS?



A lo largo del tiempo un cliente puede cambiar de nombre, o de dirección, o incluso de NIT, pero una vez que una factura ha sido emitida, esos datos **no pueden cambiar**. Lo mismo para con un producto, puede que en el futuro cambie su descripción o el impuesto que se le aplica, pero aunque el gobierno decida subir el IVA, la factura que emitimos hace 3 años **no debería verse modificada** por ello.

AGREGADOS – COMO ELEGIRLOS?



IMPORTANTE

Como principio general, un microservicio no debe ser menor que un **agregado** ni mayor que un **contexto delimitado**.



SERVICIOS DE APLICACIÓN Y DE DOMINIO

En la terminología del diseño basado en dominios, un servicio es un objeto que **implementa alguna lógica sin mantener ningún estado.**

Evans distingue entre servicios de dominio, que encapsulan la **lógica del dominio**, y servicios de aplicación, que proporcionan la **funcionalidad técnica**, como la autenticación del usuario o el envío de un mensaje SMS. Los servicios de dominio a menudo se utilizan para modelar el comportamiento que abarca varias entidades.



EVENTOS DE DOMINIO

Los eventos de dominio se pueden utilizar para **notificar** a otras partes del sistema **cuando sucede algo**. Como sugiere su nombre, los eventos de dominio deben **significar** algo dentro del dominio. Por ejemplo, "se inserta un registro en una tabla" no es un evento de dominio. "Se canceló una entrega" es un evento de dominio.

Los eventos de dominio son especialmente importantes en una arquitectura de **microservicios**. Dado que los microservicios se distribuyen y **no comparten los almacenes de datos**, los eventos de dominio proporcionan una manera de que los microservicios se coordinen entre sí.





Hay otros patrones del diseño basado en dominios que **no se mencionan aquí**, como son los generadores, los repositorios y los módulos.

Pueden ser patrones útiles cuando se vaya a **implementar** un microservicio, pero son menos importantes al **diseñar** los límites entre microservicios.

DEBEMOS SABER..

1. En general, la funcionalidad en un microservicio **no debe abarcar más de un contexto delimitado**. Por definición, un contexto delimitado marca el límite. Si encuentra que un microservicio mezcla modelos de dominio diferentes, es un signo de que puede que tenga que volver atrás para refinar el análisis de dominio.

DEBEMOS SABER..

- Es importante revisar las funciones secundarias en el modelo de dominio. Ya que estas suelen ser buenas candidatas para microservicios.
- Se deben tener en cuenta los requisitos **no funcionales**. Observar factores como el tamaño del equipo, los tipos de datos, tecnologías, requisitos de escalabilidad, requisitos de disponibilidad y requisitos de seguridad. Estos factores pueden llevar a volver a dividir un microservicio en dos o más servicios más pequeños, o a hacer lo contrario combinando varios microservicios en uno.

ANÁLISIS DEL MODELO TÁCTICO

Consiste en definir los modelos de dominio con más **precisión**. Los patrones tácticos se aplican dentro de un **único contexto delimitado**.

En una arquitectura de microservicios, interesan especialmente los **patrones de agregados** y **entidades**. Aplicar estos patrones nos ayudará a identificar los límites naturales de los servicios en nuestra aplicación

Análisis de
dominio



Definición de
contextos
acotados



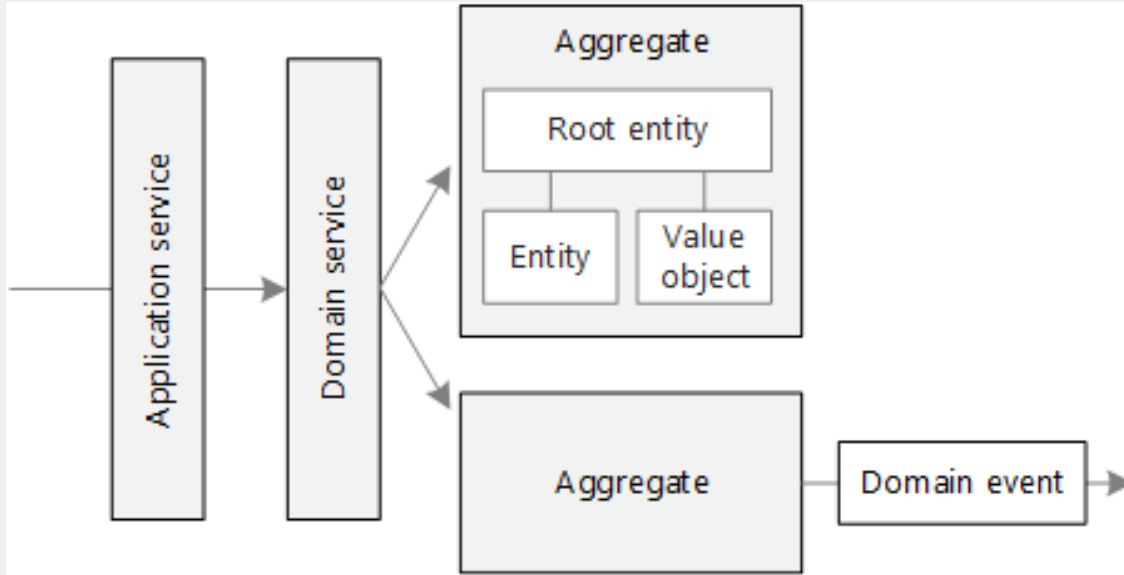
Definición de
entidades,
relaciones y
servicios



Identificación de
microservicios



REPRESENTACIÓN DEL MODELO TÁCTICO



EJEMPLO

DRONE DELIVERY



ESCENARIO — CONTEXTO DELIMITADO

1. Un cliente puede solicitar un dron para recoger las mercancías de una empresa que se registre en el servicio de entrega de drones.
2. El remitente genera una etiqueta (código de barras o RFID) para colocar en el paquete.
3. Un dron recogerá el paquete en la ubicación de origen y lo entregará en la ubicación de destino.
4. Cuando un cliente programa una entrega, el sistema proporciona un tiempo estimado de llegada (ETA) según la información de la ruta, las condiciones meteorológicas y los datos históricos.
5. Cuando el dron está en vuelo, el usuario puede realizar el seguimiento de la ubicación actual y del tiempo estimado de llegada más reciente.

ESCENARIO — CONTEXTO DELIMITADO

CONT.

1. Hasta que un dron recoja el paquete, el cliente puede cancelar una entrega.
2. Se notifica al cliente cuándo se completa la entrega.
3. El remitente puede solicitar al cliente la confirmación de la entrega, en forma de una firma o de una huella digital.
4. Los usuarios pueden ver el historial de una entrega completada.

ESCENARIO — ENTIDADES

- Entrega
- Paquete
- Dron
- Cuenta
- Confirmación
- Notificación
- Etiqueta



ESCENARIO — ENTIDADES

- **Entrega**
- **Paquete**
- **Dron**
- **Cuenta**



Las cuatro primeras (entrega, paquete, dron y cuenta) son todos agregados que representan los límites de la coherencia transaccional.

- Confirmación
- Notificación
- Etiqueta

Las confirmaciones y las notificaciones son las entidades secundarias de las entregas y las etiquetas son las entidades secundarias de los paquetes.

Los **objetos de valor** en este diseño incluyen la ubicación (Location), el tiempo estimado (ETA), el peso del paquete (PackageWeight) y su tamaño (PackageSize).

DISEÑO EN UML

Para ilustrar esto, se propone el siguiente diagrama UML del agregado de entrega. Hay que tener en cuenta que contiene referencias a otros agregados, como son la cuenta, el paquete y el dron.



LOS EVENTOS DE DOMINIO

Hay que tener en cuenta que estos eventos describen aspectos significativos dentro del modelo de dominio. Describen algo sobre este y no están vinculados a una construcción de un lenguaje de programación determinado.

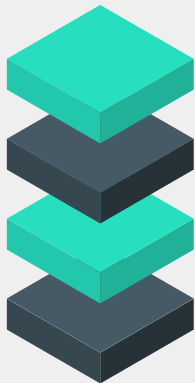
- Mientras un dron está volando, la entidad Dron envía **eventos** DroneStatus que describen la **ubicación** y el **estado del dron** (en vuelo, en tierra).
- La entidad de entrega (Delivery) envía **eventos** de **seguimiento de la entrega** (DeliveryTracking) cada vez que cambia la fase de una entrega. Entre estos, se incluyen los de creación, reprogramación, preparada y completada (DeliveryCreated, DeliveryRescheduled, DeliveryHeadedToDropoff y DeliveryCompleted, respectivamente)

LOS EVENTOS DE DOMINIO

Hay que tener en cuenta que estos eventos describen aspectos significativos dentro del modelo de dominio. Describen algo sobre este y no están vinculados a una construcción de un lenguaje de programación determinado.

- Mientras un dron está volando, la entidad Dron envía **eventos** DroneStatus que describen la **ubicación** y el **estado del dron** (en vuelo, en tierra).
- La entidad de entrega (Delivery) envía **eventos** de **seguimiento de la entrega** (DeliveryTracking) cada vez que cambia la fase de una entrega. Entre estos, se incluyen los de creación, reprogramación, preparada y completada (DeliveryCreated, DeliveryRescheduled, DeliveryHeadedToDropoff y DeliveryCompleted, respectivamente)

SERVICIOS DE DOMINIO ADICIONALES

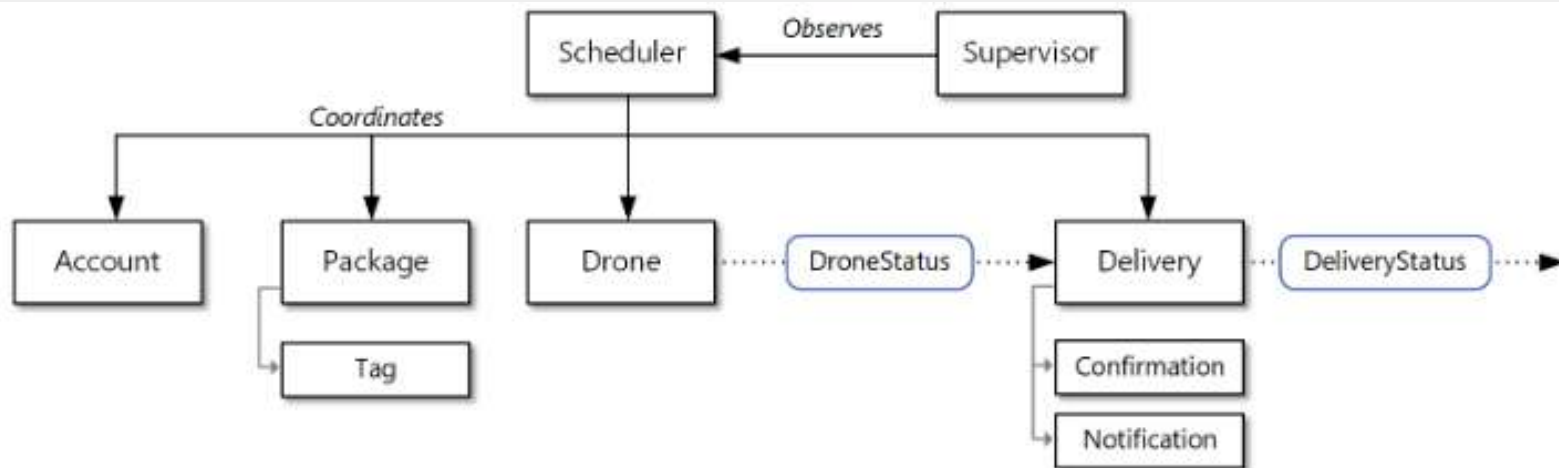


El equipo de desarrollo identificó un área más de funcionalidad, que no encaja claramente en ninguna de las entidades descritas hasta ahora.

Una parte del sistema debe **coordinar** todos los pasos implicados en la **programación** o actualización de una entrega.

Por lo tanto, el equipo de desarrollo agrega **dos servicios de dominio** al diseño: un **programador** (Scheduler) que coordina los pasos y un **supervisor** que supervisa el estado de cada paso, con el fin de detectar si en alguno se generó un error o se agotó su tiempo asignado. Esta es una variación del patrón **Scheduler Agent Supervisor** (supervisor del agente de programación).

DIAGRAMA DE MODELO TÁCTICO



IDENTIFICANDO LOS MICROSERVICIOS

PARTE II



Ahora estamos preparados para pasar del modelo de dominio al diseño de la aplicación. Este es un enfoque que puede usar para derivar microservicios desde el modelo de dominio.

IDENTIFICACION DE MICROSERVICIOS

Consiste en definir los modelos de dominio con más **precisión**. Los patrones tácticos se aplican dentro de un **único contexto delimitado**.

En una arquitectura de microservicios, interesan especialmente los **patrones de agregados y entidades**. Aplicar estos patrones nos ayudará a identificar los límites naturales de los servicios en nuestra aplicación

Análisis de
dominio



Definición de
contextos
acotados



Definición de
entidades,
relaciones y
servicios



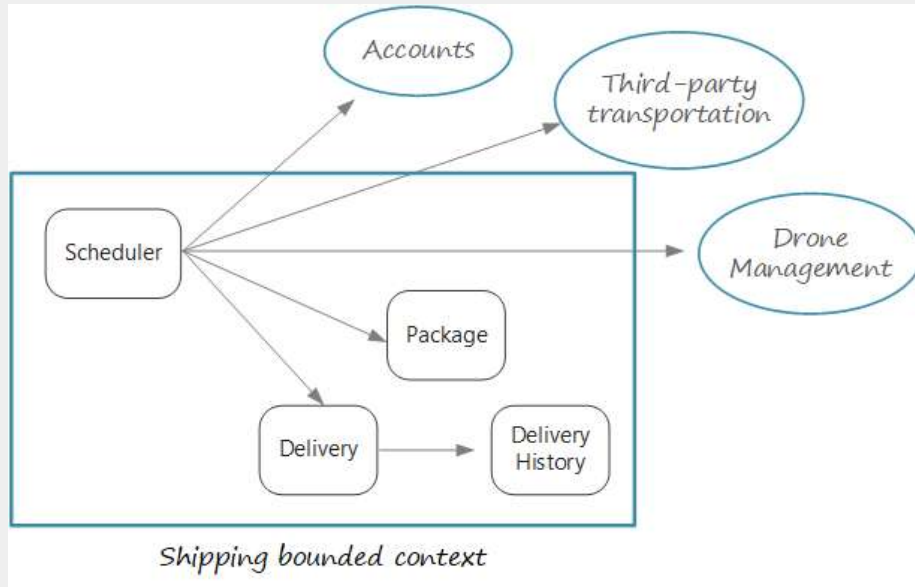
Identificación de
microservicios

COMENZAR CON UN CONTEXTO DELIMITADO

En general, la funcionalidad en un microservicio no debe abarcar más de un contexto delimitado.

Por definición, un contexto delimitado marca el límite de un modelo de dominio en particular. Si se encuentra que un microservicio mezcla modelos de dominio diferentes, es un signo de que puede que tenga que volver atrás para refinar el análisis de dominio.

COMENZAR CON UN CONTEXTO DELIMITADO



MIRAR LOS AGREGADOS

Los agregados suelen ser **buenos candidatos** para microservicios. Un agregado bien diseñado tiene muchas de las características de un microservicio bien diseñado, como:

- Un agregado se deriva de los requisitos empresariales más que de las preocupaciones técnicas, como el acceso a datos o la mensajería.
- Un agregado debe tener alta cohesión funcional.
- Un agregado es un límite de persistencia.
- Los agregados deben tener un acoplamiento flexible.

MIRAR LOS AGREGADOS

Los agregados suelen ser **buenos candidatos** para microservicios. Un agregado bien diseñado tiene muchas de las características de un microservicio bien diseñado, como:

- Un agregado se deriva de los requisitos empresariales más que de las preocupaciones técnicas, como el acceso a datos o la mensajería.
- Un agregado debe tener alta cohesión funcional.
- Un agregado es un límite de persistencia.
- Los agregados deben tener un acoplamiento flexible.

MIRAR LOS AGREGADOS


- . Entrega
- . Paquete
- . Dron
- . Cuenta



MIRAR LOS SERVICIOS DE DOMINIO

Los servicios de dominio también son buenos candidatos para microservicios. Ya que son operaciones sin estado a través de varios agregados.

Un ejemplo típico es un flujo de trabajo que implica varios microservicios

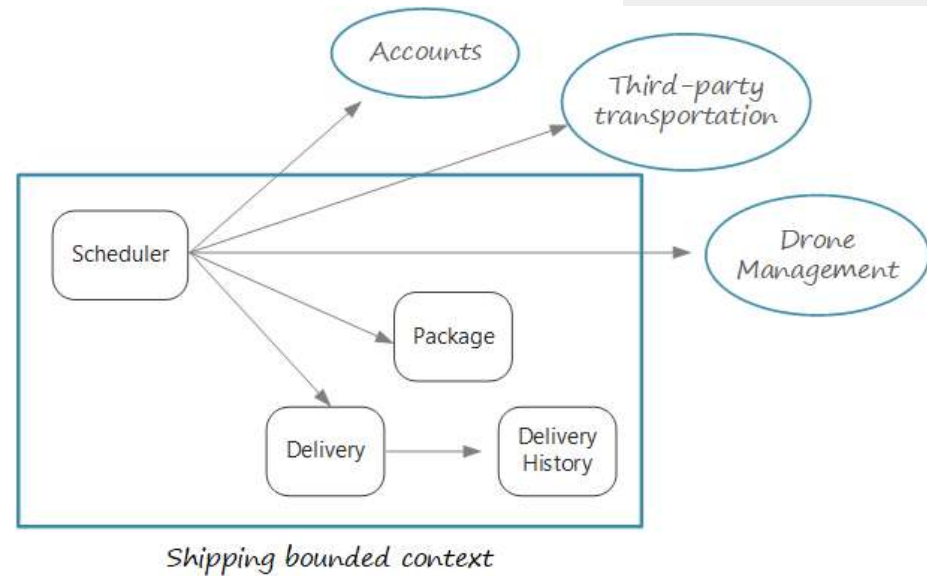
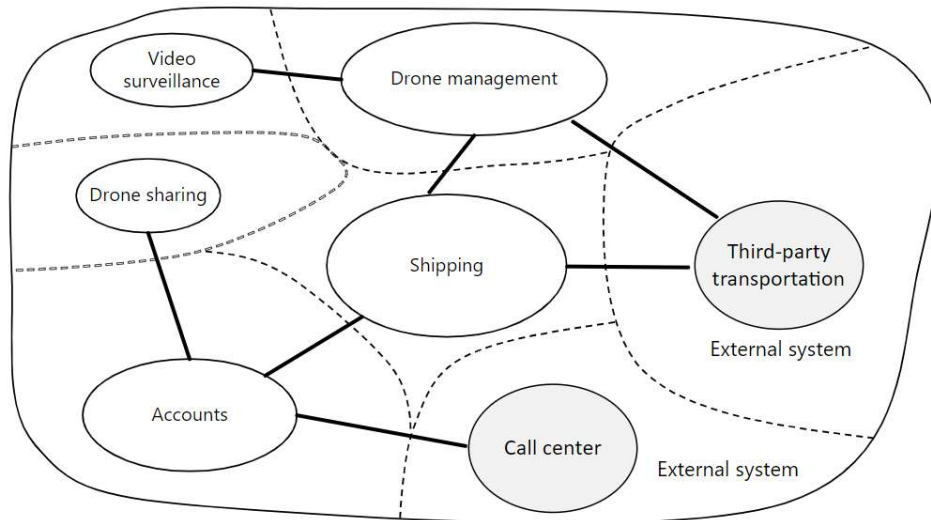
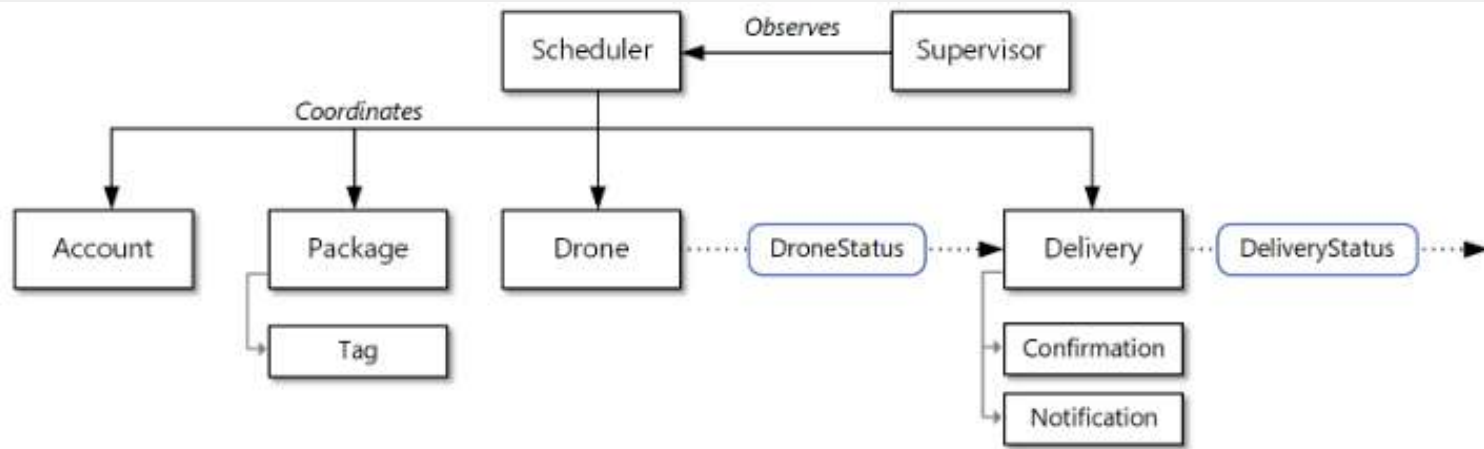


MIRAR LOS SERVICIOS DE DOMINIO

- Agendador
- Supervisor

MIRAR LOS REQUISITOS NO FUNCIONALES

- Observar factores como el tamaño del equipo, los tipos de datos, tecnologías, requisitos de escalabilidad, requisitos de disponibilidad y requisitos de seguridad.
- Estos factores pueden llevar a volver a **dividir** un microservicio en dos o más servicios más pequeños, o a hacer lo contrario **combinando** varios microservicios en uno.



VALIDANDO LOS MICROSERVICIOS



INTERDEPENDENCIAS

No hay interdependencias que requieran la implementación de dos o más servicios en sincronía.



RESPONSABILIDAD UNICA

Cada servicio tiene una única responsabilidad.



SERVICIOS MANEJABLES

Cada servicio es lo suficientemente pequeño como para que un equipo pequeño lo pueda generar trabajando de forma independiente.



BAJO ACOPLAMIENTO

Los servicios no están estrechamente acoplados y pueden evolucionar independientemente.



COHERENCIA E INTEGRIDAD


Los límites de servicio no causarán problemas con la coherencia de datos o la integridad.



CONVERSACIONES ACEPTABLES

No hay llamadas que generen mucha conversación entre servicios

Sobre todo, es importante ser práctico y recordar que el diseño basado en dominio es un proceso iterativo. En caso de duda, empezar con microservicios más generales.



Dividir un microservicio en dos servicios más pequeños es más sencillo que la refactorización de funcionalidad entre varios microservicios existentes.

DEDUCIENDO LOS MICROSERVICIOS

Delivery y Package son candidatos obvios para los microservicios.

Scheduler y Supervisor coordinan las actividades realizadas por otros microservicios, por lo que tiene sentido implementar estos servicios de dominio como microservicios.

Drone y Account son interesantes porque pertenecen a otros contextos delimitados. Una opción es que Scheduler llame a los contextos delimitados Drone y Account directamente. Otra opción es crear microservicios Drone y Account dentro del contexto delimitado Shipping. Estos microservicios mediarán entre los contextos delimitados a través de la exposición de las API o los esquemas de datos que son más adecuados para el contexto Shipping.

FACTORES A TENER EN CUENTA

- ¿Cuál es la sobrecarga de la red al llamar directamente al otro contexto delimitado?
- ¿Es el esquema de datos para el otro contexto delimitado adecuado para este contexto, o es mejor tener un esquema que se adapte a este contexto delimitado?
- ¿Es el otro contexto delimitado un sistema heredado (legacy)? Si es así, podría crear un servicio que actúe como una capa para evitar daños para traducir entre el sistema heredado y la aplicación moderna.
- ¿Cuál es la estructura del equipo? ¿Es fácil comunicarse con el equipo que se encarga del otro contexto delimitado? De lo contrario, la creación de un servicio que medie entre los dos contextos puede ayudar a mitigar el costo de la comunicación entre equipos.

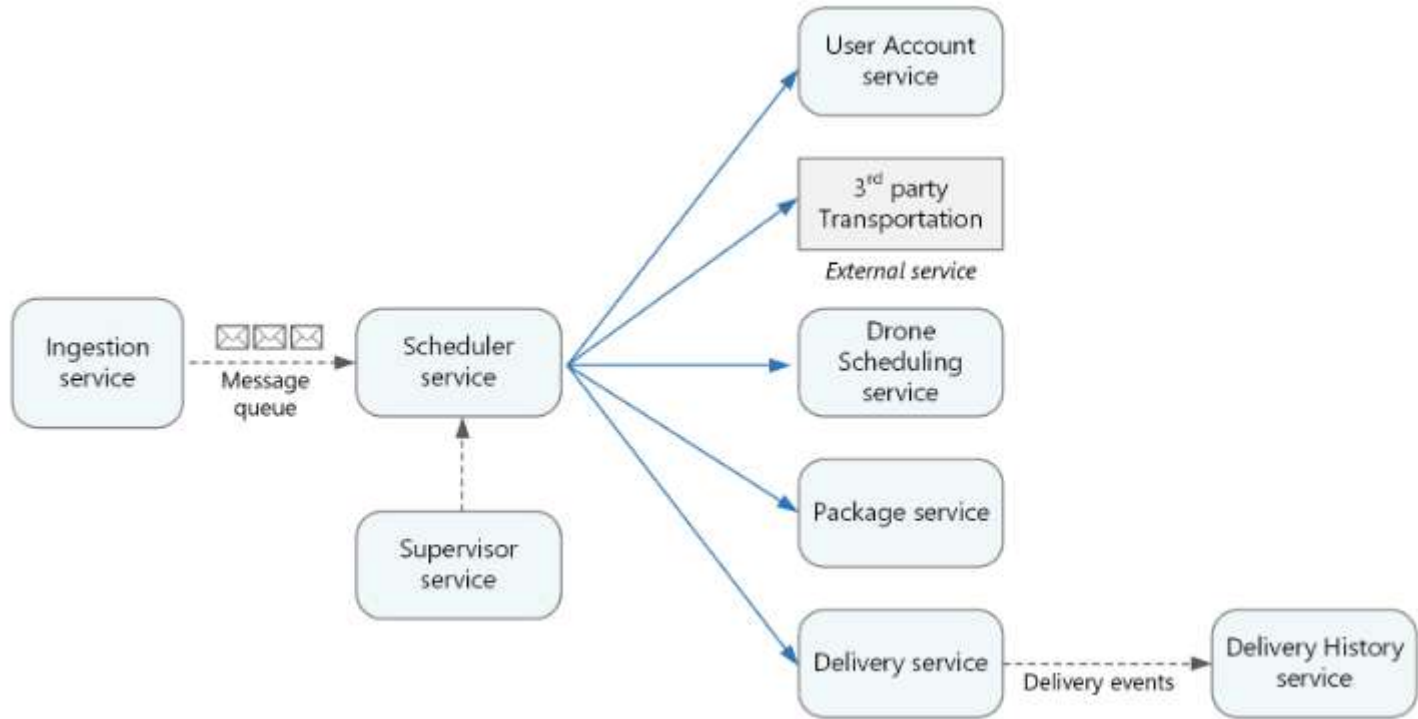
CONSIDERACIONES ADICIONALES

- Hasta ahora, no hemos considerado ningún **requisito no funcional**.
- Pensando en los requisitos de rendimiento de la aplicación, el equipo de desarrollo decidió crear un microservicio Ingestion independiente, que se encarga de la ingesta de las solicitudes de cliente. Este microservicio implementará la **redistribución de la carga** colocando las solicitudes entrantes en un búfer para su procesamiento. Scheduler leerá las solicitudes desde el búfer y ejecutará el flujo de trabajo

CONSIDERACIONES ADICIONALES

- También se necesita almacenar el historial de cada entrega en un almacenamiento a largo plazo para el análisis de datos. El equipo consideró hacer que esto fuera responsabilidad del servicio Delivery. Sin embargo, los requisitos de almacenamiento de datos para el análisis histórico son **muy distintos** a los de las operaciones en vuelo.
- Por lo tanto, el equipo decidió crear un servicio **Delivery History** independiente que realizará **escuchas de eventos** DeliveryTracking del servicio Delivery, y escribirá los eventos en un almacenamiento a largo plazo.

RESULTADO DEL EJERCICIO



MICROSERVICIOS



Your Everything Innovation Partner

Confidential and Proprietary. © 2019 UST Global Inc





USTGlobal®

MICROSERVICIOS

Presentado por:
Daniel Nieto

SESIÓN 3



OPCIONES DE PROCESO EN LA ARQUITECTURA



ELEGIR UNA OPCIÓN DE PROCESO PARA MICROSERVICIOS

El término proceso hace referencia al modelo de hospedaje para los recursos informáticos donde se ejecutan las aplicaciones. Para una arquitectura de microservicios, hay dos enfoques especialmente populares:

- Un orquestador de servicios que administra los servicios que se ejecutan en nodos dedicados (máquinas virtuales).
- Una arquitectura sin servidor mediante funciones como un servicio (FaaS).
- Aunque estas no son las únicas opciones, ambas son métodos probados para generar microservicios. Una aplicación puede incluir ambos enfoques.

ORQUESTADORES DE SERVICIOS

Un orquestador controla las tareas relacionadas con la implementación y administración de un conjunto de servicios. Estas tareas incluyen colocar servicios en los nodos, supervisar el estado de los servicios, reiniciar los servicios en mal estado, equilibrar la carga del tráfico de red en todas las instancias de servicio, detectar servicios, escalar el número de instancias de un servicio y aplicar actualizaciones de configuración. Entre los orquestadores más conocidos se encuentran Kubernetes, Service Fabric, DC/OS y Docker Swarm



Service Fabric



Azure Kubernetes Service (AKS)



ARQUITECTURA SIN SERVIDOR (FUNCIONES COMO UN SERVICIO)

Con una arquitectura sin servidor, no se administran las máquinas virtuales ni la infraestructura de red virtual. En su lugar, se implementa código y el servicio de hospedaje controla la colocación de ese código en una máquina virtual y lo ejecuta. Este enfoque tiende a favorecer a las pequeñas funciones pormenorizadas que se coordinan mediante el uso de los desencadenadores basados en eventos. Por ejemplo, un mensaje que se colocan en una cola puede desencadenar una función que lee de la cola y procesa el mensaje.

Las funciones de Azure es un servicio de proceso sin servidor que admite varios desencadenadores de función, incluidos los eventos de Event Hubs, colas de Service Bus y las solicitudes HTTP.

¿CÓMO ELEGIR? ¿ORQUESTADOR O SIN SERVIDOR?

Capacidad de administración: una aplicación sin servidor es fácil de administrar, porque la plataforma administra todos los recursos de proceso. Un orquestador aunque abstrae algunos aspectos de la administración y la configuración de un clúster, no oculta por completo las máquinas virtuales subyacentes. Con un orquestador, tendrá que pensar en problemas, como el equilibrio de carga, uso de CPU y memoria y funciones de red.

Flexibilidad y control. Un orquestador le ofrece un gran control sobre la configuración y administración de los servicios y del clúster. Como contrapartida tiene una complejidad adicional. Con una arquitectura sin servidor, se renuncia a un cierto grado de control porque se extraen estos detalles.

¿CÓMO ELEGIR? ¿ORQUESTADOR O SIN SERVIDOR?

Portabilidad. Todos los orquestadores listados (Kubernetes, DC/OS, Docker Swarm y Service Fabric) se pueden ejecutar de forma local o en varias nubes públicas.

Integración de aplicaciones. Puede resultar complicado crear una aplicación compleja con una arquitectura sin servidor, debido a la necesidad de coordinar, implementar y administrar muchas pequeñas funciones independientes. Una opción en Azure consiste en usar Azure Logic Apps para coordinar un conjunto de Azure Functions.

¿CÓMO ELEGIR? ¿ORQUESTADOR O SIN SERVIDOR?

Costo. Con un orquestador, se paga por las máquinas virtuales que se ejecutan en el clúster. Con una aplicación sin servidor, solo se paga por el consumo real de los recursos de proceso. En ambos casos, debe tener en cuenta el costo de los servicios adicionales, como almacenamiento, bases de datos y servicios de mensajería.

Escalabilidad. Azure Functions se escala automáticamente para satisfacer la demanda en función del número de eventos de entrada. Con un orquestador, puede escalar horizontalmente aumentando el número de instancias de servicio que se ejecutan en el clúster. También puede escalar agregando máquinas virtuales adicionales al clúster.



COMUNICACIÓN ENTRE MICROSERVICIOS

COMUNICACIÓN DE MICROSERVICIOS Y SUS DESAFIOS

- Resistencia
- Equilibrio de Carga
- Seguimiento Distribuido
- Versiones del servicio
- Cifrado TLS y autenticación TLS mutua



RESISTENCIA

- Puede haber decenas o incluso centenares de instancias de cualquier microservicio concreto. Se puede producir un error en una instancia por una serie de motivos. Puede haber un error de nivel de nodo, como un error de hardware o un reinicio de máquina virtual. Una instancia puede bloquearse o verse abrumada por las solicitudes, y dejar de procesar todas las nuevas solicitudes. Cualquiera de estos eventos puede provocar que una llamada de red genere un error.
- Hay dos patrones de diseño que pueden ayudar a que las llamadas de red de servicio a servicio ganen en resistencia:

RETRY

Una llamada de red puede producir un error debido a un error transitorio que desaparece por sí solo. En lugar de declarar un error directamente, el autor de la llamada debería reintentar la operación un determinado número de veces o hasta que transcurra un período de tiempo de espera configurado. Sin embargo, si una operación no es idempotente, los reintentos pueden producir efectos secundarios no deseados. La llamada original puede realizarse correctamente, pero el autor de la llamada nunca recibe una respuesta. Si el autor de la llamada vuelve a intentar realizarla, se puede invocar la operación dos veces. Por lo general, no es seguro volver a intentar los métodos POST o PATCH, porque no se garantiza que sean idempotentes.

CIRCUIT BREAKER .

Demasiadas solicitudes con error pueden causar un cuello de botella, ya que las solicitudes pendientes se acumulan en la cola. Estas solicitudes bloqueadas pueden contener recursos críticos del sistema, tales como la memoria, subprocesos o conexiones de base de datos, entre otros, que pueden causar errores en cascada. El patrón Circuit Breaker puede evitar que un servicio intente repetidamente una operación que probablemente produzca errores.

EQUILIBRIO DE CARGA.

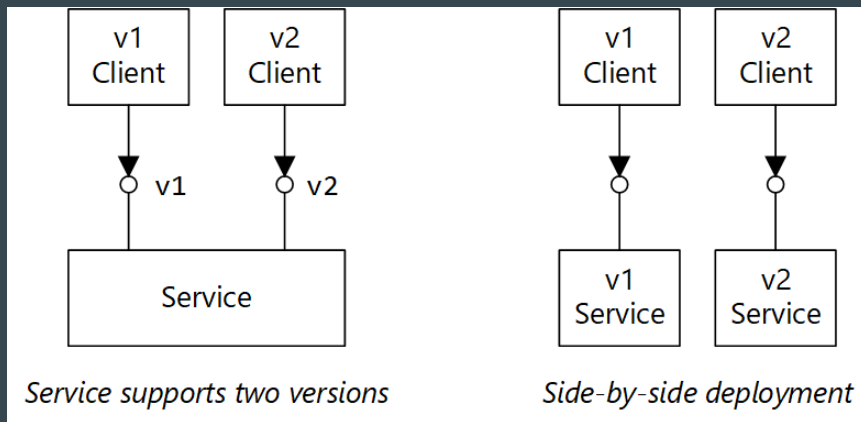
Cuando el servicio "A" llama servicio "B", la solicitud tiene que llegar a una instancia en ejecución del servicio "B". En Kubernetes, el tipo de recurso Service proporciona una dirección IP estable para un grupo de pods. El tráfico de red a la dirección IP del servicio se reenvía a un pod mediante reglas de iptable. De forma predeterminada, se elige un pod aleatorio. Una malla de servicio (ver abajo) puede proporcionar algoritmos de equilibrio de carga más inteligentes en función de la latencia observada o de otras métricas.

SEGUIMIENTO DISTRIBUIDO.

Una sola transacción puede abarcar varios servicios. Esto puede dificultar la supervisión del rendimiento general y del estado del sistema. Incluso si todos los servicios generan registros y métricas, si no hay una manera de unirlos, son de uso limitado.

VERSIONES DEL SERVICIO

Cuando un equipo implementa una nueva versión de un servicio, tiene que evitar romper otros servicios o clientes externos que dependen de él. Además, puede ejecutar varias versiones de un servicio en paralelo y enrutar las solicitudes a una versión determinada.



CIFRADO TLS Y AUTENTICACIÓN DE TLS MUTUA

Por motivos de seguridad, puede querer cifrar el tráfico entre los servicios con TLS y utilizar la autenticación de TLS mutua para autenticar a los autores de las llamadas.



AZURE FUNCTIONS

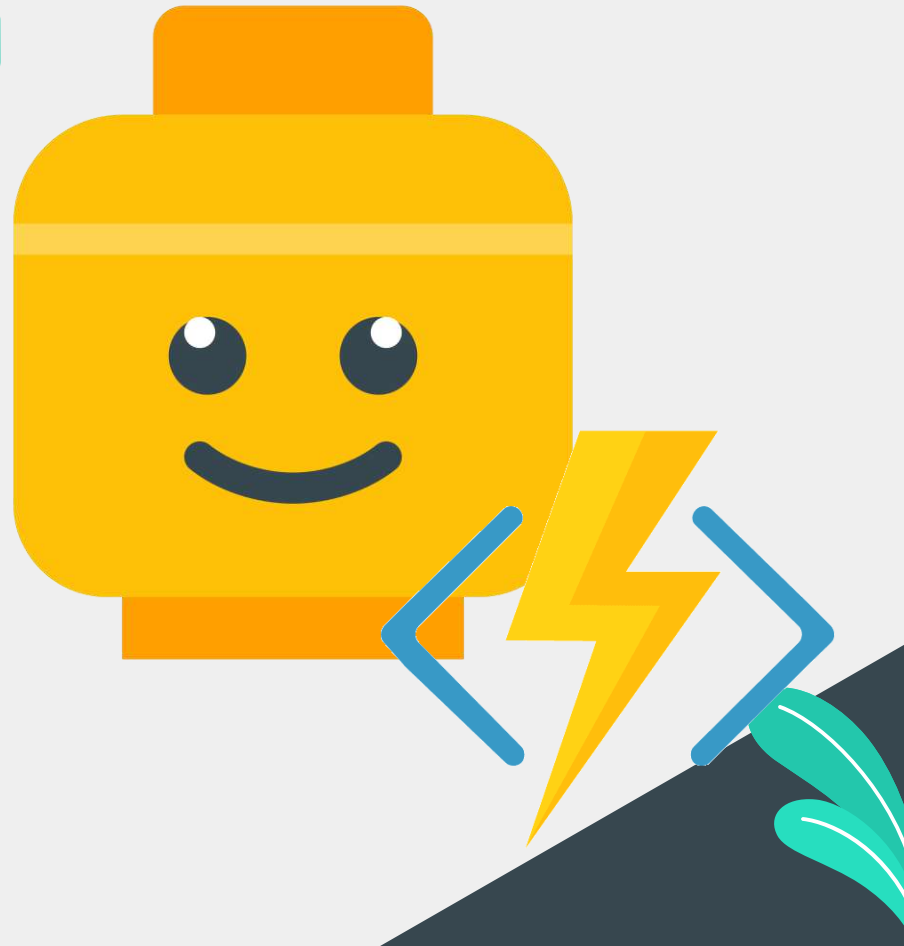
CONSTRUYENDO BLOQUES

Una de las maneras más sencillas de implementar Microservicios en la nube es a través de **AZURE FUNCTIONS**, ya que con esta forma de implementación :

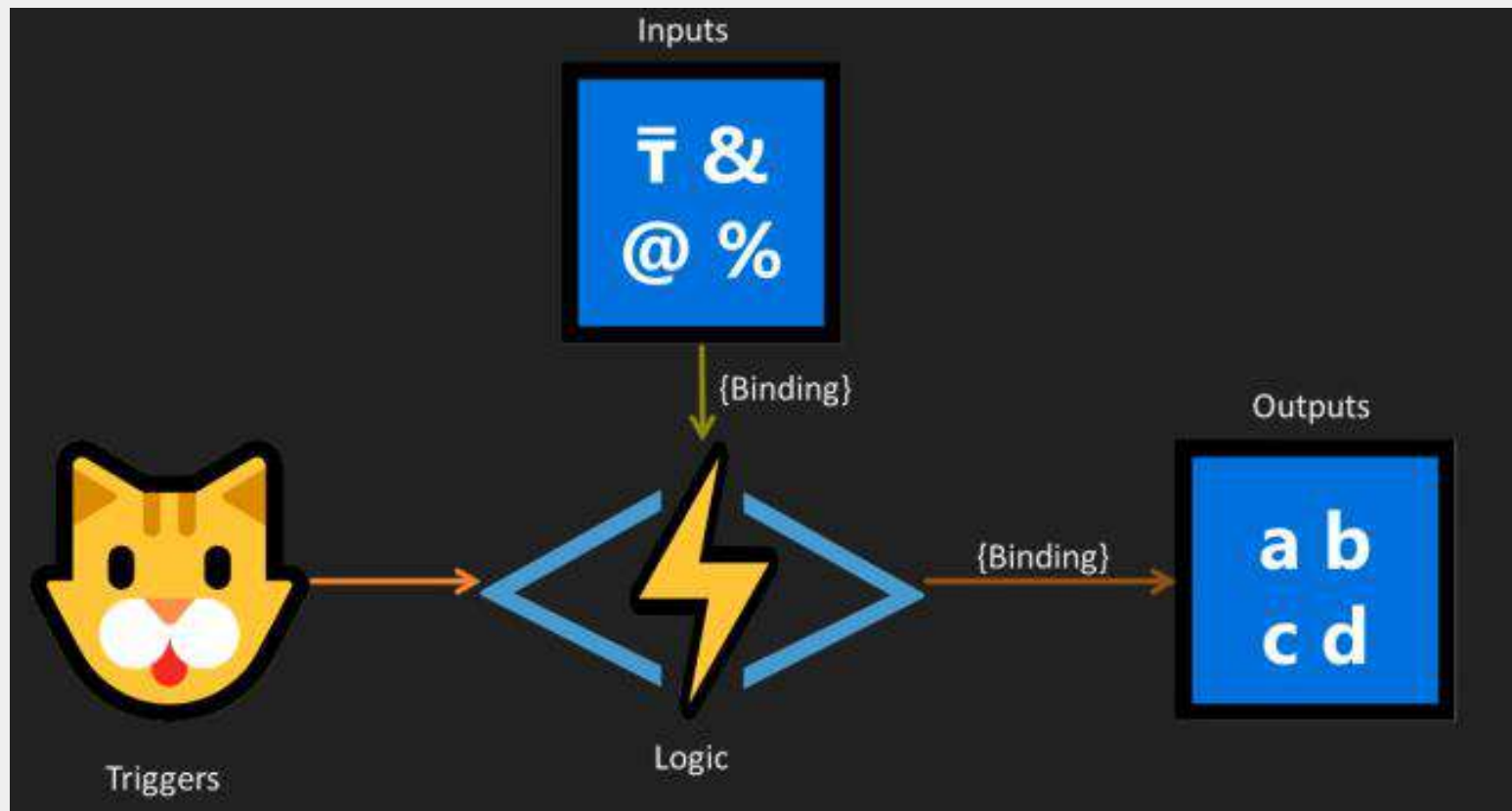
- No piensas en los Servidores ni en toda su configuración o mantenimiento.

- Escalan automáticamente de acuerdo a tus necesidades.

- Brindan conectividad e integración con otros servicios




COMPONENTES DE UNA AZURE FUNCTION



Triggers



 HTTP trigger

A function that will be run whenever it receives an HTTP request, responding based on data in the body or query string.

C# F# JavaScript

 Timer trigger

A function that will be run on a specified schedule

C# F# JavaScript

 Queue trigger

A function that will be run whenever a message is added to a specified Azure Storage queue

C# F# JavaScript

 Service Bus Queue trigger

A function that will be run whenever a message is added to a specified Service Bus queue

C# F# JavaScript

 Service Bus Topic trigger

A function that will be run whenever a message is added to the specified Service Bus Topic

C# F# JavaScript

 Blob trigger

A function that will be run whenever a blob is added to a specified container

C# F# JavaScript

 Event Hub trigger

A function that will be run whenever an event hub receives a new event

C# F# JavaScript

 Cosmos DB trigger

A function that will be run whenever documents change in a document collection

C# JavaScript

 IoT Hub (Event Hub)

A function that will be run whenever an IoT Hub delivers a new message for Event Hub-compatible endpoints

C# F# JavaScript



Queue trigger

A function that will be run whenever a message is added to a specified Azure Storage queue

C# F# JavaScript



Azure Queue Storage trigger [x delete](#)

Message parameter name ⓘ

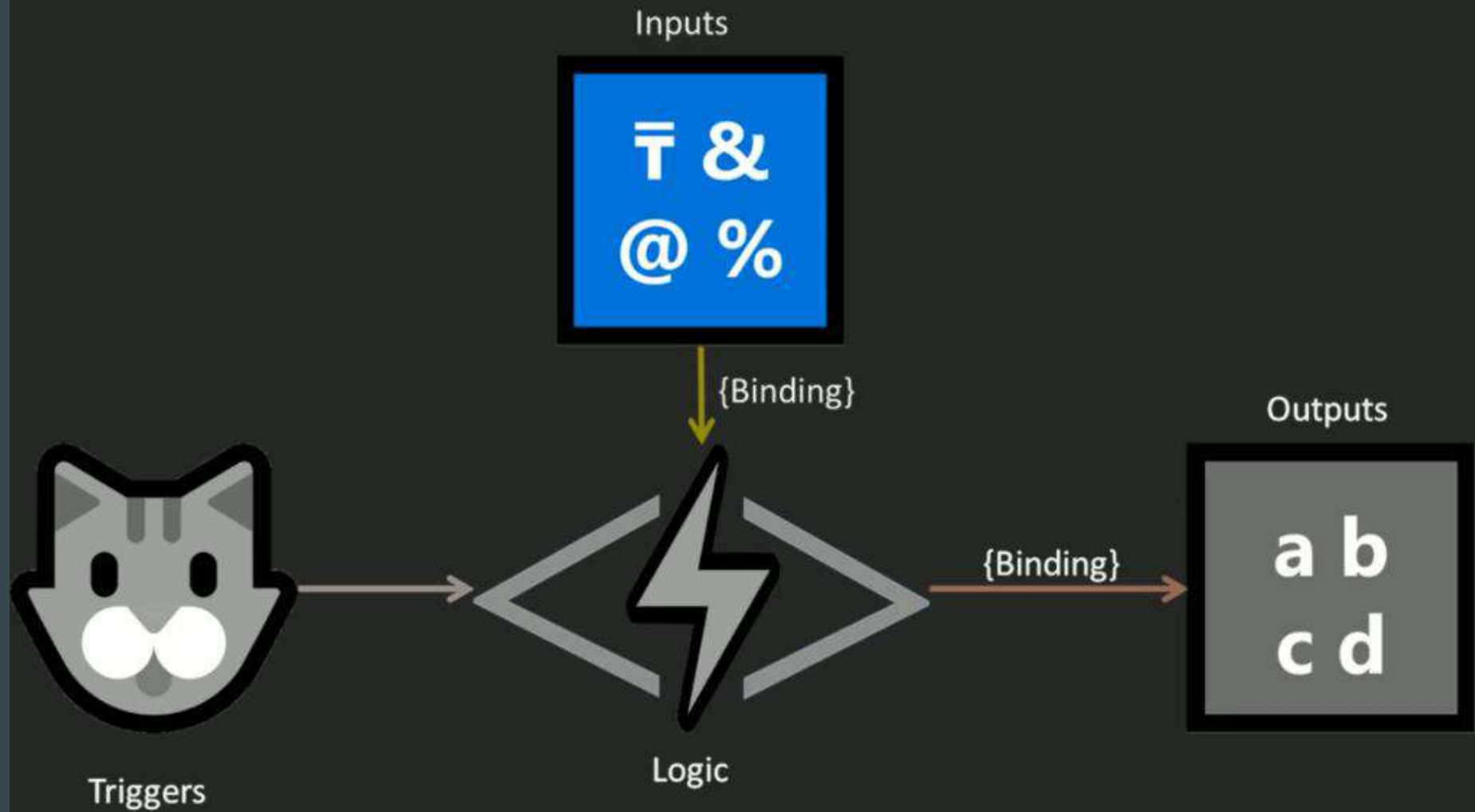
Queue name ⓘ

Storage account connection ⓘ

[show value](#)



[new](#)



INPUTS



Azure Blob Storage



External File (Preview)



External Table (Experimental)



Azure Table Storage



Azure Cosmos DB



Azure Mobile Tables

Azure Table Storage input

Table parameter name ⓘ

Partition key (optional) ⓘ

Maximum number of records to read (optional) ⓘ

Storage account connection ⓘ

[show value](#)

[new](#)

Table name ⓘ

Row key (optional) ⓘ

Query filter (optional) ⓘ

Save

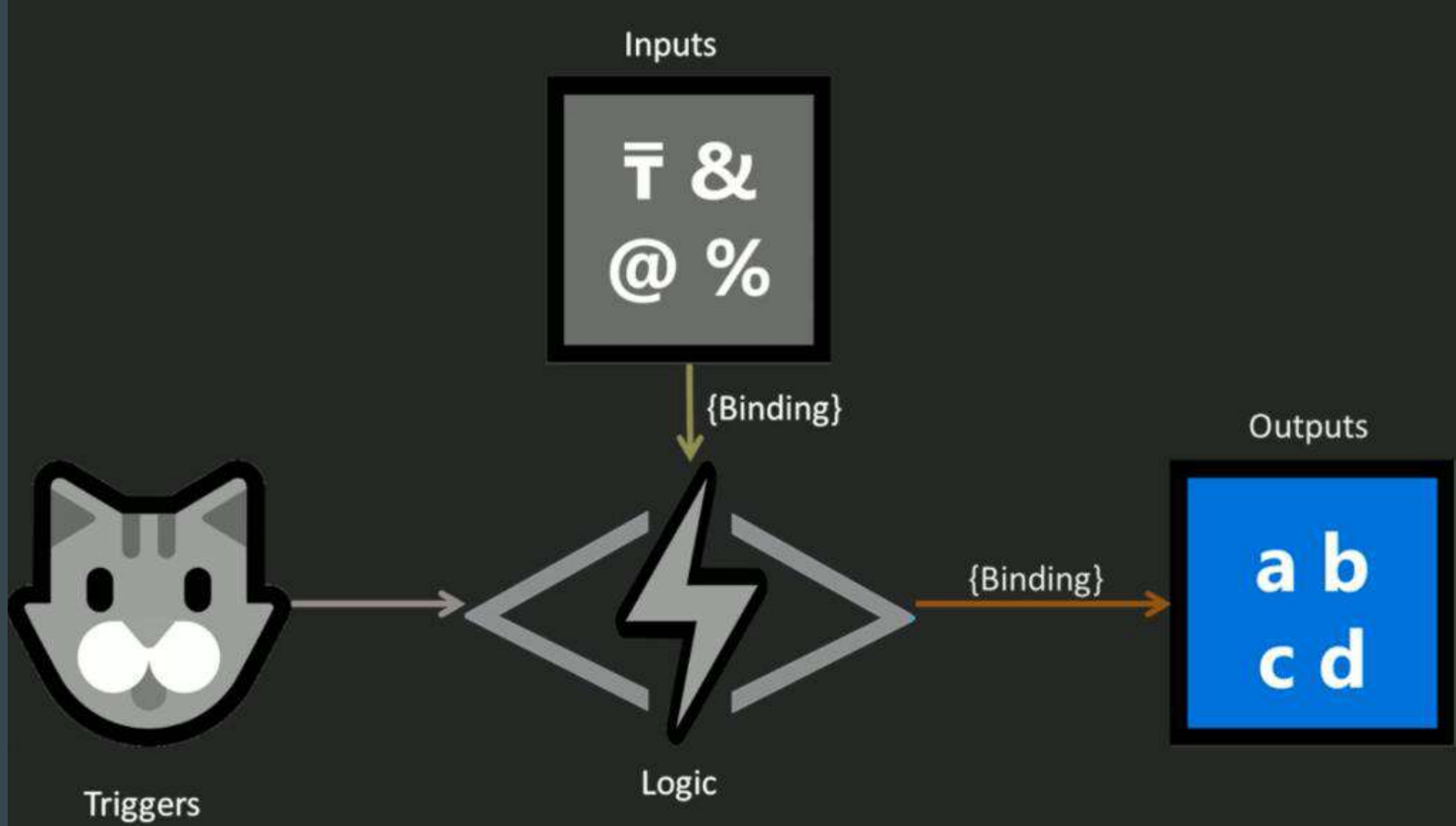
Cancel



Qué son los Bindings?

- Son atributos para decorar la función
- Estos atributos resuelven toda la lógica de conectividad y transformación de los datos
- Triggers, y parámetros de Input y Output funcionan de esta manera
- Te enfocas en la lógica, no en los detalles de conexión o formato de los objetos.
- La manera en que se configuran estos atributos depende del lenguaje y tipo de Functions que utilices





OUTPUTS



Azure Blob Storage output [x delete](#)

Blob parameter name ⓘ

☐ Use function return value

Storage account connection ⓘ [show value](#)
 [new](#)

Path ⓘ

Actions
Create a new function triggered by this output [Go](#)

ESCALABILIDAD Y HOSTING DE LAS FUNCTIONS



- Colección de Functions
- Un proyecto en VS es a la final una function app.
- La function App es hospedada en un plan
 - Consumption plan
 - App Service plan

TIPOS DE PLAN

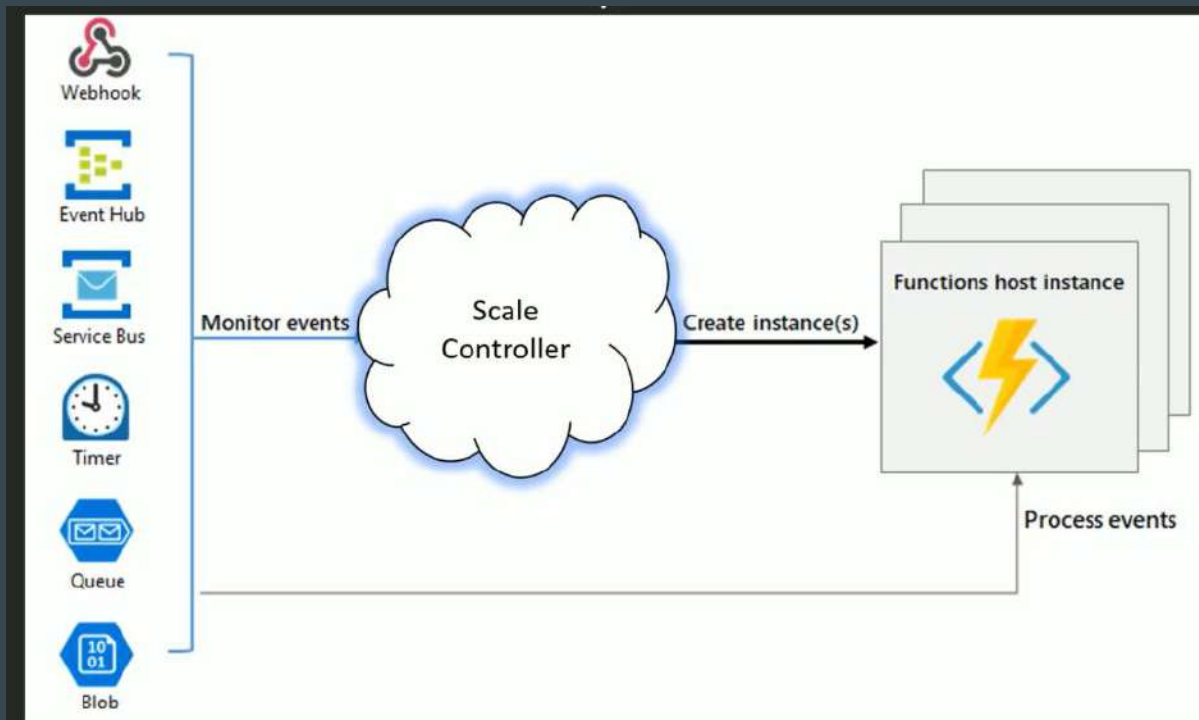
Plan App Service

- Ya tenemos App Services subutilizados
- Se requiere correr continuamente
- El proceso requiere más memoria o procesador del ofrecido por el plan de consumo
- Necesita más tiempo de ejecución

Plan de Consumo

- 1.5 GB
- 10 minutos x function
- Facturación por {ejecuciones, tiempo, memoria}
- Escala automáticamente incluso en escenarios de alta demanda
- Always On

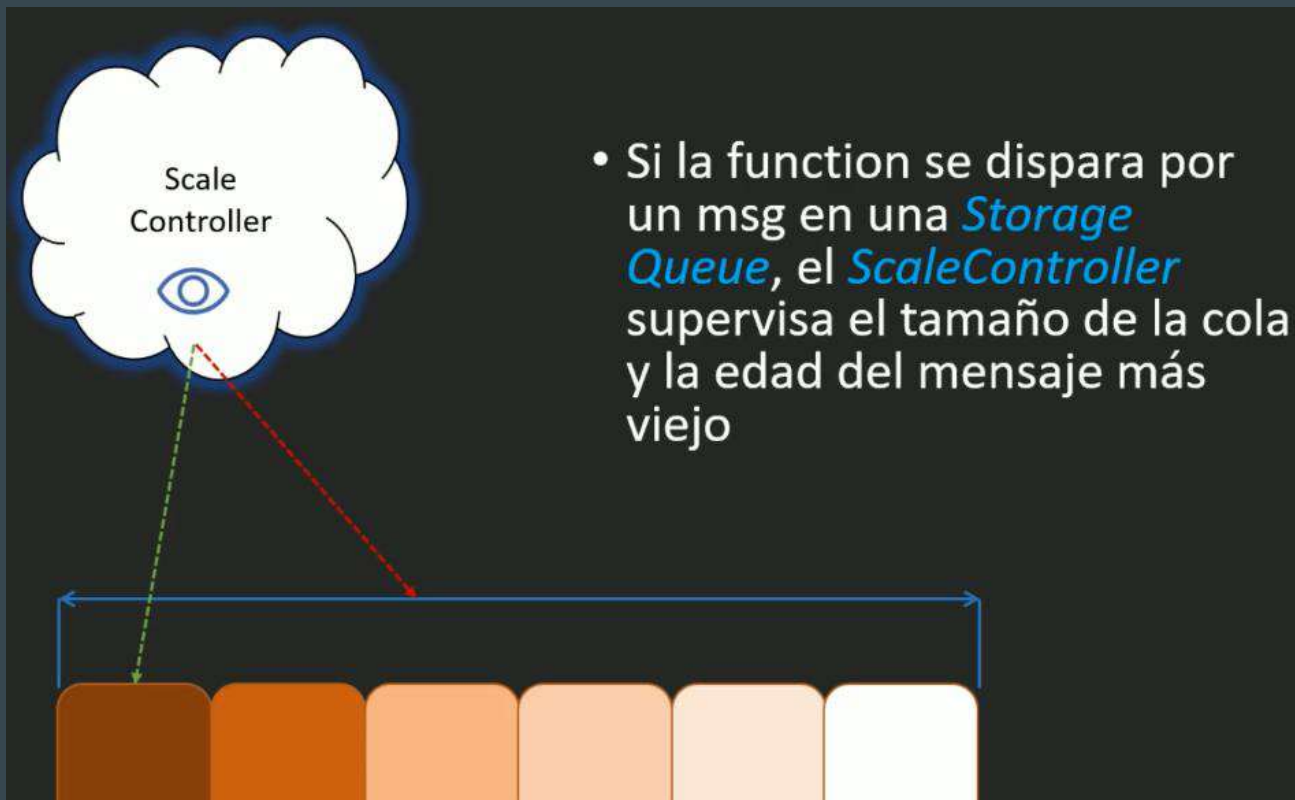
COMO FUNCIONA EL PLAN DE CONSUMO



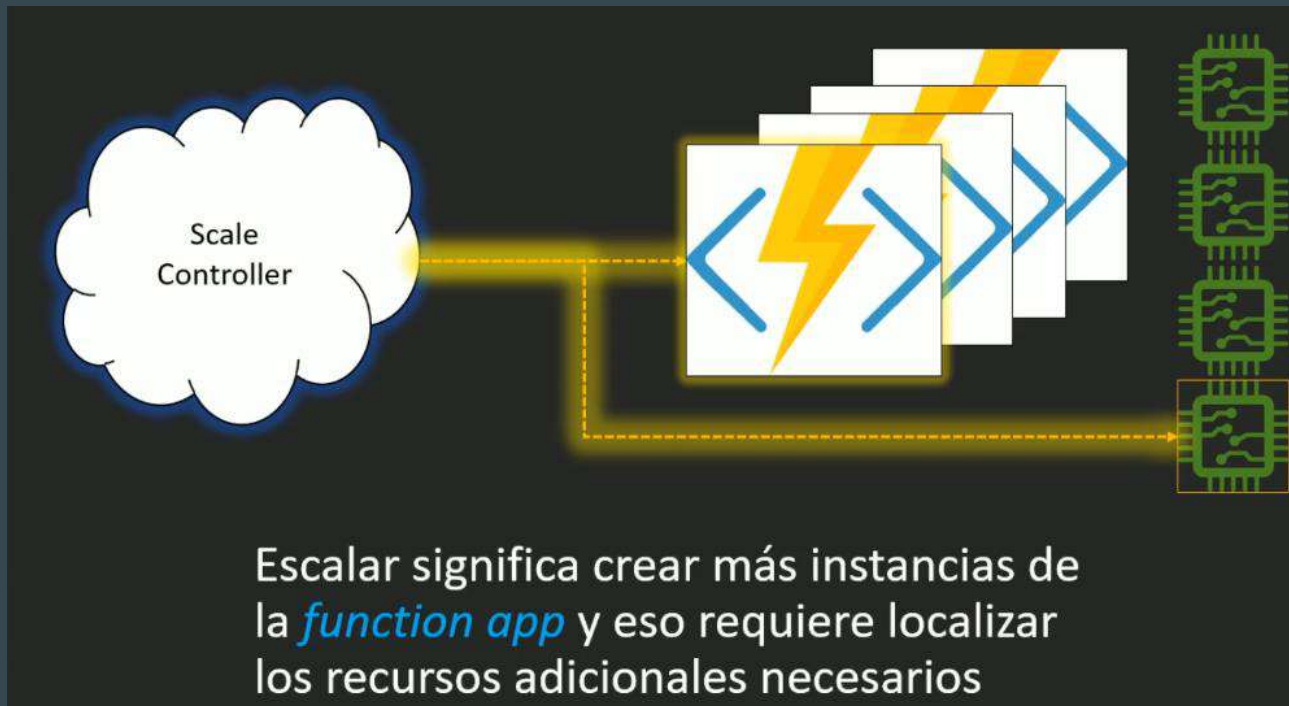
- Scale Controller
- Tiene heurísticas para cada tipo de trigger

- Cuando se cumple la condición escala
- La unidad de escala es la función app

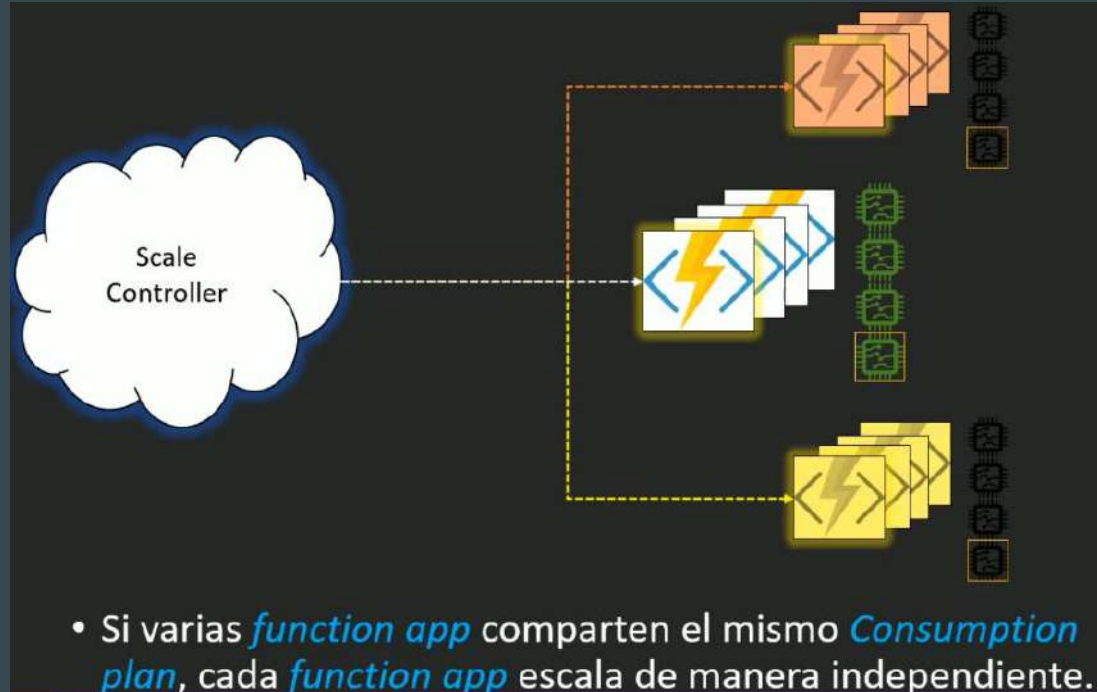
COMO FUNCIONA EL PLAN DE CONSUMO



COMO FUNCIONA EL PLAN DE CONSUMO



COMO FUNCIONA EL PLAN DE CONSUMO



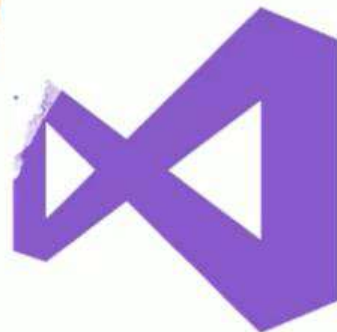
TRABAJANDO CON AZURE FUNCTIONS

Formas de Crear Functions con C#

- Desde el portal
- Visual Studio Code
- Visual Studio Community
- Notepad 



Press to exit full screen



Tipos de Functions con C#

C# Script (csx)

- Prototipado
- No necesita de un proyecto
- Los script se publican en el portal

Precompiled Functions

- Proyecto de visual studio
- Se compila antes de publicar (se publica un dll)

Durable Functions

Variante de Precompiled Functions que entrega control sobre el scale controller

SESIÓN 4



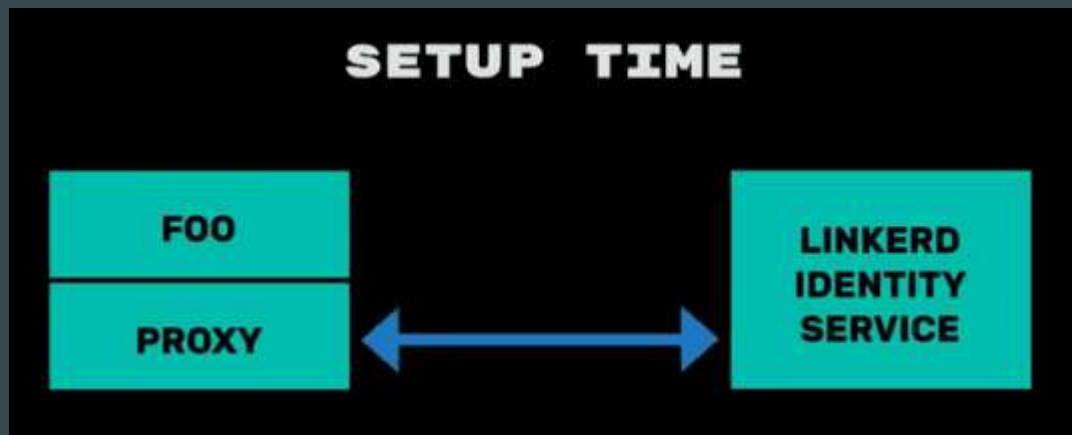


DISEÑO DE ARQUITECTURA

MALLAS DE SERVICIO

Una malla de servicio es una capa de software que controla la comunicación de servicio al servicio. Las mallas del servicio están diseñadas para tratar muchos de los problemas enumerados en la sesión anterior, y para mover la responsabilidad de estas preocupaciones fuera de los microservicios a una **capa compartida**. La malla del servicio actúa como un **proxy** que intercepta la comunicación de red entre microservicios en el clúster. Actualmente, el concepto de malla de servicio se aplica principalmente a los **organizadores de contenedores**, en lugar de las arquitecturas sin servidor.

MALLAS DE SERVICIO



FUNCIONES DE LA MALLA DE SERVICIOS

- Retry
- Circuit breaker
- Equilibrio de Carga
- Seguimiento Distribuido
- Enrutamiento
- Cifrado TLS y autenticación TLS mutua

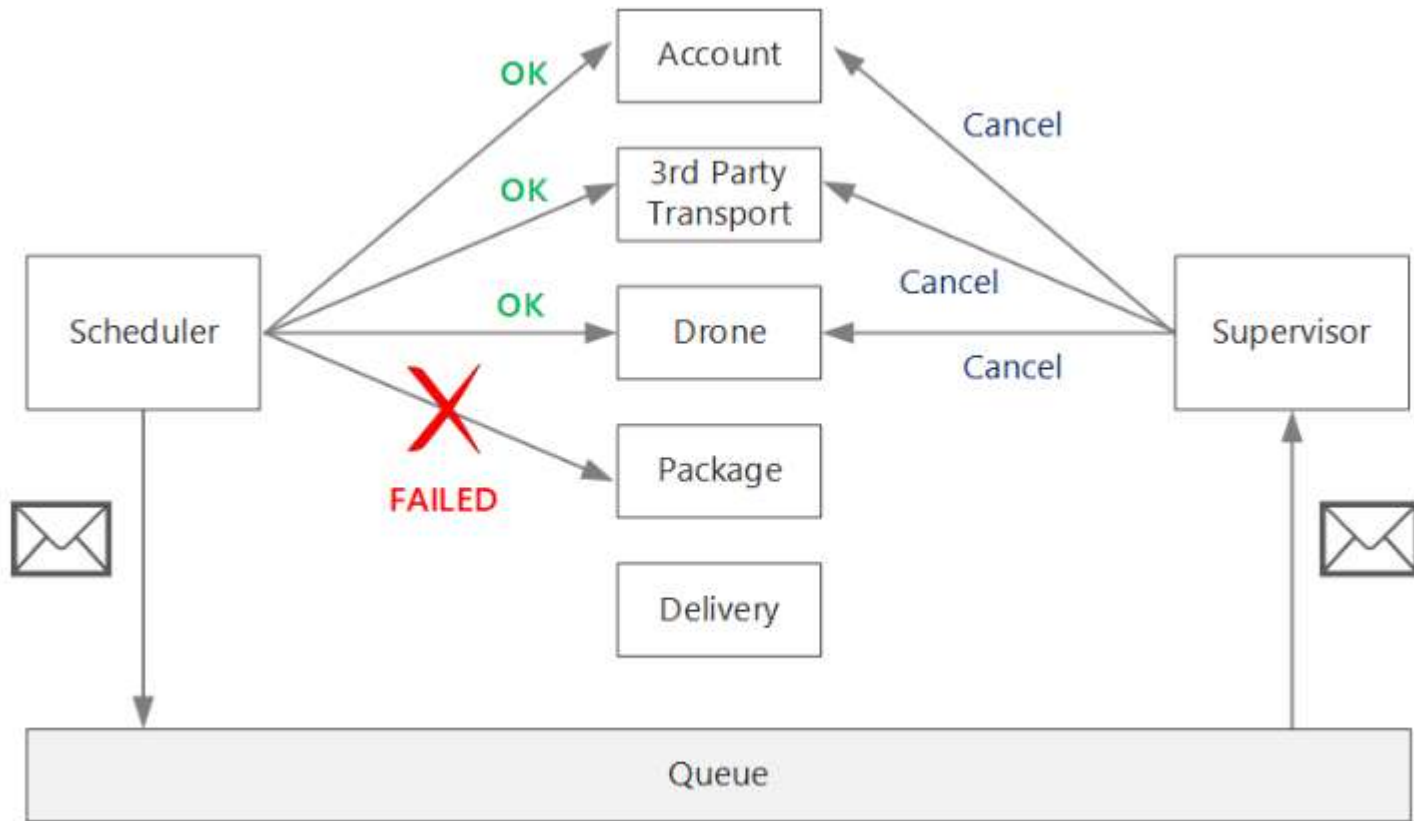


ES IMPORTANTE RECORDAR QUE...



La malla de servicio es un ejemplo del **patrón Ambassador** —, un servicio de aplicación auxiliar que envía solicitudes de red en nombre de la aplicación.

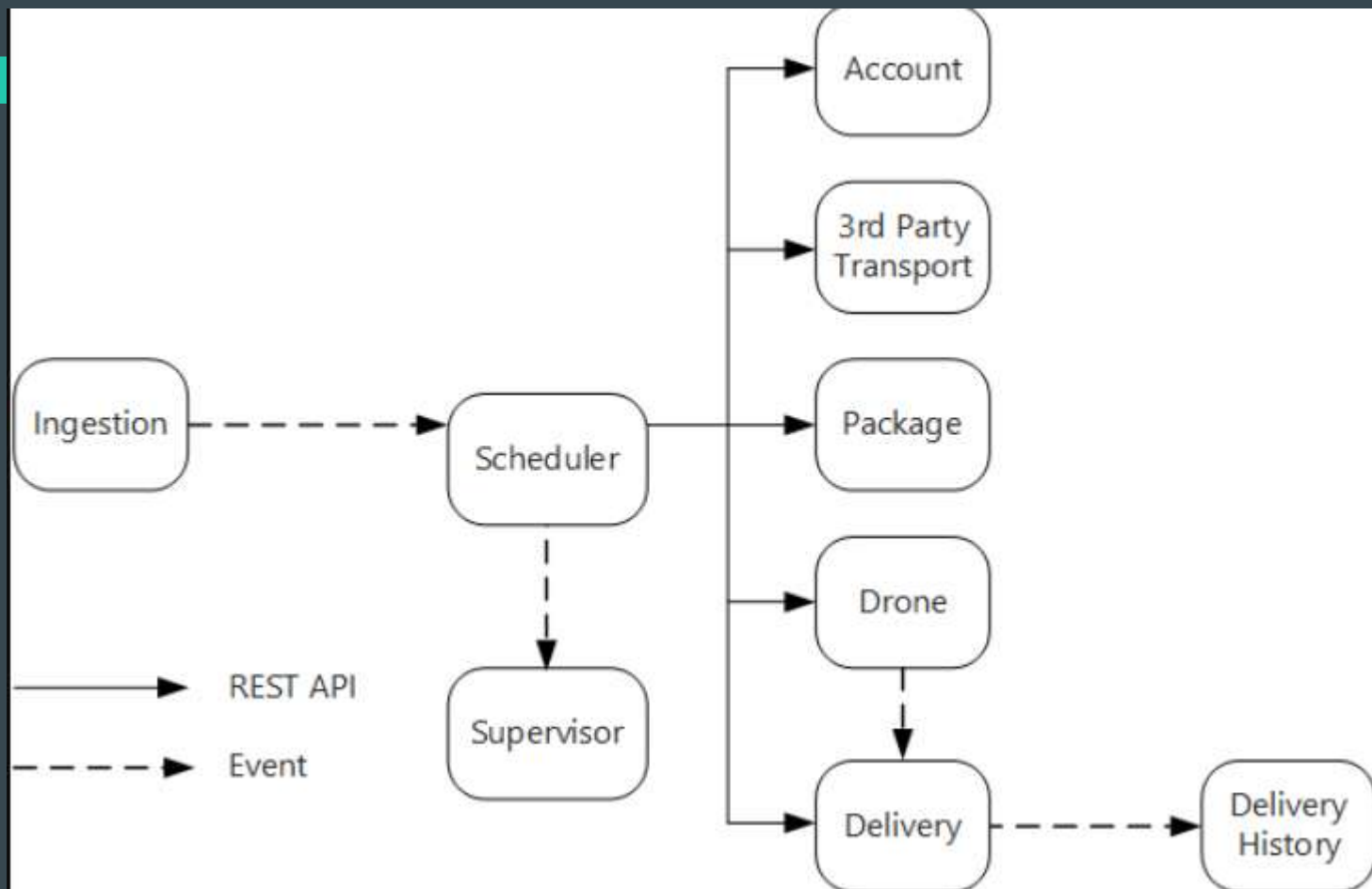
TRANSACCIONES DISTRIBUIDAS



EJEMPLO

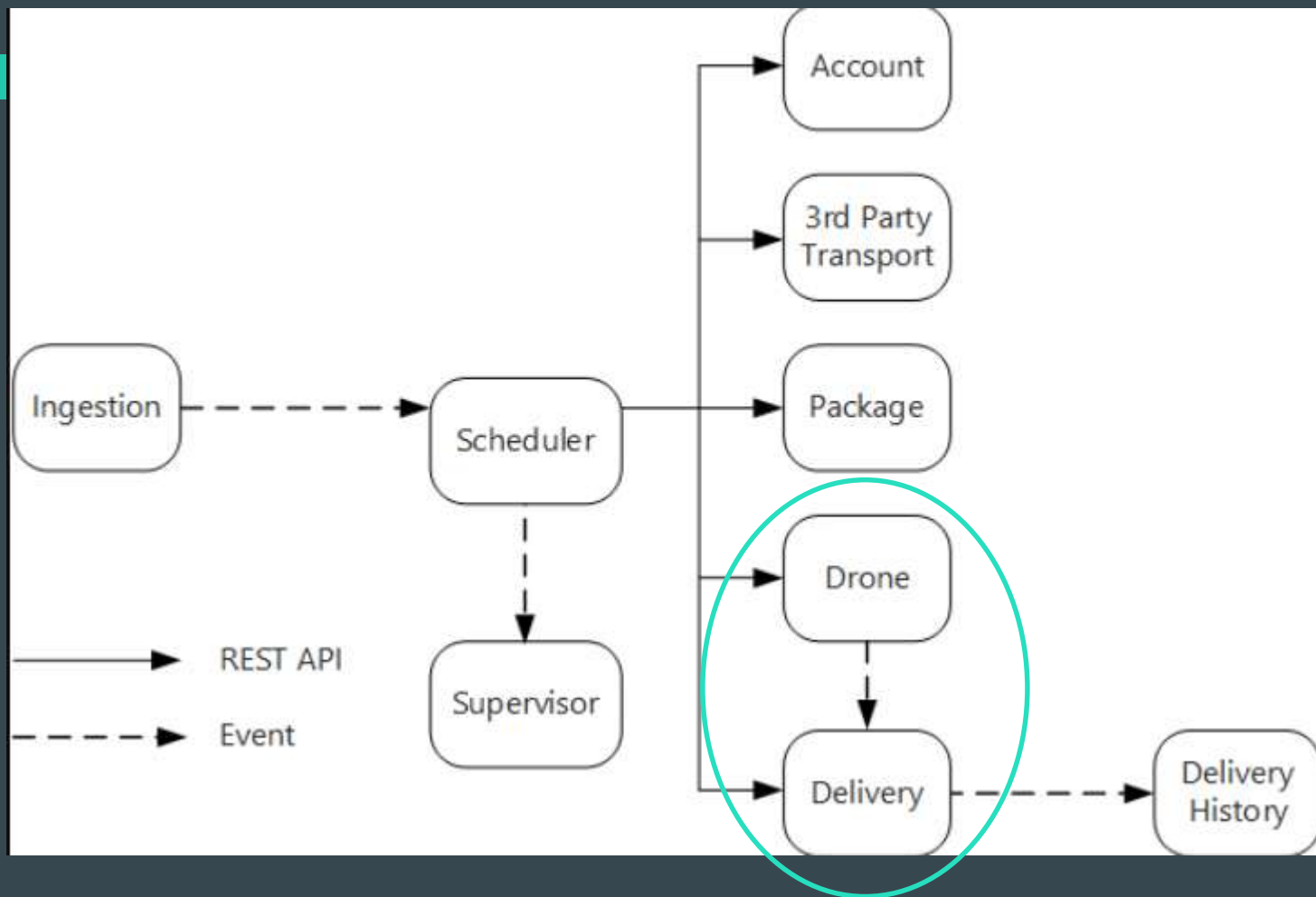
DRONE DELIVERY





ELECCIÓN DE LOS PATRONES DE MENSAJERÍA

Tenga en cuenta que los eventos de estado de entrega se derivan de los eventos de la ubicación de dron. Por ejemplo, cuando un dron llega a una ubicación de entrega y suelta un paquete, el servicio Delivery traduce esto en un evento DeliveryCompleted. Este es un ejemplo de pensamiento en términos de modelos de dominio. Como se describió anteriormente, Drone Management pertenece a un contexto independiente enlazado. Los eventos de dron proporcionan la ubicación física de un dron. Los eventos de entrega, por otro lado, representan los cambios en el estado de una entrega, que es una entidad empresarial diferente.



DISEÑO DE API PARA MICROSERVICIOS

Dado que los servicios están diseñados por equipos que trabajan de forma independiente, las API deben tener semántica bien definida y esquemas de control de versiones, de manera que las actualizaciones no interrumpan otros servicios.

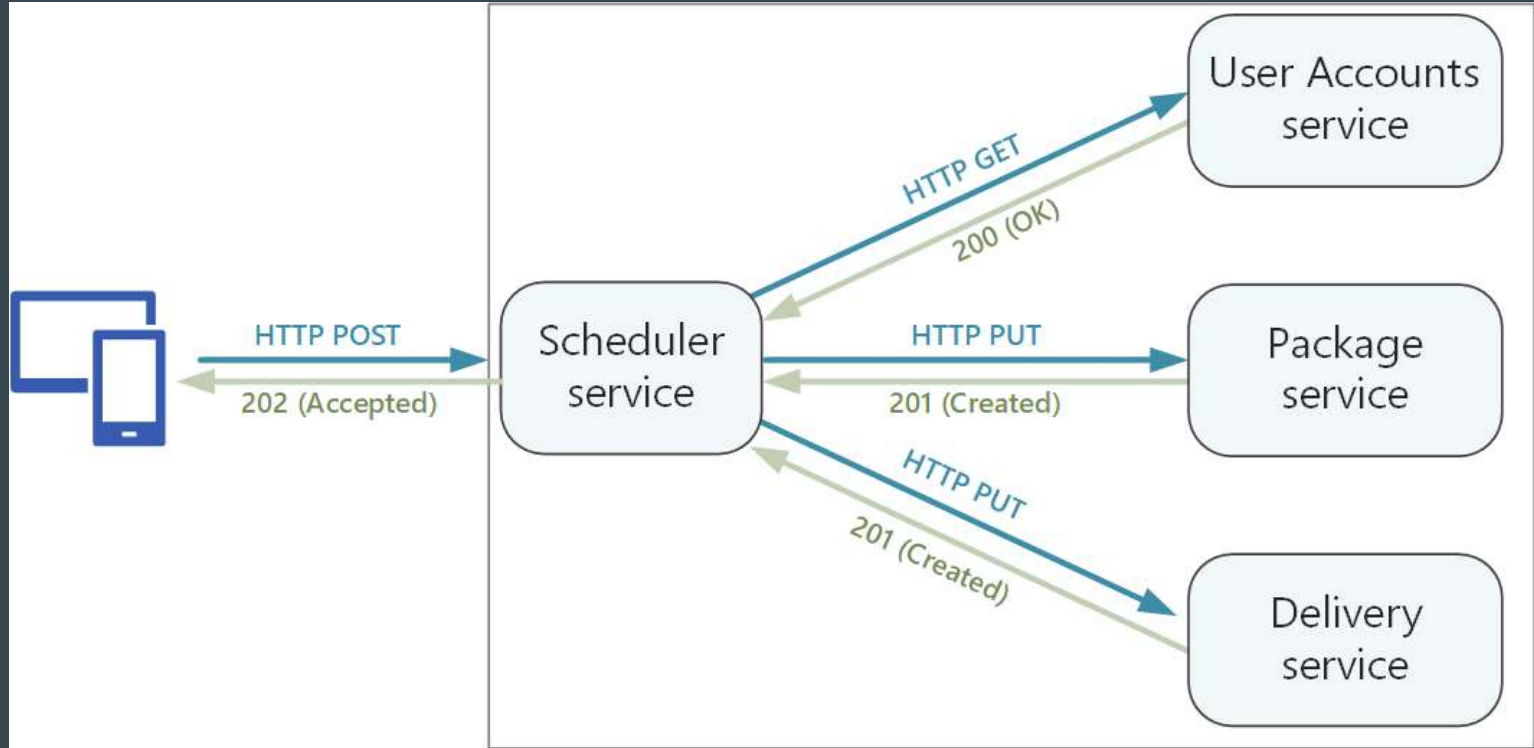
Es importante distinguir entre dos tipos de API:

- Las API públicas que llaman las aplicaciones cliente. (REST/HTTP)
- Las API de back-end que se usan para la comunicación entre servicios (gRPC, Apache Avro y Apache Thrif)

DISEÑO DE API PARA MICROSERVICIOS - CONSIDERACIONES

- **REST VS RPC:** REST modela recursos, lo cual puede ser una manera natural de expresar el modelo de dominio. RPC está más orientado a las operaciones o los comandos.
- **Eficacia:** Considere la eficacia en cuanto a tamaño de carga, memoria y velocidad. Una interfaz basada en gRPC suele ser más rápida que REST a través de HTTP.
- **Serialización:** Las opciones incluyen formatos de texto (principalmente, JSON) y formatos binarios como el búfer de protocolo. Los formatos binarios son generalmente más rápidos que los de texto.

DISEÑO DE API PARA MICROSERVICIOS

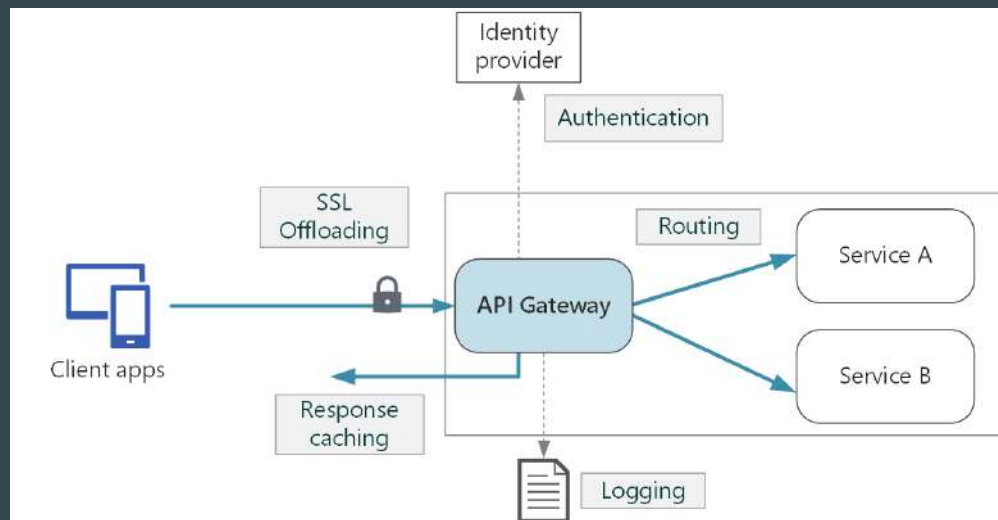


DISEÑO DE API PARA MICROSERVICIOS

Concepto de DDD	Equivalente en REST	Ejemplo
Agregado	Resource	<code>{ "1":1234, "status":"pending"... }</code>
Identidad	URL	<code>https://delivery-service/deliveries/1</code>
Entidades secundarias	Vínculos	<code>{ "href": "/deliveries/1/confirmation" }</code>
Actualización de objetos de valor	PUT o PATCH	<code>PUT https://delivery-service/deliveries/1/dropoff</code>
Repositorio	Colección	<code>https://delivery-service/deliveries?status=pending</code>

USO DE PUERTAS API PARA MICROSERVICIOS

En una arquitectura de microservicios, un cliente puede interactuar con más de un servicio front-end. Así, ¿cómo un cliente sabe a qué puntos de conexión llamar? ¿Qué ocurre cuando se introducen nuevos servicios o los servicios existentes se refactorizan? ¿Cómo abordan los servicios la terminación SSL, la autenticación y otros problemas? Una puerta de enlace de API puede ayudar a superar estos desafíos.



USO DE PUERTAS API PARA MICROSERVICIOS

Estos son algunos ejemplos de funcionalidad que puede descargar a una puerta de enlace:

- Terminación de SSL
- Autenticación
- Lista de direcciones IP permitidas
- Limitación de la tasa de clientes
- Registro y supervisión
- Almacenamiento en caché de respuesta
- Firewall de aplicaciones web
- Compresión GZIP
- Mantenimiento de contenido estático

PUERTAS API - HERRAMIENTAS

- **Servidor de proxy inverso.** Nginx y HAProxy son servidores de proxy inverso comunes que admiten características tales como el equilibrio de carga, SSL y enrutamiento de nivel 7. Ambos son productos gratuitos y de código abierto
- **Controlador de entrada del servicio de malla.** Si utiliza un servicio malla como linkerd o Istio, tenga en cuenta las características que se proporcionan con el controlador de entrada para ese servicio malla
- **Azure Application Gateway.** Servicio de equilibrio de carga administrada que puede realizar el enrutamiento de nivel 7 y la terminación SSL.
- **Azure API Management.** Azure API Management es una solución completa para publicar API para clientes externos e internos. Proporciona características que resultan útiles para administrar una API expuesta al público, incluida la limitación de velocidad, la lista de direcciones IP permitidas y la autenticación con Azure Active Directory u otros proveedores de identidad. API Management no realiza el equilibrio de carga

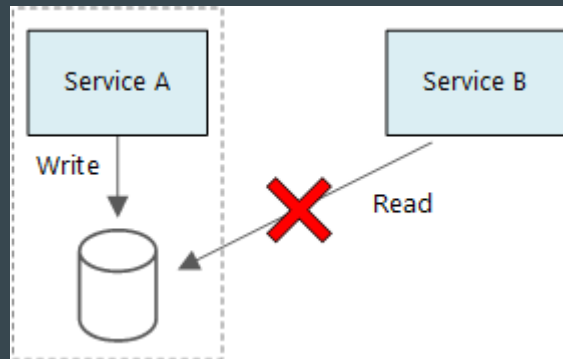
PERSISTENCIA EN MICROSERVICIOS

Un principio básico de los microservicios es que **cada servicio administra sus propios datos**. Dos servicios no deben compartir un mismo almacén de datos. En su lugar, cada servicio es responsable de su propio almacén de datos privado, al que otros servicios no pueden acceder directamente.

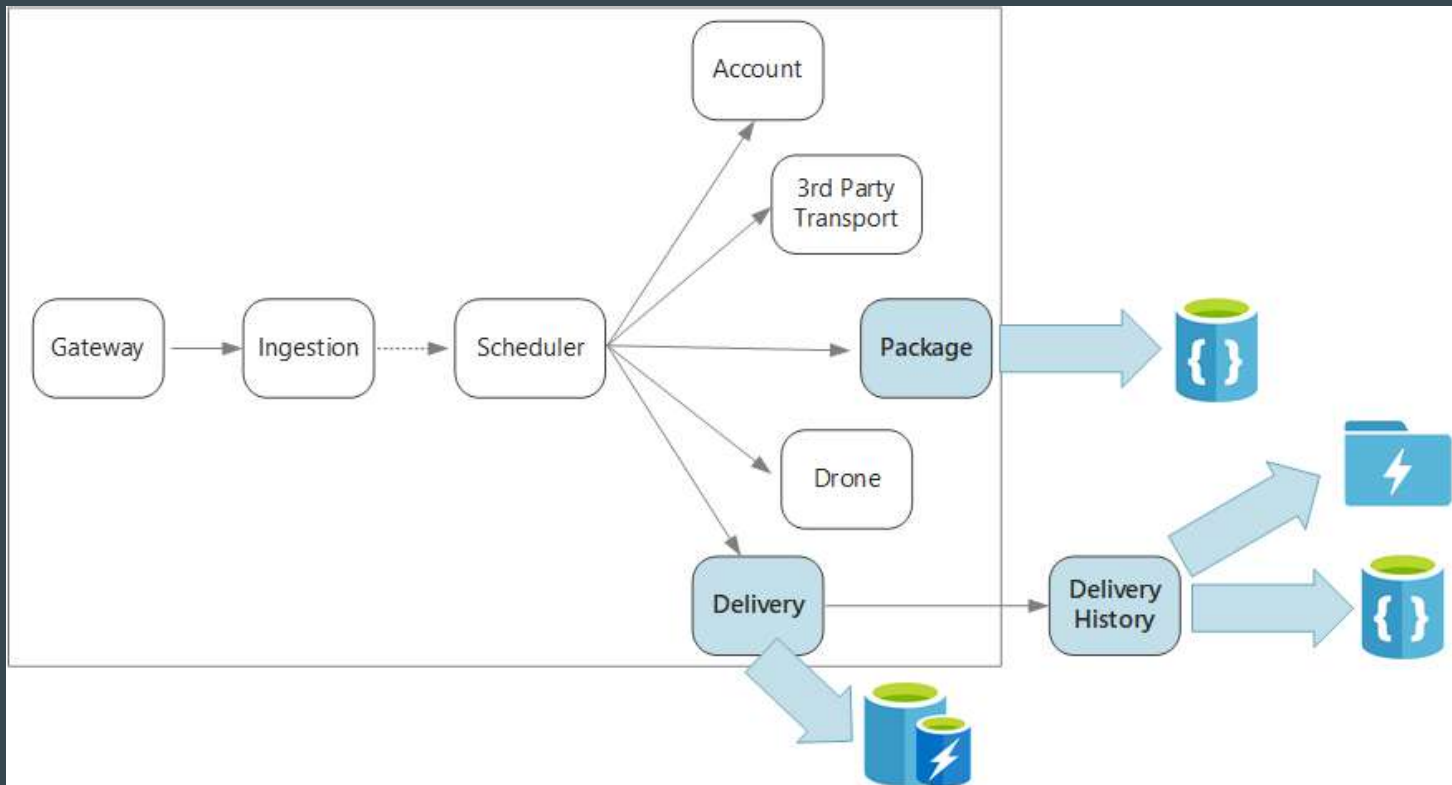
La razón de esta regla es evitar el acoplamiento involuntario entre servicios, lo que puede suceder si los servicios **comparten los mismos esquemas de datos subyacentes**. Si hay un cambio en el esquema de datos, se debe **coordinar en todos** los servicios que utilizan esa base de datos. Tener en cuenta la redundancia

IMPORTANTE

Es pasable que los servicios compartan el mismo servidor de base de datos físico. El problema se produce cuando los servicios **comparten el mismo esquema**, o leen y escriben en el **mismo conjunto** de tablas de base de datos.



MICROSERVICIOS Y PERSISTENCIA

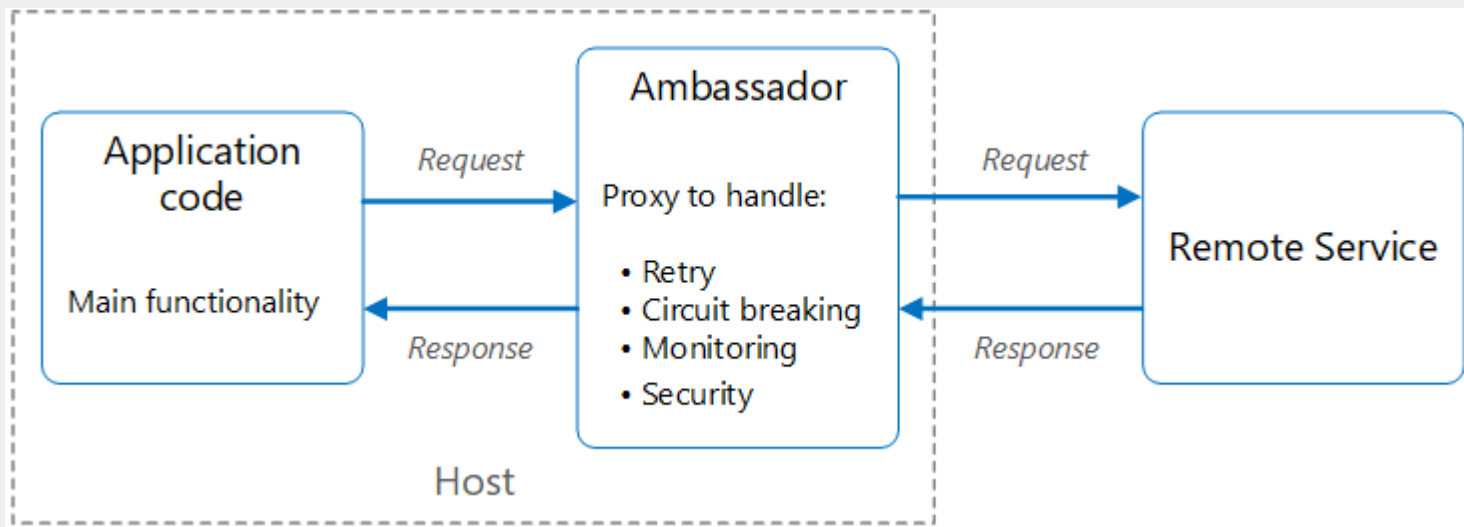




PATRONES DE DISEÑO PARA MICROSERVICIOS

AMBASSADOR

Puede usarse para descargar tareas comunes de conectividad de cliente, como la supervisión, registro, enrutamiento y seguridad (por ejemplo, TLS) en un lenguaje de forma independiente.



PROBLEMAS Y CONSIDERACIONES



El proxy agrega cierta sobrecarga de latencia. Considere si una biblioteca de cliente, que se invoca directamente por la aplicación, es un enfoque más adecuado.

Tenga en cuenta los posibles efectos de incluir características generalizadas en el proxy. Por ejemplo, Ambassador podría controlar los reintentos, pero esto podría no ser seguro (a menos que todas las operaciones sean idempotentes).

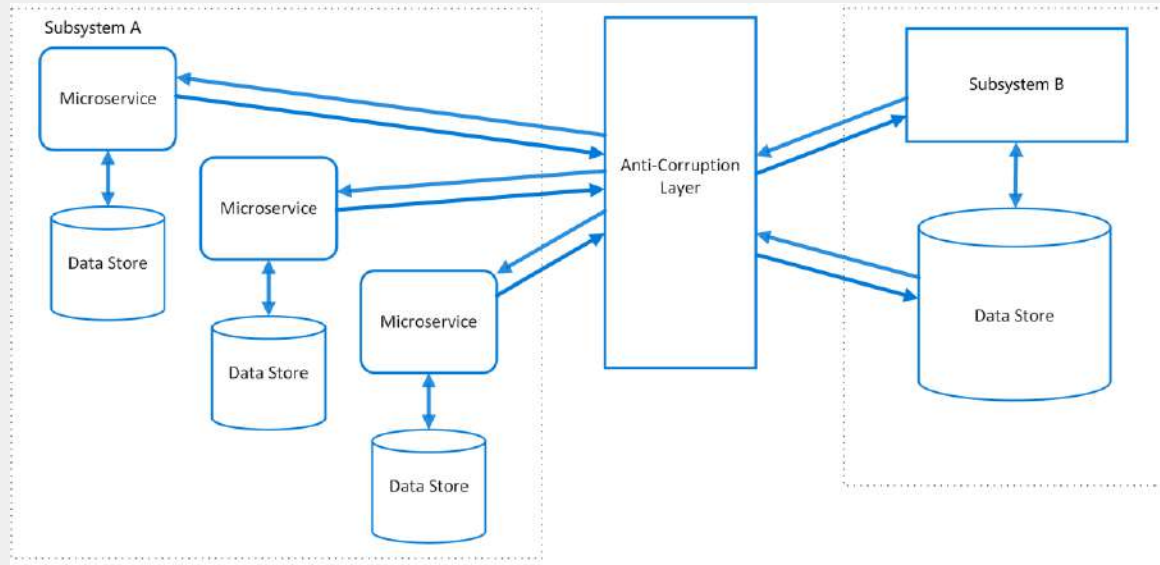
CUANDO USARLO?



- Tenga que crear un conjunto común de características de conectividad de cliente en varios lenguajes o marcos de trabajo.
- Tenga que descargar los problemas de conectividad de cliente transversales en los desarrolladores de infraestructura u otros equipos más especializados.
- Tenga que admitir los requisitos de conectividad del clúster o la nube en una aplicación heredada o una aplicación que sea difícil de modificar

CAPA ANTICORRUPCIÓN

Implementar una capa de fachada o adaptador entre los diferentes subsistemas que no comparten la misma semántica. Esta capa traduce las solicitudes que un subsistema hace al otro subsistema. Se debe usar este patrón para asegurarse de que el diseño de la aplicación no se vea limitado por dependencias de subsistemas externos.



PROBLEMAS Y CONSIDERACIONES



- La capa para evitar daños puede añadir latencia a las llamadas entre los dos sistemas.
- La capa para evitar daños añade un servicio adicional que debe administrarse y mantenerse.
- Tenga en cuenta cómo se escalará la capa para evitar daños.
- Asegúrese de mantener la coherencia de la transacción y los datos y de que se pueda supervisar.
- Si la capa anticorrupción es parte de una estrategia de migración de aplicaciones, considere si será permanente o se retirará después de migrar toda la funcionalidad heredada.

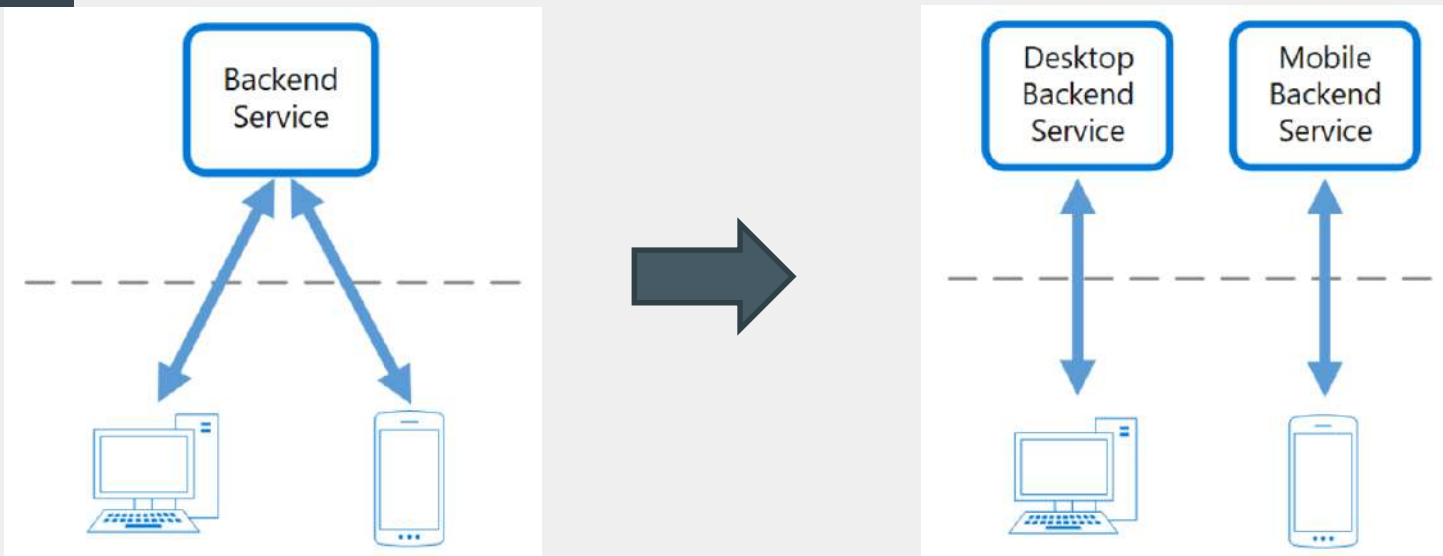
CUANDO USARLO?



- Se haya planificado una migración en varias fases, pero deba mantenerse la integración entre el sistema nuevo y el heredado.
- Dos o más subsistemas usan una semántica diferente, pero necesitan comunicarse.

PATRÓN BACKENDS FOR FRONTENDS

Crea servicios independientes de back-end que determinadas aplicaciones de front-end o interfaces puedan usar. Este patrón es útil cuando desea evitar personalizar un único back-end para varias interfaces. Sam Newman describió por primera vez este patrón.



PROBLEMAS Y CONSIDERACIONES



- Tenga en cuenta cuántos servidores back-end va a implementar.
- Si distintas interfaces (por ejemplo, clientes móviles) van a hacer las mismas solicitudes, calcule si es necesario implementar un back-end para cada interfaz o si será suficiente con un back-end único.
- Es muy probable que se produzca una duplicación de código en los servicios al implementar este patrón.
- Piense en cómo puede reflejarse este patrón en las responsabilidades de un equipo de desarrollo.
- Tenga en cuenta el tiempo que se tardará en implementar este patrón. ¿Incurrirá el esfuerzo de creación de los nuevos backend en una deuda técnica, mientras continúa apoyando el back-end genérico existente?

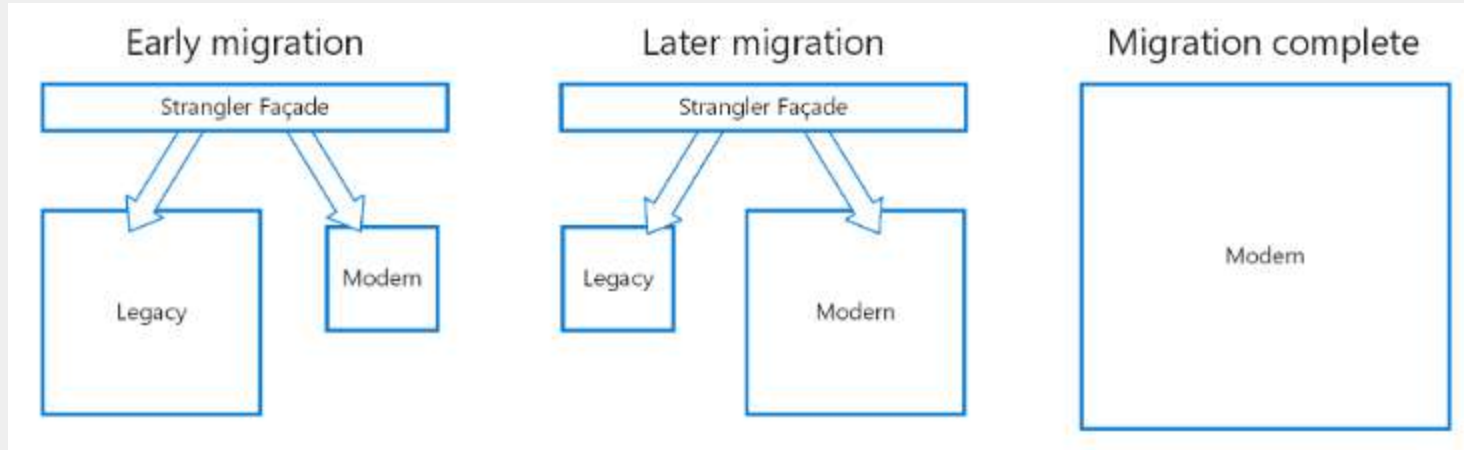
CUANDO USARLO?



- Un servicio back-end de uso general o compartido debe mantenerse con una sobrecarga de desarrollo importante.
- Desea optimizar el back-end para los requisitos de interfaces de cliente específicas.
- Las personalizaciones se realizan en un back-end de uso general para dar cabida a varias interfaces.
- Un lenguaje de programación alternativo es más adecuado para el back-end de una interfaz de usuario diferente.

PATRÓN STRANGLER

Migra de forma incremental un sistema heredado reemplazando gradualmente funciones específicas por los servicios y aplicaciones nuevas. A medida que se reemplaza el sistema heredado, el nuevo sistema sustituye eventualmente todas las características del sistema anterior, suprimiéndolo y permitiéndole su desmantelamiento.



PROBLEMAS Y CONSIDERACIONES



- Piense en cómo administrar los servicios y los almacenes de datos que potencialmente pueden utilizar tanto los sistemas nuevos como los heredados. Asegúrese de que ambos pueden tener acceso a estos recursos en paralelo.
- Estructure las nuevas aplicaciones y servicios de forma que se puedan interceptar y reemplazar fácilmente en el futuro por migraciones de Strangler.
- Asegúrese de que la fachada se mantiene al día con la migración.
- Asegúrese de que la fachada no se convierte en un único punto de error o un cuello de botella para el rendimiento.

CUANDO USARLO?

- Utilice este patrón cuando migre gradualmente una aplicación de back-end a una nueva arquitectura



PATRÓN GATEWAY OFFLOADING

Descarga una funcionalidad de servicio compartida o especializada en un proxy de puerta de enlace. Este patrón puede simplificar la implementación de la aplicación al mover la funcionalidad del servicio compartido, como el uso de certificados SSL, de otras partes de la aplicación a la puerta de enlace.

Simplifica el desarrollo de servicios mediante la eliminación de la necesidad de distribuir y mantener los recursos de compatibilidad y permite equipos expertos



PROBLEMAS Y CONSIDERACIONES



- Asegúrese de que la puerta de enlace de API es resistente a errores y de alta disponibilidad. Ejecute varias instancias de la puerta de enlace de API para evitar los puntos únicos de error.
- Asegúrese de que la puerta de enlace está diseñada para los requisitos de capacidad y escalado de la aplicación y los puntos de conexión. Asegúrese de que la puerta de enlace no se convierte en cuello de botella para la aplicación y de que es lo suficientemente escalable.
- Descargue solo las características que use toda la aplicación, como la seguridad o la transferencia de datos.
- La lógica de negocios no debe descargarse a la puerta de enlace de API.
- Si necesita realizar un seguimiento de las transacciones, considere la posibilidad de generar identificadores de correlación para fines de registro.

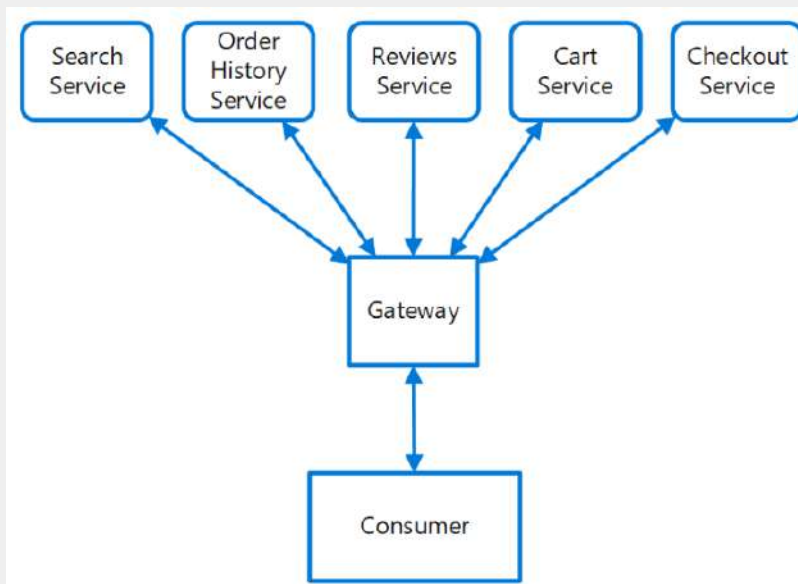
CUANDO USARLO?



- Una implementación de una aplicación tenga un problema compartido, como de los certificados SSL o el cifrado.
- Una característica común a las implementaciones de la aplicación pueda tener requisitos de recursos diferentes, como los recursos de memoria, la capacidad de almacenamiento o las conexiones de red.
- Quiera que la responsabilidad de problemas como la seguridad de red, la limitación u otros problemas de límite de red recaiga en un equipo más especializado.

PATRÓN GATEWAY ROUTING

Enruta las solicitudes a varios servicios mediante un solo punto de conexión. Este patrón es útil cuando desea exponer varios servicios en un único punto de conexión y enrutarlos al servicio adecuado en función de la solicitud.



CONTEXTO Y PROBLEMA

Cuando un cliente debe consumir varios servicios, puede resultar difícil configurar un punto de conexión diferente para cada servicio y hacer que el cliente administre cada punto de conexión. Por ejemplo, una aplicación de comercio electrónico podría proporcionar servicios como búsqueda, revisiones, carro, finalización de la compra e historial de pedidos. Cada servicio tiene una API diferente con la que el cliente debe interactuar, y el cliente debe conocer cada punto de conexión para conectarse a los servicios. Si cambia una API, también se debe actualizar el cliente. Si refactoriza un servicio en dos o más servicios independientes, el código debe cambiar en el servicio y en el cliente.

PROBLEMAS Y CONSIDERACIONES



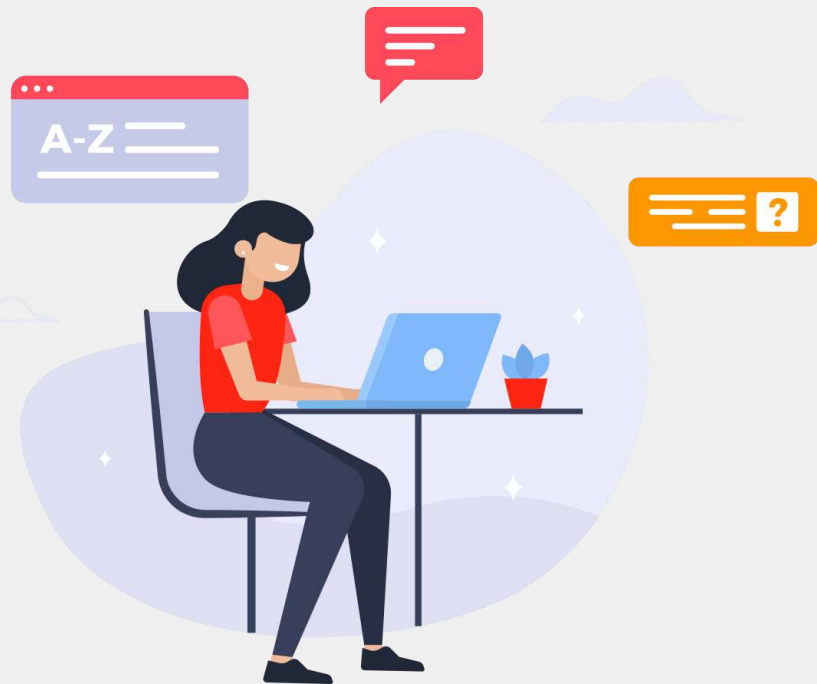
- El servicio de puerta de enlace puede introducir un único punto de error. Asegúrese de que esté diseñado correctamente para satisfacer sus necesidades de disponibilidad. A la hora de la implementación, tenga en cuenta las funcionalidades de resistencia y tolerancia a errores.
- El servicio de puerta de enlace puede introducir un cuello de botella. Asegúrese de que la puerta de enlace tenga un rendimiento adecuado para administrar la carga y que pueda escalarse fácilmente en línea con sus expectativas de crecimiento.
- Realice pruebas de carga en la puerta de enlace para asegurarse de que no introduce errores en cascada en los servicios.
- El enrutamiento de puerta de enlace es de nivel 7. Se puede basar en la dirección IP, el puerto, el encabezado o la dirección URL.

CUANDO USARLO?



- Un cliente deba usar varios servicios a los que se pueda acceder detrás de una puerta de enlace.
- Quiera simplificar las aplicaciones cliente usando un único punto de conexión.
- Necesite enrutar las solicitudes desde puntos de conexión externamente direccionables hasta puntos de conexión virtuales internos, por ejemplo, exponer los puertos de una máquina virtual a las direcciones IP virtuales del clúster.

POR APRENDER



FUNCIONAMIENTO DE MICROSERVICIOS EN PRODUCCIÓN

- Martin Fowler
- Sam Newman
- Microsoft

MIGRACIÓN A UNA ARQUITECTURA DE MICROSERVICIOS

- James Lewis
- Martin Omander, Adam Ross

PATRONES DE DISEÑO EN LA NUBE

- Microsoft
- Bill Wilder



GRACIAS!

Presentado por:

Catalina Cano — contacto: catalinacano08@Hotmail.com

Daniel Nieto — contacto: dnietog2@Gmail.com