

INICIO

Desarrollo de Microservicios con Spring Cloud Netflix OSS

ISC. Ivan Venor García Baños





+

NETFLIX
OSS

Agenda

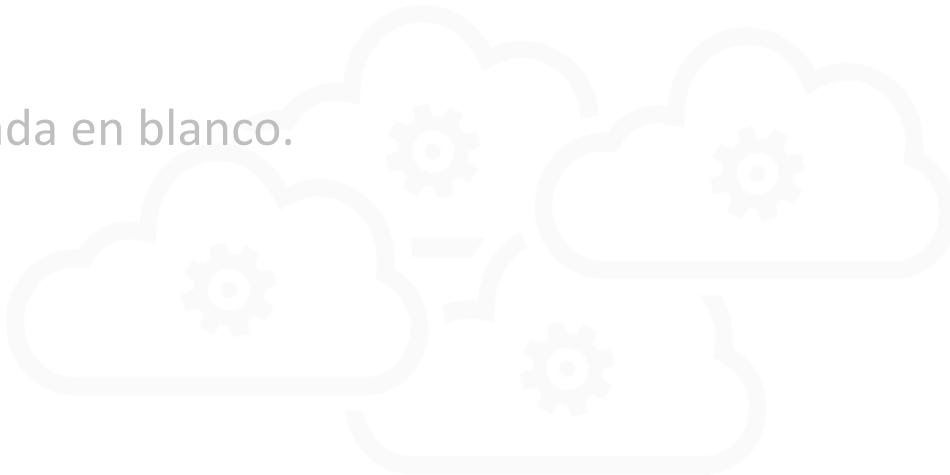
1. Presentación
2. Objetivos
3. Contenido
4. Despedida



Microservices



Esta página fue intencionalmente dejada en blanco.



Microservices



+

NETFLIX
OSS

3. Contenido

- i. Arquitectura de sistemas monolíticos
- ii. Introducción a la Arquitectura Orientada a Servicios
- iii. Fundamentos Spring Boot 2.x
- iv. Arquitectura de Microservicios
- v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS



3. Contenido

- i. Arquitectura de sistemas monolíticos
- ii. Introducción a la Arquitectura Orientada a Servicios
- iii. Fundamentos Spring Boot 2.x
- iv. **Arquitectura de Microservicios**
- v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS



+

NETFLIX
OSS

iv. Arquitectura de Microservicios



Microservices



+

iv. Arquitectura de Microservicios

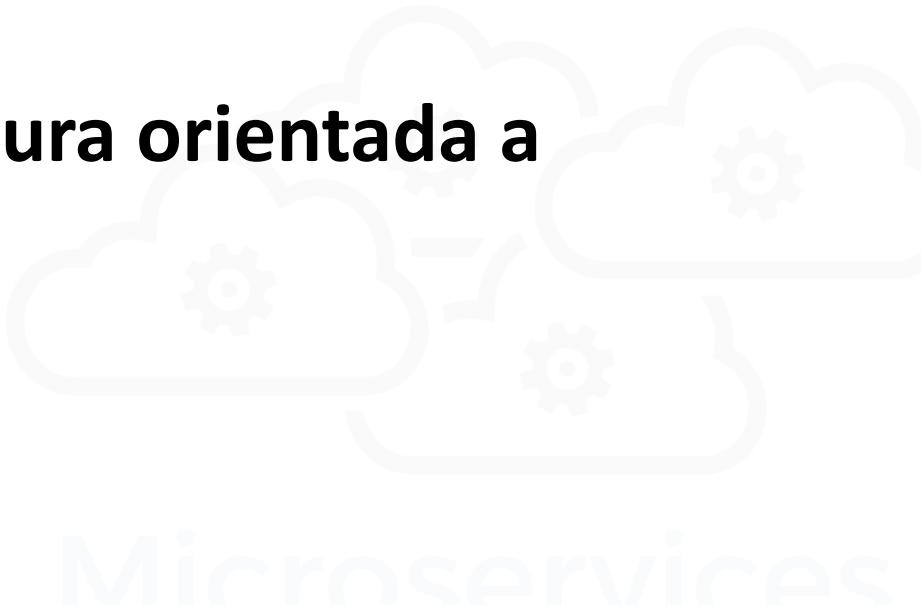
- iv.i ¿Qué es la arquitectura orientada a microservicios?**
- iv.ii Descomponiendo aplicaciones monolíticas.
- iv.iii Protocolos ligeros de comunicación para microservicios.
- iv.iv Aplicaciones “cloud-native”.
- iv.v Principios de diseño para aplicaciones en la nube.
- iv.vi Orquestación vs Coreografía.
- iv.vii Gestión de Transacciones ACID vs BASE.
- iv.viii API Manager



+

NETFLIX
OSS

iv.i ¿Qué es la arquitectura orientada a microservicios?



Microservices



+

NETFLIX
OSS

Objetivos de la lección

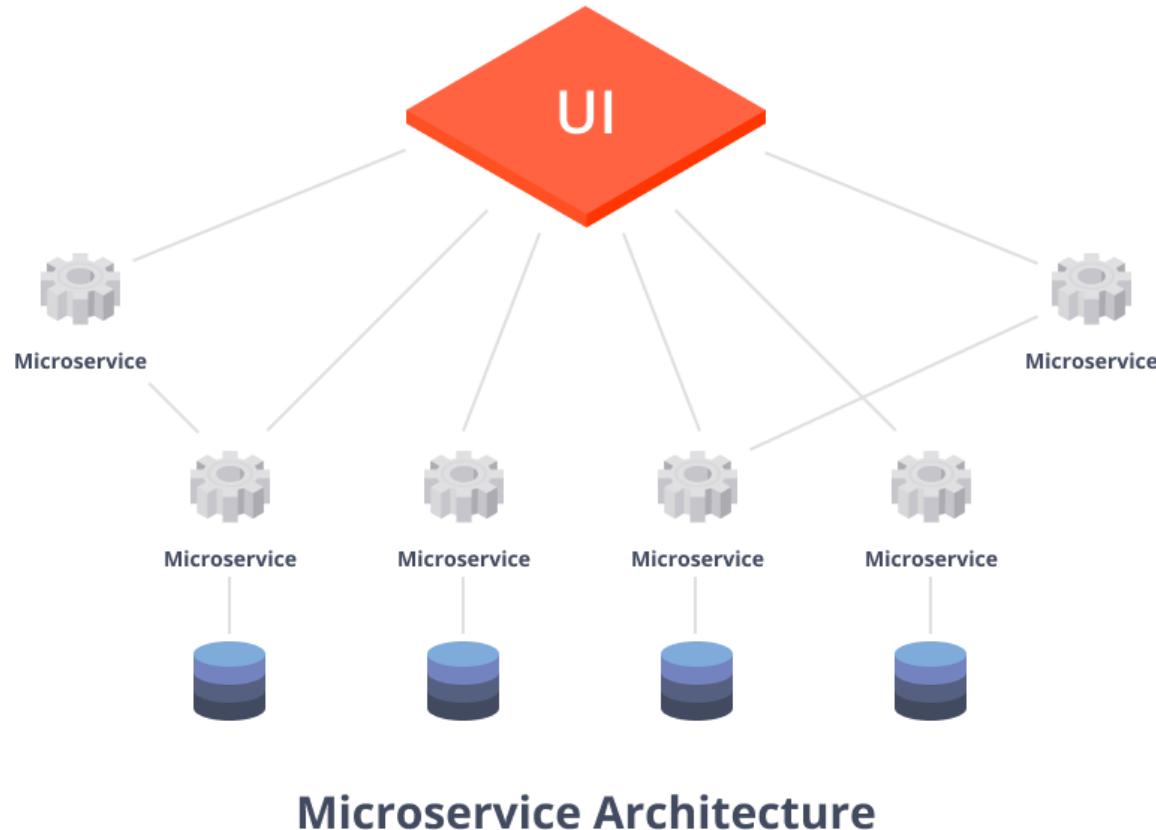
iv.i ¿Qué es la arquitectura orientada a microservicios?

- Comprender qué es una arquitectura orientada a microservicios.
- Contrastar una arquitectura tradicional o monolítica contra una arquitectura orientada a microservicios.
- Comprender las características de las arquitecturas orientadas a microservicios.
- Analizar ventajas y desventajas de las arquitecturas orientadas a microservicios.

iv.i ¿Qué es la arquitectura orientada a microservicios? (a)

- Los microservicios, o la arquitectura orientada a microservicios, es un tipo de arquitectura, que sirve para diseñar aplicaciones donde las funciones o funcionalidades del sistema están desplegadas de forma independiente a diferencia de los sistemas basados en arquitecturas tradicionales o monolíticas.
- Cada función se denomina servicio y se puede diseñar, codificar e implementar de forma independiente, permitiendo que funcionen por separado, de forma aislada, y que también fallen por separado sin afectar a los demás servicios.

iv.i ¿Qué es la arquitectura orientada a microservicios? (b)



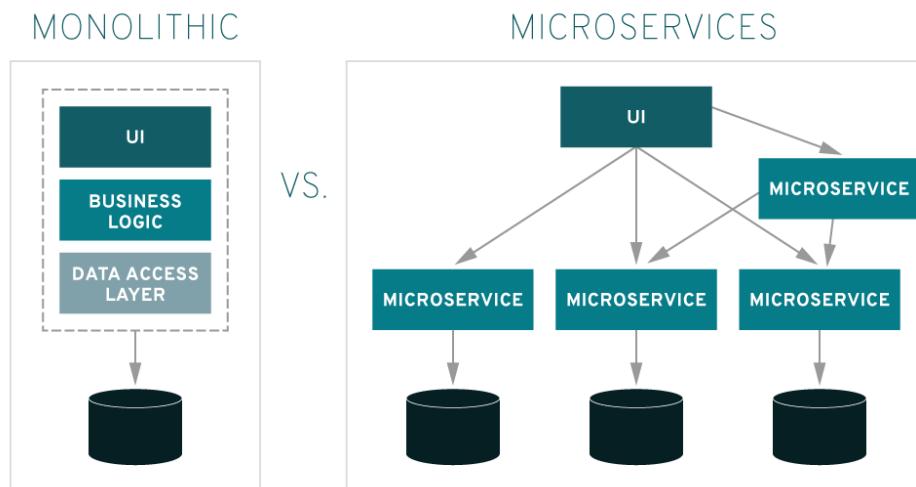
iv.i ¿Qué es la arquitectura orientada a microservicios? (c)

- Una arquitectura orientada a microservicios consta de un conjunto de servicios pequeños que hacen una sola cosa bien.
- Los microservicios son autónomos, es decir, son independientes entre sí y cada uno debe de implementar una funcionalidad de negocio individual.
- Desde un punto de vista técnico, los microservicios son la evolución natural de las arquitecturas orientadas a servicios.



iv.i ¿Qué es la arquitectura orientada a microservicios? (d)

- Las arquitecturas de microservicios consisten en desarrollar una sola aplicación como un conjunto de pequeños servicios, cada uno con su propio proceso y cada uno independiente de los demás, lo cual los hace más flexibles al cambio, escalables y robustos.





iv.i ¿Qué es la arquitectura orientada a microservicios? (e)

- Características que definen un microservicio (a):
 - Los microservicios son pequeños e independientes y se mantienen desacoplados mediante interfaces bien definidas (o acoplados de forma flexible).
 - Cada microservicio tiene un código base independiente, que puede administrarse por un equipo de desarrollo pequeño (two-pizza-rule).
 - Los microservicios pueden implantarse de manera independiente, es decir, un equipo puede actualizar un servicio existente sin tener que volver a compilar y desplegar toda la aplicación.



+

iv.i ¿Qué es la arquitectura orientada a microservicios? (f)

- Características que definen un microservicio (b):
 - Los microservicios son responsables de conservar sus propios datos o estado externo. Esto difiere del modelo tradicional, donde una capa de datos independiente controla la persistencia de los datos.
 - Los microservicios se comunican entre sí mediante protocolos conocidos, simples y APIs bien definidas, ocultando los detalles de la implementación interna de cada microservicio frente a otros microservicios.
 - No es necesario que los microservicios compartan el mismo “stack” tecnológico, bibliotecas o frameworks. Los microservicios pueden ser políglotas.



iv.i ¿Qué es la arquitectura orientada a microservicios? (g)

- Ventajas:
 - Desarrollo independiente.
 - Equipos pequeños y centrados, metodologías ágiles.
 - Despliegue independiente
 - Escalabilidad independiente.
 - Reusabilidad.
 - Aislamiento de errores
 - Stack tecnológico mixto (políglotas).
 - Facilidad de mantenimiento.
 - Facilidad de ejecución de pruebas unitarias.
 - Tolerancia a fallos, alta disponibilidad y replicación.
 - Distribución de carga, concurrencia y tiempos de respuesta.

iv.i ¿Qué es la arquitectura orientada a microservicios? (h)

- ¿Cuándo implementar una arquitectura orientada a microservicios?:
 - Aplicaciones grandes que requieran una **alta velocidad** de publicación de nuevos "**releases**" o funcionalidades.
 - Aplicaciones complejas que requieren de gran **escalabilidad y elasticidad**.
 - Aplicaciones que den soporte a negocios complejos donde se definan **múltiples dominios o sub-dominios** de negocio.
 - Organizaciones que dispongan de **pequeños equipos de trabajo**.
 - Organizaciones que dispongan de **equipos de trabajo distribuidos**.
- La arquitectura orientada a microservicios no es una "bala de plata".



iv.i ¿Qué es la arquitectura orientada a microservicios? (i)

- Desventajas (a):
 - **Complejidad:** Una aplicación de microservicios tiene más partes en movimiento que la aplicación monolítica equivalente. Cada servicio es más sencillo, pero el sistema como un todo es más complejo.
 - **Dependencias para desarrollo y pruebas:** El rápido desarrollo de aplicaciones orientadas a microservicios suponen un mayor esfuerzo en el desarrollo (y pruebas) de microservicios dependientes de otros. La refactorización en las interfaces y en los límites del servicio puede resultar catastróficos. Debido a lo anterior, el versionado de los componentes es muy importante.



iv.i ¿Qué es la arquitectura orientada a microservicios? (j)

- Desventajas (b):
 - **Falta de Gobierno:** Debido a la falta de gobernabilidad, una arquitectura basada en microservicios puede acabar con tantos lenguajes y frameworks diferentes causando que la aplicación sea difícil de mantener.
 - **Congestión y latencia de red:** Uno de los mayores problemas de las arquitecturas basadas en microservicios son las **falacias de la computación distribuida**. La comunicación entre muchos microservicios pequeños y detallados dar lugar a una mayor congestión y latencia en la red.



iv.i ¿Qué es la arquitectura orientada a microservicios? (k)

- Desventajas (c):
 - **Integridad de los datos:** Cada microservicio es responsable de la conservación de sus propios datos, como consecuencia, la coherencia de los datos puede suponer un problema (eventual consistencia).
 - **Administración:** Para tener éxito con los microservicios se necesita una cultura de DevOps consolidada debido a que el registro correlacionado entre microservicios puede resultar un desafío.



iv.i ¿Qué es la arquitectura orientada a microservicios? (I)

- Desventajas (d):
 - **Control de versiones:** Las actualizaciones de un servicio no deben interrumpir servicios que dependen del mismo. Es posible que varios microservicios se actualicen en cualquier momento, por lo tanto, sin un cuidadoso diseño entre sus interfaces y sin un adecuado versionamiento, podrían surgir problemas con la compatibilidad con versiones anteriores y/o posteriores del software.
 - **Conjunto de habilidades:** Los microservicios son sistemas muy distribuidos. Es necesario evaluar si el equipo de desarrollo tiene los conocimientos y la experiencia para desenvolverse correctamente en el desarrollo de sistemas basados en arquitectura de microservicios.

iv.i ¿Qué es la arquitectura orientada a microservicios? (m)

- Desventajas (e):
 - **Mayor complejidad para los operadores:** Para los operadores, o equipos de monitoreo, se origina una explosión de mayores procesos a administrar y monitorear debido a que pueden desplegarse decenas, cientos o miles de microservicios en ejecución donde cada uno de ellos, se ejecuta en un proceso independiente.



iv.i ¿Qué es la arquitectura orientada a microservicios? (n)

- Desventajas (f):
 - **Mayor complejidad en la delimitación de los microservicios:** La complejidad del negocio puede aparentar que, sobre el papel, los microservicios estén bien delimitados, sin embargo, mientras se va desarrollando e implementando posibles caminos alternos, se descubre que los microservicios no son tan independientes entre. Ejemplo, compartición de los mismos datos.
 - **Obviar la complejidad del estado entre microservicios:** El manejo de microservicios sin estado es efectivo, es decir que los servicios son “stateless”. El manejo de estado dificulta la escalabilidad. Los microservicios deberían de recibir como entrada todos los datos requeridos para operar.



iv.i ¿Qué es la arquitectura orientada a microservicios? (ñ)

- Desventajas (g):
 - **Transaccionabilidad:** Amplia dificultad para implementar operaciones transaccionales entre llamadas a microservicios. Supone un gran esfuerzo y ello conyeva aumentar los tiempos de respuesta de los microservicios, habilitando “cuellos de botella”. No se recomienda implementar transaccionabilidad entre microservicios, para ello se recomienda implementar servicios idempotentes o implementar “eventual consistencia”.
 - **Monolítos disfrazados, microservicios o nanoservicios:** Delimitar los microservicios es crucial. ¿Qué tan micro es un microservicio?



iv.i ¿Qué es la arquitectura orientada a microservicios? (o)

- Desventajas (h):
 - **Dificultad para el despliegue:** Dado el alto número de microservicios que puede suponer un sistema en su totalidad, la administración para el despliegue de los microservicios supone un reto. Requiere automatización y una cultura DevOps madura.
 - Falacias de la computación distribuida, entre otras ...



+

NETFLIX
OSS

iv.i ¿Qué es la arquitectura orientada a microservicios? (p)

- Falacias de la computación distribuida
 - La red es confiable.
 - La latencia es cero.
 - El ancho de banda es infinito.
 - La red es segura.
 - La topología no cambia.
 - Hay uno y sólo un administrador.
 - El costo de transporte es cero.
 - La red es homogénea





+

NETFLIX
OSS

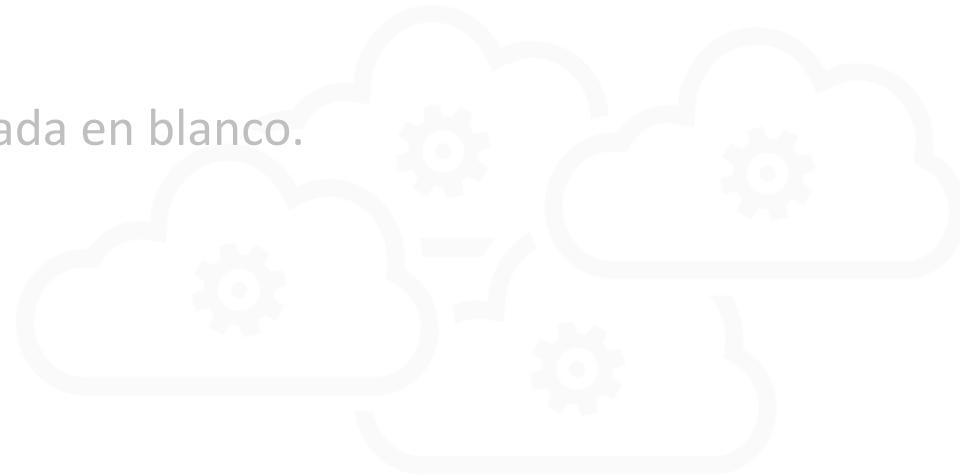
Resumen de la lección

iv.i ¿Qué es la arquitectura orientada a microservicios?

- Aprendimos qué es una arquitectura orientada a microservicios.
- Analizamos qué son los microservicios.
- Comprendimos las características que definen a los microservicios.
- Aprendimos cuáles son las ventajas y desventajas de implementar una arquitectura de microservicios y en qué casos es recomendable aplicarla.



Esta página fue intencionalmente dejada en blanco.



Microservices



+

iv. Arquitectura de Microservicios

- iv.i ¿Qué es la arquitectura orientada a microservicios?
- iv.ii Descomponiendo aplicaciones monolíticas.
- iv.iii Protocolos ligeros de comunicación para microservicios.
- iv.iv Aplicaciones “cloud-native”.
- iv.v Principios de diseño para aplicaciones en la nube.
- iv.vi Orquestación vs Coreografía.
- iv.vii Gestión de Transacciones ACID vs BASE.
- iv.viii API Manager



+

iv.ii Descomponiendo aplicaciones monolíticas.

Microservices



Objetivos de la lección

iv.ii Descomponiendo aplicaciones monolíticas

- Analizar algunos puntos a considerar para saber si es conveniente o no implementar una aplicación mediante una arquitectura orientada a microservicios.
- Comprender como implementar la migración de un sistema monolítico a una arquitectura orientada a microservicios.
- Aprender ciertas recomendaciones a la hora de descomponer aplicaciones monolíticas.
- Analizar lo que es el Domain-Driven Design y los "Bounded-context".

iv.ii Descomponiendo aplicaciones monolíticas (a)

- Hoy en día, las organizaciones ya cuentan con múltiples sistemas, monolíticos o no, escritos en diversos lenguajes de programación o normados y desarrollados bajo uno o más lenguajes de programación.
- Por lo general, las organizaciones han invertido mucho dinero en el desarrollo de sus sistemas y no pueden darse el lujo de tirar todo y volver a empezar desarrollando, nuevamente, sus sistemas orientados a microservicios.
- Sin embargo refactorizar una aplicación monolítica a una arquitectura de microservicios, puede resultar una buena idea, para disfrutar de sus beneficios y afrontar sus desventajas.

iv.ii Descomponiendo aplicaciones monolíticas (b)

- ¿Es conveniente adoptar una arquitectura orientada a microservicios en mi empresa o proyecto? Sí, si...
 - Es una aplicación grande que requiera una **alta velocidad** de publicación de nuevos “**releases**” o funcionalidades.
 - Es una aplicación compleja que requiere de gran **escalabilidad y elasticidad**.
 - Es una aplicación que da soporte a un negocio complejos donde se definen **múltiples dominios o sub-dominios** de negocio.
 - La organización dispone de **pequeños equipos de trabajo**.
 - La organización dispone de **equipos de trabajo distribuidos**.
 - La organización esta dispuesta a afrontar los desafios inherentes.



+

iv.ii Descomponiendo aplicaciones monolíticas (c)

- La refactorización de un código fuente o aplicación sugiere: “Introducir una transformación de código preservando el comportamiento existente”.
- Durante la refactorización de un monolito a microservicios se resume a mantener iguales sus APIs externas mientras se cambia la manera en la que el sistema se compila, empaqueta, despliega y opera internamente.
- No se añade nueva funcionalidad mientras se refactoriza a microservicios.

iv.ii Descomponiendo aplicaciones monolíticas (d)

- Para evaluar si una aplicación es apta para ser refactorizada a microservicios es necesario considerar:
 - ¿Cómo esta empaquetada la aplicación?
 - ¿Cómo funciona el código de la aplicación?
 - ¿Con qué tipo de arquitectura está implementada la aplicación?
 - ¿Se cuenta con la arquitectura, “blueprints”, de la aplicación?
 - ¿Cómo están definidos los orígenes de datos de la aplicación?
 - ¿Cómo estan estructurados los datos persistidos en la aplicación?

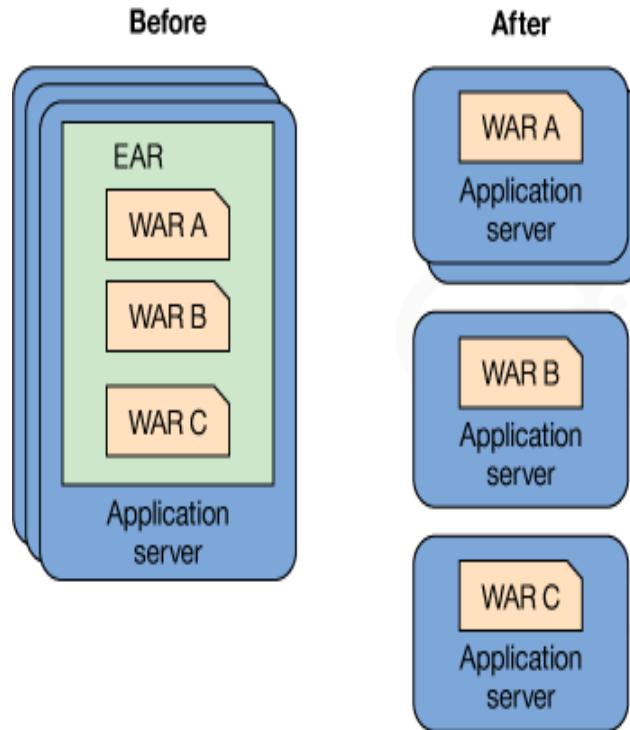


iv.ii Descomponiendo aplicaciones monolíticas (e)

- ¿Cómo re-empaquetar la aplicación? (aplicaciones Java)
- **Analizar su diagrama de despliegue.**
- **Revisar la estructura de la aplicación.**
- **Dividir archivos EAR:** En lugar de empaquetar en un EAR todos los WAR y/o JARs requeridos, dividir en archivos WAR independientes, lo cual causaría cambios en el código.
- **Implementar contenedores por servicio:** Desplegar cada WAR en un contenedor de Servlets independiente.
- **Crear, desplegar y gestionar de forma independiente.**

iv.ii Descomponiendo aplicaciones monolíticas (f)

- Re-empaquetado y re-despliegue de la aplicación.

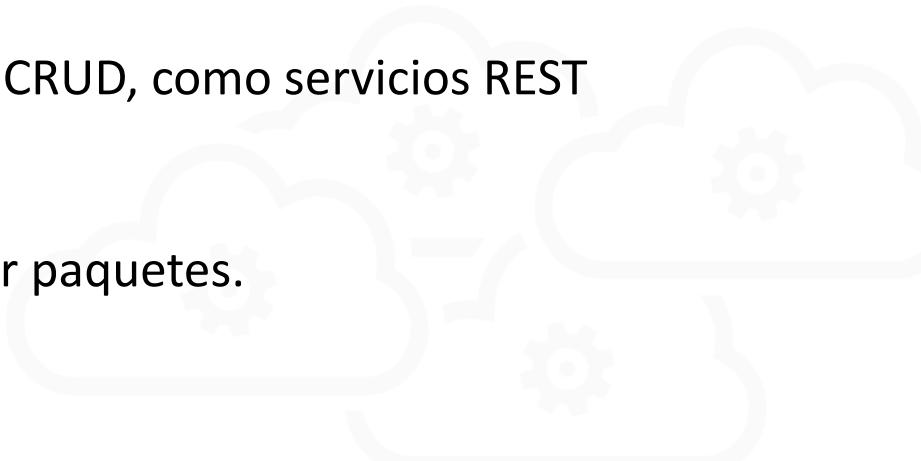


iv.ii Descomponiendo aplicaciones monolíticas (g)

- Refactorizar código.
- Debido a que el aplicativo ha sido desplegado en diferentes archivos WAR y en diferentes contenedores, las interfaces de comunicación entre ellos deberán refactorizarse también.
- Posiblemente muchas llamadas entre servicios se hacían en forma de procesos, es decir, mediante una simple llamada desde un objeto “caller” a un “worker”, ahora deberá hacerlas mediante protocolos de comunicación síncronos o asíncronos entre distintos microservicios.

iv.ii Descomponiendo aplicaciones monolíticas (h)

- Refactorizar código.
- Exponer operaciones simples, tipo CRUD, como servicios REST orientados a dominio.
- Distribución de funcionalidades por paquetes.



Microservices



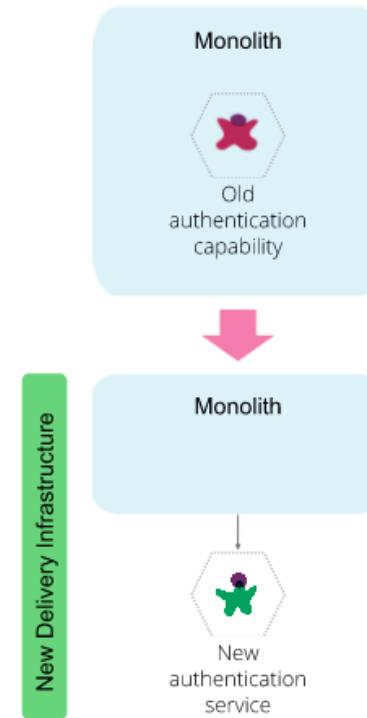
+

iv.ii Descomponiendo aplicaciones monolíticas (i)

- Refactorizar los datos y su persistencia.
- Analizar como refactorizar los datos.
- Verificar como se realizan las búsquedas en la aplicación:
 - Si se realizan mediante claves primarias, posiblemente migrar a una Base de Datos NoSQL, resulta bien y con mejor performance a una Base de Datos relacional.
 - Si se realizan búsquedas complejas en base a múltiples uniones entre tablas, una Base de Datos SQL puede seguir resultando bien.
- Comprender apliamente como se distribuyen los datos entre tablas y/o colecciones, dado que serán particionadas.

iv.ii Descomponiendo aplicaciones monolíticas (j)

- Recomendaciones.
- Iniciar con una capacidad del sistema simple y bastante desacoplada.



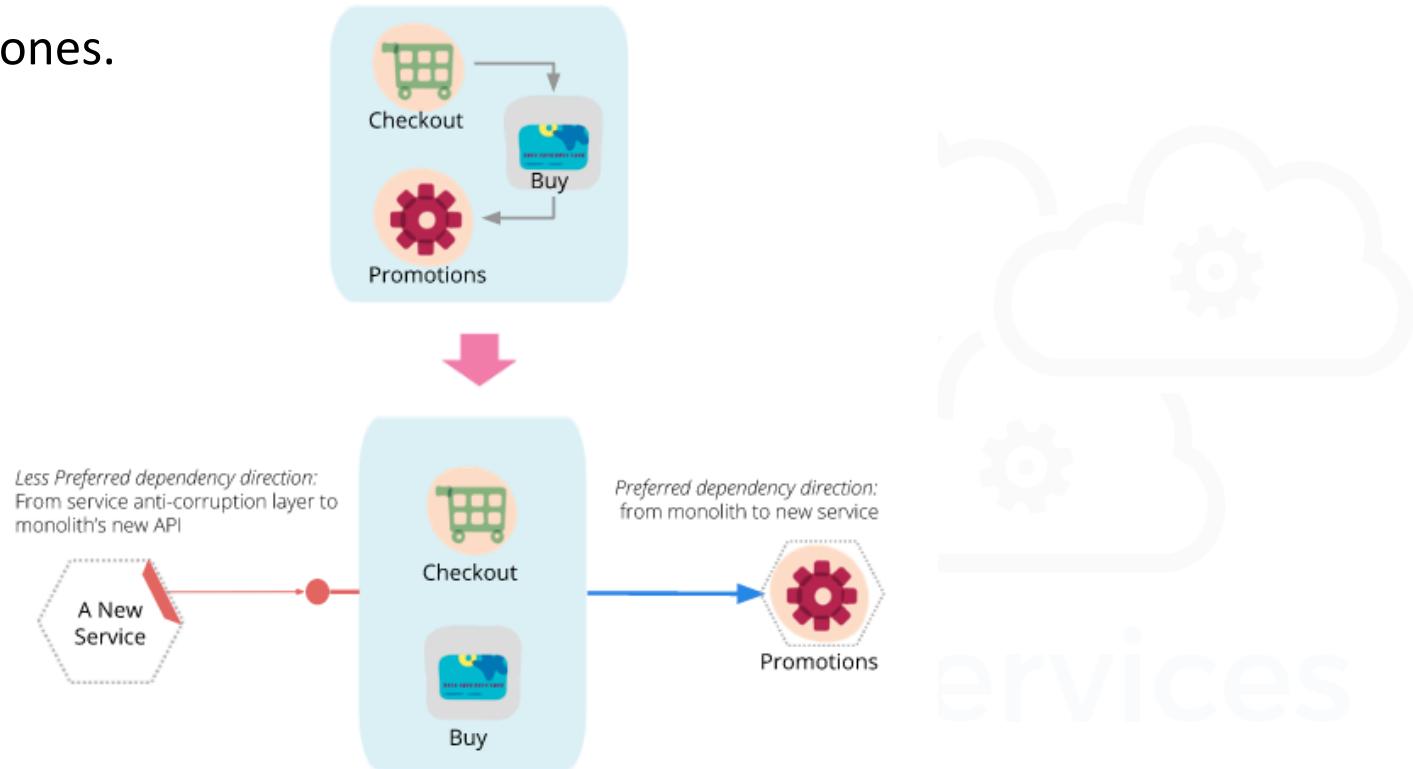


iv.ii Descomponiendo aplicaciones monolíticas (k)

- Recomendaciones.
- Minimizar la dependencia de los microservicios hacia el monolito y tratar de aumentar la dependencia del monolito hacia los microservicios, ello ocasionará continuar desacoplando los componentes dependientes hacia los microservicios, creando nuevos microservicios.
- Desacoplar las funcionalidades acopladas mediante interfaces.

iv.ii Descomponiendo aplicaciones monolíticas (I)

- Recomendaciones.





+

NETFLIX
OSS

iv.ii Descomponiendo aplicaciones monolíticas (m)

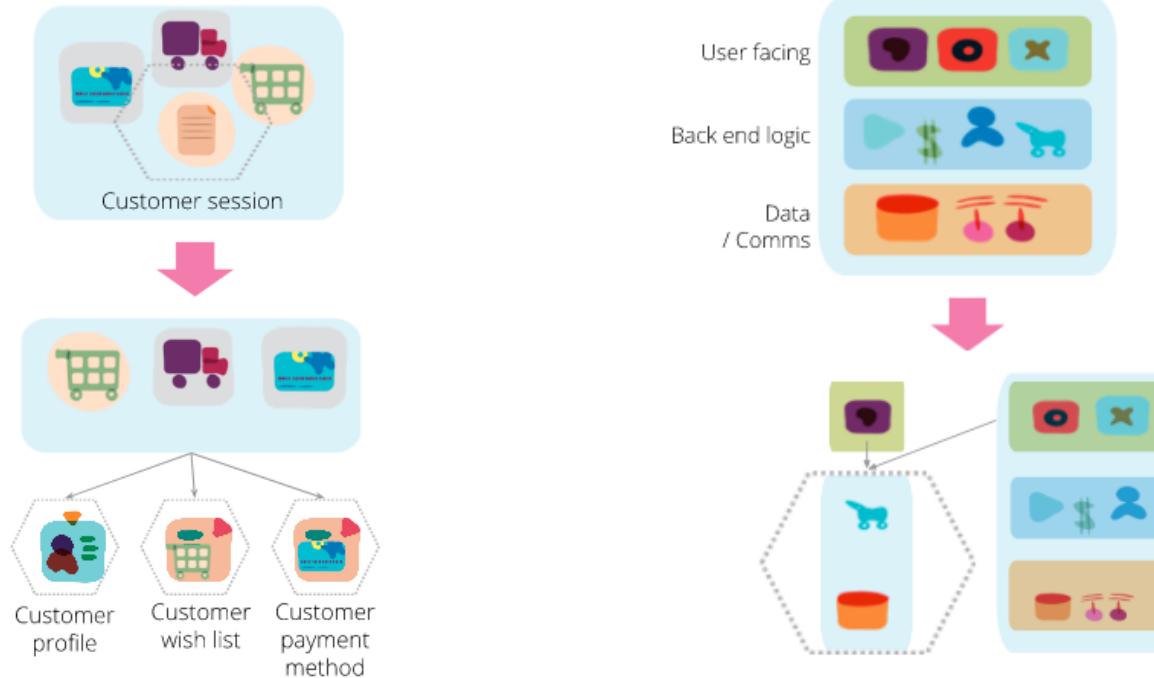
- Recomendaciones.
- Implementar cache distribuido en lugar de sesiones (web).
- Desacoplar grupos de funcionalidades por contexto de negocio (bounded-context).
- Desacoplar funcionalidades sin re-escribir código, es viable refactorizar utilizando interfaces, pero sin re-implementar lógica de negocio actualmente implementadas.
- Mantener clases cohesivas.

iv.ii Descomponiendo aplicaciones monolíticas (n)

- Recomendaciones.
- Refactorizar funcionalidad a nivel macro y luego a micro, empezando primero con pocos microservicios y después segregando a más microservicios definidos por un contexto delimitado (bounded-context).
- Refactorizar a pasos evolutivos e iterativos, es decir, refactorice la aplicación a nivel macro en su totalidad y luego a micro en su totalidad.

iv.ii Descomponiendo aplicaciones monolíticas (ñ)

- Recomendaciones.



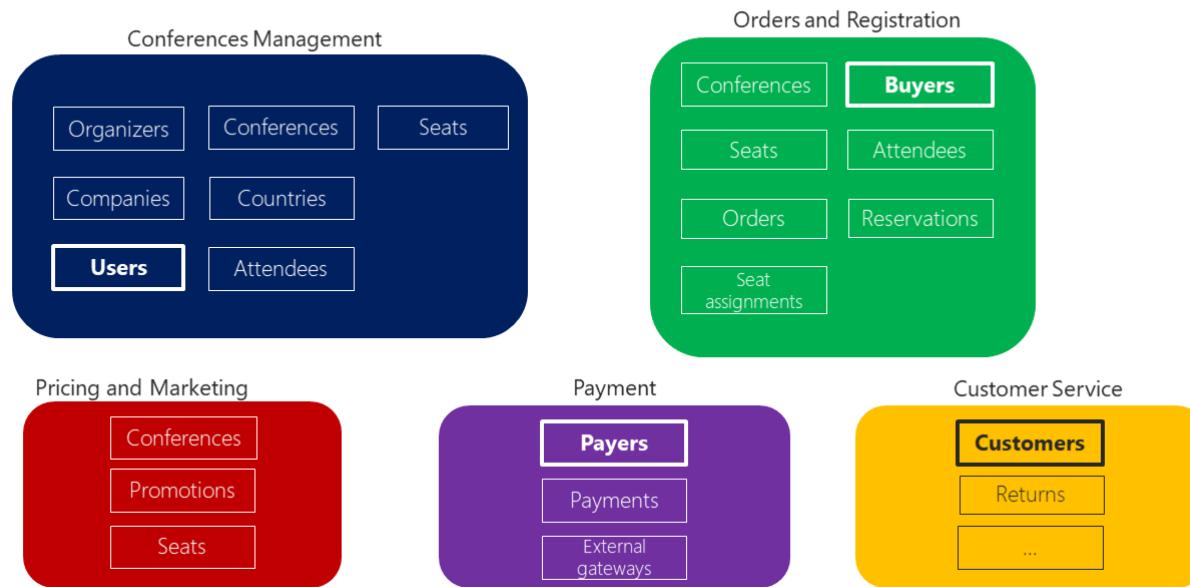
iv.ii Descomponiendo aplicaciones monolíticas (o)

- Domain Driven Design (Diseño Guiado por el Dominio).
- El DDD propone un modelado de objetos basado en la realidad de negocio con relación a sus casos de uso.
- En el contexto del desarrollo de aplicaciones, DDD hace referencia a los problemas como dominios y describe áreas del negocio independientes como contextos delimitados (cada contexto delimitado está correlacionado con un microservicio) resultando en un lenguaje común para hablar del negocio y de las clases o implementaciones que dan soporte tecnológico al negocio.

iv.ii Descomponiendo aplicaciones monolíticas (p)

- Domain Driven Design (Diseño Guiado por el Dominio).

Identifying a Domain Model per Microservice or Bounded Context





+

iv.ii Descomponiendo aplicaciones monolíticas (q)

- Domain Driven Design (Diseño Guiado por el Dominio).
- Por lo regular los patrones y técnicas del DDD se perciben como obstáculos al delimitar contextos para la implementación de los microservicios, pero los patrones y las técnicas no son lo importante, sino organizar el código para que esté en línea con los problemas de negocio y utilizar los mismos términos empresariales a nivel de código (lenguaje ubicuo).
- Lenguaje ubicuo: Lenguaje común entre programadores/técnicos y usuarios o negocio.

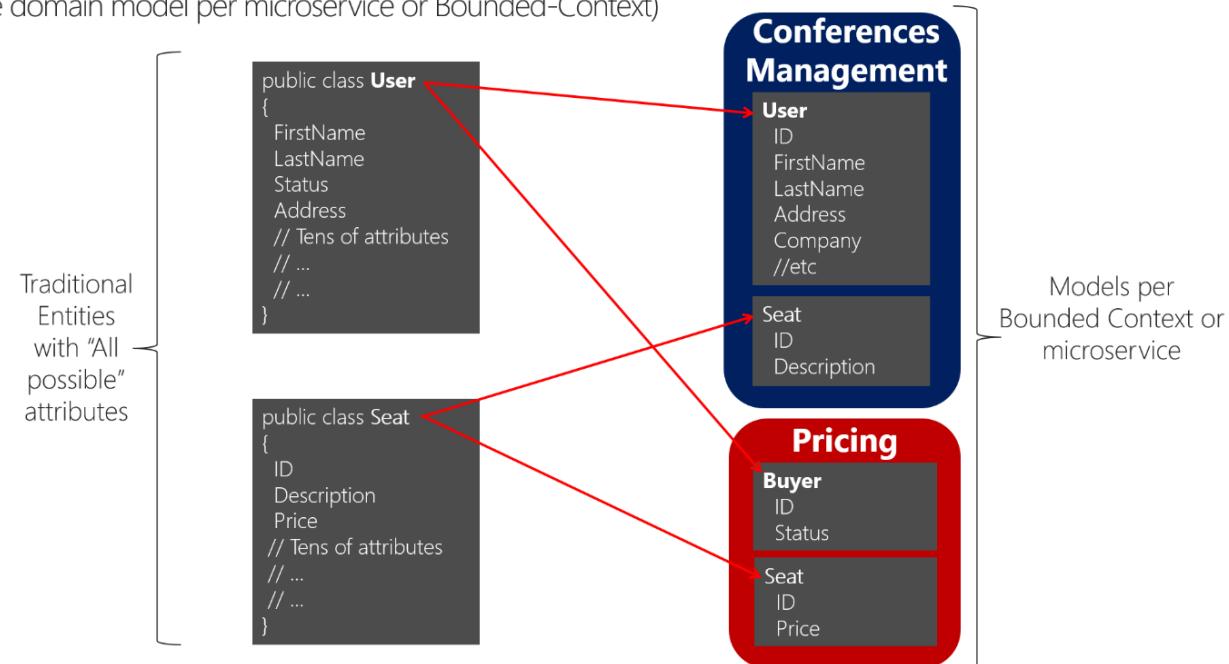
iv.ii Descomponiendo aplicaciones monolíticas (r)

- Domain Driven Design (Diseño Guiado por el Dominio).
- La clave para una correcta delimitación de contextos para la implementación de microservicios esta en situar los límites en el contexto de negocio y trasladar ese contexto y sus problemas a resolver a un microservicio.
- El DDD afecta a los límites en el contexto de negocio y por tanto afecta a los límites en la implementación de microservicios, es muy importante delimitar los contextos.

iv.ii Descomponiendo aplicaciones monolíticas (s)

- Domain Driven Design (Diseño Guiado por el Dominio).

Decomposing a traditional data model into multiple domain models
(One domain model per microservice or Bounded-Context)





+

NETFLIX
OSS

iv.ii Descomponiendo aplicaciones monolíticas (t)

- “**Bounded-context**” estrechos. Delimitar contextos relativamente estrechos.
- Determinar dónde colocar los límites entre contextos delimitados contrapone dos objetivos.
 1. Es importante delimitar los microservicios lo más pequeños posibles aunque el principal objetivo no es que sean pequeños, sino que sean delimitados alrededor de un contexto de negocio y que sean coherentes.
 2. Evitar un alto volumen de intercomunicación entre microservicios, lo cual ocasionará:
 - Saturación en la red.
 - Dado un nula comunicación entre microservicios, puede originar microservicios-monolíticos.



+

NETFLIX
OSS

iv.ii Descomponiendo aplicaciones monolíticas (u)

- “**Bounded-context**” estrechos. Delimitar contextos relativamente estrechos.
- La cohesión, no la sencillez, ni pequeñez, ni la atomicidad del microservicio, es la única clave para delimitar un contexto de negocio.
- Si dos microservicios requieren colaborar mucho entre si, son altamente dependientes, posiblemente ambos tengan que ser un mismo microservicio.
- Si un microservicio debe depender de otro servicio para satisfacer directamente una solicitud, entonces no es realmente autónomo.



iv.ii Descomponiendo aplicaciones monolíticas (v)

- Práctica 8. Descomponiendo ACME HR System
- Analiza la aplicación Acme-HR-System.
- Descompón la aplicación Acme-HR-System en dos microservicios: **8-Employee-Acme-HR-Microservice** y **8-Workstation-Acme-HR-Microservice** cumpliendo con las mismas (casi) interfaces REST de la aplicación Acme-HR-System.



+

NETFLIX
OSS

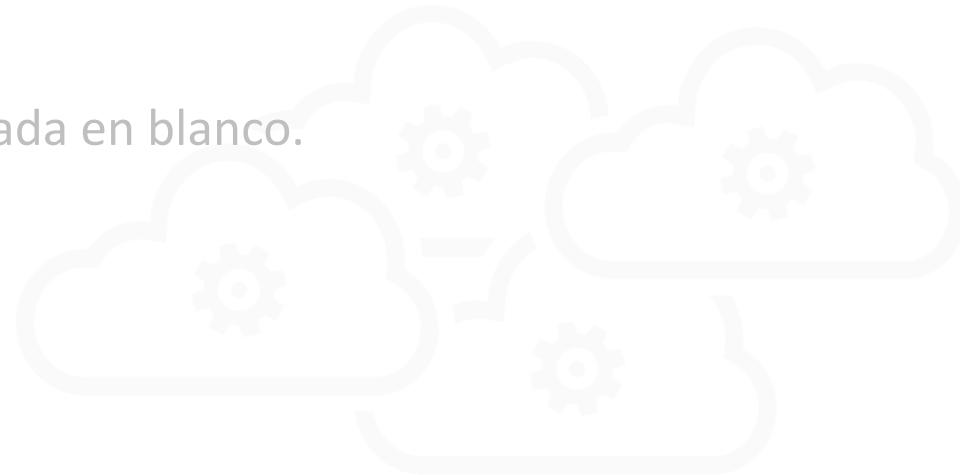
Resumen de la lección

iv.ii Descomponiendo aplicaciones monolíticas

- Comprendemos los puntos a observar para migrar una aplicación monolítica a una orientada a microservicios.
- Aprendimos las recomendaciones a la hora de descomponer aplicaciones monolíticas.
- Verificamos lo que es el “**Domain-Driven Design**” y los “**Bounded-context**”.



Esta página fue intencionalmente dejada en blanco.



Microservices



+

NETFLIX
OSS

iv. Arquitectura de Microservicios

- iv.i ¿Qué es la arquitectura orientada a microservicios?
- iv.ii Descomponiendo aplicaciones monolíticas.
- iv.iii **Protocolos ligeros de comunicación para microservicios.**
- iv.iv Aplicaciones “cloud-native”.
- iv.v Principios de diseño para aplicaciones en la nube.
- iv.vi Orquestación vs Coreografía.
- iv.vii Gestión de Transacciones ACID vs BASE.
- iv.viii API Manager



+

NETFLIX
OSS

iv.iii Protocolos ligeros de comunicación para microservicios.



Objetivos de la lección

iv.iii Protocolos ligeros de comunicación para microservicios

- Analizaremos los distintos mecanismos de comunicación entre microservicios.
- Comprenderemos las diferencias entre comunicación síncrona y asíncrona.
- Comprenderemos las ventajas y desventajas de cada protocolo de comunicación.
- Implementaremos comunicación asíncrona mediante broker de mensajes ActiveMQ.

iv.iii Protocolos ligeros de comunicación para microservicios (a)

- En una aplicación monolítica, las comunicaciones entre componentes se ejecutan en un único proceso, no necesariamente en un mismo hilo de ejecución, donde los componentes se invocan entre sí mediante llamadas de funciones (o llamadas a métodos) a nivel de lenguaje de forma acoplada o desacoplada.
- Lo más complicado al refactorizar una aplicación monolítica a microservicios es cambiar el mecanismo de comunicación entre los componentes mediante comunicación entre procesos (Inter Process Communication, IPC).



iv.iii Protocolos ligeros de comunicación para microservicios (b)

- En una arquitectura de microservicios, es preferible disminuir la cantidad de comunicaciones que tendrá un microservicio con respecto de los demás.
- No existe una única solución, para evitar las comunicaciones, una posible solución puede ser delimitar los contextos de negocio lo más aislados posibles, de tal grado que no exista comunicación entre microservicios.
- Una aplicación basada en microservicios es un sistema distribuido el cual se ejecuta en varios procesos o servicios e incluso en diferentes servidores, físicos o virtuales donde, cada instancia de un microservicio es un proceso independiente.

iv.iii Protocolos ligeros de comunicación para microservicios (c)

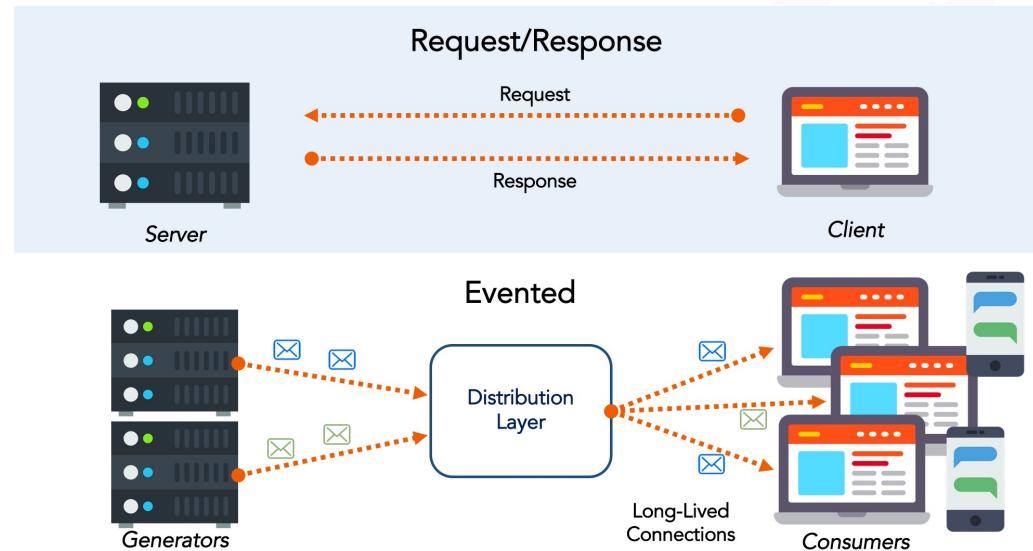
- Dado que cada instancia de un microservicio es un proceso independiente, debe implementar un protocolo de comunicación entre procesos como son HTTP, AMQP, gRPC o TCP/IP, dependiendo de la función de cada microservicio.
- Una sana implementación de microservicios promueve que la intercomunicación entre microservicios sea mediante “**smart endpoints and dumb pipes**” (Puntos de conexión inteligentes y tuberías tontas) fomentando así un diseño desacoplado entre microservicios.

iv.iii Protocolos ligeros de comunicación para microservicios (d)

- En una correcta implementación de microservicios, cada microservicio es responsable de sus propios datos y de implementar su lógica de negocio dado su delimitación de contexto sin embargo, las aplicaciones basadas en microservicios, partiendo de un caso de uso “**end-to-end**”, establecen comunicaciones entre microservicios y prevalecen los protocolos ligeros tales como:
 - HTTP mediante REST en lugar de protocolos complejos como WS-* o SOAP.
 - JMS, AMQP o comunicaciones guiadas por eventos en lugar de orquestadores de procesos de negocio centralizados como ESB.
 - gRPC o “Google open-source remote procedure call”, para llamadas internas entre microservicios.

iv.iii Protocolos ligeros de comunicación para microservicios (e)

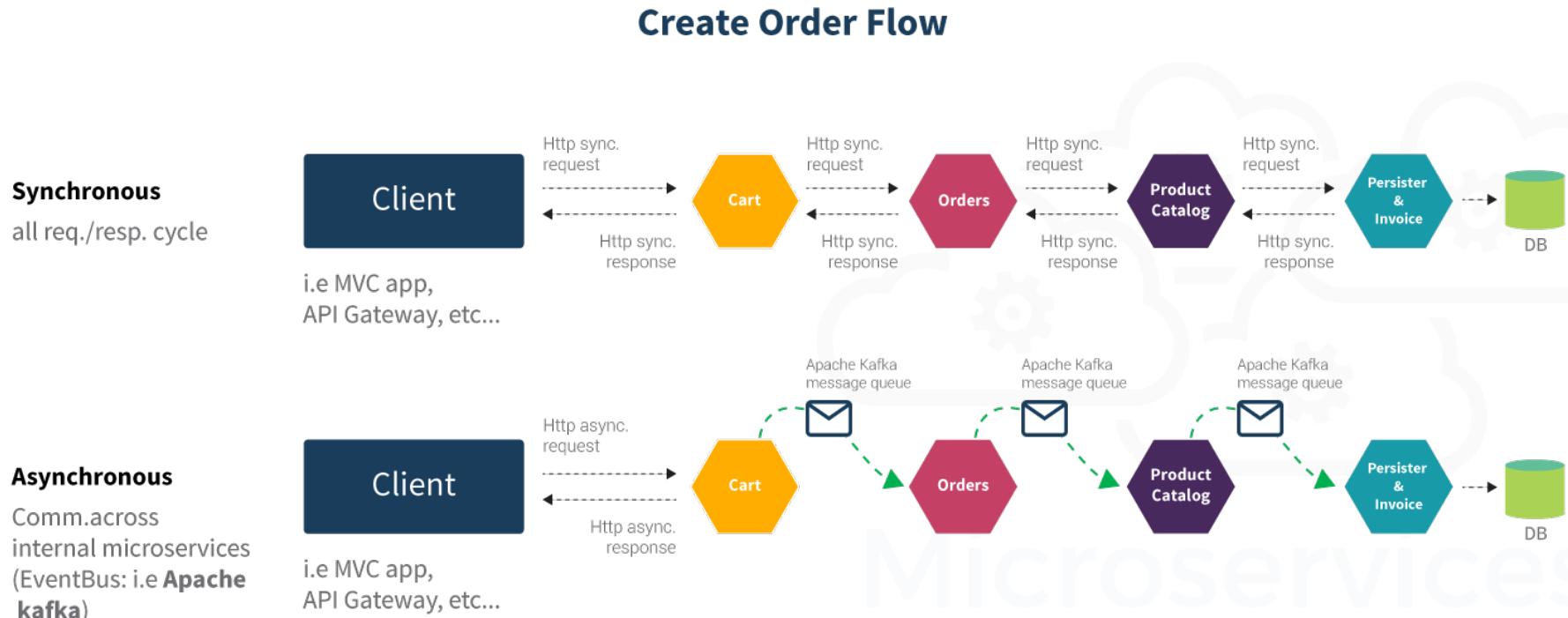
- Habitualmente se sugieren el protocolo HTTP para situaciones "**request-response**" mediante APIs REST y mensajería asíncrona ligera para comunicación de tipo "**fire-and-forget**".



iv.iii Protocolos ligeros de comunicación para microservicios (f)

- La comunicación entre microservicios se divide en dos ejes principales:
 1. Comunicación síncrona o asíncrona.
 - **Protocolos síncronos:** HTTP / HTTPS.
 - **Protocolos asíncronos:** JMS (TCP/IP), AMQP, entre otros.
 2. Comunicación a un único receptor o a varios receptores.
 - **Único receptor:** Colas de mensajes o “**queues**” que implementan el patrón “**Producer/Consumer**” y balanceo de carga (es posible que existan múltiples “**listeners**” de la “**queue**” pero sólo uno recibe el mensaje).
 - **Varios receptores:** La comunicación entre varios receptores debe ser asíncrona, Topicos o “**topics**” que implementan el patrón “**Publish/Subscribe**” o arquitectura controlada por eventos.

iv.iii Protocolos ligeros de comunicación para microservicios (g)





iv.iii Protocolos ligeros de comunicación para microservicios (h)

- Una aplicación basada en arquitectura orientada a servicios acostumbra utilizar una combinación entre los tipos de comunicación síncrona y asíncrona.
- Lo más común en la comunicación desde la aplicación cliente, consumidora de un microservicio es mediante protocolo síncrono como HTTP/HTTPS, mientras que, internamente, los microservicios se comunican entre sí de forma asíncrona, lo cual habilita su independencia, obligando su autonomía, facilidad de escalabilidad, reusabilidad, tolerancia a fallos y disponibilidad.



+

iv.iii Protocolos ligeros de comunicación para microservicios (i)

- Ventajas de comunicación síncrona:
 - Baja complejidad en el diseño.
 - Facilidad para el manejo de errores.
 - Recepción de respuestas en tiempo real (on-the-go).
- Desventajas de comunicación síncrona:
 - El servicio debe estar disponible todo el tiempo, si el servicio no está disponible, el hilo “caller” puede bloquearse por un tiempo, hasta que ocurra un error por “time-out”, causando problemas de performance.
 - Posibilidad de propagación no controlada de errores de comunicación.
 - Respuestas lentas debido a que los servicios deben esperar a que termine de recibir la respuesta de los demás servicios involucrados.
 - Necesidad de un protocolo orientado a conexión (TCP).

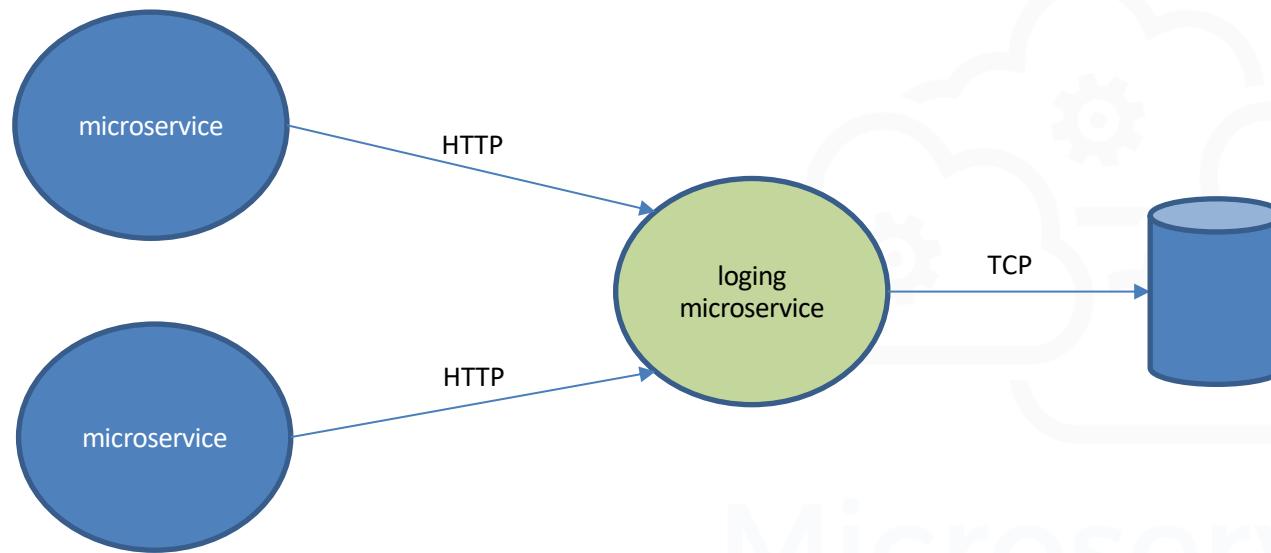


iv.iii Protocolos ligeros de comunicación para microservicios (j)

- Autonomía de microservicios (a).
- Tal como se ha mencionado, el punto más crítico de una aplicación basada en microservicios es como integrar/comunicar los microservicios entre sí.
- Idealmente es preferible evitar las comunicaciones entre si, utilizar código reusable (librerías) en lugar de depender de llamadas a microservicios de uso general.

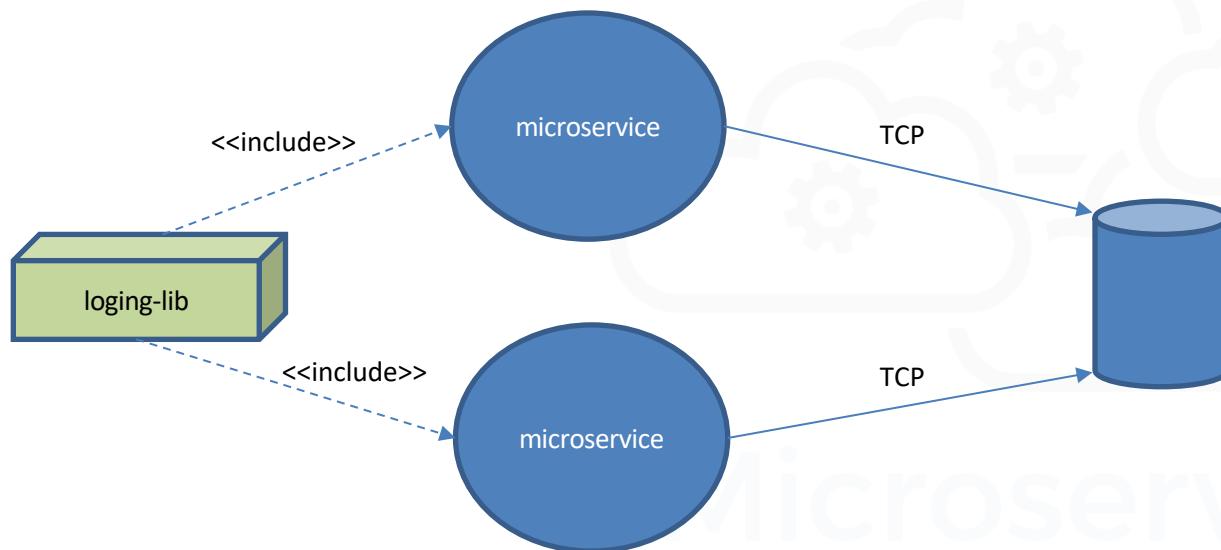
iv.iii Protocolos ligeros de comunicación para microservicios (k)

- Autonomía de microservicios (b).



iv.iii Protocolos ligeros de comunicación para microservicios (I)

- Autonomía de microservicios (c).



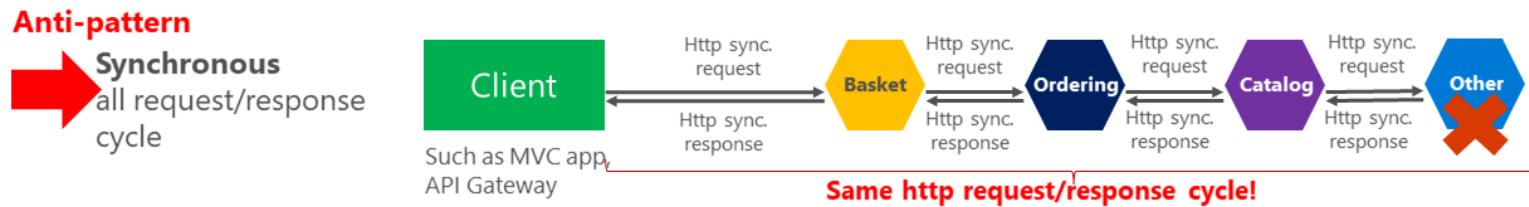
iv.iii Protocolos ligeros de comunicación para microservicios (m)

- Autonomía de microservicios (d).
- Cuando sea necesaria la comunicación entre microservicios, dicha comunicación, como regla fundamental debe ser asíncrona. Siendo posible jamás depender de una comunicación síncrona.
- La comunicación síncrona entre microservicios promueve una alta dependencia entre microservicios, lo cual evita su independencia y autonomía, indistintamente de que el despliegue de los microservicios sea autónomo e independiente.

iv.iii Protocolos ligeros de comunicación para microservicios (n)

- Autonomía de microservicios (e).

Comunicación síncrona vs asíncrona entre microservicios



iv.iii Protocolos ligeros de comunicación para microservicios (ñ)

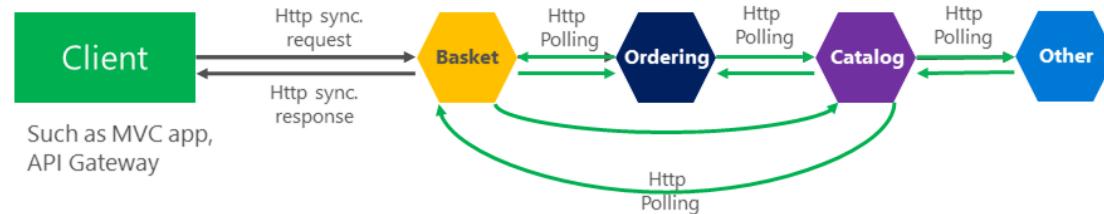
- Autonomía de microservicios (f).

Comunicación síncrona vs asíncrona entre microservicios

Asynchronous
Comm. across internal
microservices
(EventBus: like **AMQP**)



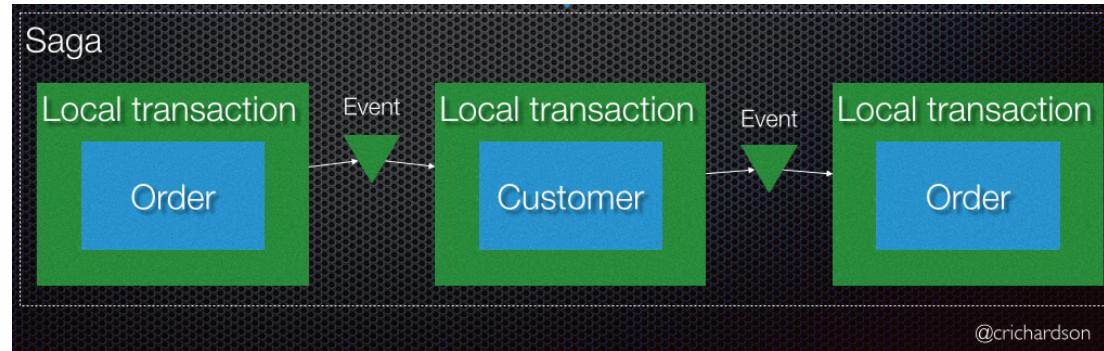
"Asynchronous"
Comm. across
internal microservices
(Polling: **Http**)



iv.iii Protocolos ligeros de comunicación para microservicios (o)

- Autonomía de microservicios (g).
- En caso que un microservicio deba accionar alguna operación en otro microservicio, como por ejemplo, una actualización de datos, ejecute dicha acción orientada a eventos (o implementando **Saga Pattern**), es decir, de forma asíncrona.

iv.iii Protocolos ligeros de comunicación para microservicios (p)





iv.iii Protocolos ligeros de comunicación para microservicios (q)

- Ventajas de comunicación asíncrona:
 - Sin necesidad de protocolos orientados a conexión ya que la información viaja mediante “**brokers**” de mensajes.
 - Sin necesidad de esperar una respuesta del lado del consumidor, no hay problemas de performance.
 - El productor (quien envía el mensaje) no necesita saber quien o cuantos son los consumidores (quien recibe el mensaje) del mensaje. Existe un bajo acoplamiento entre servicios.
 - No es necesaria una alta disponibilidad del servicio consumidor.
 - La comunicación asíncrona mejora la tolerancia a fallos, aísla los fallos.



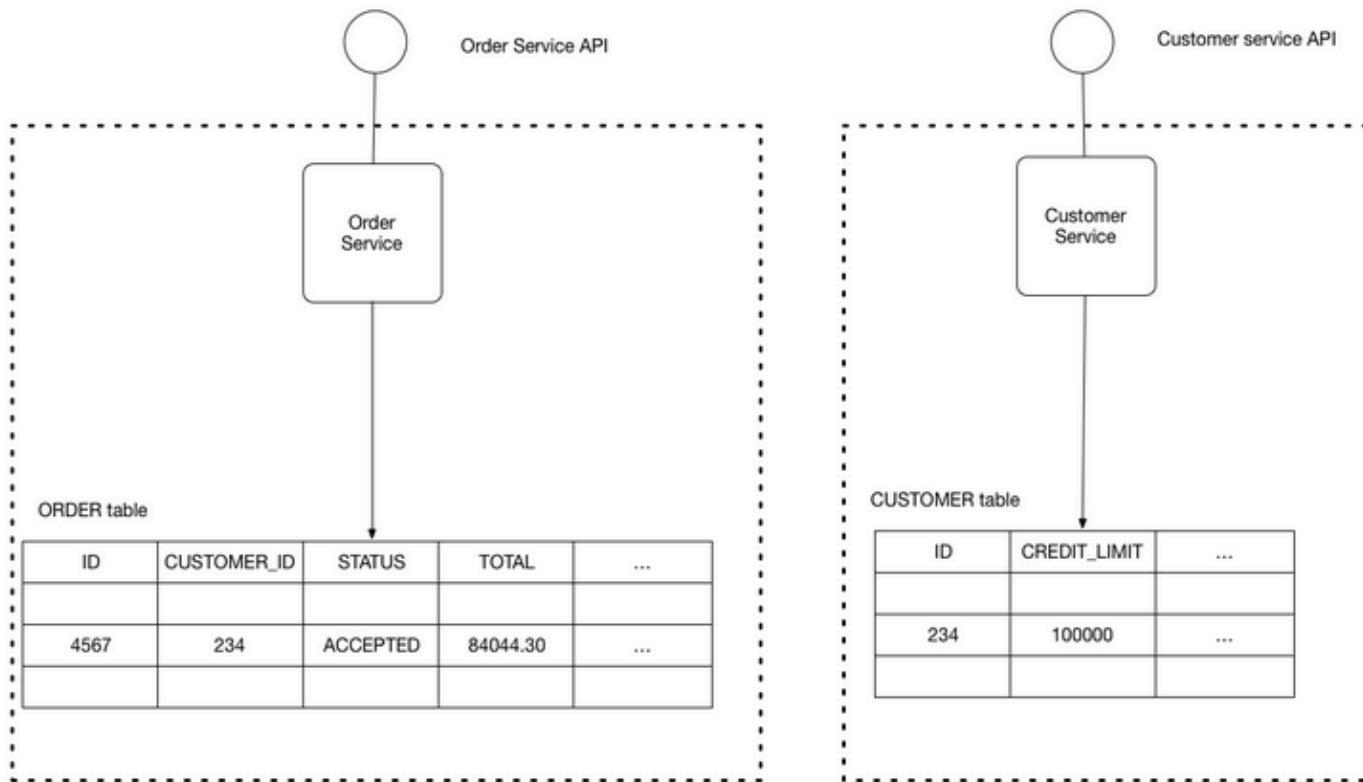
iv.iii Protocolos ligeros de comunicación para microservicios (r)

- Desventajas de comunicación síncrona:
 - Mayor complejidad en el diseño de sistemas distribuidos mediante comunicación asíncrona.
 - Dificultad de manejar errores en componentes con comunicación asíncrona.
 - Alto acoplamiento con el "**broker**" de mensajes.
 - Alto costo de latencia si la bandeja (cola) de mensajes alcanza su límite.
 - Alto costo para el monitoreo y "**debug**" de la infraestructura de mensajes.

iv.iii Protocolos ligeros de comunicación para microservicios (s)

- Autonomía de microservicios (i).
- Cuando un microservicio requiera datos para operar, cuyo propietario de los datos es otro microservicio, no dependa de la solicitud de dichos datos a ese otro microservicio. Replique los datos en ambos microservicios (**Database per Service Pattern**).

iv.iii Protocolos ligeros de comunicación para microservicios (t)





iv.iii Protocolos ligeros de comunicación para microservicios (u)

- Práctica 9. Comunicación asíncrona
- Analiza la aplicación **0-Embedded-ActiveMQ-Broker-Service**, **9-Order-Producer-Microservice** y **9-Order-Consumer-Microservice**.
- Ingresar a la ruta: **{tu-workspace}/0-Embedded-ActiveMQ-Broker-Service**
- Importar el proyecto **0-Embedded-ActiveMQ-Broker-Service** en STS.
- Define el Bean **BrokerService** para exponer el broker ActiveMQ embebido como un broker de mensajes JMS en local a través del conector “**tcp://localhost:61616**”.



iv.iii Protocolos ligeros de comunicación para microservicios (v)

- **Práctica 9. Comunicación asíncrona**
- Ejecuta la clase principal del proyecto, anotada con **@SpringBootApplication**, se deberá visualizar una salida en consola similar a lo siguiente:

```
No active profile set, falling back to default profiles: default
Using Persistence Adapter: KahaDBPersistenceAdapter[/Users/xvhx/mgt-ws/ws-curso-microservicios-spring-cloud-netflix-profesor/9]
JMX consoles can connect to service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
KahaDB is version 6
PListStore:[/Users/xvhx/mgt-ws/ws-curso-microservicios-spring-cloud-netflix-profesor/9]
Apache ActiveMQ 5.15.8 (localhost, ID:trial-62922-1558910531705-0:2) is starting
Listening for connections at: tcp://localhost:61616
Connector tcp://localhost:61616 started
Apache ActiveMQ 5.15.8 (localhost, ID:trial-62922-1558910531705-0:2) started
For help or more information please see: http://activemq.apache.org
LiveReload server is running on port 35729
Started EmbeddedBrokerServiceMicroservice in 0.162 seconds (JVM running for 29.44)
Condition evaluation unchanged
```



+

iv.iii Protocolos ligeros de comunicación para microservicios (w)

- Práctica 9. Comunicación asíncrona
- Ingresar a la ruta: **{tu-workspace}/9-Order-Consumer-Microservice**
- Importar el proyecto **9-Order-Consumer-Microservice** en STS.
- Analiza la clase **Order** del paquete
com.consulting.mgt.springboot.practica9.embedded.broker.service.model.
- Analiza la clase de configuración **ActiveMQConfig** del paquete
com.consulting.mgt.springboot.practica9.embedded.broker.service._config; habilita mensajería JMS mediante la anotación **@EnableJms**.



+

iv.iii Protocolos ligeros de comunicación para microservicios (x)

- Práctica 9. Comunicación asíncrona
- Define una constante String **ORDER_QUEUE** con el valor “**order-queue**” en la clase **ActiveMQConfig**.
- Define el Bean **OrderConsumer**, mediante configuración por anotaciones, directamente sobre la clase **OrderConsumer** en el paquete **com.consulting.mgt.springboot.practica9.embedded.broker.service.consumer**.
- Define un listener JMS, mediante la anotación **@JmsListener** que escuche mensajes directamente de la “**queue**” destino definida por la constante **ORDER_QUEUE**. Procesa el mensaje **Order** mediante la anotación **@Payload** y loggea el cuerpo del mensaje.



iv.iii Protocolos ligeros de comunicación para microservicios (y)

- **Práctica 9. Comunicación asíncrona**
- Ingresa en una consola o terminal a la ubicación **{tu-workspace}/9-Order-Consumer-Microservice** y, compila y empaqueta la aplicación mediante el comando: “**mvn clean package**”.
- Posteriormente ejecuta la aplicación mediante el comando “**java -jar target/9-Order-Consumer-Microservice-0.0.1-SNAPSHOT.jar**”.
- Es posible iniciar uno o más instancias del servicio **9-Order-Consumer-Microservice**, lo cual mejoraría la alta disponibilidad del servicio.



+

NETFLIX
OSS

iv.iii Protocolos ligeros de comunicación para microservicios (z)

- Práctica 9. Comunicación asíncrona
- Ingresar a la ruta: **{tu-workspace}/9-Order-Producer-Microservice**
- Importar el proyecto **9-Order-Producer-Microservice** en STS.
- Analiza la clase **Order** del paquete
com.consulting.mgt.springboot.practica9.embedded.broker.service.model.
- Analiza la clase de configuración **ActiveMQConfig** del paquete
com.consulting.mgt.springboot.practica9.embedded.broker.service._config; habilita mensajería JMS mediante la anotación **@EnableJms**.



+

NETFLIX
OSS

iv.iii Protocolos ligeros de comunicación para microservicios (a')

- Práctica 9. Comunicación asíncrona
- Define el Bean **OrderProducer**, mediante configuración por anotaciones, directamente sobre la clase **OrderProducer** en el paquete **com.consulting.mgt.springboot.practica9.embedded.broker.service.producer**.
- Define inyección de dependencias de un objeto **JmsTemplate** sobre el Bean **OrderProducer**.
- Define un método para enviar un mensaje de tipo **Order** mediante el template **JmsTemplate** definido.



+

NETFLIX
OSS

iv.iii Protocolos ligeros de comunicación para microservicios (b')

- Práctica 9. Comunicación asíncrona
- Define el Bean controlador REST sobre la clase **OrderController**, mediante configuración por anotaciones, directamente sobre la clase **OrderController** en el paquete **com.consulting.mgt.springboot.practica9.embedded.broker.service.restcontroller**.
- Define inyección de dependencias del objeto **OrderProducer** sobre el Bean **OrderController**.
- Define un “**handler method**” que reciba las peticiones HTTP entrantes mediante la URI **/place-order** a través del método **GET** y envía un mensaje de tipo **Order** utilizando la dependencia **OrderProducer**.



iv.iii Protocolos ligeros de comunicación para microservicios (c')

- **Práctica 9. Comunicación asíncrona**
- Ingresa en una consola o terminal a la ubicación **{tu-workspace}/9-Order-Producer-Microservice** y, compila y empaqueta la aplicación mediante el comando: “**mvn clean package**”.
- Posteriormente ejecuta la aplicación mediante el comando “**java -jar target/9-Order-Producer-Microservice-0.0.1-SNAPSHOT.jar**”.



iv.iii Protocolos ligeros de comunicación para microservicios (d')

- **Práctica 9. Comunicación asíncrona**
- Desde un cliente HTTP, ejecuta una petición GET al servicio expuesto por el servicio **/place-order**, mediante la URL <http://localhost:8080/place-order>.
- Como ejemplo, mediante consola, ejecuta el comando “**http localhost:8080/place-order**” (requiere **httpie** instalado). Es posible utilizar **cURL** mediante el comando “**curl -i -X GET <http://localhost:8080/place-order>**”.
- Analiza el comportamiento de los servicios y la comunicación asíncrona.



Resumen de la lección

iv.iii Protocolos ligeros de comunicación para microservicios

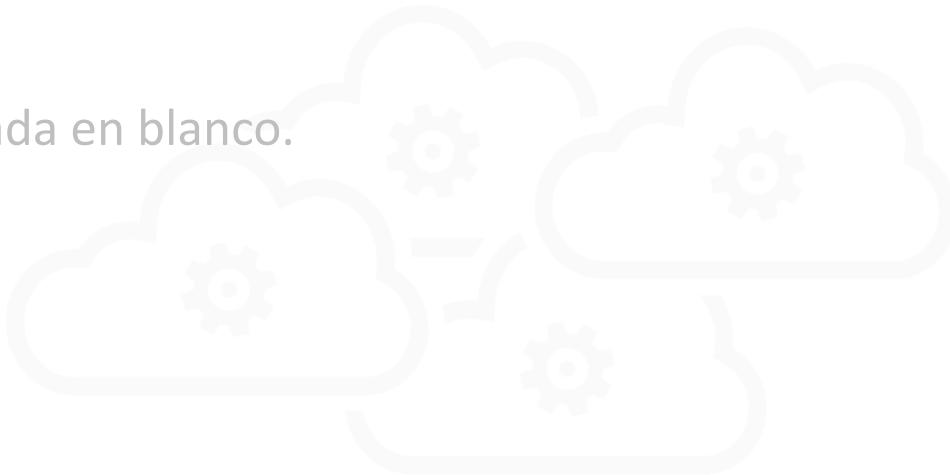
- Comprendemos los distintos mecanismos de comunicación entre microservicios diferenciando entre síncronos y asíncronos.
- Analizamos y discutimos las diferencias entre comunicación síncrona y asíncrona y cuál es la recomendación para los distintos casos de uso.
- Aplicamos comunicación asíncrona mediante broker de mensajes ActiveMQ entre dos microservicios.



+

NETFLIX
OSS

Esta página fue intencionalmente dejada en blanco.



Microservices



+

iv. Arquitectura de Microservicios

- iv.i ¿Qué es la arquitectura orientada a microservicios?
- iv.ii Descomponiendo aplicaciones monolíticas.
- iv.iii Protocolos ligeros de comunicación para microservicios.
- iv.iv **Aplicaciones “cloud-native”.**
- iv.v Principios de diseño para aplicaciones en la nube.
- iv.vi Orquestación vs Coreografía.
- iv.vii Gestión de Transacciones ACID vs BASE.
- iv.viii API Manager



+

NETFLIX
OSS

iv.iv Aplicaciones “cloud-native”.



Objetivos de la lección

iv.iv Aplicaciones “cloud-native”.

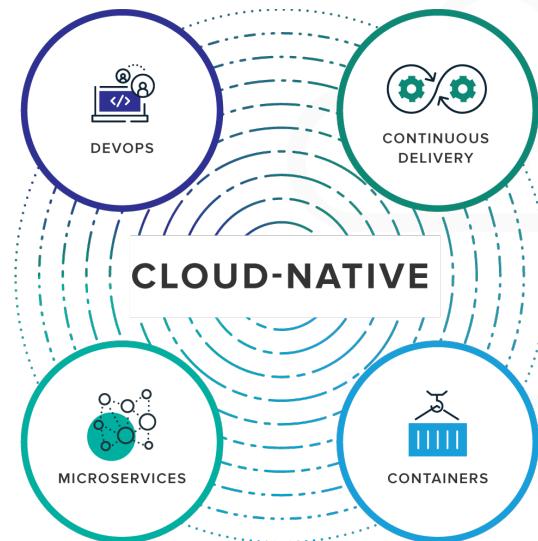
- Aprenderemos cuales son los pilares a considerar de las aplicaciones “cloud-native”.
- Analizaremos la metodología DevOps y sus beneficios.
- Revisaremos lo que son los contenedores y como aplican en una arquitectura de microservicios.
- Aprenderemos que es necesaria la integración continua en ambientes de aplicaciones para la nube.

iv.iv Aplicaciones “cloud-native” (a)

- “**Cloud-native**” es un nuevo enfoque de desarrollar y ejecutar aplicaciones las cuales explotan las ventajas del modelo de entrega de la computación en la nube (**cloud-computing**).
- Las aplicaciones “**cloud-native**” tratan de cómo se crean y despliegan las aplicaciones, más no en dónde.
- Cuando las empresas crean y operan aplicaciones de una manera nativa para la nube, éstas traen nuevas ideas al mercado más rápido y responden más rápido a las demandas de los clientes.

iv.iv Aplicaciones “cloud-native” (b)

- Las organizaciones requieren una plataforma para crear y operar aplicaciones y servicios “**cloud-native**” que automaticen e integren los conceptos de DevOps, entrega continua (**Continuous Delivery**), microservicios y contenedores.



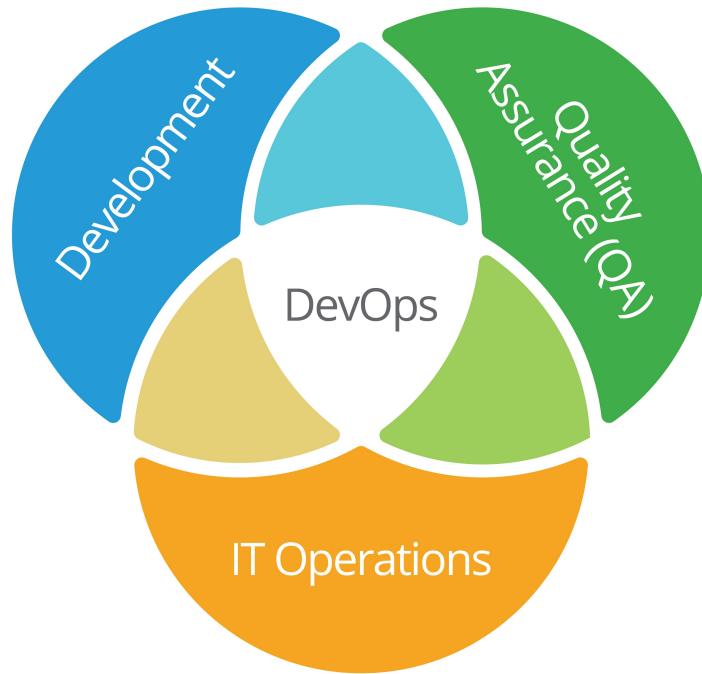


iv.iv Aplicaciones “cloud-native” (c)

- DevOps.
- Es una metodología que permite la colaboración entre desarrolladores y personal de operaciones (administradores de sistemas), sin embargo, requiere un fuerte cambio cultural y de organización, para su correcta implementación, permitiendo la colaboración y la comunicación que permita integrar las áreas de desarrollo y de sistemas.
- Una buena práctica de DevOps liberá a los desarrolladores para centrarse en hacer lo que mejor saben hacer: escribir software, debido a que DevOps elimina el trabajo y las preocupaciones de la puesta en producción del software una vez que está escrito.

iv.iv Aplicaciones “cloud-native” (d)

- DevOps.



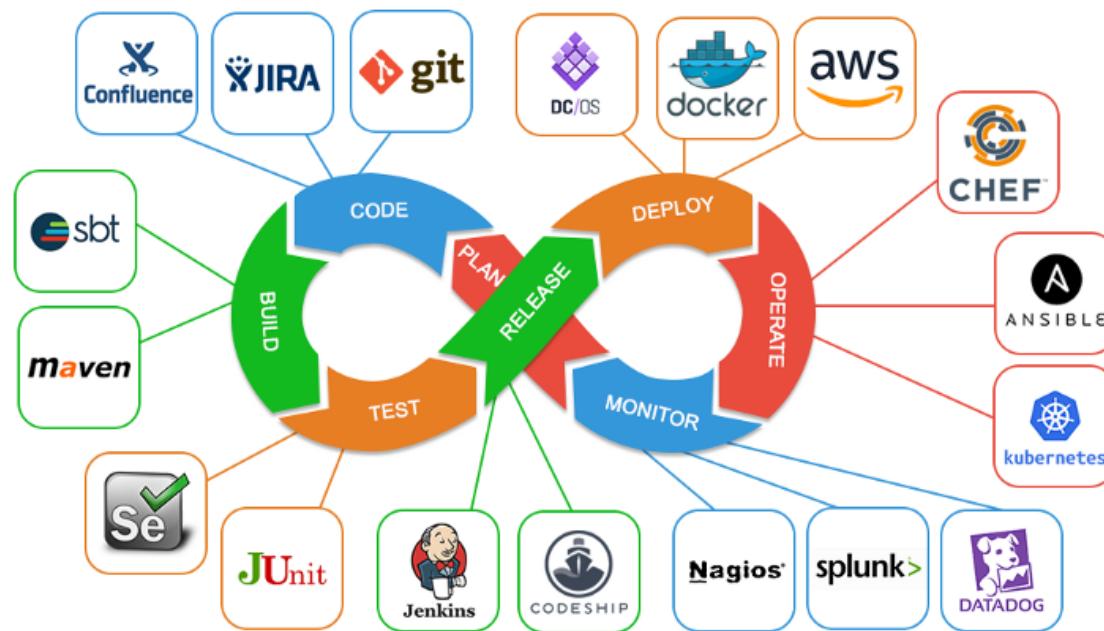


iv.iv Aplicaciones “cloud-native” (e)

- Entrega Continua (Continuous Delivery).
- Otro de los pilares de las aplicaciones “**cloud-native**” es que, mediante DevOps, todos los pasos en la construcción y despliegue de las aplicaciones estén automatizados, para así evitar el error humano y, agilizar y mejorar el proceso de construcción y despliegue aumentando la calidad del software.
- La entrega continua facilita que el producto de software se despliegue y ejecute en entornos productivos en el menor tiempo posible y con mayor continuidad, con el menor costo y la máxima garantía de calidad.

iv.iv Aplicaciones “cloud-native” (f)

- Entrega Continua (Continuous Delivery).



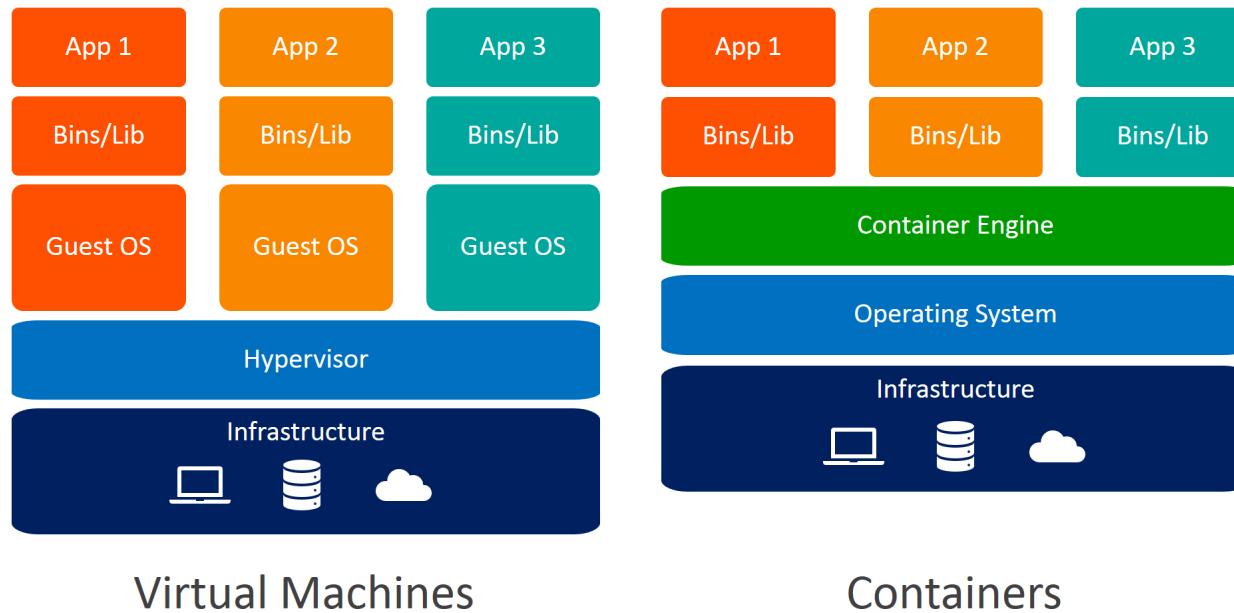


iv.iv Aplicaciones “cloud-native” (g)

- Contenedores (Containers).
- Los contenedores habilitan garantizar que un producto de software se ejecuta de la misma manera en cualquier entorno en el que éste se despliegue.
- Se evita el famoso dicho “en mi máquina si funciona”.
- Se simplifica ampliamente la virtualización de máquinas debido a que se omite el sistema operativo Host de las VMs, ofreciendo mayor eficiencia y velocidad de procesamiento.

iv.iv Aplicaciones “cloud-native” (h)

- Contenedores (Containers).





+

NETFLIX
OSS

iv.iv Aplicaciones “cloud-native” (i)

- Microservicios.
- Los microservicios, o la arquitectura orientada a microservicios, es un tipo de arquitectura, que sirve para diseñar aplicaciones donde las funciones o funcionalidades del sistema están desplegadas de forma independiente ejecutándose en procesos independientes.
- Las arquitecturas de microservicios adquieren desafíos técnicos o no funcionales los cuales deben ser mitigados mediante patrones de diseño orientados a la nube.



iv.iv Aplicaciones “cloud-native” (j)

- Los 10 principales atributos de las aplicaciones “**cloud-native**”.
- 1. **Empaquetados como contenedores ligeros:** Las aplicaciones “**cloud-native**” son una colección de servicios independientes y autónomos que se empaquetan como contenedores livianos los cuales, pueden escalarse rápidamente.
- 2. **Desarrollado con los mejores lenguajes y frameworks de trabajo:** Cada servicio de una aplicación “**cloud-native**” se desarrolla utilizando el lenguaje y/o framework más adecuado para la funcionalidad específica. Las aplicaciones “**cloud-native**” son políglotas.

iv.iv Aplicaciones “cloud-native” (k)

- Los 10 principales atributos de las aplicaciones “**cloud-native**”.
3. **Diseñados como microservicios de bajo acoplamiento:** Los servicios que pertenecen a la misma aplicación se descubren en tiempo de ejecución. Los microservicios existen de forma independiente a otros servicios. Una correcta arquitectura de microservicios y la elasticidad de los servicios permite que los mismos se integren correctamente, puedan ser escalados con eficiencia y alto rendimiento.

Los servicios débilmente acoplados permiten a los desarrolladores tratar cada servicio de manera independiente.

iv.iv Aplicaciones “cloud-native” (I)

- Los 10 principales atributos de las aplicaciones “**cloud-native**”.
4. **Centrado en las API para la interacción y la colaboración:** Los servicios “**cloud-native**” utilizan APIs ligeras que se basan en protocolos ligeros y abiertos de comunicación como HTTP/REST, gRPC (Google Remote Procedure Call), AMQP, TCP o NATS (open-source, cloud-native messaging system).

REST sobre HTTP, se utiliza ampliamente para exponer las APIs hacia el exterior de la aplicaciones y, para mejorar el rendimiento, se sugiere la implementación de gRPC, AMQP o NATS para la comunicación interna entre los microservicios.



iv.iv Aplicaciones “cloud-native” (m)

- Los 10 principales atributos de las aplicaciones “cloud-native”.
5. **Arquitectura diseñada con una clara separación entre servicios con y sin estado:** Los servicios que son persistentes y duraderos (con estado o stateful) siguen un patrón diferente que asegura una mayor disponibilidad y resistencia. Los servicios sin estado (o stateless) existen independientemente de los servicios con estado.



+

iv.iv Aplicaciones “cloud-native” (n)

- Los 10 principales atributos de las aplicaciones “**cloud-native**”.
- 6. Microservicios aislado de dependencias del servidor y del sistema operativo:** Las aplicaciones “**cloud-native**” no tienen afinidad con ningún sistema operativo en particular o máquina concreta.

La única excepción es cuando un microservicio necesita ciertas capacidades de unidades de estado sólido (SSD) o unidades de procesamiento de gráficos (GPU), que pueden ser ofrecidas exclusivamente por un conjunto de máquinas especializadas.

iv.iv Aplicaciones “cloud-native” (ñ)

- Los 10 principales atributos de las aplicaciones “cloud-native”.
- 7. **Desplegados a través de autoservicio (self-service), mediante infraestructura elástica y en la nube:** Las aplicaciones “cloud-native” se despliegan en infraestructura virtual, compartida y elástica, la cual es provista por el mismo equipo de desarrollo en un entorno de auto-servicio.

iv.iv Aplicaciones “cloud-native” (o)

- Los 10 principales atributos de las aplicaciones “**cloud-native**”.
- 8. **Gestionado a través de procesos ágiles de DevOps:** Cada servicio de una aplicación “**cloud-native**” pasa por un ciclo de vida (**pipe-line**) independiente, que se gestiona a través de un proceso ágil de DevOps.

Múltiples “**pipe-lines**” de integración continua/entrega continua (CI / CD) trabajan en conjunto para desplegar y administrar una aplicación “**cloud-native**”.



iv.iv Aplicaciones “cloud-native” (p)

- Los 10 principales atributos de las aplicaciones “**cloud-native**”.
- 9. **Capacidades automatizadas:** Las aplicaciones “**cloud-native**” deben ser altamente automatizadas; se implementan correctamente con el concepto de infraestructura como código mediante herramientas como Ansible o Terraform.
- 10. **Asignación de recursos definida y basada en políticas de consumo:** Las aplicaciones “**cloud-native**” se alinean con el modelo de gobierno definido a través de un conjunto de políticas de consumo. Se adhieren a políticas de consumo de CPU, cuotas de almacenamiento y, políticas de red que asignan recursos a los servicios.



Resumen de la lección

iv.iv Aplicaciones “cloud-native”

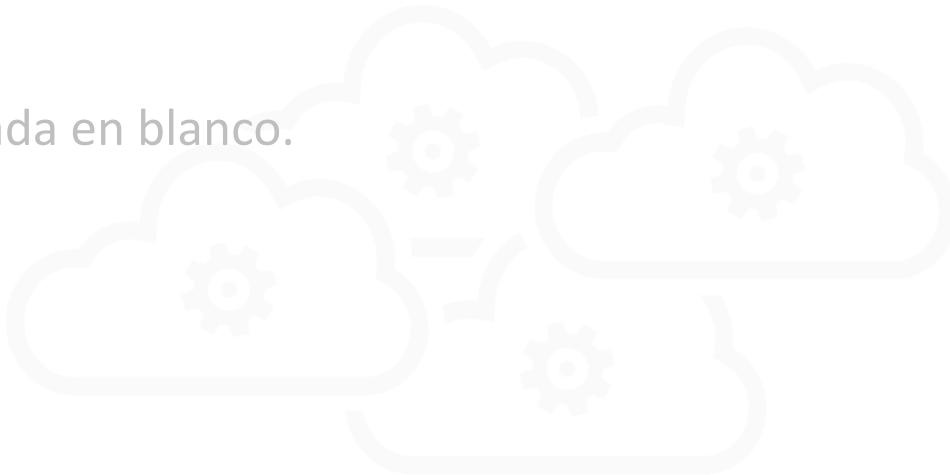
- Aprendimos que los microservicios, una cultura DevOps, contenedores y entrega continua (CI/CD) forman parte de los pilares de las aplicaciones “**cloud-native**”.
- Comprendimos las principales características que deben considerarse en la implementación de aplicaciones “**cloud-native**”.



+

NETFLIX
OSS

Esta página fue intencionalmente dejada en blanco.



Microservices



+

iv. Arquitectura de Microservicios

- iv.i ¿Qué es la arquitectura orientada a microservicios?
- iv.ii Descomponiendo aplicaciones monolíticas.
- iv.iii Protocolos ligeros de comunicación para microservicios.
- iv.iv Aplicaciones “cloud-native”.
- iv.v **Principios de diseño para aplicaciones en la nube.**
- iv.vi Orquestación vs Coreografía.
- iv.vii Gestión de Transacciones ACID vs BASE.
- iv.viii API Manager



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube.



Objetivos de la lección

iv.v Principios de diseño para aplicaciones en la nube.

- Comprender los 10 principios de diseño para implementar aplicaciones basadas en la nube.
- Implementar patrones de diseño especializados para la nube.
- Conocer las recomendaciones para diseñar aplicaciones resistentes, escalables, administrables, mantenibles y con alta disponibilidad para la nube.



iv.v Principios de diseño para aplicaciones en la nube. (a)

- Para una correcta implementación de aplicaciones en la nube integre los 10 principios de diseño para obtener aplicaciones escalables, resistentes y administrables.
 - Diseñe para la recuperación automática.
 - Haga todo redundante.
 - Minimizar la coordinación.
 - Diseñe para facilitar el escalamiento horizontal.
 - Particione alrededor de límites.
 - Diseñe para las operaciones.
 - Use servicios administrados.
 - Use el repositorio correcto para el trabajo correcto.
 - Diseñe para evolucionar el servicio.
 - Desarrolle considerando las necesidades del negocio.



iv.v Principios de diseño para aplicaciones en la nube. (b)

- Diseñe para la recuperación automática.
- Haga todo redundante.
- Minimizar la coordinación.
- Diseñe para facilitar el escalamiento horizontal.
- Particione alrededor de límites.
- Diseñe para las operaciones.
- Use servicios administrados.
- Use el repositorio correcto para el trabajo correcto.
- Diseñe para evolucionar el servicio.
- Desarrolle considerando las necesidades del negocio.



iv.v Principios de diseño para aplicaciones en la nube. (c)

- Diseñe para la recuperación automática.
- En un sistema distribuido todo puede fallar, por tanto, diseñe una aplicación que se recupere automáticamente cuando suceda.
- Enfoque en tres puntos:
 - Detectar errores.
 - Responder a los errores correctamente (No lanzar excepciones en cascada).
 - Registrar y supervisar los errores a fin de conseguir una perspectiva operativa (bitácora y pistas de auditoría).

iv.v Principios de diseño para aplicaciones en la nube. (d)

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Reintentos sobre operaciones que resultaron en algún error.**
 - Controles de error transitorios (perdida momentánea de conectividad en la red en componentes y servicios).
 - **Retry Pattern.**
 - Verificar si la operación es adecuada para un reinicio.
 - Determinar un número adecuado de reintentos e intervalos.
 - Evitar reintentos inmediatos.
 - Probar la estrategia de reintentos injectando errores transitorios y no transitorios en el servicio.



iv.v Principios de diseño para aplicaciones en la nube. (e)

- Retry Pattern.
- **Categoría:** Resistencia.
 - Permite a la aplicación manejar fallas transitorias de recursos externos a través de reintentos.
- **Intención.**
 - Reintentar de forma transparente ciertas operaciones que involucran comunicación con recursos externos, particularmente a través de la red.
 - Aísla el código de llamada al servicio externo, de los detalles de la implementación del reintentó.



iv.v Principios de diseño para aplicaciones en la nube. (f)

- Retry Pattern.
- **Aplicabilidad.**
 - Cada vez que una aplicación necesite comunicarse con un recurso externo, especialmente en un entorno de ejecución en la nube y, si los requerimientos del negocio lo permiten (el servicio es idempotente).



+

NETFLIX
OSS

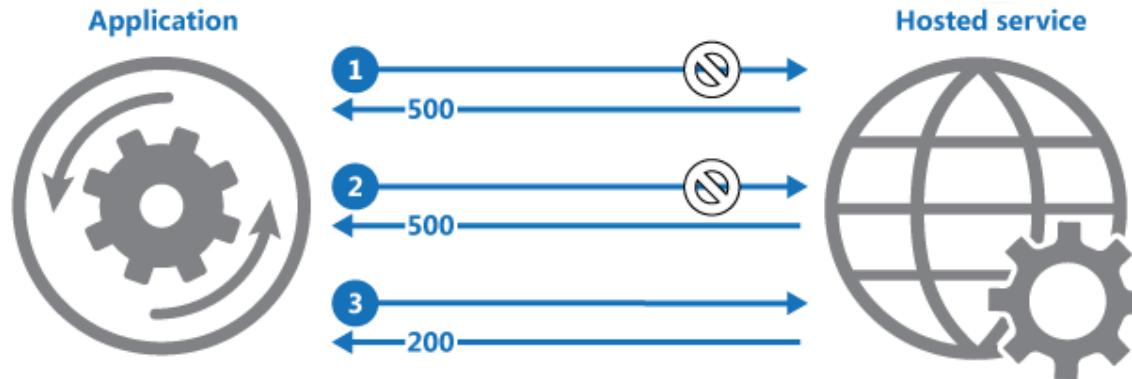
iv.v Principios de diseño para aplicaciones en la nube. (g)

- Retry Pattern.
- **Ventajas:**
 - Resistencia.
- **Desventajas:**
 - Complejidad de implementación desacoplada.
 - Mantenimiento de operaciones.



iv.v Principios de diseño para aplicaciones en la nube. (h)

- Retry Pattern.



- 1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
- 2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
- 3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).



+

iv.v Principios de diseño para aplicaciones en la nube. (i)

- Práctica 10. Retry Pattern
- Analiza la aplicación **10-Failing-Microservice** y **10-Retry-Microservice**.
- Ingresar a la ruta: **{tu-workspace}/10-Failing-Microservice**
- Importar el proyecto **10-Failing-Microservice** en STS.
- Analiza la clase **StatusResponse** del paquete base del proyecto.
- Sobre la clase **Application**, clase principal del proyecto, implementa un **@RestController** que responda un código status HTTP **200** o **500** dependiendo de una entrada en el path “**/{statusCode}**”.



+

iv.v Principios de diseño para aplicaciones en la nube. (j)

- Práctica 10. Retry Pattern
- Define las propiedades **server.servlet.context-path** con el valor “**/failing-service**” y la propiedad **server.port** con el valor **8082**.
- Ejecuta la clase principal **Application**, anotada con **@SpringBootApplication**, la aplicación debe arrancar en el puerto **8082**.
- Prueba la aplicación ingresando en el navegador a la URL <http://localhost:8082/failing-service/200> o <http://localhost:8082/failing-service/500>



+

iv.v Principios de diseño para aplicaciones en la nube. (k)

- Práctica 10. Retry Pattern
- Utilice un cliente HTTP como cURL o httpie para probar el servicio.
- curl -i -X GET <http://localhost:8082/failing-service/200>
- http GET <http://localhost:8082/failing-service/200>



iv.v Principios de diseño para aplicaciones en la nube. (I)

- Práctica 10. Retry Pattern
- Ingresar a la ruta: **{tu-workspace}/10-Retry-Microservice**
- Importar el proyecto **10-Retry-Microservice** en STS.
- Analiza la clase **StatusResponse** del paquete
com.consulting.mgt.springboot.practica10.retry.controller.model.
- Analiza la interface **IBusinessService**, del paquete
com.consulting.mgt.springboot.practica10.retry.service, esta interface define los métodos **setRetries** y **setAttempts** únicamente para pruebas.



+

iv.v Principios de diseño para aplicaciones en la nube. (m)

- Práctica 10. Retry Pattern
- Analiza la clase **BusinessService**, del paquete **com.consulting.mgt.springboot.practica10.retry.service.impl**, implementación de la interface **IBusinessService**.
- En la clase **BusinessService** implementa inyección de dependencias, de un **RestTemplate** y un **String** que contenga la URL del servicio <http://localhost:8082/failing-service/{statusCode}>, mediante constructor.



iv.v Principios de diseño para aplicaciones en la nube. (n)

- Práctica 10. Retry Pattern
- En la clase **BusinessService** implementa el método **perform()** donde se invoque al servicio <http://localhost:8082/failing-service/500> un número determinado de reintentos menos 1 y al final invocar al servicio <http://localhost:8082/failing-service/200>. Lleva en una variable el número de reintentos. En caso de un error al servicio, cacha la excepción, escribe en el log y lanza una excepción personalizada **FailingServiceException**.
- La clase **BusinessService** no implementa reintentos, los reintentos deberán desacoplarse de la clase de negocio.



+

iv.v Principios de diseño para aplicaciones en la nube. (ñ)

- Práctica 10. Retry Pattern
- Analiza las clases **FailingServiceException** y **ServiceException** del paquete **com.consulting.mgt.springboot.practica10.retry.service.exception**.
- Sobre la clase **ApplicationConfig**, del paquete **com.consulting.mgt.springboot.practica10.retry._config**, define los beans **RestTemplate restTemplate**, **String failingServiceURL** y **IBusinessService noRetriableBusinessService** mediante **JavaConfig**.
- Define la clase **RetryController** como un **@RestController** e inyecta la dependencia **IBusinessService bs**, mediante **@Autowired**. Analiza la clase.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (o)

- Práctica 10. Retry Pattern
- Define las propiedades **server.servlet.context-path** con el valor “**/retry-service**” y la propiedad **server.port** con el valor **8081**.
- Define la propiedad **failing.service.url** con el valor “**http://localhost:8082/failing-service/**”
- Ejecuta la clase principal **Application**, anotada con **@SpringBootApplication**, la aplicación debe arrancar en el puerto **8081**.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (p)

- Práctica 10. Retry Pattern
- Prueba la aplicación ingresando en el navegador a la URL <http://localhost:8081/retry-service/{retries}> donde **retries** es el número de intentos que ejecutará el servicio para realizar el llamado al servicio remoto ejecutándose en el puerto **8082**.
- El servicio **retry-service** no implementa reintentos por el momento.
- Prueba la aplicación **retry-service** sin reintentos.



+

iv.v Principios de diseño para aplicaciones en la nube. (q)

- Práctica 10. Retry Pattern
- Analiza la clase **RetryBusinessService**, del paquete **com.consulting.mgt.springboot.practica10.retry.service.impl**, esta clase implementa la interface **IBusinessService** y agrega composición de un objeto **IBusinessService targetBusinessService** implementando un patrón **Proxy**.
- La clase **RetryBusinessService** agrega el número máximo de reintentos que ejecutará (**int maxAttempts**), el retraso o “**delay**” entre cada reinicio y el número de reintentos ejecutados (**AtomicInteger attempts**).



iv.v Principios de diseño para aplicaciones en la nube. (r)

- Práctica 10. Retry Pattern
- Implemente el método **perform()** de la clase **RetryBusinessService** llamando al método **perform()** del objeto **targetBusinessService** donde reintente un número máximo de reintentos (int maxAttempts) e implemente un "delay" entre cada reinicio. En caso de que se llegue al número máximo de reintentos lance la excepción que sea lanzada por el método **perform()** del objeto **targetBusinessService**.



+

iv.v Principios de diseño para aplicaciones en la nube. (s)

- Práctica 10. Retry Pattern
- Sobre la clase **ApplicationConfig**, del paquete **com.consulting.mgt.springboot.practica10.retry._config**, define el bean **IBusinessService retriableBusinessService** de tipo concreto **RetryBusinessService** mediante **JavaConfig**.
- Defina el bean **IBusinessService retriableBusinessService** como primario (**@Primary**).
- Observe que existen dos beans de tipo **IBusinessService** definidos, **noRetriableBusinessService** y **retriableBusinessService**, donde **retriableBusinessService** es un bean intercambiable proxy de **noRetriableBusinessService**.



+

iv.v Principios de diseño para aplicaciones en la nube. (t)

- Práctica 10. Retry Pattern
- Prueba la aplicación ingresando en el navegador a la URL <http://localhost:8081/retry-service/{retries}> donde **retries** es el número de intentos que ejecutará el servicio para realizar el llamado al servicio remoto ejecutándose en el puerto **8082**.
- El servicio **retry-service** implementa reintentos de forma automática.
- Prueba la aplicación **retry-service** con reintentos.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (u)

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Proteger los servicios remotos defectuosos.**
 - Realizar reintentos es conveniente después de un error transitorio, pero si el error persiste, se puede originar errores en cascada.
 - **Circuit Breaker Pattern.**
 - Implemente el patrón “circuit breaker” para fallar y responder rápido a los errores, sin realizar la llamada remota cuando es altamente probable que la operación generará error.



iv.v Principios de diseño para aplicaciones en la nube. (v)

- Circuit Breaker Pattern.
- **Categoría:** Resistencia.
 - Permite a la aplicación manejar fallas transitorias de recursos externos a través de reintentos y de forma similar a un disyuntor eléctrico.
 - El patrón “**circuit breaker**” permite crear aplicaciones que fallen rápido (“**fail-fast**”) temporalmente, deshabilitando la ejecución de la llamada al recurso externo, previniendo la sobrecarga del sistema, evitando el consumo de recursos para una operación que posiblemente falle.



iv.v Principios de diseño para aplicaciones en la nube. (w)

- Circuit Breaker Pattern.
- **Categoría:** Resistencia.
 - Dada la situación donde el número de ejecuciones fallidas a un recurso externo sobrepasa el límite establecido, el “**circuit breaker**” se abre para habilitar el “**fail-fast**” del servicio, durante un periodo de tiempo.
 - Cuando el “**circuit breaker**” se abre, el patrón habilita una respuesta rápida alternativa para mantener operable el sistema, dicha respuesta alternativa se conoce como “**fallback**”.



iv.v Principios de diseño para aplicaciones en la nube. (x)

- Circuit Breaker Pattern.
- **Intención.**
 - Evitar llamadas consecutivas a un servicio externo que no ha respondido satisfactoriamente y que es altamente probable que falle.
 - Ejecutar el “**fail-fast**” de servicios y evitar el consumo de recursos computacionales durante la espera de la respuesta a la invocación del servicio externo.
 - Habilitar una respuesta rápida alternativa o “**default**”, de la llamada remota, que permita continuar operando al sistema en forma de “**fallback**”.



iv.v Principios de diseño para aplicaciones en la nube. (y)

- Circuit Breaker Pattern.
- **Intención.**
 - Reintentar un conjunto de llamadas al servicio después de que se ha alcanzado el periodo de tiempo esperado para que el servicio externo vuelva a estar disponible habilitando el **“circuit breaker”** como **“half-open”**.
 - Si los reintentos realizados son satisfactorios, el **“circuit breaker”** se cierra y el contador de ejecuciones fallidas se reinicia.



iv.v Principios de diseño para aplicaciones en la nube. (z)

- Circuit Breaker Pattern.
- **Aplicabilidad.**
 - Cada vez que una aplicación necesite comunicarse con un recurso externo, especialmente en un entorno de ejecución en la nube y, si los requerimientos del negocio lo permiten (el servicio es idempotente).
 - Prevenir a la aplicación de invocar ejecuciones remotas a servicios que son altamente probables de fallar.



+

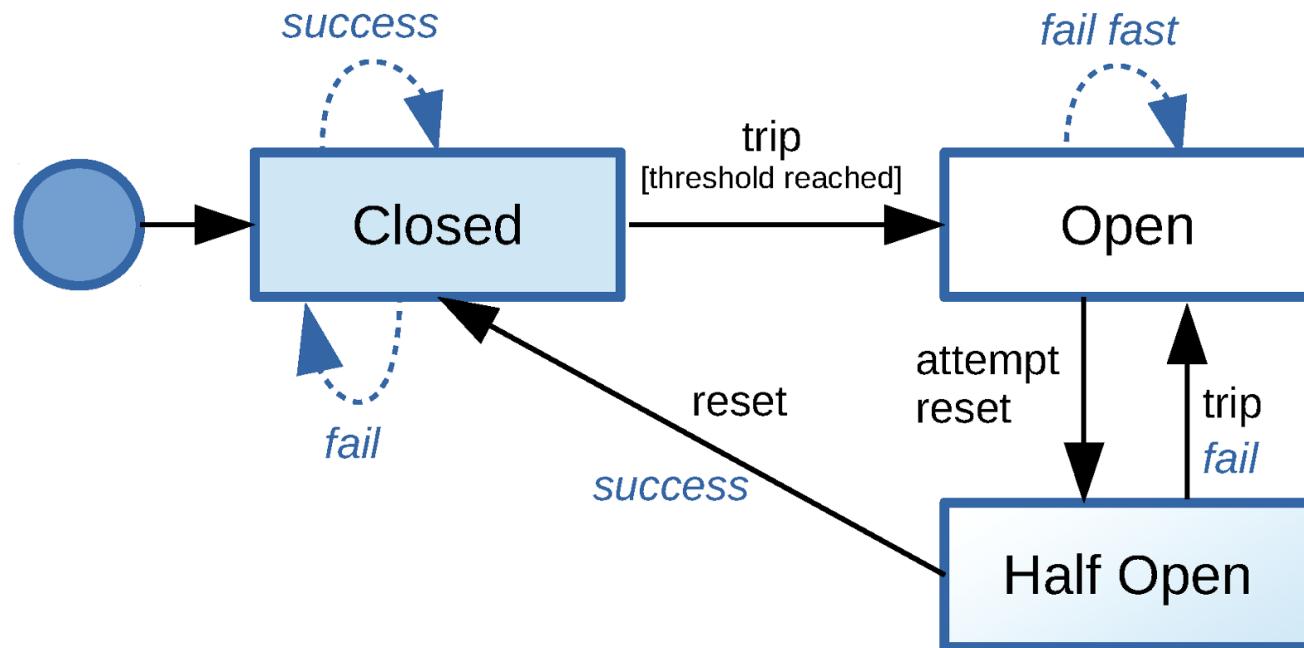
iv.v Principios de diseño para aplicaciones en la nube. (a')

- Circuit Breaker Pattern.
- **Ventajas:**
 - Resistencia / tolerancia a fallos.
 - Proporciona datos duros sobre fallas externas.
 - Habilita el monitoreo.
 - Evita sobrecarga.
- **Desventajas:**
 - Complejidad de implementación desacoplada.
 - Mantenimiento de operaciones.



iv.v Principios de diseño para aplicaciones en la nube. (b')

- Circuit Breaker Pattern.





iv.v Principios de diseño para aplicaciones en la nube. (c')

- Práctica 11. Circuit Breaker Pattern
- Analiza la aplicación **11-Failing-Microservice** y **11-Circuit-Breaker-Microservice**.
- Ingresar a la ruta: **{tu-workspace}/11-Failing-Microservice**
- Importar el proyecto **11-Failing-Microservice** en STS.
- Analiza la clase **StatusResponse** que se encuentra en el paquete principal del proyecto, **com.consulting.mgt.springboot.practica11.circuitbreaker.failingservice**.



iv.v Principios de diseño para aplicaciones en la nube. (d')

- Práctica 11. Circuit Breaker Pattern
- Define un controlador REST sobre la clase principal **Application**, anotada con la anotación **@SpringBootApplication**, sobre el paquete **com.consulting.mgt.springboot.practica11.circuitbreaker.failingservice**.
- Define un “**handler-method**” mediante la anotación **@GetMapping** que atienda las peticiones entrantes por “**root**” (“**/**”) y que responda un **ResponseEntity** con código status HTTP **200** y, sobre el “**body**” del response, un objeto nuevo **StatusResponse** con **statusCode=200** y **status=“UP”**.



+

iv.v Principios de diseño para aplicaciones en la nube. (e')

- Práctica 11. Circuit Breaker Pattern
- Sobre el archivo “**application.properties**”, define las propiedades correspondientes para definir el “**context-path**” del **DispatcherServlet** con el valor “**/failing-service**” y define el puerto de la aplicación web al **8082**.
- Compila la aplicación **11-Failing-Microservice** mediante línea de comandos utilizando “**mvn clean package**” y ejecútala mediante el comando “**java -jar**”.
- Prueba la aplicación ingresando en el navegador a la URL
<http://localhost:8082/failing-service/>



+

iv.v Principios de diseño para aplicaciones en la nube. (f')

- Práctica 11. Circuit Breaker Pattern
- Ingresar a la ruta: **{tu-workspace}/11-Circuit-Breaker-Microservice**
- Importar el proyecto **11-Circuit-Breaker-Microservice** en STS.
- El proyecto **11-Circuit-Breaker-Microservice** utiliza la librería **resilience4j** para implementar el patrón Circuit Breaker, no reinventar la rueda. Más información <https://github.com/resilience4j/resilience4j>
- Analiza la interface funcional **IBusinessService** del paquete **com.consulting.mgt.springboot.practica11.circuitbreaker.service**.



iv.v Principios de diseño para aplicaciones en la nube. (g')

- Práctica 11. Circuit Breaker Pattern
- Analiza las excepciones **ServiceException** y **FailingServiceException** del paquete **com.consulting.mgt.springboot.practica11.circuitbreaker.service.exception**.
- Analiza la clase **StatusResponse** del paquete **com.consulting.mgt.springboot.practica11.circuitbreaker.controller.model**; dicha clase es la misma implementada en el microservicio **11-Failing-Microservice**.
- Analiza la clase **BusinessService**, del paquete **com.consulting.mgt.springboot.practica11.circuitbreaker.service.impl**, que es implementación de la interface **IBusinessService**.



iv.v Principios de diseño para aplicaciones en la nube. (h')

- Práctica 11. Circuit Breaker Pattern
- Sobre la clase **BusinessService**, implemente inyección de dependencias por constructor de un objeto **RestTemplate** y un **String** (cuyo valor será la URL del servicio a consultar del microservicio “<http://localhost:8082/failing-service/>”).
- Sobre la clase **BusinessService**, implemente la llamada mediante **RestTemplate** hacia la URI definida por el **String** injectado en el constructor. Ya están implementados los bloques “try-catch”.
- Note que en caso de haber un error transitorio, los bloques “catch” lanzan una excepción de tipo **FailingServiceException**.



+

iv.v Principios de diseño para aplicaciones en la nube. (i')

- **Práctica 11. Circuit Breaker Pattern**
- De igual forma, note que la implementación del método “**perform()**”, de la clase **BusinessService**, no implementa resistencia, ni reintentos, ni “**fallback**”; en otras palabras no implementa resistencia mediante “**Circuit Breaker Pattern**”, que ofrezca las características de resistencia (“**resiliencia**”) requeridas.
- Sobre la clase **ApplicationConfig**, del paquete **com.consulting.mgt.springboot.practica11.circuitbreaker._config**, defina el bean **String failingServiceURL** e inyecte, mediante anotación **@Value**, la propiedad “**failing.service.url**” la cual debe ser definida sobre el archivo “**application.properties**”.



iv.v Principios de diseño para aplicaciones en la nube. (j')

- Práctica 11. Circuit Breaker Pattern
- De forma análoga, sobre la clase **ApplicationConfig**, defina el bean **IBusinessService noCircuitBreakerBusinessService** e inyecte, mediante su constructor, el bean **RestTemplate** definido y el **String failingServiceURL**.
- Analice el controlador **CircuitBreakerController**, del paquete **com.consulting.mgt.springboot.practica11.circuitbreaker.controller**; la clase ya está implementada.
- Sobre el archivo “**application.properties**”, define las propiedades correspondientes para definir el “**context-path**” del **DispatcherServlet** con el valor “**/circuit-breaker-service**” y define el puerto de la aplicación web al **8081**.



+

iv.v Principios de diseño para aplicaciones en la nube. (k')

- Práctica 11. Circuit Breaker Pattern
- Compila la aplicación **11-Circuit-Breaker-Microservice** mediante línea de comandos utilizando “**mvn clean package**” y ejecútala mediante el comando “**java -jar**”.
- Prueba la aplicación ingresando en el navegador a la URL
<http://localhost:8081/circuit-breaker-service/>
- El microservicio <http://localhost:8081/circuit-breaker-service/> llama al servicio <http://localhost:8082/failing-service/> y responde un mensaje
“**{"statusCode":200,"status":"UP"}**”.



+

iv.v Principios de diseño para aplicaciones en la nube. (I')

- Práctica 11. Circuit Breaker Pattern
- Tire el proceso del microservicio <http://localhost:8082/failing-service/> y vuelva a probar el microservicio <http://localhost:8081/circuit-breaker-service/> ¿Cuál es el resultado?
- Implemente un “Circuit Breaker Pattern” de forma desacoplada al servicio que ejecuta la llamada externa mediante la clase **BusinessService**.
- Analice la documentación de **resilience4j** (<https://github.com/resilience4j/resilience4j>) e implemente los beans necesarios sobre la clase **ApplicationConfig**, del paquete **com.consulting.mgt.springboot.practica11.circuitbreaker._config**.



+

iv.v Principios de diseño para aplicaciones en la nube. (m')

- Práctica 11. Circuit Breaker Pattern
- Implemente el patrón “proxy” mediante la implementación de la misma interface del servicio **BusinessService**. Llame a esta clase **CircuitBreakerBusinessService**.
- Sobre la clase **CircuitBreakerBusinessService**, implemente inyección de dependencias del objeto “target-object” (objeto **IBusinessService** original (target), que es el que realiza la llamada remota) y del objeto **CircuitBreaker** definido.
- Implemente el patrón “decorator”, para decorar un **Supplier<StatusResponse>** que envuelva a la llamada del método **perform()** del “target-object” mediante el objeto **CircuitBreaker**. Analice la documentación de **resilience4j**.



+

iv.v Principios de diseño para aplicaciones en la nube. (n')

- Práctica 11. Circuit Breaker Pattern
- Implemente el método **perform()**, de la clase **CircuitBreakerBusinessService**, mandando a llamar al objeto **Supplier<StatusResponse>** que “decora” la llamada al método **perform()** del “target-object” mediante el **CircuitBreaker**.
- Implemente la recuperación de la llamada fallida del **CircuitBreaker** mediante una llamada a un método “**fallback**” sobre la misma clase **CircuitBreakerBusinessService** mediante el API **Try.ofSupplier** de la librería **io.vavr** (incluida en **resilience4j**). Analice documentación.
- En el método “**fallback**” devuelva un objeto nuevo **StatusResponse** con **statusCode=204** y **status=“DEFAULT STATUS”**.



+

iv.v Principios de diseño para aplicaciones en la nube. (ñ')

- Práctica 11. Circuit Breaker Pattern
- Defina sobre la clase **ApplicationConfig** el bean **IBusinessService circuitBreakerBusinessService** implementado por la clase **CircuitBreakerBusinessService** e inyecte, mediante su constructor, el bean **IBusinessService noCircuitBreakerBusinessService** y el **CircuitBreaker**.
- Defina el bean **IBusinessService circuitBreakerBusinessService**, implementación de la clase **CircuitBreakerBusinessService**, como primario, mediante **@Primary**.



+

iv.v Principios de diseño para aplicaciones en la nube. (o')

- Práctica 11. Circuit Breaker Pattern
- Compila nuevamente la aplicación **11-Circuit-Breaker-Microservice** mediante línea de comandos utilizando “**mvn clean package**” y ejecútala mediante el comando “**java -jar**”.
- Prueba la aplicación ingresando en el navegador a la URL
<http://localhost:8081/circuit-breaker-service/>
- El microservicio <http://localhost:8081/circuit-breaker-service/> llama al servicio (caído) <http://localhost:8082/failing-service/> y responde un mensaje
“**{"statusCode":204,"status":"DEFAULT STATUS"}**”.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (p')

- Práctica 11. Circuit Breaker Pattern
- Levante nuevamente el microservicio <http://localhost:8082/failing-service/> y realice pruebas el microservicio **11-Circuit-Breaker-Microservice** nuevamente ¿Cuál es la respuesta del servicio?
- Tire y levante el proceso <http://localhost:8082/failing-service/> y revise que la implementación del **CircuitBreaker** funciona.



+

iv.v Principios de diseño para aplicaciones en la nube. (q')

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Realizar redistribución de carga.**
 - Redistributions the load of a "back-end" service to avoid overload.
 - **Queue-Based Load Leveling Pattern.**
 - The pattern implements a message queue (or buffer) that reduces work load.
 - Acts as "backpressure" (counter-pressure) to avoid overwork.



iv.v Principios de diseño para aplicaciones en la nube. (r')

- Queue-Based Load Leveling Pattern.
- **Categorías:** Resistencia, rendimiento, escalabilidad, mensajería y disponibilidad.
 - Implemente una cola (“queue”) que actúe como búfer entre la comunicación de una tarea o proceso a un servicio de invocación interna o remota; nivelando la sobrecarga de trabajo del consumidor de la “queue”.



iv.v Principios de diseño para aplicaciones en la nube. (s')

- Queue-Based Load Leveling Pattern.
- **Intención.**
 - La implementación de la “queue”, entre el productor y el consumidor, permite disminuir la sobrecarga al servicio invocado evitando que exista una comunicación intermitente la cual puede ocasionar que el servicio llamado, en la invocación remota, falle o que el proceso que invoca la llamada se bloquee y falle mediante un “time-out”.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (t')

- Queue-Based Load Leveling Pattern.
- **Intención.**
 - La nivelación de carga basada en "**queues**" puede ayudar a minimizar el impacto de los picos en la demanda de disponibilidad y capacidad de respuesta tanto para la tarea o proceso que invoca al servicio, como para el servicio invocado en cuestión.



iv.v Principios de diseño para aplicaciones en la nube. (u')

- Queue-Based Load Leveling Pattern.
- **Aplicabilidad.**
 - Este patrón es útil para cualquier aplicación que utilice servicios sujetos a sobrecarga ya sea mediante llamadas internas o mediante una ejecución remota del servicio.
 - Este patrón no es útil si la aplicación espera una respuesta del servicio, interno o externo, con una latencia mínima.



iv.v Principios de diseño para aplicaciones en la nube. (v')

- Queue-Based Load Leveling Pattern.
- **Ventajas:**
 - Maximiza disponibilidad del servicio invocado.
 - Maximiza la escalabilidad del servicio invocado.
 - Habilita bajo acoplamiento entre el productor y el consumidor.
 - Evita sobrecarga del servicio invocado.
 - Evita tiempos de espera largos del proceso que invoca la llamada remota.

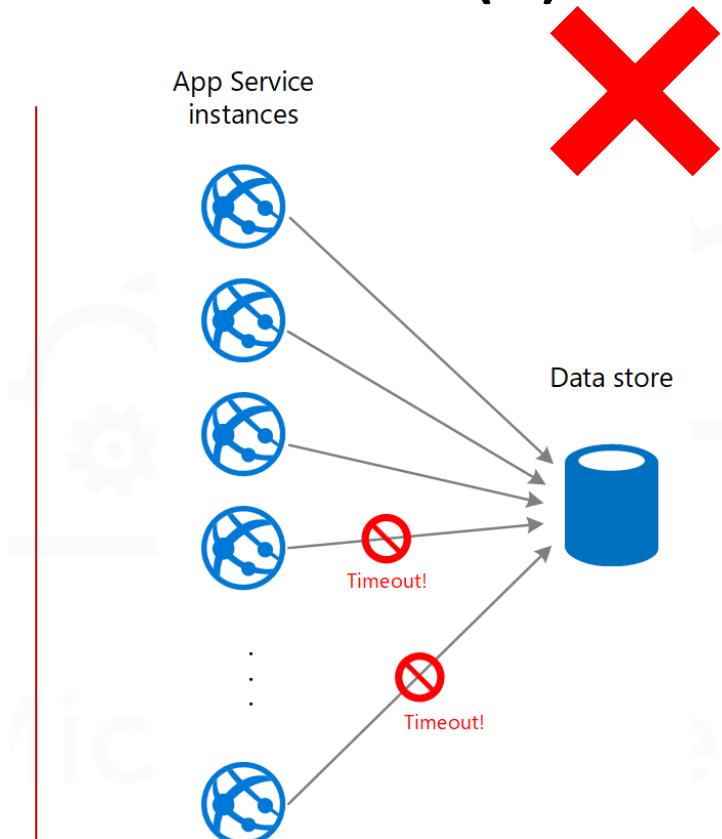
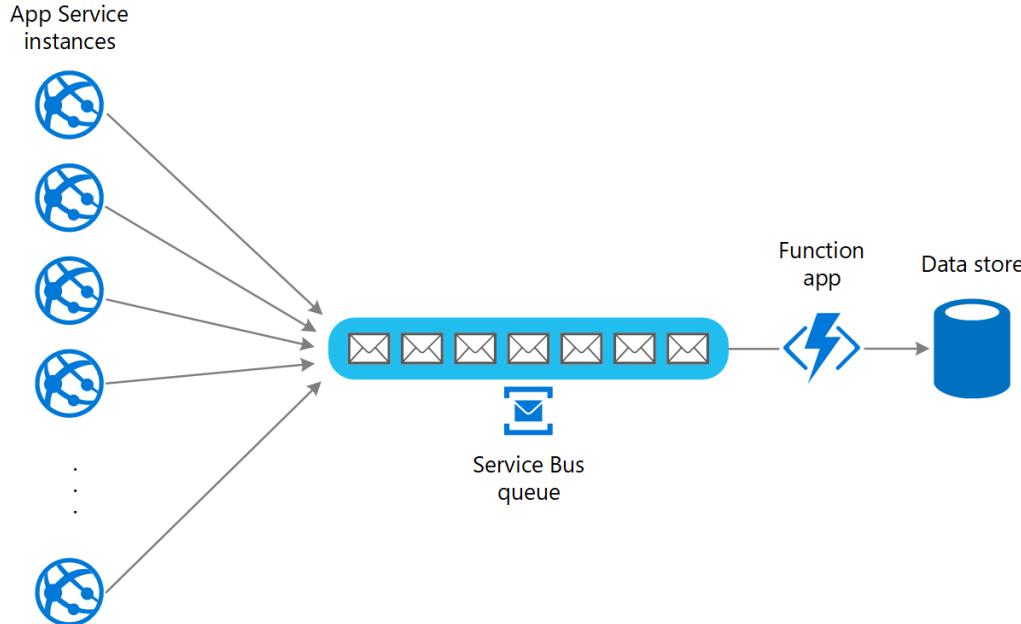


iv.v Principios de diseño para aplicaciones en la nube. (w')

- Queue-Based Load Leveling Pattern.
- Desventajas:
 - Complejidad de implementación de lógica necesaria para controlar la tasa de producción de mensajes contrastada con la tasa de consumo de los mismos.
 - Comunicación hacia sólo un lado (“one-way-communication”).

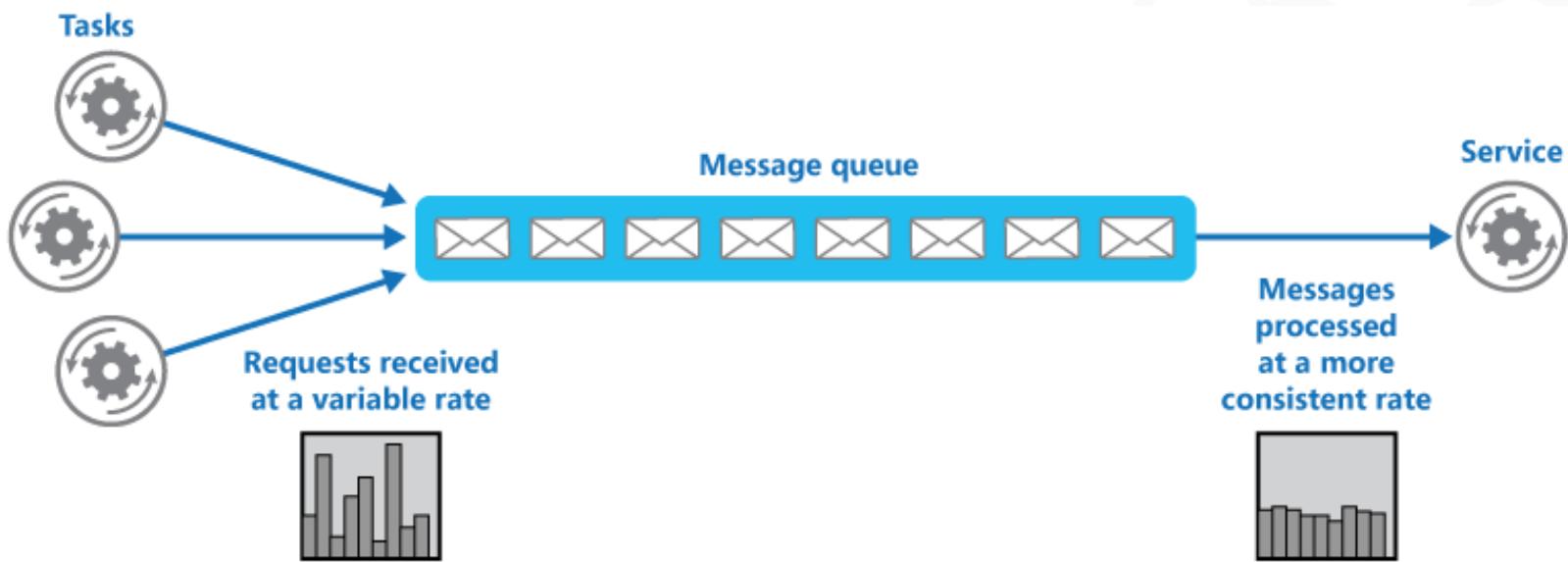
iv.v Principios de diseño para aplicaciones en la nube. (x')

- Queue-Based Load Leveling Pattern.



iv.v Principios de diseño para aplicaciones en la nube. (y')

- Queue-Based Load Leveling Pattern.





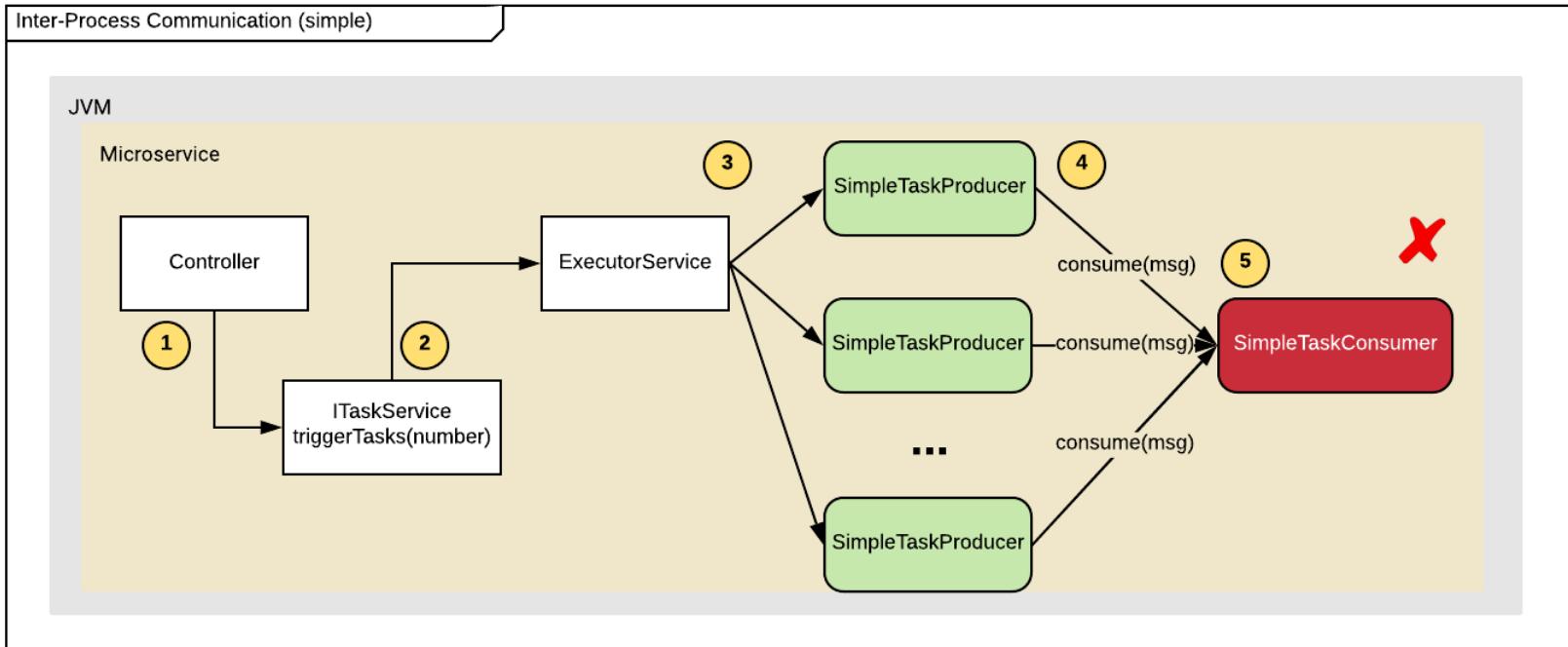
iv.v Principios de diseño para aplicaciones en la nube. (z')

- Práctica 12. Queue Based Load Leveling Pattern
- Analiza la aplicación **12-Queue-Based-Load-Leveling-Microservice**.
- Ingresar a la ruta: **{tu-workspace}/12-Queue-Based-Load-Leveling-Microservice**
- Importar el proyecto **12-Queue-Based-Load-Leveling-Microservice** en STS.
- Analizar el caso de uso, ver imágenes del siguiente slide.



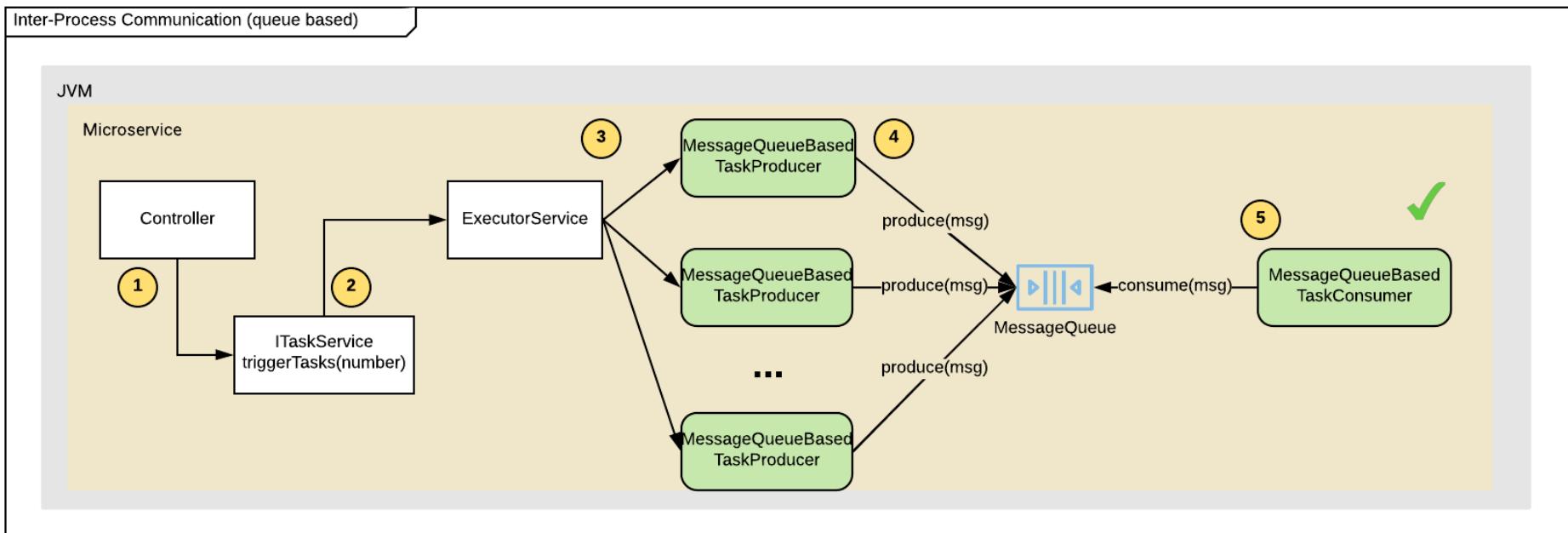
iv.v Principios de diseño para aplicaciones en la nube. (a'')

- Práctica 12. Queue Based Load Leveling Pattern



iv.v Principios de diseño para aplicaciones en la nube. (b'')

- Práctica 12. Queue Based Load Leveling Pattern





+

iv.v Principios de diseño para aplicaciones en la nube. (c’)

- Práctica 12. Queue Based Load Leveling Pattern
- Analice las clases de la aplicación.
- Analice el funcionamiento de la aplicación cuando el perfil activo es “simple”.
- Implemente el desacoplamiento de clases mediante un nuevo perfil (“queued-based”) mediante una clase **MessageQueue** que contenga una estructura **BlockingQueue<Message>** la cual sea utilizada como “cola” **FIFO** para la comunicación asíncrona entre los componentes productor y consumidor.
- Práctica guiada por instructor.



iv.v Principios de diseño para aplicaciones en la nube. (d'')

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Realizar commutación por error.**
 - Despliegue replicas de los servicios, utilice un balanceador para reintentar la llamada remota a otra instancia (replica del servicio).
 - **Compence transacciones con error.**
 - Evite transacciones distribuidas, debido a que requiere coordinacion a través de servicios y recursos compartidos.
 - Componga transacciones individuales atómicas, que sean fáciles de deshacer. Utilice transacciones de compensación para deshacer cualquier paso previo completado en caso de error.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (e'')

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Compence transacciones con error.**
 - Implemente eventual consistencia mediante compensación de transacciones con error.
 - **Saga Pattern**
 - Implemente el patrón Saga para ejecutar compensación de transacciones.
 - Es posible implementar el patrón Saga mediante Orquestación y Coreografía.

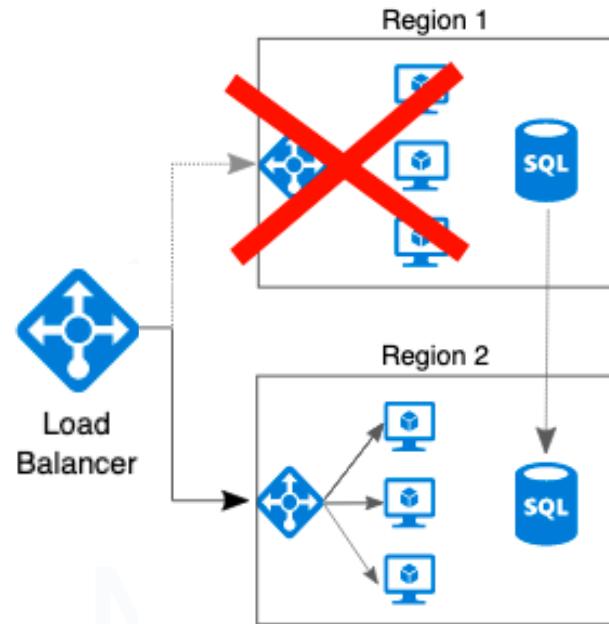
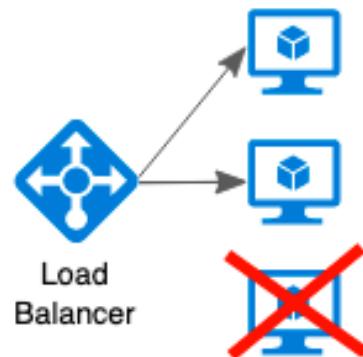


+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (f'')

- Comutación por error.





iv.v Principios de diseño para aplicaciones en la nube. (g’’)

- Compensating Transaction Pattern.
- **Categorías:** Resistencia, escalabilidad, mensajería y disponibilidad.
 - Deshaga o ejecute “**rollback**” sobre los pasos ejecutados de un trabajo que se ha realizado de forma distribuida realizando una operación transaccional eventualmente consistente.
 - En aplicaciones en la nube se sugiere implementar eventual consistencia en lugar de transacciones distribuidas mediante “**two-phase commit**” (2PC).



iv.v Principios de diseño para aplicaciones en la nube. (h’')

- Compensating Transaction Pattern.
- **Intención.**
 - Habilitar eventual consistencia entre servicios independientes, mediante compensación de transacciones donde se promueva deshacer los cambios aplicados por un flujo no transaccional distribuido.
 - Permitir diseñar un flujo de trabajo complejo como una serie de pequeños pasos transaccionales de forma local, fáciles de deshacer en caso de algún error.
 - El deshacer los cambios previamente aplicados no significa eliminarlos sino, ejecutar las operaciones correspondientes de compensación.



iv.v Principios de diseño para aplicaciones en la nube. (i’')

- Compensating Transaction Pattern.
- **Intención.**
 - La compensación de transacciones no significa reestablecer el estado de alguna entidad a su estado anterior debido a que podría estarse sobre-escribiendo algún cambio realizado por otra operación concurrente del mismo flujo de trabajo.
 - El flujo de compensación de transacciones debe compensar todos los pasos previos ejecutados de forma inversa a como fueron ejecutados para mantener una eventual consistencia.



iv.v Principios de diseño para aplicaciones en la nube. (j’')

- Compensating Transaction Pattern.
- **Aplicabilidad.**
 - Utilice este patrón solo para las operaciones realizadas de un flujo ejecutado de forma distribuida que deben deshacerse si algún paso falla.
 - De ser posible, diseñe soluciones que evite la complejidad de requerir compensación de transacciones.



iv.v Principios de diseño para aplicaciones en la nube. (k'')

- Compensating Transaction Pattern.
- **Ventajas:**
 - Implementar eventual consistencia entre servicios distribuidos.
- **Desventajas:**
 - Dificultad para determinar que un paso en el flujo del proceso distribuido a fallado.
 - Definición lógica definición de compensación de transacciones es dependiente del modelo de negocio.
 - Dificultad para la definición de compensación de transacciones como ejecución de comandos idempotentes.



+

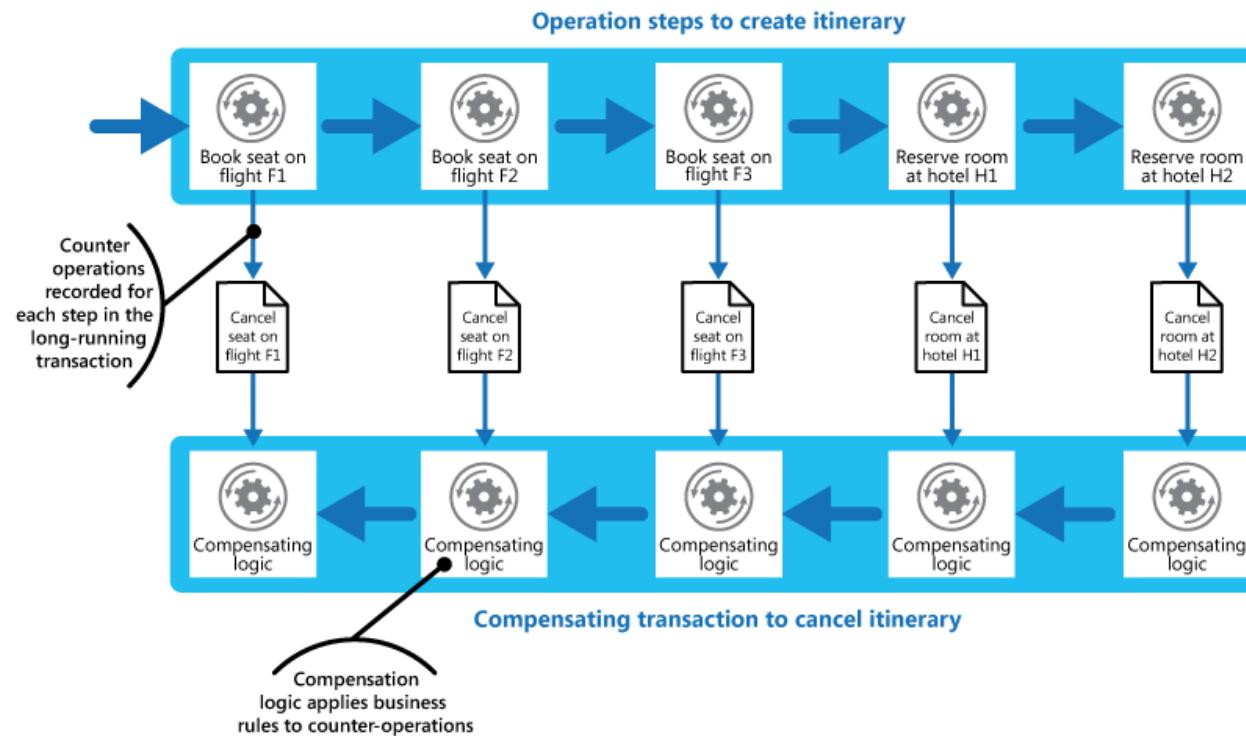
NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (I’')

- Compensating Transaction Pattern.
- **Desventajas:**
 - La ejecución de una operación de compensación de transacciones puede fallar también.
 - Dificultad para diseñar compensación de transacciones que sean resistentes a fallos.
 - Existen casos donde no es posible realizar la compensación de transacciones mas que de forma manual.

iv.v Principios de diseño para aplicaciones en la nube. (m'')

- Compensating Transaction Pattern.





+

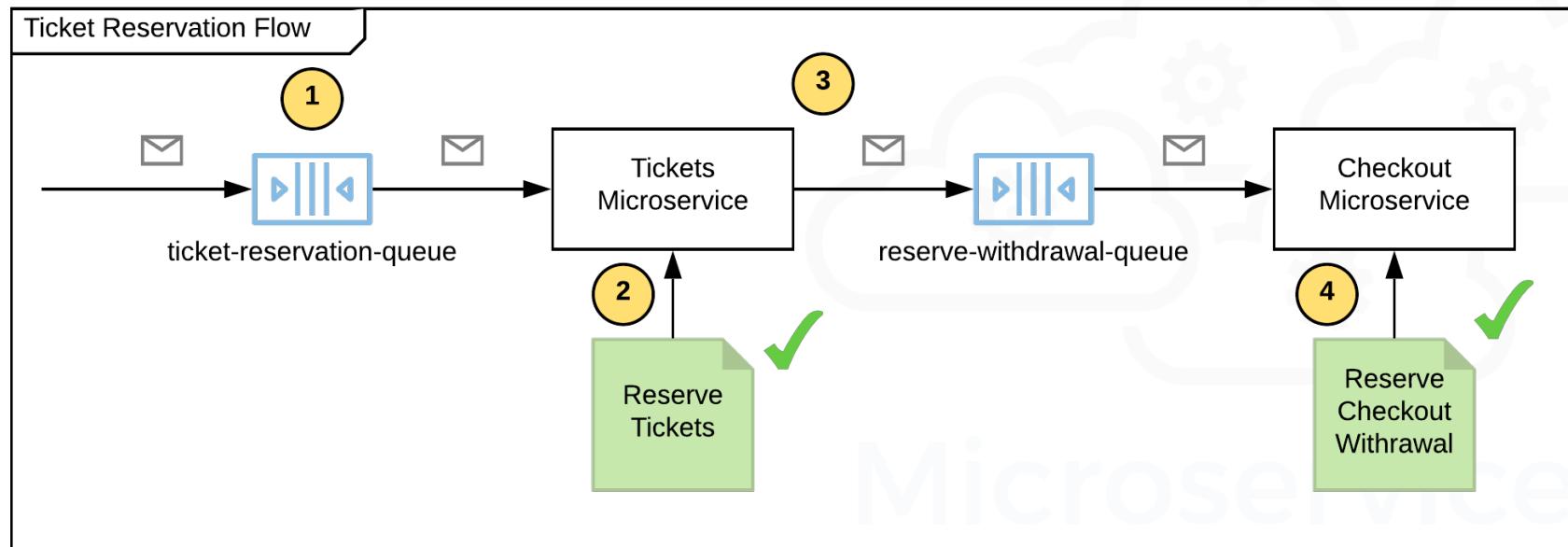
NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (n")

- Práctica 13. Compensation Transaction Pattern
- Analiza la aplicación **13-Checkout-Microservice** y **13-Tickets-Microservice**.
- Inicia el servidor **ActiveMQ** embebido del proyecto **0-Embedded-ActiveMQ-Broker-Service**.
- Importar los proyectos **13-Checkout-Microservice** y **13-Tickets-Microservice** en STS.
- Analizar el caso de uso, ver imágenes del siguiente slide.

iv.v Principios de diseño para aplicaciones en la nube. (ñ”)

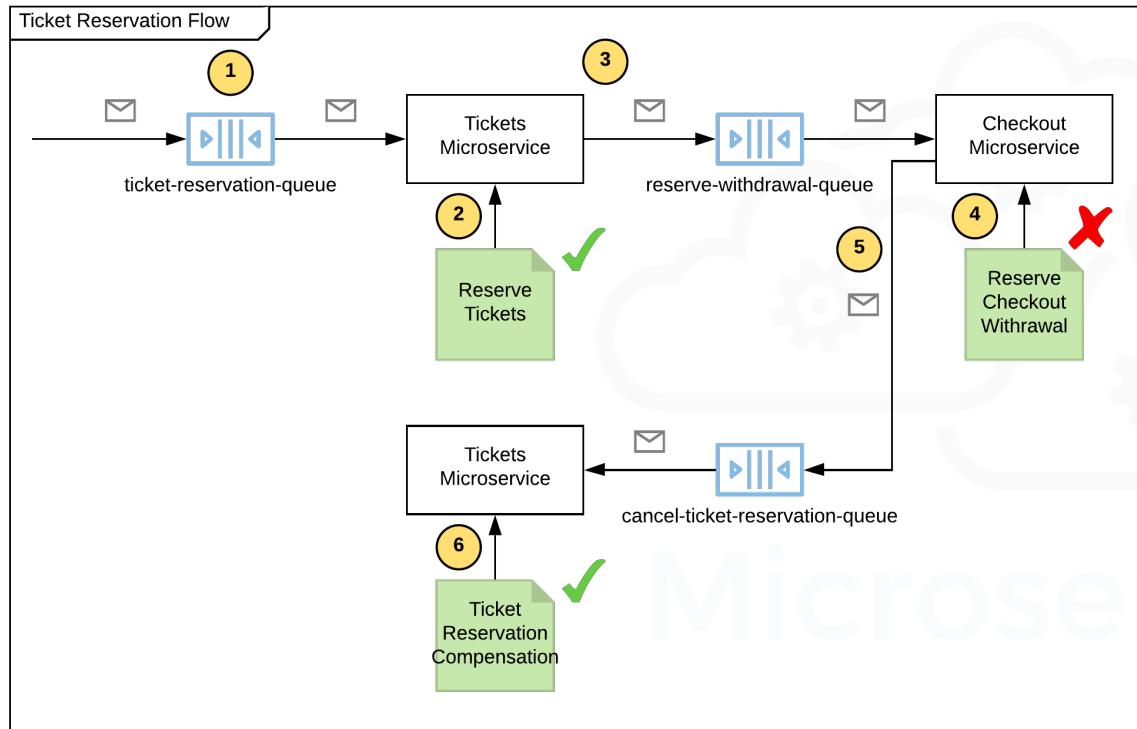
- Práctica 13. Compensation Transaction Pattern

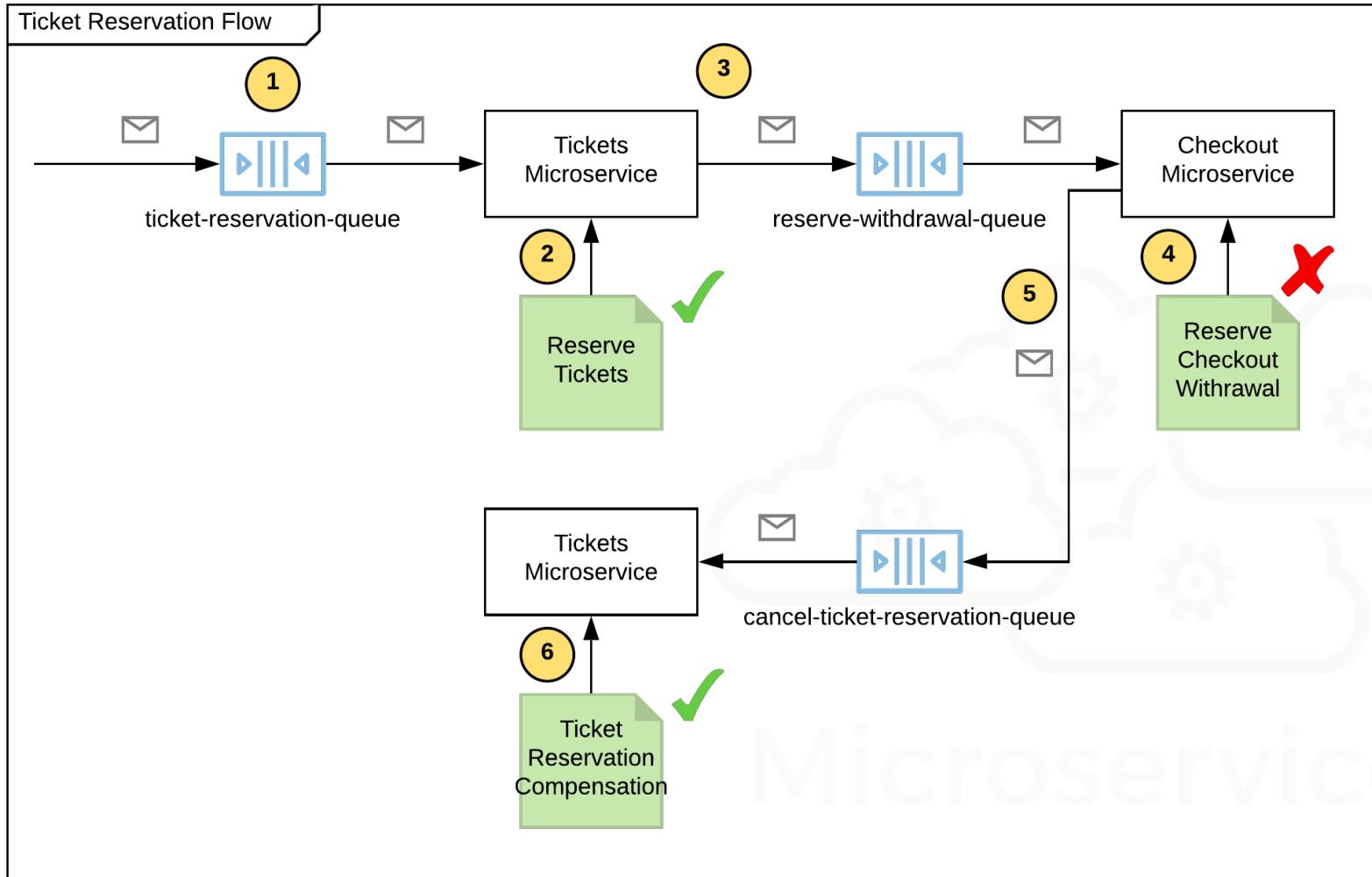




iv.v Principios de diseño para aplicaciones en la nube. (o")

- Práctica 13. Compensation Transaction Pattern







+

iv.v Principios de diseño para aplicaciones en la nube. (p'')

- **Práctica 13. Compensation Transaction Pattern**
- En el microservicio **13-Tickets-Microservice** implementa el "listener", sobre la clase **TicketsQueueListener**, de los eventos **TicketReservationEvent** y **CancelTicketReservationEvent** del paquete **com.consulting.mgt.springboot.practica13.compensatingtransactions.tickets.queue.event**.
- Analiza la clase de configuración **TicketsMicroserviceApplicationConfig**.
- Analiza la clase **TicketsMicroserviceQueues** del paquete **com.consulting.mgt.springboot.practica13.compensatingtransactions.tickets.queue**.



iv.v Principios de diseño para aplicaciones en la nube. (q’’)

- Práctica 13. Compensation Transaction Pattern
- Sobre la clase **TicketsQueueListener**, al momento de procesar el evento **TicketReservationEvent**, dispara el evento **ReservationCheckoutWithdrawalEvent**, el cual deberá ser atendido por el microservicio **13-Checkout-Microservice**.
- Analizar la clase **CheckoutMicroserviceQueues** del paquete **com.consulting.mgt.springboot.practica13.compensatingtransactions.checkout.queue**.
- Implementar la clase de servicio **CheckoutMicroserviceQueueProducer**, encargada de enviar, disparar, el evento **ReservationCheckoutWithdrawalEvent**.



+

iv.v Principios de diseño para aplicaciones en la nube. (r'')

- Práctica 13. Compensation Transaction Pattern
- Analiza la clase **AppDemoService**, del paquete
`com.consulting.mgt.springboot.practica13.compensatingtransactions.tickets.appdemo.service`.
- Prueba tu trabajo implementando un bean **CommandLineRunner**, sobre la clase principal del proyecto, que simule el envío de un evento para la reservación de un ticket y otro para el envió de un evento para la cancelación de un ticket. Utiliza perfiles.
- El envío del evento de reservación de un ticket debe disparar el evento de reservación de retiro de dinero.



iv.v Principios de diseño para aplicaciones en la nube. (s'')

- **Práctica 13. Compensation Transaction Pattern**
- Sobre el microservicio **13-Checkout-Microservice** implemente el "listener", sobre la clase **CheckoutQueueListener**, del evento **ReservationCheckoutWithdrawalEvent**.
- Analiza el proyecto e implementa las funcionalidades faltantes para implementar el envío y recepción de eventos para completar el flujo descrito en el diagrama del caso de uso.
- No es necesario implementar persistencia mediante Spring Data JPA, sólo es requerido realizar el modelo de integración Saga Pattern para implementar compensación de transacciones.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (t'')

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Degrade la funcionalidad del servicio.**
 - A veces los errores no son posibles de solucionar en un corto plazo, degrade la funcionalidad del servicio a una versión reducida que siga siendo útil y procese la operación completa en otro momento de forma asíncrona, tal como operaciones batch.



iv.v Principios de diseño para aplicaciones en la nube. (u’')

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Limite los clientes.**
 - En algunas ocasiones un número reducido de usuarios sobrecargan la aplicación reduciendo su disponibilidad para otros usuarios.
 - **Throttling Pattern**
 - Limite los clientes durante un periodo de tiempo determinado.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (v'')

- Throttling Pattern.
- **Categorías:** Rendimiento, escalabilidad y disponibilidad.
 - Asegurar que un cliente o consumidor de un servicio no le sea permitido el acceso al mismo, más de lo establecido en su límite de consumo.



iv.v Principios de diseño para aplicaciones en la nube. (w'')

- Throttling Pattern.
- **Intención.**
 - Controlar el consumo de recursos utilizados por una instancia de la aplicación o de un aplicativo consumidor particular.
 - Permitir que el sistema continúe funcionando, sin degradar los acuerdos de nivel de servicio (SLAs) inclusive aún cuando la demanda del servicio sea extensa para los recursos solicitados.



iv.v Principios de diseño para aplicaciones en la nube. (x'')

- Throttling Pattern.
- **Aplicabilidad.**
 - Utilice este patrón cuando servicios críticos expuestos, que son consumidos con alta demanda, necesiten asegurar el acuerdo de nivel de servicio (SLAs) pactado.
 - Cuando se desee prevenir de un único cliente de monopolizar el consumo del servicio provisto por la aplicación.
 - Asegurar el funcionamiento del servicio con cargas de trabajo masivas.
 - Mejorar el costo-beneficio de los servicios productivos limitando el consumo desmesurado de recursos.

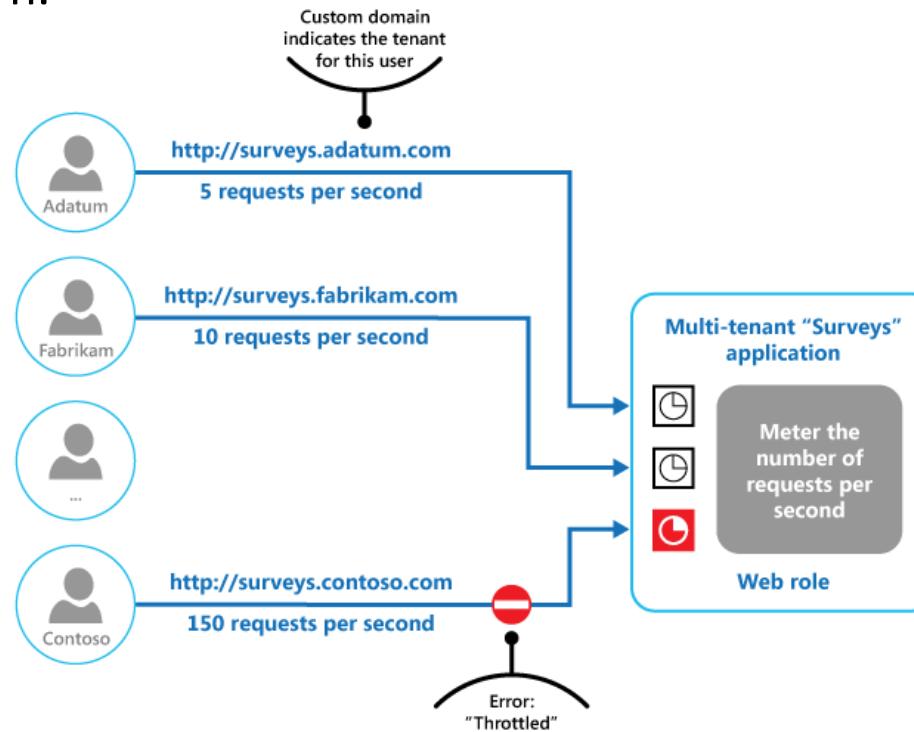


iv.v Principios de diseño para aplicaciones en la nube. (y'')

- Throttling Pattern.
- **Ventajas:**
 - Evitar la sobrecarga y degradación de servicio.
 - Mantener los acuerdos de nivel de servicios (SLAs) pactados.
 - Permite utilizar limitación de clientes de forma temporal mientras el servicio escala.
- **Desventajas:**
 - Debe ser considerado en el inicio del desarrollo del servicio dada su dificultad para su implementación una vez implementados los servicios.

iv.v Principios de diseño para aplicaciones en la nube. (z'')

- Throttling Pattern.





iv.v Principios de diseño para aplicaciones en la nube. (a'')

- Práctica 14. Throttling Pattern
- Analiza la aplicación **14-Throttling-Microservice**.
- Ingresar a la ruta: **{tu-workspace}/14-Throttling-Microservice**
- Importar el proyecto **14-Throttling-Microservice** en STS.
- Analiza las clases **User** y **Users** del paquete
com.consulting.mgt.springboot.practica14.throttling.restcontroller.model.
- Analiza la clase **UserController** del paquete
com.consulting.mgt.springboot.practica14.throttling.restcontroller.



iv.v Principios de diseño para aplicaciones en la nube. (b'')

- Práctica 14. Throttling Pattern
- Sobre la clase de configuración **ApplicationConfig** define un bean **Users** y un Bean **List<User>** con 3 elementos **User** dentro de la lista. Inyecta el bean de tipo **List<User>** sobre el bean **Users**.
- Analiza la clase **ValidateAuthenticationFilter** del paquete **com.consulting.mgt.springboot.practica14.throttling.filters**, ¿Cuál es el header requerido en el request para que el filtro continue su ejecución hacia el recurso solicitado?



iv.v Principios de diseño para aplicaciones en la nube. (c'')

- Práctica 14. Throttling Pattern
- Asigna la propiedad servlet context-path el valor “/throttling-service” y establece el puerto de la aplicación web como **8081**.
- Compila desde línea de comandos el proyecto, mediante comando “**mvn clean package**” y ejecuta la aplicación mediante comando “**java -jar**”.
- Utiliza el un cliente HTTP, como Postman, RestClient, HTTPie o cURL para ejecutar una petición al servicio <http://localhost:8081/throttling-service/users>.
 - curl -GET -X http http://localhost:8081/throttling-service/users
 - http <http://localhost:8081/throttling-service/users>
- ¿Qué respuesta devuelve el servidor?



iv.v Principios de diseño para aplicaciones en la nube. (d'')

- **Práctica 14. Throttling Pattern**
- Envía el header requerido para poder acceder al recurso. Prueba de nuevo.
 - curl -GET -H "x-authenticated-id: ivan" http://localhost:8081/throttling-service/users
 - http http://localhost:8081/throttling-service/users X-AUTHENTICATED-ID:Ivan
- Es posible ejecutar "n" cantidad de veces la petición previa.
- A continuación limite el consumo del cliente al API a 1 petición por segundo.



iv.v Principios de diseño para aplicaciones en la nube. (e'')

- **Práctica 14. Throttling Pattern**
- Analiza la clase **Tenant** y **CallsCount** del paquete **com.consulting.mgt.springboot.practica14.throttling.throttler**.
- Analiza la interface **Throttler** y realiza la implementación de **ThrottleTimerImpl** en el paquete **com.consulting.mgt.springboot.practica14.throttling.throttler**.
- Sobre la clase **ThrottlingConfig** define los beans **CallsCount** y **Throttler**, recuerda configurar el **Throttler** para que el periodo de regulación de peticiones sea de 1 segundo; a su vez define un bean de tipo **Map<String, Tenant>** donde almacenes un conjunto de tenants (clientes, ocupantes) referenciados por **llave = “nombre del tenant”** y **valor = “el objeto tenant”**.



+

iv.v Principios de diseño para aplicaciones en la nube. (f'')

- Práctica 14. Throttling Pattern
- Analiza las clases **TenantException** y **ThrottlerException** del paquete **com.consulting.mgt.springboot.practica14.throttling.throttler.exception**.
- Implementa el filtro **ThrottlingFilter**, del paquete **com.consulting.mgt.springboot.practica14.throttling.filters**, el cual obtenga el valor del header “**x-authenticated-id**” y ubique el **Tenant** correspondiente mediante el **Mapa<String, Tenant>** injectado. Posteriormente obtenga del bean **CallsCount** el número de cuentas de ese tenant en particular y comparelo con el numero de llamadas permitidas para el tenant correspondiente. Límite el consumo del API, evitando llamar al metodo **doFilter** del objeto **FilterChain**.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (g'')

- **Práctica 14. Throttling Pattern**
- Ejecute la aplicación nuevamente, corrobore que se ha limitado el consumo del API mediante la verificación del Tenant y la cuenta de llamadas permitidas por segundo.



iv.v Principios de diseño para aplicaciones en la nube. (h'')

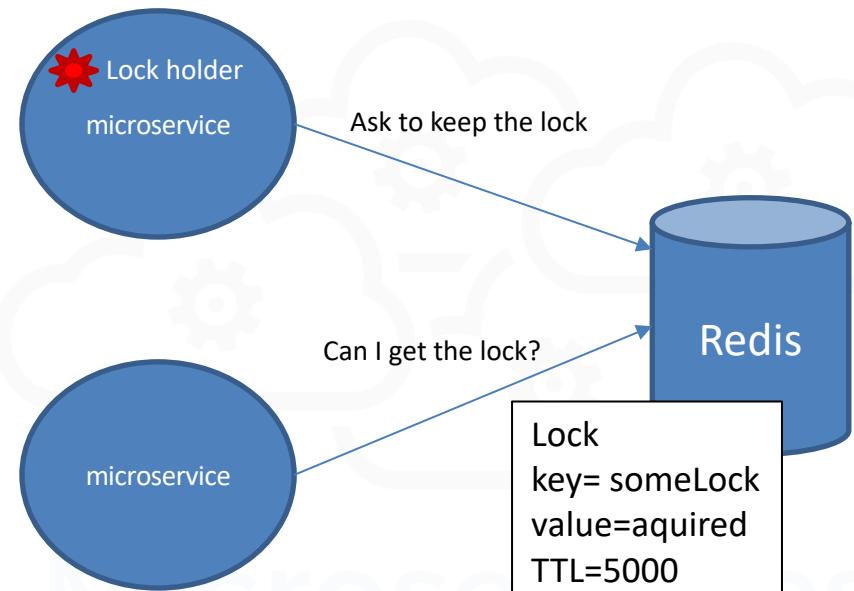
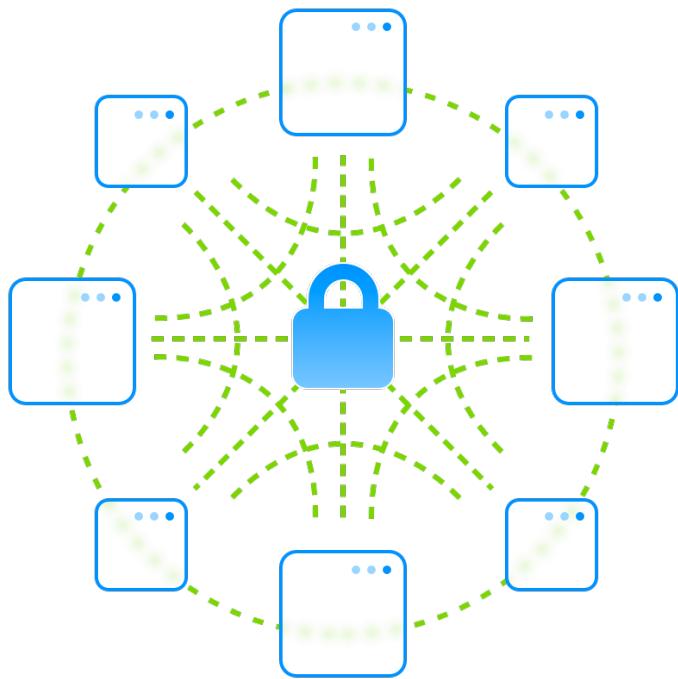
- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Bloquee clientes no válidos.**
 - El hecho de limitar clientes no significa que esté actuando de manera malintencionada, significa que el cliente ha superado su cuota de servicio.
 - Bloquee clientes que alcanzan su cuota de servicio de manera constante.
 - Proporcione un mecanismo de desbloqueo para el cliente en cuestión.

iv.v Principios de diseño para aplicaciones en la nube. (i'')

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Implemente Leader Election.**
 - Utilice el patrón "**leader election**" para seleccionar un coordinador cuando se necesite implementar una tarea de forma paralela entre varias instancias las cuales acceden a recursos de forma aleatoria y compartida.
 - **Distributed Lock**
 - Permite asegurar que una tarea sea ejecutada una única vez por un conjunto de instancias que la disparan al mismo tiempo asegurando la alta disponibilidad del servicio.

iv.v Principios de diseño para aplicaciones en la nube. (j'')

- Distributed Lock.





+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (k'')

- Diseñe para la recuperación automática.
- Recomendaciones:
 - **Realice pruebas con inserción de errores.**
 - Con frecuencia la ruta “happy-path” siempre es efectivamente probada.
 - Realice extensivas pruebas insertando errores en los puntos críticos del flujo funcional del servicio.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube.

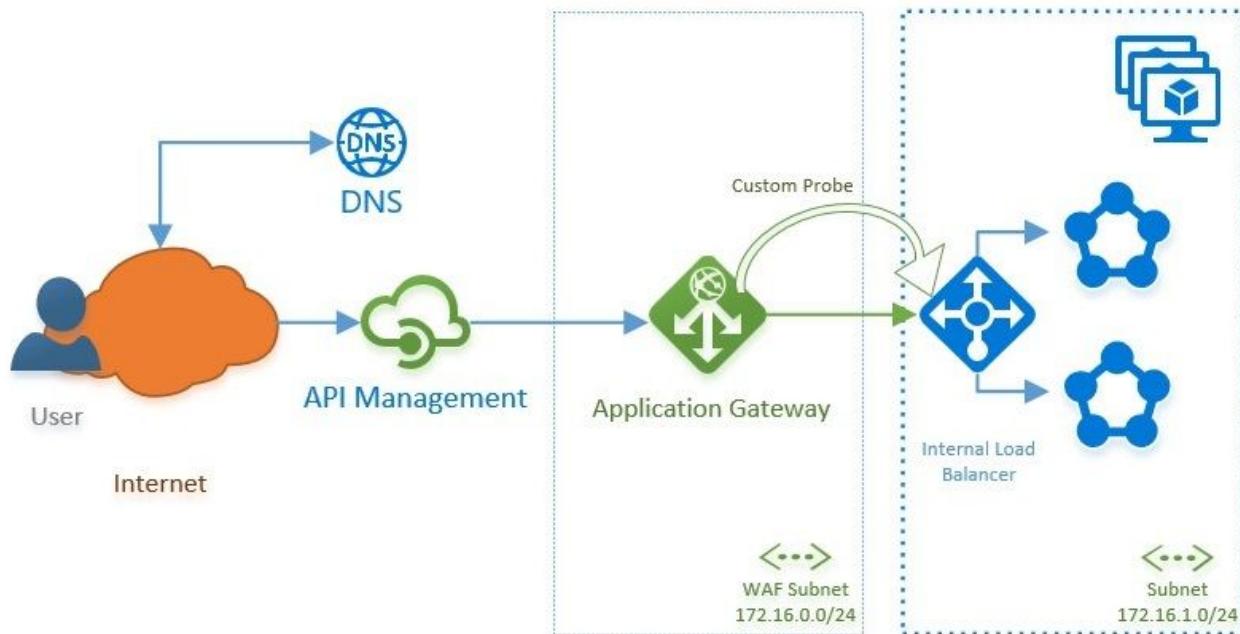
- Diseñe para la recuperación automática.
- **Haga todo redundante.**
- Minimizar la coordinación.
- Diseñe para facilitar el escalamiento horizontal.
- Particione alrededor de límites.
- Diseñe para las operaciones.
- Use servicios administrados.
- Use el repositorio correcto para el trabajo correcto.
- Diseñe para evolucionar el servicio.
- Desarrolle considerando las necesidades del negocio.

iv.v Principios de diseño para aplicaciones en la nube. (a)

- Haga todo redundante.
- Las aplicaciones resistentes (resilientes) evitan tener puntos únicos de falla.
- Recomendaciones:
 - **Replique los servicios y utilice un balanceador de carga.**
 - Nunca implemente un solo contenedor, maquina virtual o servicio en la nube para cargas de trabajo críticas.
 - Utilice un balanceador de carga para distribuir el trabajo entre los diferentes nodos o instancias del servicio.

iv.v Principios de diseño para aplicaciones en la nube. (b)

- Replique los servicios y utilice un balanceador de carga.





iv.v Principios de diseño para aplicaciones en la nube. (c)

- Haga todo redundante.
- Recomendaciones:
 - **Considere los requerimientos de negocio.**
 - Una baja cantidad de redundancia puede afectar los SLAs (Service Level Agreement) que requiere el negocio.
 - Una alta cantidad de redundancia por servicio puede afectar el ROI (Retorno de inversión) del negocio.
 - Integre redundancia por región geográfica.
 - Implemente conmutación por error llamando a procedimientos remotos en reintentos a otras regiones.



+

iv.v Principios de diseño para aplicaciones en la nube. (d)

- Haga todo redundante.
- Recomendaciones:
 - **Replique bases de datos.**
 - Replique todas las bases de datos de sus servicios.
 - **Habilite replicación geográfica.**
 - Conmute por error en regiones secundarias.
 - **Use particiones para conseguir disponibilidad.**
 - Utilice particiones o “sharding” para habilitar alta disponibilidad de los datos ya que aunque una partición deje de funcionar, puede acceder a las demás particiones y solo provocará la interrupción de un subconjunto de transacciones/datos.

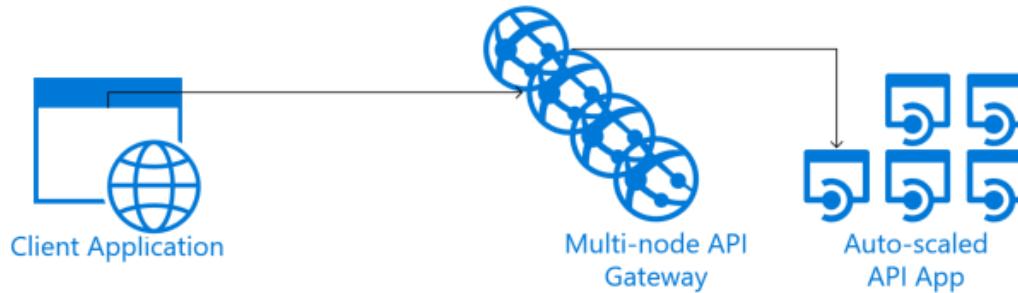


iv.v Principios de diseño para aplicaciones en la nube. (e)

- Haga todo redundante.
- Recomendaciones:
 - **Inlcuya redundancia en el API Gateway.**
 - Revise los SLAs de la herramienta API Gateway utilizada y verifique si cumple con los requerimientos de negocio.
 - Replique el API Gateway en múltiples instancias y utilice un balanceador de carga del lado del cliente para distribuir la carga en los múltiples llamados a los servicios a través del API Gateway siempre y cuando los servicios sean sin estado.

iv.v Principios de diseño para aplicaciones en la nube. (f)

- Incluya redundancia en el API Gateway.





+

iv.v Principios de diseño para aplicaciones en la nube.

- Diseñe para la recuperación automática.
- Haga todo redundante.
- **Minimizar la coordinación.**
- Diseñe para facilitar el escalamiento horizontal.
- Particione alrededor de límites.
- Diseñe para las operaciones.
- Use servicios administrados.
- Use el repositorio correcto para el trabajo correcto.
- Diseñe para evolucionar el servicio.
- Desarrolle considerando las necesidades del negocio.



iv.v Principios de diseño para aplicaciones en la nube. (a)

- Minimizar la coordinación.
- Para asegurar la escalabilidad y confiabilidad de una arquitectura de microservicios, es necesario ejecutar servicios, bases de datos, procesos, etc en múltiples instancias, es decir, con replica.
- Será necesaria implementar coordinación entre servicios para asegurar la integridad de los datos y evitar los bloqueos de acceso a los recursos.



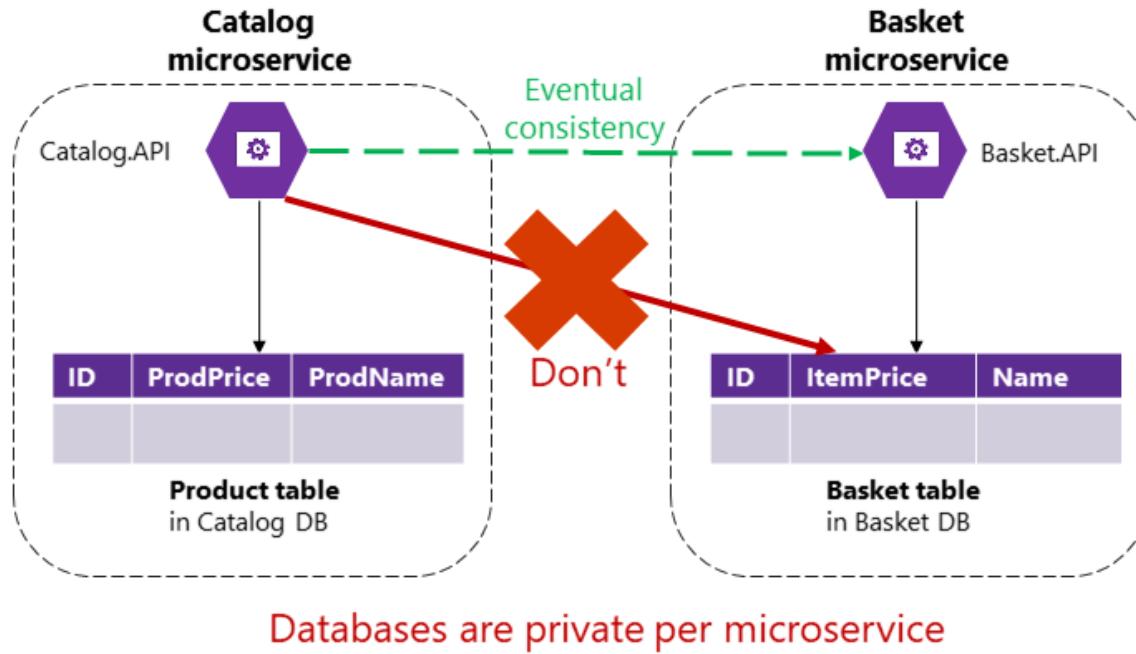
+

iv.v Principios de diseño para aplicaciones en la nube. (b)

- Minimizar la coordinación.
- Recomendaciones:
 - **Adoptar la eventual consistencia.**
 - Cuando los datos en las transacciones son distribuidas, es necesario implementar coordinación para exigir garantizar la consistencia.
 - Evitar 2PC (two-phase commit pattern) debido a que maximiza la coordinación para implementar transacciones distribuidas.
 - Es preferible integrar eventual consistencia dividiendo operaciones transaccionales en otras más pequeñas y habilitar la **compensación de transacciones** en caso de error.

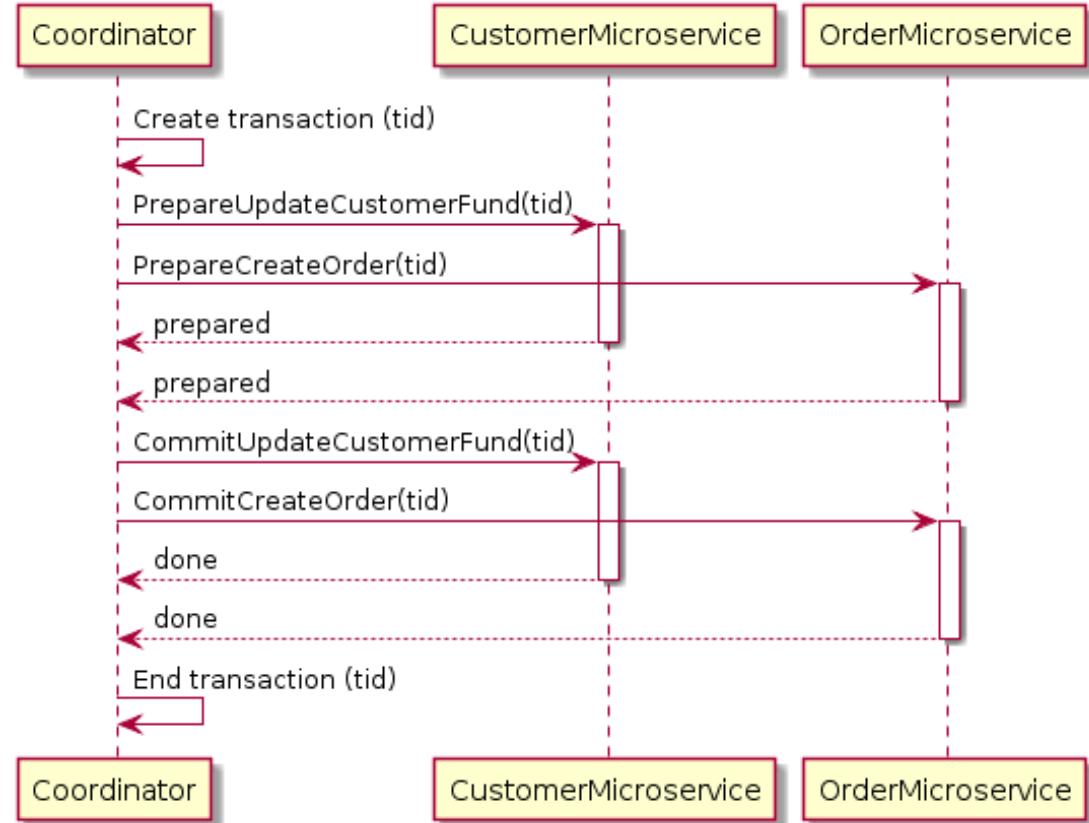
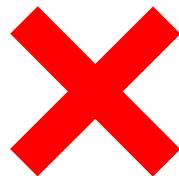
iv.v Principios de diseño para aplicaciones en la nube. (c)

- Adoptar la eventual consistencia.



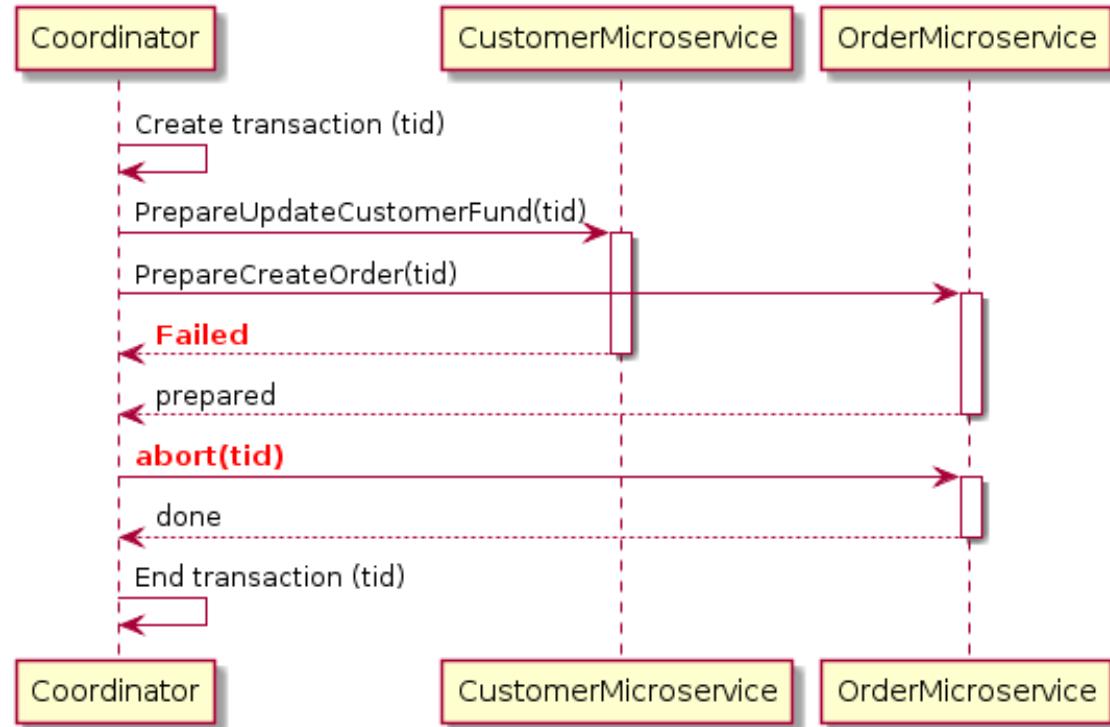
iv.v Principios de diseño para aplicaciones en la nube. (d)

- Two-Phase Commit (a).



iv.v Principios de diseño para aplicaciones en la nube. (e)

- Two-Phase Commit (b).



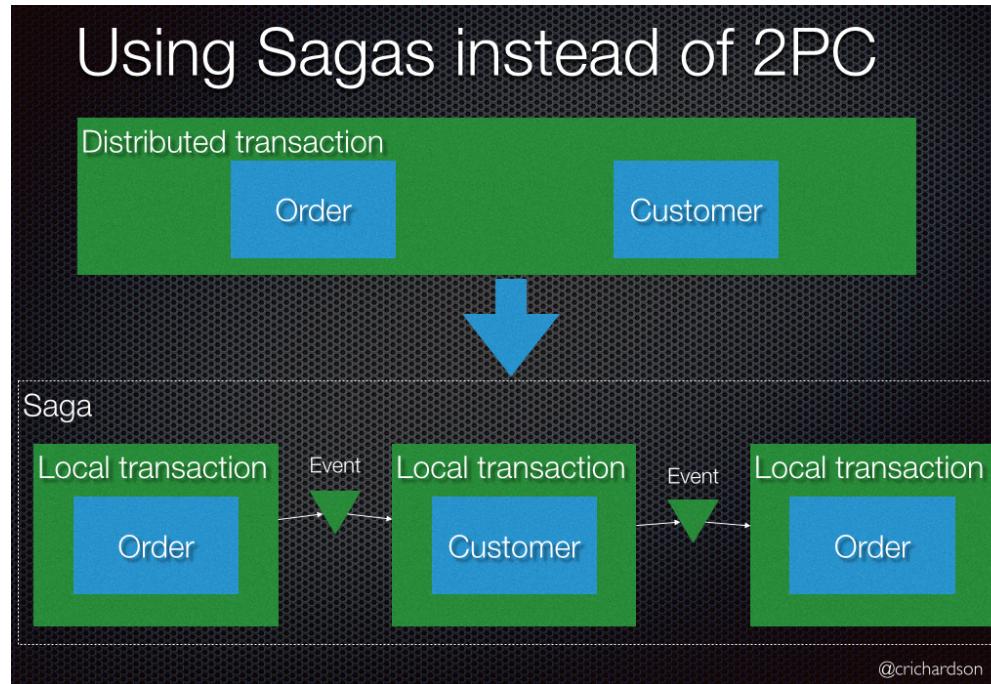


iv.v Principios de diseño para aplicaciones en la nube. (f)

- Minimizar la coordinación.
- Recomendaciones:
 - **Implementar arquitecturas dirigidas por eventos para sincronizar estado (Event-Driven Architecture).**
 - Desacoplar al productor y consumidor de eventos minimiza la coordinación entre servicios.
 - EDA habilita minimizar la coordinación para la sincronización de estado entre servicios.
 - **Saga Pattern**
 - Altamente recomendable para habilitar la **compensación de transacciones** para deshacer cambios transaccionales.

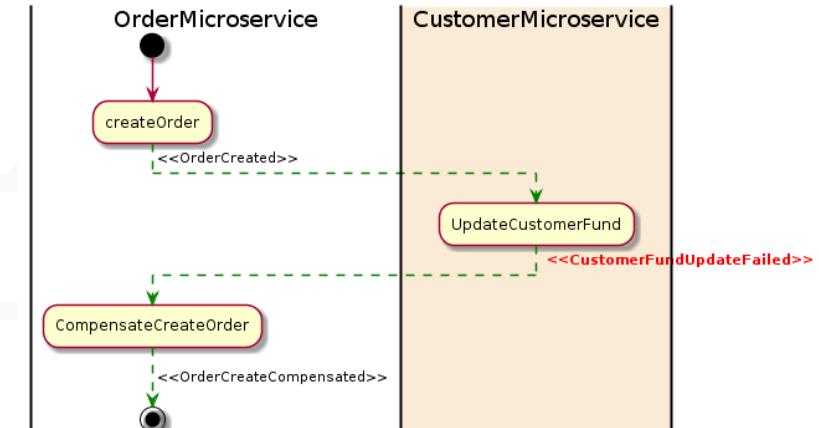
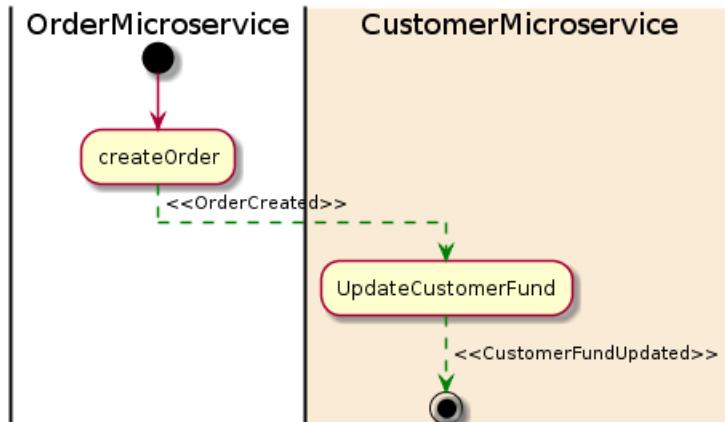
iv.v Principios de diseño para aplicaciones en la nube. (g)

- Saga Pattern vs 2PC.



iv.v Principios de diseño para aplicaciones en la nube. (h)

- Saga Pattern.



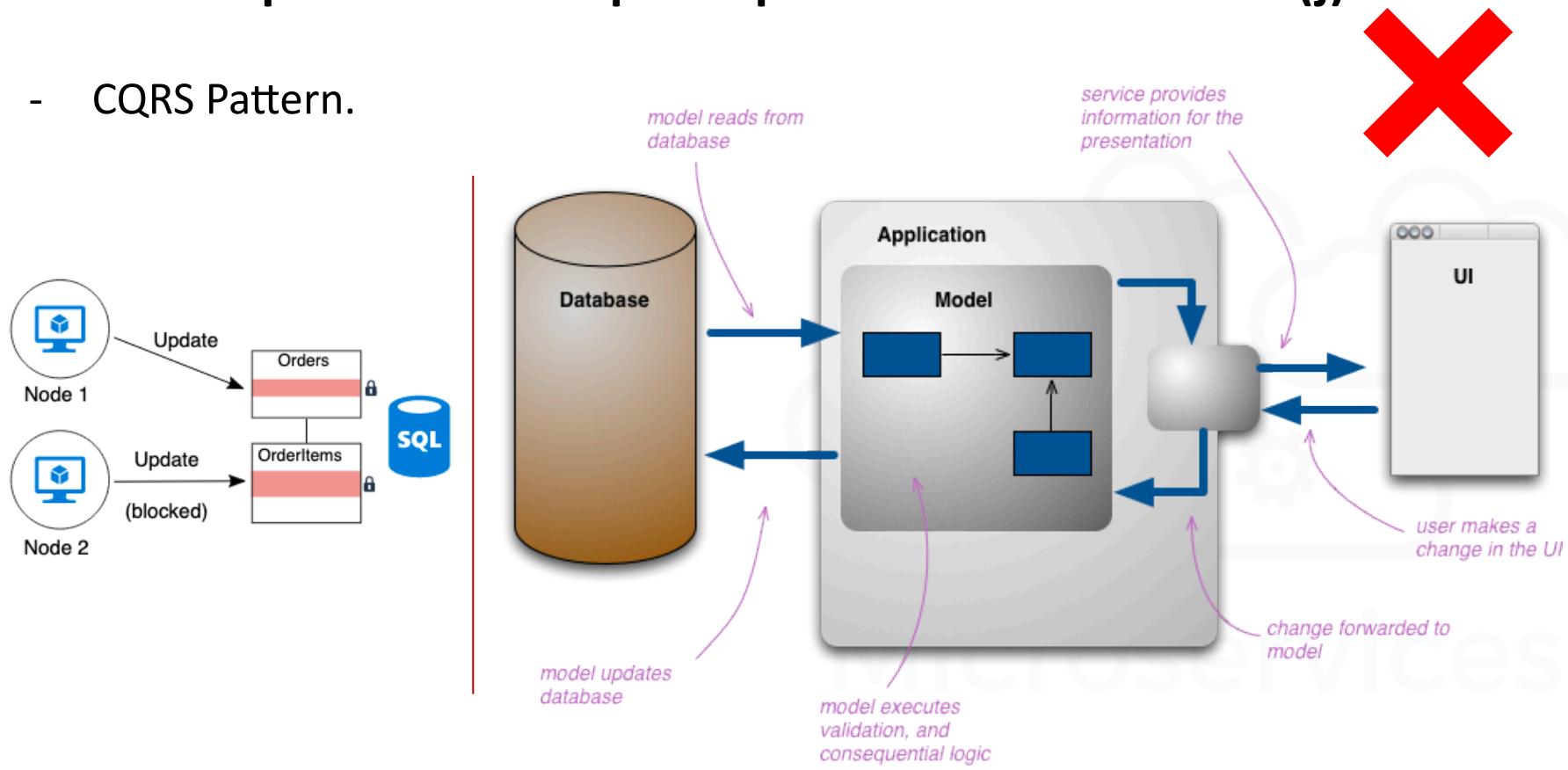


iv.v Principios de diseño para aplicaciones en la nube. (i)

- Minimizar la coordinación.
- Recomendaciones:
 - **Implementar patrones como CQRS y Event Sourcing.**
 - Evitar el bloqueo de dos o más instancias cuando tratan acceder y ejecutar operaciones de lectura y/o escritura sobre un mismo set de datos.
 - **CQRS Pattern (Command and Query Responsibility Segregation)**
 - Separación de operaciones de lectura y escritura.
 - **Event Sourcing Pattern**
 - Almacenar los cambios de estado de una entidad como una serie de eventos que pueden ser recuperables en el tiempo.

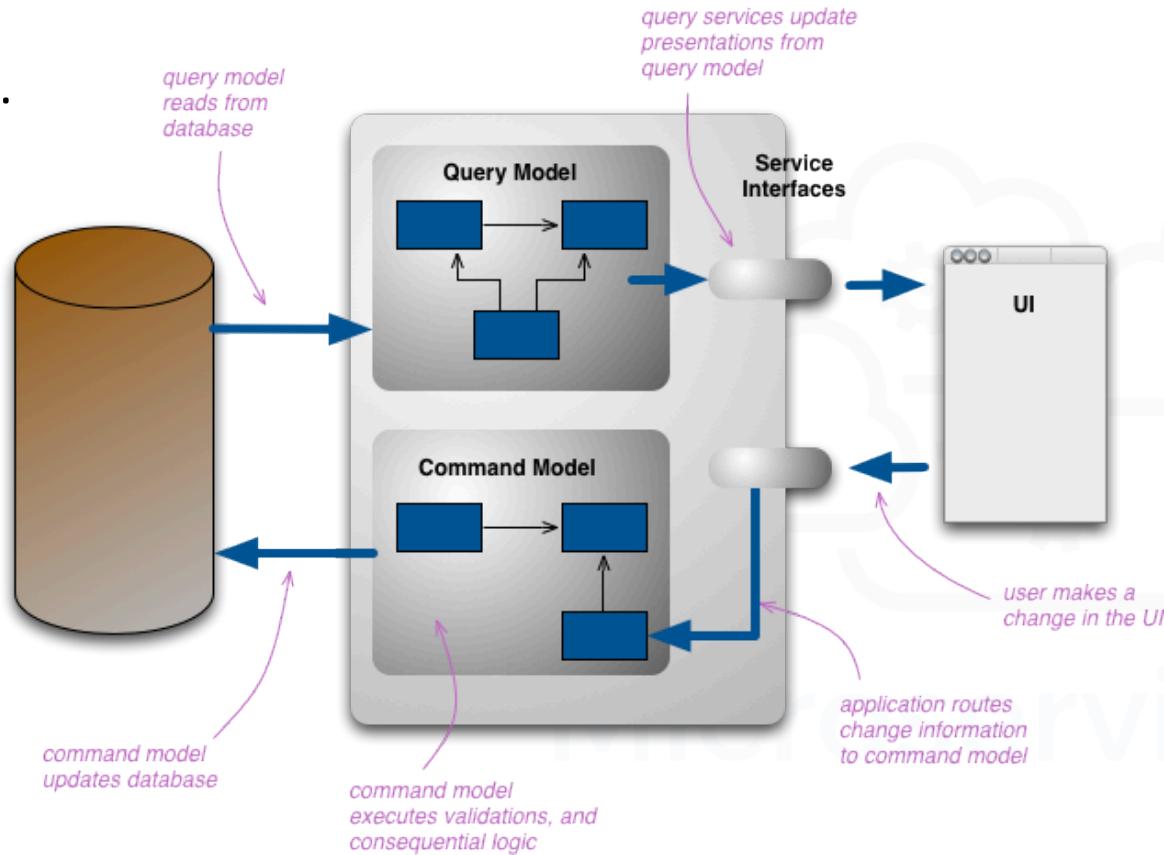
iv.v Principios de diseño para aplicaciones en la nube. (j)

- CQRS Pattern.



iv.v Principios de diseño para aplicaciones en la nube. (k)

- CQRS Pattern.





iv.v Principios de diseño para aplicaciones en la nube. (I)

- Event Sourcing Pattern.
- **Categorías:** Rendimiento, escalabilidad y administración de datos.
 - Las aplicaciones comúnmente almacenan el estado de un objeto sin embargo, existen casuísticas que no solo requieren saber el estado de un objeto en particular sino también, como llegó dicho objeto a ese estado.
 - **Event Sourcing Pattern** asegura que los cambios aplicados a un objeto son almacenados como una secuencia de eventos, los cuales pueden ser consultados y reconstruidos, como parte de un log, en caso de alguna falla o en caso de requerir comprobar el estado final de un objeto a partir de replicar, los eventos aplicados al mismo.



iv.v Principios de diseño para aplicaciones en la nube. (m)

- Event Sourcing Pattern.
- **Intención.**
 - En lugar de almacenar sólo el estado actual de un objeto de dominio, implemente un repositorio con tipo de almacenamiento “**append-only**”, para registrar la serie completa de eventos ejecutados sobre los objetos.
 - El repositorio actúa como el sistema de registro o “**log**” el cuál, puede ser utilizado para materializar los objetos del dominio en el futuro.

iv.v Principios de diseño para aplicaciones en la nube. (n)

- Event Sourcing Pattern.
- **Intención.**
 - El repositorio o “log” de eventos, puede simplificar tareas en dominios de negocio complejos, evitando la necesidad de sincronizar el modelo de datos y/o el estado actual de un objeto; al tiempo que mejora el rendimiento, la escalabilidad y la capacidad de respuesta del sistema.
 - También brinda eventual consistencia para transacciones distribuidas, y permite mantener registros de auditoría completos y su historial que pueda habilitar acciones compensatorias.



iv.v Principios de diseño para aplicaciones en la nube. (ñ)

- Event Sourcing Pattern.
- **Aplicabilidad.**
 - Cuando es necesario capturar los cambios de estado de un objeto de dominio como una serie de eventos.
 - Cuando sea vital minimizar o eliminar conflictos de concurrencia sobre la actualización de los datos persistidos.
 - Cuando sea requerido almacenar los eventos ocurridos sobre el estado de un objeto o sistema y sea necesario reproducirlos, en el futuro, para restaurar el estado del sistema o, para aplicar cambios de tipo “**roll-back**” o, simplemente para mantener un historial o pistas de auditoria confiables.



iv.v Principios de diseño para aplicaciones en la nube. (o)

- Event Sourcing Pattern.
- **Aplicabilidad.**
 - Cuando el manejo de eventos es una funcionalidad natural de operación en el sistema; la implementación de **Event Sourcing Pattern** no requiere mayor esfuerzo.
 - Cuando sea requerido desacoplar los procesos de entrada de datos, con los de actualización de datos y desea mantener un estado consistente de los mismos.
 - Cuando requiera implementar mecanismos de eventual consistencia es fundamental la aplicabilidad de **Event Sourcing Pattern**.



iv.v Principios de diseño para aplicaciones en la nube. (p)

- Event Sourcing Pattern.
- **Ventajas:**
 - Implementar eventual consistencia entre servicios distribuidos.
 - Mantener el historial de cambios de estados sobre objetos de dominio los cuales pueden ser replicables en el futuro.
 - Fácil integración en sistemas con arquitecturas orientada a eventos.
 - Facilita la compensación de transacciones.
 - Incrementa la resistencia del sistema a fallas de software y/o hardware habilitando la recuperación del sistema mediante ejecutar la serie de eventos previos.

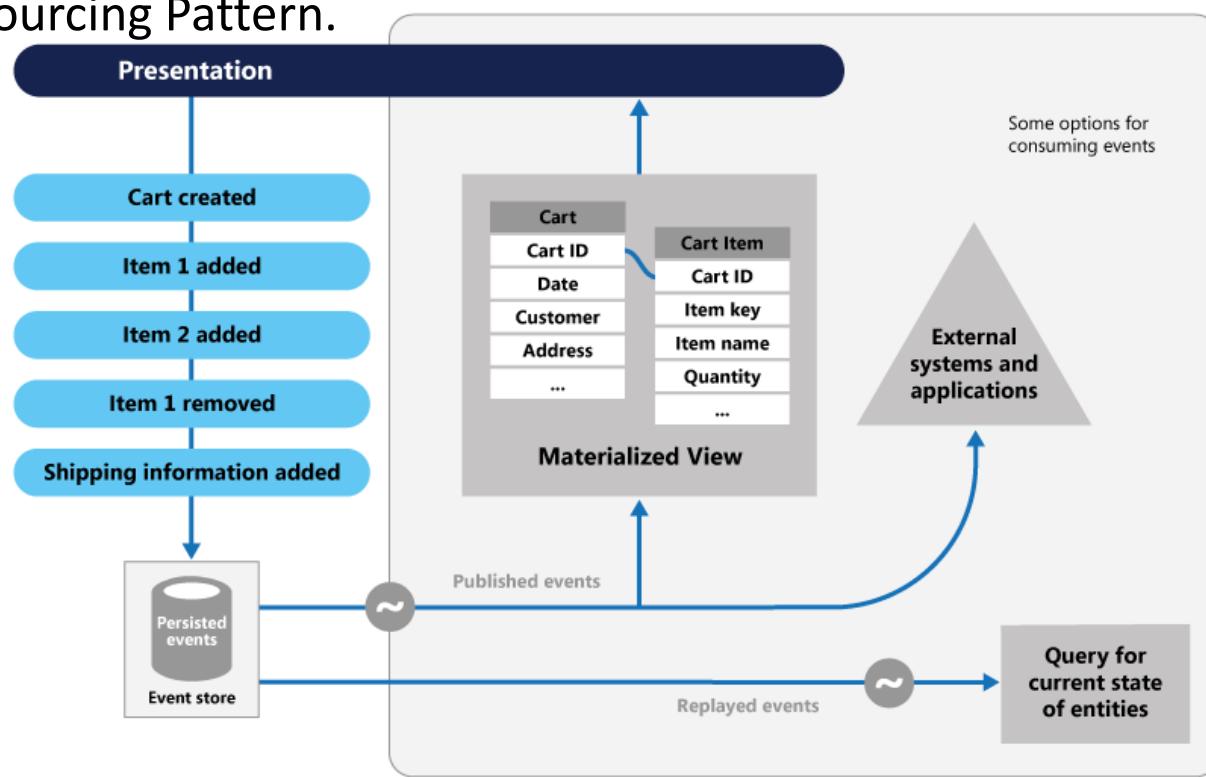


iv.v Principios de diseño para aplicaciones en la nube. (q)

- Event Sourcing Pattern.
- Desventajas:
 - Latencia de la red puede ocasionar que un evento llegue al repositorio de eventos de forma tal que sea necesaria una reconciliación de datos.
 - Dificultad para implementar consumidores de eventos de tipo “at-least-once” debido a manejar un mal diseño no **Idempotente** de servicios.

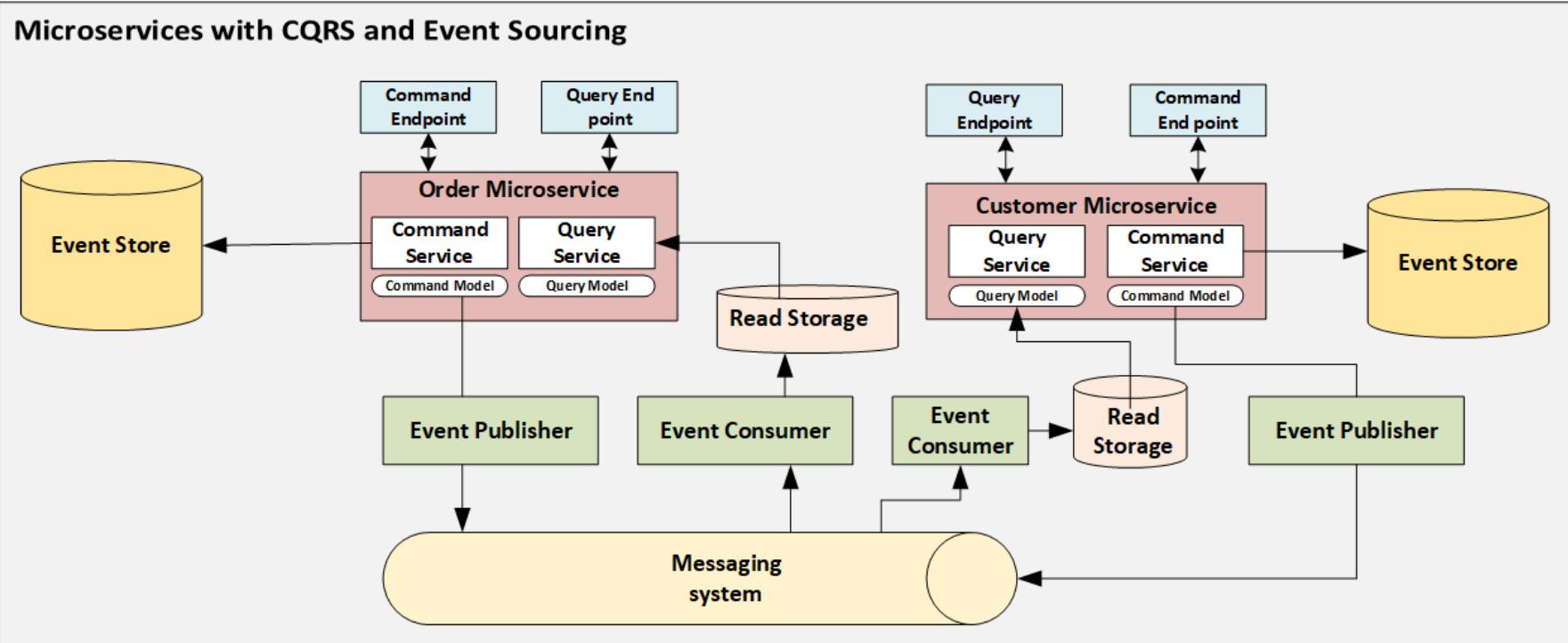
iv.v Principios de diseño para aplicaciones en la nube. (r)

- Event Sourcing Pattern.



iv.v Principios de diseño para aplicaciones en la nube. (s)

- CQRS Pattern y Event Sourcing Pattern.





+

iv.v Principios de diseño para aplicaciones en la nube. (t)

- Práctica 15. Event Sourcing Pattern
- Analiza la aplicación **15-Event-Sourcing-Microservice**. Analiza el caso de uso.
- Ingresar a la ruta: **{tu-workspace}/15-Event-Sourcing-Microservice**
- Importar el proyecto **15-Event-Sourcing-Microservice** en STS.
- Analiza las clases **AccountException** y **JournalException** del paquete **com.consulting.mgt.springboot.practica15.eventsourcing.events.exception**.
- Analiza la clase **AccountHolder** del paquete **com.consulting.mgt.springboot.practica15.eventsourcing.holder**.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (u)

- Práctica 15. Event Sourcing Pattern
- Analiza la clase **Account** del paquete
com.consulting.mgt.springboot.practica15.eventsourcing.domain.
- Analiza la clase base **DomainEvent** del paquete
com.consulting.mgt.springboot.practica15.eventsourcing.events.
- Sobre el paquete
com.consulting.mgt.springboot.practica15.eventsourcing.events.domainevents,
define los eventos:
 - AccountCreateEvent
 - MoneyDepositEvent
 - MoneyWithdrawalEvent
 - MoneyTransferEvent



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (v)

- Práctica 15. Event Sourcing Pattern
- Analiza la clase **JsonFileJournal** del paquete
com.consulting.mgt.springboot.practica15.eventsourcing.processor.
- Implementa el vean componente **DomainEventProcessor** del paquete
com.consulting.mgt.springboot.practica15.eventsourcing.processor. Guiado por instructor.
- Define el bean Gson sobre la clase de configuración **ApplicationConfig** del paquete
com.consulting.mgt.springboot.practica15.eventsourcing._config.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (w)

- **Práctica 15. Event Sourcing Pattern**
- Asigna la propiedad que establece el servlet context path con el valor “/event-sourcing-service” y la define el puerto de la aplicación web al **8081**.
- Analiza la clase **HomeController** del paquete **com.consulting.mgt.springboot.practica15.eventsourcing.restcontroller**; en ella se definen las APIs que se expondrán a través del aplicativo.
- Define la implementación de las APIs definidas sobre el controlador **AccountController** del paquete **com.consulting.mgt.springboot.practica15.eventsourcing.restcontroller**.



+

iv.v Principios de diseño para aplicaciones en la nube. (x)

- **Práctica 15. Event Sourcing Pattern**
- Define la implementación de las APIs definidas sobre el controlador **TransferController** del paquete **com.consulting.mgt.springboot.practica15.eventsourcing.restcontroller.**
- Define la implementación de las APIs definidas sobre el controlador **SystemController** del paquete **com.consulting.mgt.springboot.practica15.eventsourcing.restcontroller.**
- Compila el proyecto desde línea de comando y ejecuta el aplicativo.
- Prueba tu trabajo.



iv.v Principios de diseño para aplicaciones en la nube. (y)

- Minimizar la coordinación.
- Recomendaciones:
 - **Particionar los datos.**
 - Evitar utilizar un mismo repositorio o esquema de base de datos compartido entre diferentes o distintos aplicativos o servicios.
 - **Database per Service Pattern**
 - Implementa transacciones locales para un único repositorio de datos por servicio.
 - Facilidad para implementar **compensación de transacciones** mediante **Saga Pattern**.
 - Permite asegurar servicios con bajo acoplamiento.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (z)

- Minimizar la coordinación.
- Recomendaciones:
 - **Diseñar operaciones idempotentes.**
 - **Idempotente:** En informática se refiere a una operación que produce los mismos resultados si se ejecuta una o varias veces.
 - Implementar idempotencia de operaciones transaccionales garantiza evitar coordinación transaccional entre microservicios.
 - Evitar operaciones “**exactly-once**” y preferir operaciones “**at-least-once**” para garantizar la entrega del mensaje a través de reintentos.



+

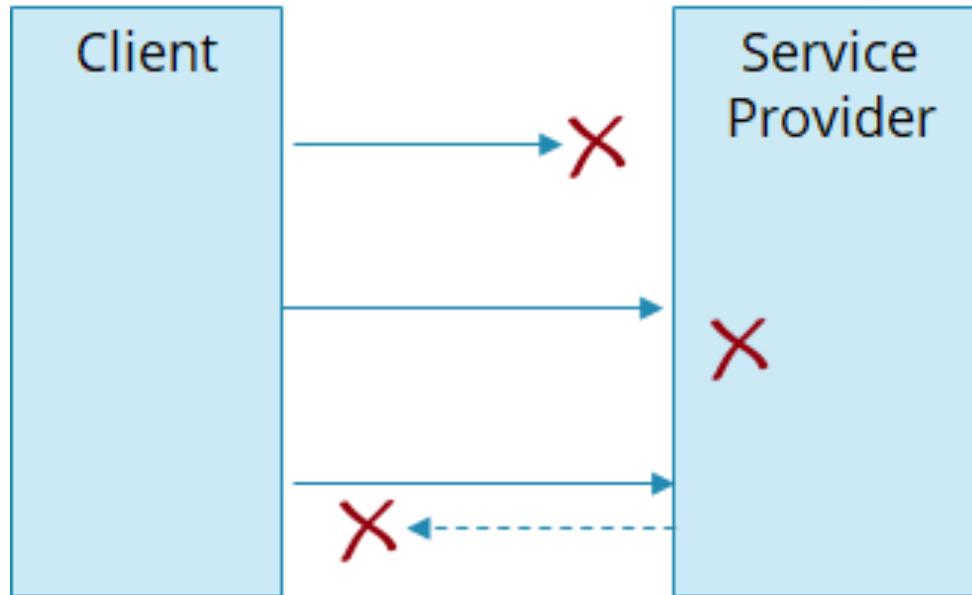
iv.v Principios de diseño para aplicaciones en la nube. (a')

- Minimizar la coordinación.
- Recomendaciones:
 - **Diseñar operaciones idempotentes.**
 - No implementar operaciones idempotentes dificulta la implementación de reintentos en línea o reintentos en "background".



iv.v Principios de diseño para aplicaciones en la nube. (b')

- Problemas al implementar reintentos.





+

iv.v Principios de diseño para aplicaciones en la nube. (c')

- Minimizar la coordinación.
- Recomendaciones:
 - **Diseñar operaciones idempotentes.**
 - **Identificador Unico.**
 - Generar un identificador único para cada llamada a servicio y agregarlo al cuerpo del mensaje.
 - Almacenar en Base de Datos los eventos enviados tanto en el productor como en el consumidor.
 - Utilizar mensajería asíncrona.

iv.v Principios de diseño para aplicaciones en la nube. (d')

- Minimizar la coordinación.
- Recomendaciones:
 - **Diseñar operaciones idempotentes.**
 - **Hash del mensaje de solicitud.**
 - Generar un hash del mensaje de la solicitud, en el consumidor y almacenarlo en BD como solicitud atendida.
 - Calcular el hash de cada mensaje entrante y verificar si este ya fue atendido.
 - **Siempre es necesario confirmar el procesamiento de un mensaje al productor/cliente del mismo.**

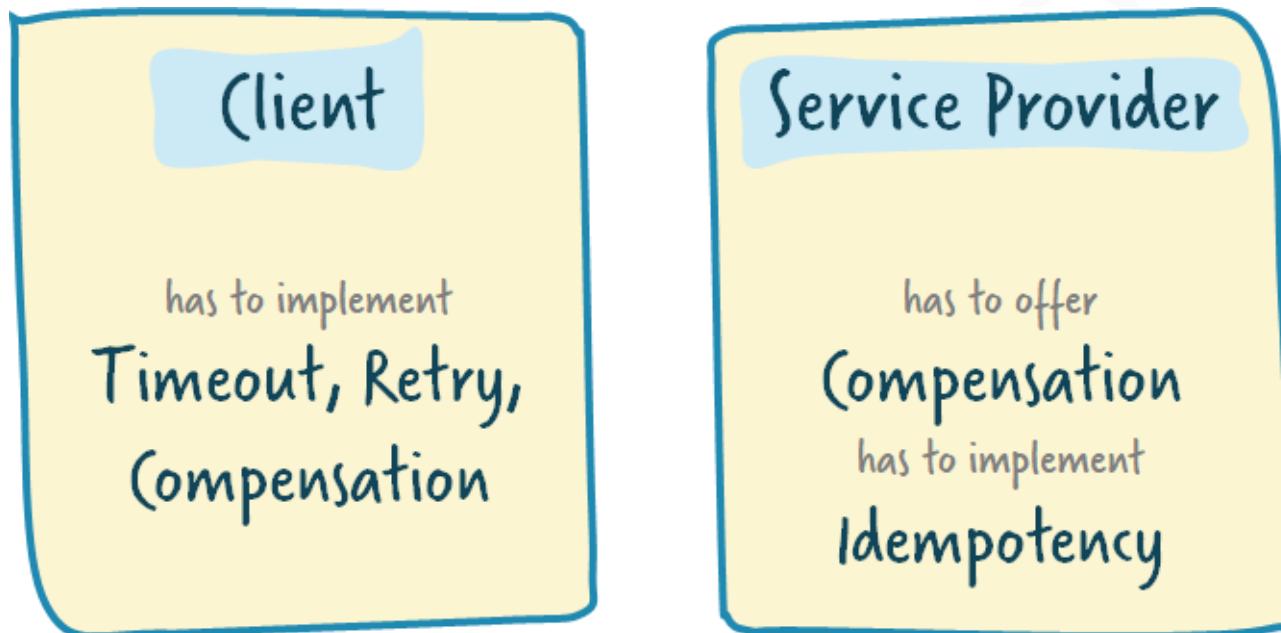


+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (e')

- Idempotencia.





iv.v Principios de diseño para aplicaciones en la nube. (f')

- Minimizar la coordinación.
- Recomendaciones:
 - **Usar procesamiento paralelo asíncrono.**
 - Implementar procesamiento de una operación en paralelo siempre y cuando, el resultado de dicho proceso, pueda agregarse de forma separada, es decir, el orden de cada paso en el procesamiento paralelo no depende de pasos anteriores.

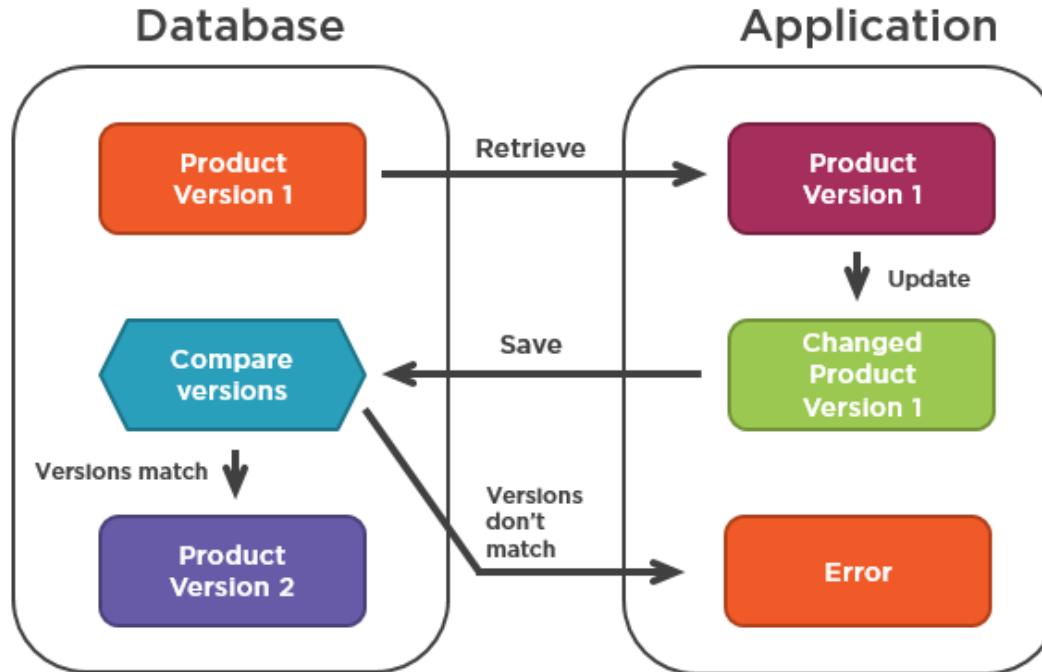
iv.v Principios de diseño para aplicaciones en la nube. (g')

- Minimizar la coordinación.
- Recomendaciones:
 - **Usar bloqueo optimista.**
 - Utilice bloqueo optimistas para evitar el problema de “**last transaction wins**” en una situación de concurrencia.
 - Es posible implementar bloqueo pesimista donde sólo un cliente puede alterar un conjunto de datos al mismo tiempo, ocasionando un cuello de botella lo cual degrada en performance y disponibilidad del servicio. Utilice bloqueo pesimista cuando el costo de integrar cambios concurrentes, por múltiples usuarios, sea alto (venta de boletos numerados, stock de productos, etc).



iv.v Principios de diseño para aplicaciones en la nube. (h')

- Usar bloqueo optimista.





iv.v Principios de diseño para aplicaciones en la nube.

- Diseñe para la recuperación automática.
- Haga todo redundante.
- Minimizar la coordinación.
- **Diseñe para facilitar el escalamiento horizontal.**
- Particione alrededor de límites.
- Diseñe para las operaciones.
- Use servicios administrados.
- Use el repositorio correcto para el trabajo correcto.
- Diseñe para evolucionar el servicio.
- Desarrolle considerando las necesidades del negocio.



iv.v Principios de diseño para aplicaciones en la nube. (a)

- Diseñe para facilitar el escalamiento horizontal.
- Uno de los beneficios de las aplicaciones “**cloud-native**” es la posibilidad de escalar servicios de forma elástica, es decir, de forma automática conforme a la carga de trabajo del servicio.
- El escalamiento horizontal permite el escalamiento elástico de un servicio agregando nuevas instancias, cuando la carga lo amerite, o reduciendo el número de instancias cuando la carga de trabajo del servicio disminuya.

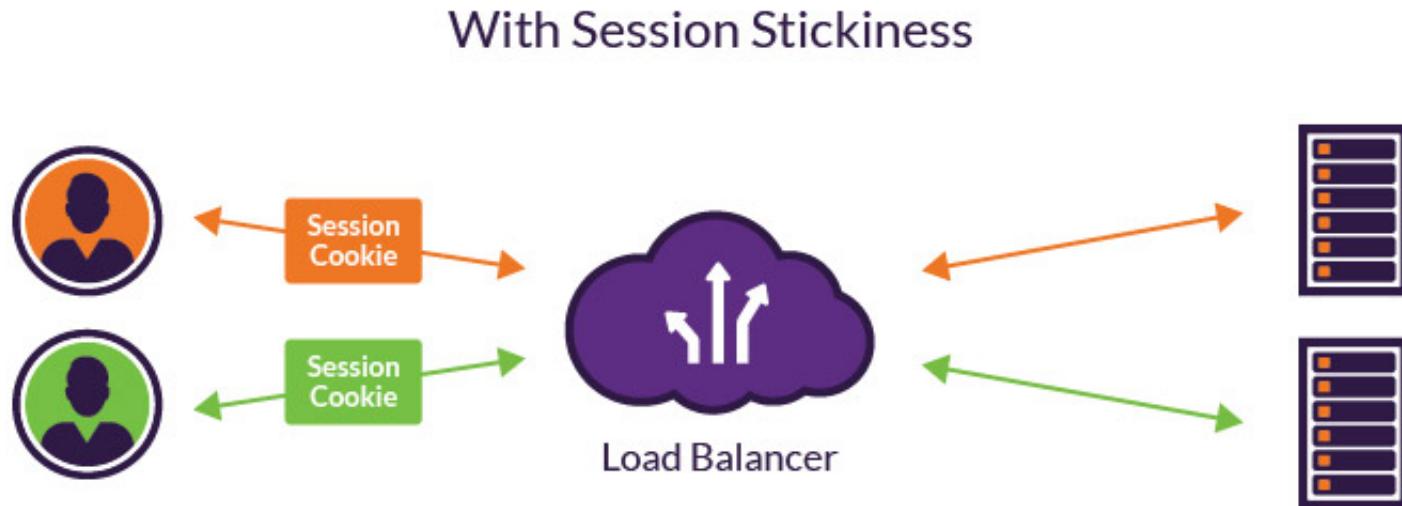
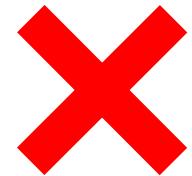


iv.v Principios de diseño para aplicaciones en la nube. (b)

- Diseñe para facilitar el escalamiento horizontal.
- Recomendaciones:
 - **Evite la afinidad de sesión.**
 - La afinidad de sesión (**sticky-sessions**) limita a la aplicación la habilidad de escalar horizontalmente.
 - El alto volumen de tráfico en servicios que mantienen afinidad de sesión evita que la carga se distribuya correctamente a través de las instancias.
 - Asegure que cualquier instancia o replica de un servicio pueda atender cualquier solicitud entrante.

iv.v Principios de diseño para aplicaciones en la nube. (c)

- Evite la afinidad de sesión.



iv.v Principios de diseño para aplicaciones en la nube. (d)

- Evite la afinidad de sesión.

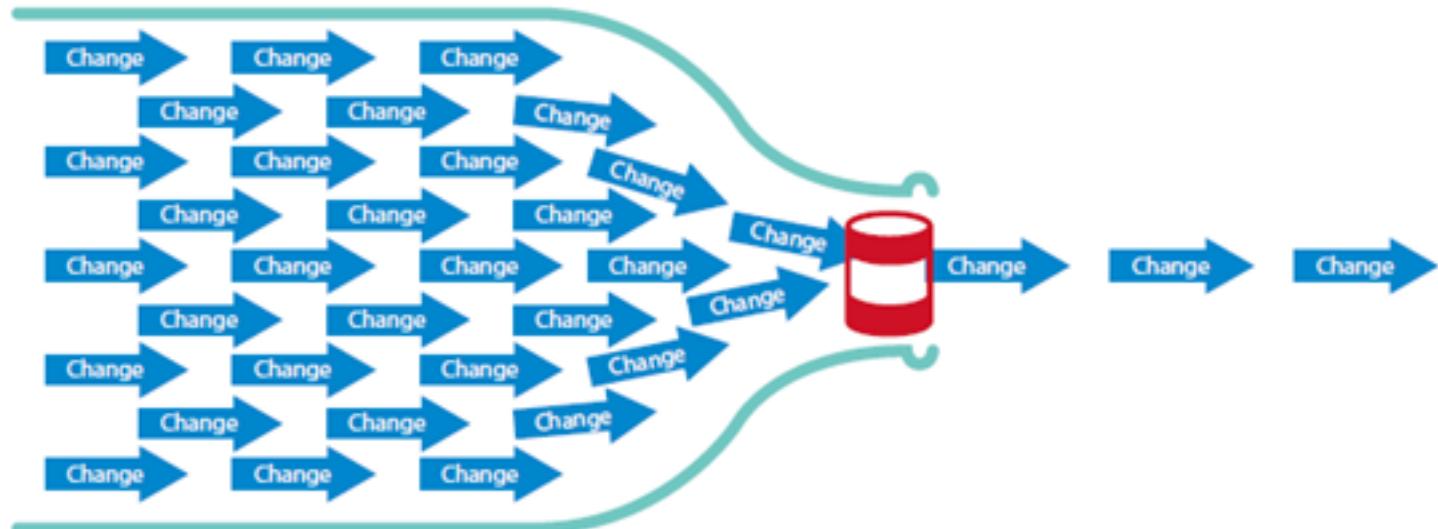


iv.v Principios de diseño para aplicaciones en la nube. (e)

- Diseñe para facilitar el escalamiento horizontal.
- Recomendaciones:
 - **Identifique cuellos de botella.**
 - El escalado horizontal no beneficia el performance ni disponibilidad de un servicio.
 - Es necesario revisar donde están los cuellos de botella y solucionarlos.
 - Comunmente los cuellos de botella se dan en la base de datos y, en este caso, aumentar el número de instancias del servicio (que consume la base de datos) solo empeorará el performance y disponibilidad del servicio de la aplicación.

iv.v Principios de diseño para aplicaciones en la nube. (f)

- Identifique cuellos de botella.



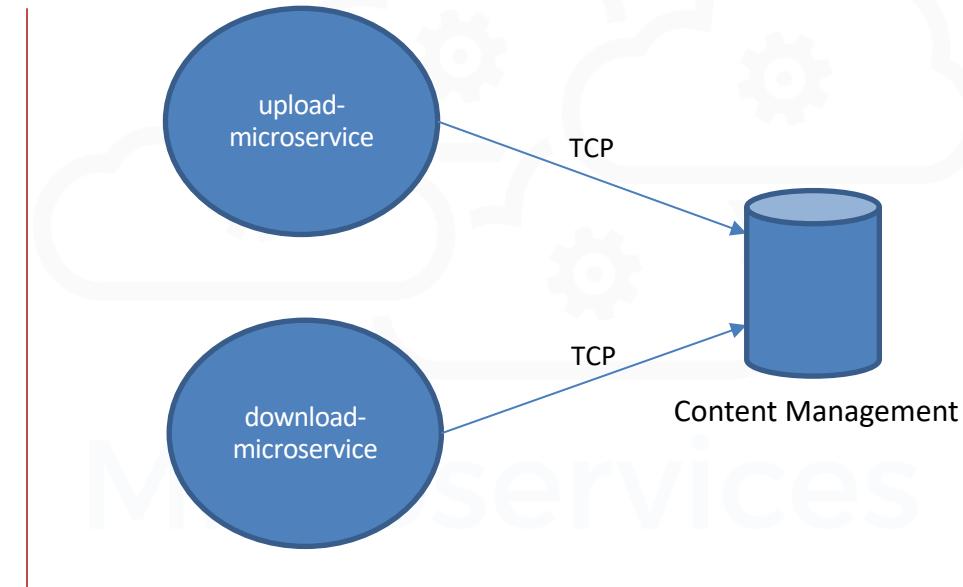
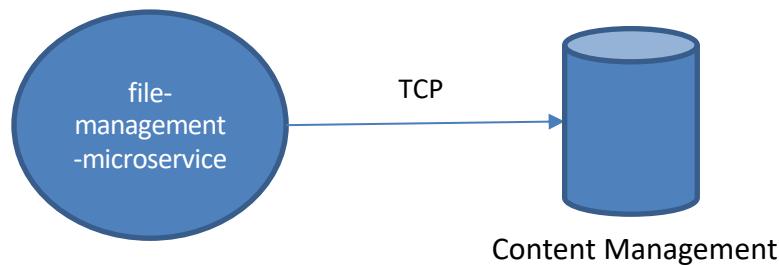


iv.v Principios de diseño para aplicaciones en la nube. (g)

- Diseñe para facilitar el escalamiento horizontal.
- Recomendaciones:
 - **Diseñe servicios descomponiendo por contexto “Bounded-context” y por requerimientos de escalabilidad.**
 - Consentir que las aplicaciones estan compuestas por múltiples requerimientos funcionales, que tienen diferentes requerimientos de escalado.
 - Posiblemente un mismo requerimiento funcional definido en un microservicio, pueda descomponerse en dos microservicios si algún servicio contenido en el microservicio, tiene diferentes requerimientos de escalabilidad.

iv.v Principios de diseño para aplicaciones en la nube. (h)

- Diseñe servicios descomponiendo por contexto “Bounded-context” y por requerimientos de escalabilidad.





iv.v Principios de diseño para aplicaciones en la nube. (i)

- Diseñe para facilitar el escalamiento horizontal.
- Recomendaciones:
 - **Descomponga cargas intensivas de trabajo que consumen muchos recursos.**
 - Descomponga tareas que requieran amplia cantidad de recursos de CPU o I/O en ejecuciones en segundo plano, de forma asíncrona.
 - Ejecutar las operaciones que consumen muchos recursos informáticos en horarios inhábiles de usabilidad de la aplicación en forma de procesamiento en “background” o por lotes (batch).

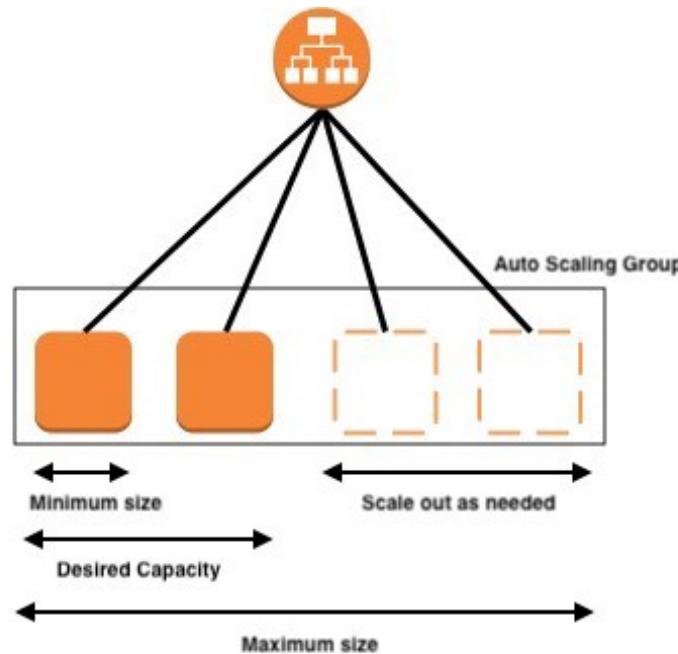


iv.v Principios de diseño para aplicaciones en la nube. (j)

- Diseñe para facilitar el escalamiento horizontal.
- Recomendaciones:
 - Utilice funcionalidades de auto-escalado integradas en la plataforma.
 - Implante funcionalidades de auto-escalado (elasticidad o “autoscaling”) de servicios en la plataforma de ejecución en la nube (PaaS).
 - Si la aplicación tiene un comportamiento de carga de trabajo predecible, utilice un “scheduler” para escalar automáticamente la aplicación; en caso de que no sea predecible, utilice métricas de uso de CPU o RAM para activar el “auto-escalado” de servicios.

iv.v Principios de diseño para aplicaciones en la nube. (k)

- Utilice funcionalidades de auto-escalado integradas en la plataforma.





+

iv.v Principios de diseño para aplicaciones en la nube. (I)

- Diseñe para facilitar el escalamiento horizontal.
- Recomendaciones:
 - **Diseñe para escalar (hacia arriba, pero también hacia abajo) reducción horizontal.**
 - Durante el proceso de escalado horizontal, al agregar instancias, mientras las instancias nuevas son creadas, dichas instancias no podrán ser utilizadas hasta que estén completamente inicializadas.
 - Al remover instancias, durante el proceso de escalado hacia abajo, la aplicación deberá de administrar correctamente las instancias que se eliminan.

iv.v Principios de diseño para aplicaciones en la nube. (m)

- Diseñe para facilitar el escalamiento horizontal.
- Recomendaciones:
 - **Diseñe para escalar (hacia arriba, pero también hacia abajo) reducción horizontal.**
 - Implemente eventos de apagado en cada servicio, para que se apaguen correctamente.
 - Los clientes o consumidores de los servicios deben manejar correctamente errores transitorios y reintentos.
 - Para trabajos de ejecución prolongada, considere dividir el trabajo mediante el patrón “**Pipes and Filters**”.



iv.v Principios de diseño para aplicaciones en la nube. (n)

- Diseñe para facilitar el escalamiento horizontal.
- Recomendaciones:
 - **Diseñe para escalar (hacia arriba, pero también hacia abajo) reducción horizontal.**
 - Habilite el uso de cola de mensajes para colocar los elementos de trabajo procesados a fin de que otra instancia pueda continuar el procesamiento del trabajo, en caso de que la instancia que lo procesaba haya sido eliminada debido a un error o a un apagado originado de una disminución en el escalado del servicio.



iv.v Principios de diseño para aplicaciones en la nube. (ñ)

- Pipes and Filters Pattern.
- **Categorías:** Mensajería, diseño e implementación.
 - Descomponga el procesamiento complejo de una tarea en una serie de tareas específicas, separadas, que puedan ser reutilizables.
 - Esto puede mejorar el rendimiento, la escalabilidad y la reutilización al permitir que los elementos de tareas que realizan el procesamiento se implementen y escalan de forma independiente.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (o)

- Pipes and Filters Pattern.
- **Intención.**
 - Mejorar el rendimiento, escalabilidad y la reutilización de componentes al permitir que los elementos que realizan la tarea del procesamiento se implementen y escalen de forma independiente.
 - Reutilizar componentes que ejecutan una tarea en distintos flujos de procesamiento.
 - Mejorar el rendimiento del aplicativo mediante la implementación de un flujo de tareas simples.
 - Mejorar el mantenimiento de los componentes mediante SOLID.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (p)

- Pipes and Filters Pattern.
- **Aplicabilidad.**
 - Cuando el procesamiento requerido por una aplicación se pueda dividir fácilmente en un conjunto de pasos independientes.
 - Cuando los distintos pasos de un flujo de procesamiento de una aplicación tiene diferentes requisitos de escalabilidad.
 - Cuando se requiera flexibilidad para permitir la reordenación de los pasos del flujo de un proceso, realizado por una aplicación, o cuando se requiera la capacidad de agregar y eliminar pasos al flujo del proceso.



iv.v Principios de diseño para aplicaciones en la nube. (q)

- Pipes and Filters Pattern.
- **Aplicabilidad.**
 - Cuando el sistema puede beneficiarse de la distribución del procesamiento de los pasos a través de diferentes nodos.
 - Cuando se requiere una solución confiable que minimice los efectos de la falla en un paso mientras se procesan los datos en pasos anteriores y/o posteriores.



+

iv.v Principios de diseño para aplicaciones en la nube. (r)

- Pipes and Filters Pattern.
- **Ventajas:**
 - Permite comprender el comportamiento de entrada / salida de un sistema como la composición del comportamiento de los filtros (“pipes”) individuales.
 - Alta reutilización de componentes.
 - Facilita el mantenimiento y evolución del aplicativo.
 - Permiten realizar análisis de “**bottle-neck**” y rendimiento.
 - Soporte para la ejecución concurrente de tareas.



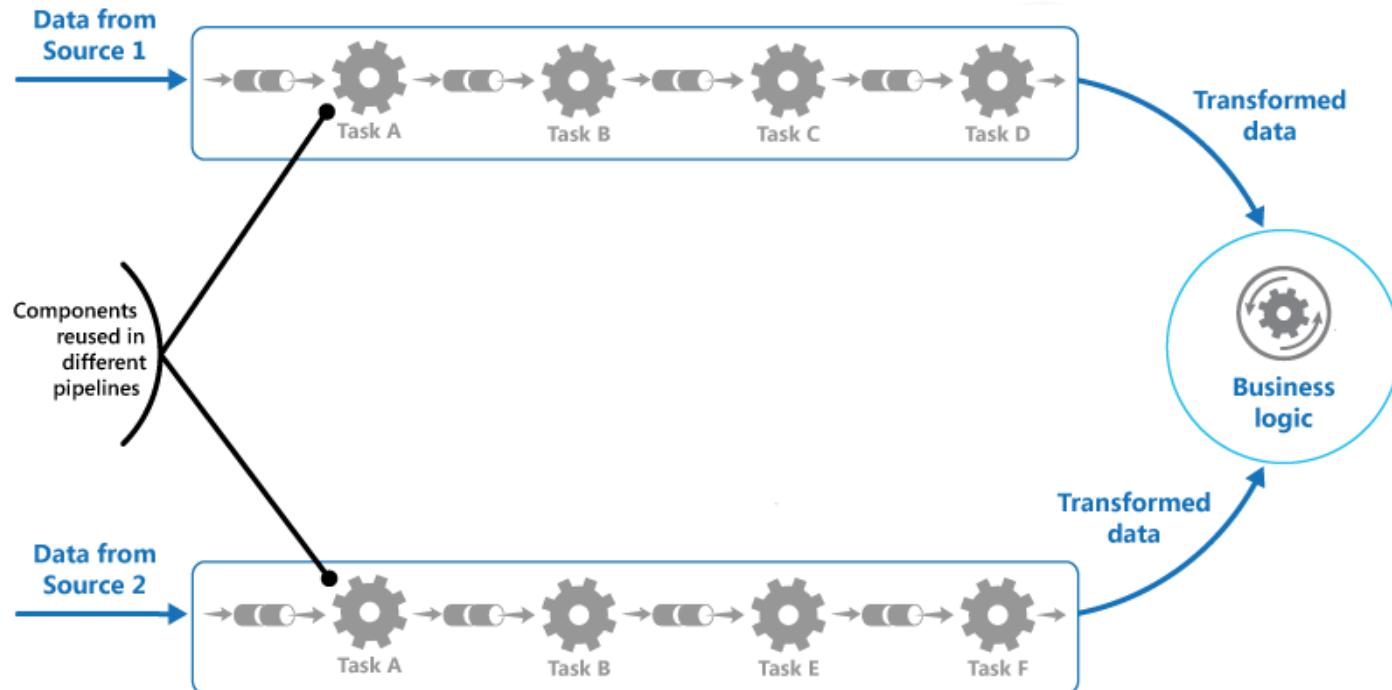
+

iv.v Principios de diseño para aplicaciones en la nube. (s)

- Pipes and Filters Pattern.
- **Desventajas:**
 - Frecuentemente tienden a una implementación de procesamiento batch.
 - No recomendable cuando sea requerida la interactividad humana durante el flujo del proceso. Puede subdividirse en diferentes “pipes”.
 - Complejidad de mantenimiento al modificar componentes reutilizables.
 - Puede ser necesario agregar componentes de conversión de datos de entrada y salida.

iv.v Principios de diseño para aplicaciones en la nube. (t)

- Pipes and Filters Pattern.





+

iv.v Principios de diseño para aplicaciones en la nube. (u)

- Práctica 16. Pipes And Filters Pattern
- Analiza la aplicación **16-Pipes-And-Filters-Microservice**. Analiza el caso de uso.
- Ingresar a la ruta: **{tu-workspace}/16-Pipes-And-Filters-Microservice**
- Importar el proyecto **16-Pipes-And-Filters-Microservice** en STS.
- Se implementará el patrón **Pipes and Filters** mediante **Spring Integration** por simplicidad.



iv.v Principios de diseño para aplicaciones en la nube. (v)

- Práctica 16. Pipes And Filters Pattern
- Implementa la clase **FilePrinterService** del paquete **com.consulting.mgt.springboot.practica16.pipesandfilters.service**, la cual recibe un objeto **File** y un mapa de headers. Imprime en el log el contenido del archivo, línea por línea.
- Analiza los archivos que se encuentran en la carpeta “**read**” de “**src/main/resources**”. Dichos archivos se copiaran a la carpeta “**write**” de la misma ubicación.

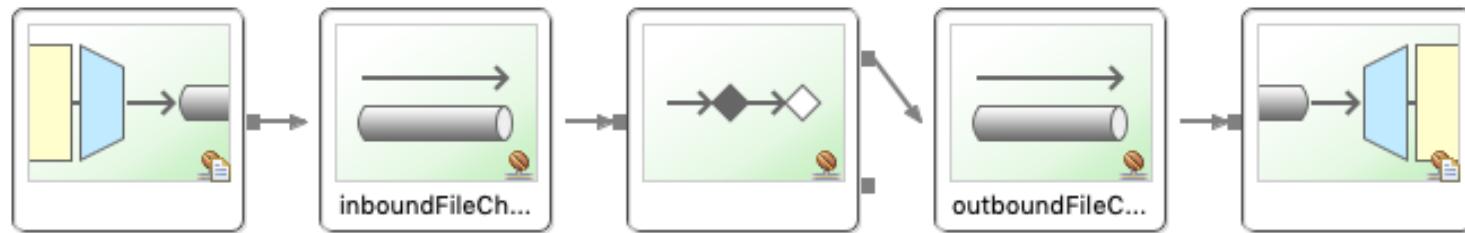


+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (w)

- **Práctica 16. Pipes And Filters Pattern**
- La intención de la práctica es crear un proceso que lea archivos desde una ubicación específica, imprima su contenido y, luego, los copie en otra ruta definida.
- El proceso simula ser pesado, por lo tanto se propone la separación de cada parte del proceso en un “task” más pequeño (y específico), conectando las tareas mediante tuberías (“pipes”).





+

iv.v Principios de diseño para aplicaciones en la nube. (x)

- Práctica 16. Pipes And Filters Pattern
- Implementa el flujo previamente descrito mediante **Spring Integration** sobre el archivo **integration-context.xml** definido en el **classpath** del proyecto.
- Carga el archivo de configuración **integration-context.xml** en la clase principal del proyecto.
- Ejecuta la clase principal del proyecto, analiza los resultados.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube.

- Diseñe para la recuperación automática.
- Haga todo redundante.
- Minimizar la coordinación.
- Diseñe para facilitar el escalamiento horizontal.
- **Particione alrededor de límites.**
- Diseñe para las operaciones.
- Use servicios administrados.
- Use el repositorio correcto para el trabajo correcto.
- Diseñe para evolucionar el servicio.
- Desarrolle considerando las necesidades del negocio.



iv.v Principios de diseño para aplicaciones en la nube. (a)

- Particione alrededor de límites.
- Utilice particiones para evitar limites de recursos informaticos sobre bases de datos, redes y procesamiento.
- En desarrollo de software todos los servicios que se mantienen a traves de hardware tienen limitaciones en cuanto a escalamiento vertical. Los límites incluyen número de vCPU asignados, memoria RAM, tamaño de disco asignado a Bases de datos, ancho de banda, etcetera.



iv.v Principios de diseño para aplicaciones en la nube. (b)

- Particione alrededor de límites.
- Si un sistema de software crece lo suficiente, puede alcanzar fácilmente los límites de los servicios informáticos que utiliza.
- Implemente particiones sobre los recursos / servicios para evitar alcanzar los límites establecidos.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (c)

- Particione alrededor de límites.
- Recomendaciones:
 - **Crear particiones de distintos componentes de la aplicación.**
 - Las bases de datos son candidatas obvias para la creacion de particiones.
 - Particione elementos de almacenamiento (volumenes, file system, etc).
 - Particione o distribuya motores de cache y colas de mensajes
 - Particione instancias de procesamiento (no se refiere a aumentar réplicas, duplique servicios en caso de ser requerido).



iv.v Principios de diseño para aplicaciones en la nube. (d)

- Particione alrededor de límites.
- Recomendaciones:
 - **Diseñe la llave de partición para evitar zonas inactivas o altamente activas.**
 - Asegurarse de que la distribución de la carga entre las particiones de los servicios (componentes) se realice de forma uniforme.
 - **Código hash por Id de cliente entre el número de instancias.**



iv.v Principios de diseño para aplicaciones en la nube. (e)

- Particione alrededor de límites.
- Recomendaciones:
 - **Establecer límites de consumo de recursos en la Plataforma como Servicio (PaaS).**
 - Los servicios y el consumo de recursos pueden limitarse por región o de forma individual por servicio.
 - Establecer límites de consumo de CPU, RAM y recursos permitira medir el costo real del aplicativo en producción.
 - Establecer límites a nivel de IaaS es más complicado.



iv.v Principios de diseño para aplicaciones en la nube.

- Diseñe para la recuperación automática.
- Haga todo redundante.
- Minimizar la coordinación.
- Diseñe para facilitar el escalamiento horizontal.
- Particione alrededor de límites.
- **Diseñe para las operaciones.**
- Use servicios administrados.
- Use el repositorio correcto para el trabajo correcto.
- Diseñe para evolucionar el servicio.
- Desarrolle considerando las necesidades del negocio.



+

iv.v Principios de diseño para aplicaciones en la nube. (a)

- Diseñe para las operaciones.
- El rol de administradores de sistemas u operaciones ha evolucionado dramáticamente conforme la industria ha adoptado la nube. Ya no solamente se encargan de administrar el hardware y la infraestructura que da soporte a las aplicaciones, ahora también ejecutan funciones como:
 - Despliegue
 - Monitoreo
 - Escalado
 - Respuesta a incidentes
 - Auditoría de seguridad



+

iv.v Principios de diseño para aplicaciones en la nube. (b)

- Diseñe para las operaciones.
- En las aplicaciones para la nube las prácticas de “**logging**” y “**tracing**” son extremadamente necesarias para una correcto seguimiento de operaciones.
- Recomendaciones:
 - SLF4J
 - Log4J
 - Spring Cloud Sleuth
 - Zipkin
 - ELK (ElasticSearch, Logstash y Kibana), entre otros.



iv.v Principios de diseño para aplicaciones en la nube. (c)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Implementar registro y seguimiento.**
 - Dado que una aplicación a sido desplegada y esta se encuentra en ejecución, la práctica de implementar “**logging**” (registro) y “**tracing**” (seguimiento) serán la información principal (y posiblemente única) sobre el estado del sistema.
 - El seguimiento o “**tracing**” registra una ruta o huella de una petición a través del sistema, la cuál es útil para dar seguimiento a la operación; es útil para identificar “**bottle-necks**”, problemas de rendimiento y puntos de error.



iv.v Principios de diseño para aplicaciones en la nube. (d)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Implementar registro y seguimiento.**
 - El registro o “**logging**” captura eventos individuales importantes en el transcurso de la ejecución de la aplicación, como cambios de estado de la aplicación, errores y excepciones.
 - Lleve un registro de “**logging**” y “**tracing**” en ambientes productivos para facilitar la tarea al equipo de operaciones.
 - **SLf4J, Log4J**
 - **Pistas de Auditoría**
 - **Bitácoras**



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (e)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Hacer que todo sea observable.**
 - Exponga información coherente y comprensible del estado de la aplicación o de algún proceso en particular en modo de API.
 - Implemente mecanismos de ingestión de dicha información para el monitoreo (Utilice herramientas para el monitoreo).
 - **Spring Boot Actuator.**
 - **Actuadores / Actuators.**
 - **APIs personalizadas, diseñadas para las operaciones.**

iv.v Principios de diseño para aplicaciones en la nube. (f)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Utilice herramientas para el monitoreo.**
 - El monitoreo permite brindar una vista de cómo se comporta la aplicación (bien o mal) en términos de performance, disponibilidad o, salud de la aplicación.
 - El monitoreo de la aplicación se debe de realizar mientras la aplicación esta en ejecución por ello, utilice herramientas que lean u obtengan estos indicadores / métricas y muestrelos en una interface de usuario que facilite las operaciones.
 - **Spring Boot Admin, Eureka, Hystrix / Turbine, Micrometer / Prometheus.**



+

iv.v Principios de diseño para aplicaciones en la nube. (g)

- Práctica 17. Spring Boot Actuator y Spring Boot Admin
- Analiza la aplicación **17-Spring-Boot-Admin-Server** y **17-Spring-Boot-Admin-Client**.
- Ingresar a la ruta: **{tu-workspace}/17-Spring-Boot-Admin-Server**
- Importar el proyecto **17-Spring-Boot-Admin-Server** en STS.
- Agregar la dependencia **de.codecentric:spring-boot-admin-starter-server:2.1.5** al archivo pom.xml



iv.v Principios de diseño para aplicaciones en la nube. (h)

- Práctica 17. Spring Boot Actuator y Spring Boot Admin
- Habilitar Spring Boot Admin Server sobre la clase principal **Application**, del paquete **com.consulting.mgt.springboot.practica17.springbootadmin**, mediante la anotación **@EnableAdminServer**.
- Configurar el servlet context-path del proyecto a: “**/spring-boot-admin-server-service**” y el puerto escucha de tomcat al **9192**.
- Accede desde el navegador a la ruta: <http://localhost:9192/spring-boot-admin-server-service/#/applications>



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (i)

- Práctica 17. Spring Boot Actuator y Spring Boot Admin
- Ingresar a la ruta: **{tu-workspace}/17-Spring-Boot-Admin-Client**
- Importar el proyecto **17-Spring-Boot-Admin-Client** en STS.
- Agregar la dependencia **de.codecentric:spring-boot-admin-starter-client:2.1.5** al archivo pom.xml



+

iv.v Principios de diseño para aplicaciones en la nube. (j)

- **Práctica 17. Spring Boot Actuator y Spring Boot Admin**
- Define las propiedades servlet context-path al valor “**/spring-boot-admin-client-service**” y configura el puerto escucha de tomcat al **8080**.
- Define la propiedad **spring.application.name** con el valor “**spring-boot-admin-client**”.
- Define la propiedad **spring.boot.admin.client.url** para que se conecte al Spring Boot Admin Server asignando como valor la URI del servidor Spring Boot Admin [“http://localhost:9192/spring-boot-admin-server-service”](http://localhost:9192/spring-boot-admin-server-service).



iv.v Principios de diseño para aplicaciones en la nube. (k)

- **Práctica 17. Spring Boot Actuator y Spring Boot Admin**
- Expón todos los endpoints de Spring Boot Actuator mediante la propiedad **management.endpoints.web.exposure.include** con valor *****.
- Expón el endpoint “**health.show-details**” mediante la propiedad **management.endpoint.health.show-details** con valor **always**.
- Define información de validéz para el proyecto mediante la propiedad **info.***.
- Define un **RestController** simple, sobre la clase **SpringBootAdminClientController**, que mediante método **GET** devuelva el String "**Hello I'm " + appName + "!!!**", donde **appName** es el valor de la propiedad **spring.application.name**.



+

iv.v Principios de diseño para aplicaciones en la nube. (I)

- **Práctica 17. Spring Boot Actuator y Spring Boot Admin**
- Ejecuta la clase principal del proyecto **17-Spring-Boot-Admin-Client**, la cual está anotada con la anotación **@SpringBootApplication**.
- Prueba la aplicación cliente, abriendo en el navegador la URI
<http://localhost:8080/spring-boot-admin-client-service/>



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (m)

- **Práctica 17. Spring Boot Actuator y Spring Boot Admin**
- Regresa a la ventana de navegador donde está abierto el aplicativo Spring Boot Admin Server, deberás encontrar que existe una aplicación en ejecución.

The screenshot shows the Spring Boot Admin interface with the following data:

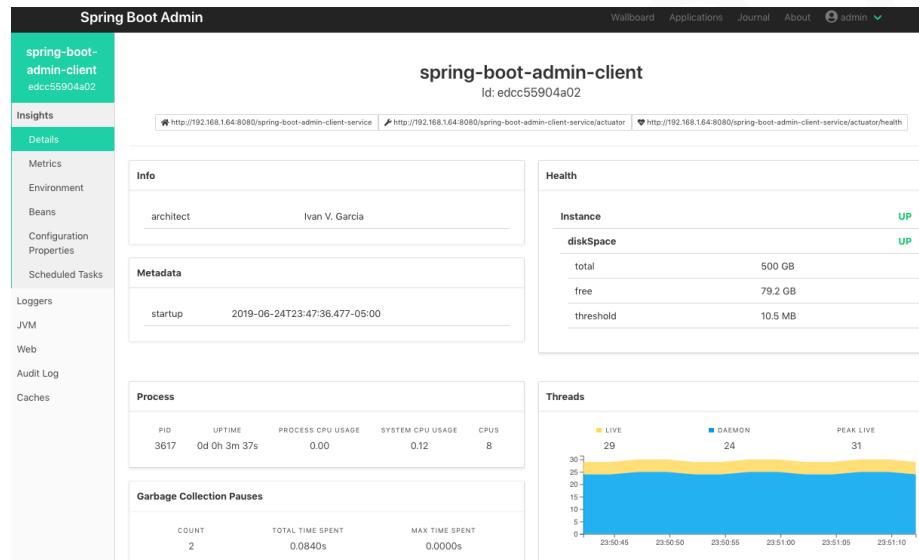
APPLICATIONS	INSTANCES	STATUS
1	1	all up

Below the table, there is a list of applications:

- spring-boot-admin-client
1m http://192.168.1.64:8080/spring-boot-admin-client-service

iv.v Principios de diseño para aplicaciones en la nube. (n)

- **Práctica 17. Spring Boot Actuator y Spring Boot Admin**
- Navega mediante la aplicación Spring Boot Admin Server y descubre las múltiples funcionalidades que presenta.



iv.v Principios de diseño para aplicaciones en la nube. (ñ)

Spring Boot Admin

Wallboard Applications Journal About admin ▾

spring-boot-admin-client
Id: edcc55904a02

http://192.168.1.64:8080/spring-boot-admin-client-service http://192.168.1.64:8080/spring-boot-admin-client-service/actuator http://192.168.1.64:8080/spring-boot-admin-client-service/actuator/health

Info

architect	Ivan V. Garcia
-----------	----------------

Metadata

startup	2019-06-24T23:47:36.477-05:00
---------	-------------------------------

Health

Instance	UP
diskSpace	UP
total	500 GB
free	79.2 GB
threshold	10.5 MB

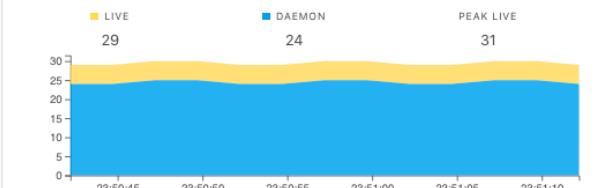
Process

PID	UPTIME	PROCESS CPU USAGE	SYSTEM CPU USAGE	CPUS
3617	0d 0h 3m 37s	0.00	0.12	8

Garbage Collection Pauses

COUNT	TOTAL TIME SPENT	MAX TIME SPENT
2	0.0840s	0.0000s

Threads



TIME	LIVE	DAEMON
23:50:45	29	24
23:50:50	28	25
23:50:55	27	26
23:51:00	26	27
23:51:05	25	28
23:51:10	24	29

iv.v Principios de diseño para aplicaciones en la nube. (o)

- Diseñe para las operaciones.
- Recomendaciones:
 - Utilice herramientas para el análisis de problemas con causa principal.
 - El análisis de problemas con causa principal ("root cause analysis") es el proceso de búsqueda manual de la causa de los errores lanzados. Este análisis se produce después de que ha surgido algún error (mejor dicho excepción), por lo general son errores transitorios.
 - Implemente mecanismos o herramientas para la visualización de dichos problemas con causa raíz (Dashboard de conexión a BD).



iv.v Principios de diseño para aplicaciones en la nube. (p)

- Diseñe para las operaciones.
- Recomendaciones:
 - Utilice herramientas de seguimiento distribuido (**distributed tracing**).
 - Utilice herramientas que permitan monitorear el seguimiento o “tracing” de las peticiones en sistemas basados en la nube donde imperan la concurrencia, la asíncronicidad y el escalamiento de componentes.
 - La trazabilidad del seguimiento de una petición debe incluir un identificador de correlación y una marca de tiempo.
 - El identificador de correlación debe propagarse entre los límites de cada componente o servicio.



iv.v Principios de diseño para aplicaciones en la nube. (q)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Utilice herramientas de seguimiento distribuido (distributed tracing).**
 - Una simple operación puede componerse de múltiples llamadas a servicios. En caso de que alguna llamada a alguna operación falle, es posible darle el seguimiento correcto y verificar la causa del error mediante su identificador de correlación.
 - **Spring Cloud Sleuth y Zipkin**
 - **ELK - ElasticSearch, Logstash y Kibana**



iv.v Principios de diseño para aplicaciones en la nube. (r)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Estandaríce logs y metricas.**
 - Al momento del monitoreo y búsqueda de errores, los equipos de operaciones necesitan agregar y analizar múltiples “**logs**”, de diversos servicios. Si dichos “**logs**” tienen su propio formato, se vuelve más difícil (a veces imposible) para la gente de operaciones rastrear la trazabilidad (seguimiento) del error.



iv.v Principios de diseño para aplicaciones en la nube. (s)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Estandaríce logs y metricas.**
 - Para facilitar la trazabilidad de errores y análisis de métricas se sugieren herramientas de “tracing” y habilitar en el “log”, datos coherentes, comprensibles que faciliten el análisis de los mismos agregando dirección IP del cliente, nombre del evento, Identificador de correlación (trazabilidad), entre otros datos, en un formato homogéneo entre los diversos servicios.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (t)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Automatizar las tareas de administración.**
 - Automatice todo incluyendo aprovisionamiento, despliegue y monitoreo.
 - Automatizar las tareas de administración permite repetirlas de forma constante, para el despliegue de múltiples servicios con una probabilidad menos propensa a errores humanos.
 - Implemente ALM (Application Lifecycle Management).
 - Jira, Jenkins, UML, Git, Maven, Gradle, SVN, Unit Tests, Integration Tests, DDD, CI / CD, etc.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (u)

- Diseñe para las operaciones.
- Recomendaciones:
 - **Manipule la configuración como código.**
 - Utilizar la configuración de servicios como código, es decir, mediante un sistema de control de versiones, de esta forma podrá ejecutar un “**rollback**” en caso de que alguna configuración posterior sea incorrecta o necesite ser cambiada a su configuración previa.
 - SVN, Git.



+

iv.v Principios de diseño para aplicaciones en la nube.

- Diseñe para la recuperación automática.
- Haga todo redundante.
- Minimizar la coordinación.
- Diseñe para facilitar el escalamiento horizontal.
- Particione alrededor de límites.
- Diseñe para las operaciones.
- **Use servicios administrados.**
- Use el repositorio correcto para el trabajo correcto.
- Diseñe para evolucionar el servicio.
- Desarrolle considerando las necesidades del negocio.



+

iv.v Principios de diseño para aplicaciones en la nube. (a)

- Use servicios administrados.
- Preferentemente utilice la plataforma como servicio (**PaaS**) en lugar de la infraestructura como servicio (**IaaS**).
- IaaS es como tener un “**tool-box**”, puede crear cualquier cosa pero debe de ser ensamblada a mano, por el equipo mismo.
- Los servicios administrados son más fáciles de configurar y administrar. Evita la necesidad de aprovisionar máquinas virtuales, configurar redes virtuales, administrar revisiones y/o actualizaciones de SO y toda la sobrecarga asociada a la ejecución de software en una máquina virtual.



iv.v Principios de diseño para aplicaciones en la nube. (b)

- Use servicios administrados.
- Todos los proveedores de PaaS proveen imágenes o servicios administrados para, por ejemplo, aprovisionar una bandeja (“queue”) de mensajes.
- Sobre IaaS, puede configurar un servicio propio de mensajería en una máquina virtual, utilizando RabbitMQ ó, sobre PaaS, desplegar un servicio administrado con un broker de mensajes similar a RabbitMQ (e incluso el mismo).



+

iv.v Principios de diseño para aplicaciones en la nube. (c)

- Use servicios administrados.
- Es posible que la aplicación a desplegar, en la nube, pueda tener requisitos específicos de software que hacen que un enfoque de implementación mediante IaaS sea más adecuada.
- Procure incorporar servicios administrados de la mejor manera posible utilizando servicios como cachés, colas de mensajes y almacenamiento de datos.



iv.v Principios de diseño para aplicaciones en la nube.

- Diseñe para la recuperación automática.
- Haga todo redundante.
- Minimizar la coordinación.
- Diseñe para facilitar el escalamiento horizontal.
- Particione alrededor de límites.
- Diseñe para las operaciones.
- Use servicios administrados.
- **Use el repositorio correcto para el trabajo correcto.**
- Diseñe para evolucionar el servicio.
- Desarrolle considerando las necesidades del negocio.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (a)

- Use el repositorio correcto para el trabajo correcto.
- Las bases de datos relacionales o SQL no son la solución correcta para todos las casuísticas. Únicamente son muy buenas para las casuísticas para las que fueron diseñadas: ofrecer garantías de **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad).
- Las bases de datos SQL refieren los siguientes inconvenientes:
 - La consulta de información puede requerir “**joins**” costosos.
 - La información debe ser normalizada y debe ser ajustada a un esquema específico (esquema de escritura).
 - El bloqueo de filas, tablas y registros para **ACID** afecta el rendimiento.



iv.v Principios de diseño para aplicaciones en la nube. (b)

- Use el repositorio correcto para el trabajo correcto.
- Para las aplicaciones de gran tamaño probablemente una tecnología sencilla, como SQL, para almacen de datos no sea satisfactoria para todos los requerimientos. Considere también alternativas como:
 - **Almacenes de llave/valor:** Redis, Raik, Oracle NoSQL
 - **Bases de datos de documentos:** MongoDB, CouchDB, Couchbase
 - **Bases de datos con motores de búsqueda:** ElasticSearch, Solr, Lucene
 - **Bases de datos de series de tiempo:** Kairos DB, Timescale DB
 - **Bases de datos de familias de columnas:** Cassandra, Hbase, BigTable
 - **Bases de datos de gráficos:** GraphDB, Neo4J, ArangoDB, OrientDB



iv.v Principios de diseño para aplicaciones en la nube. (c)

- Use el repositorio correcto para el trabajo correcto.
- Recomendaciones:
 - **No utilice una base de datos relacional para todo.**
 - Concidere otros tipos de almacenes de datos para el requerimiento apropiado.
 - **Adopte persistencia políglota.**
 - Cualquier aplicación para la nube, grande o pequeña, una única solución de almacenamiento de datos no va a satisfacer todos los requerimientos no funcionales.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (d)

- Use el repositorio correcto para el trabajo correcto.
- Recomendaciones:
 - **No utilice una base de datos relacional para todo.**
 - Concidere otros tipos de almacenes de datos para el requerimiento apropiado.
 - Adopte persistencia políglota.
 - Considere los tipos de datos para seleccionar el tipo de almacen de datos.



iv.v Principios de diseño para aplicaciones en la nube. (e)

- Use el repositorio correcto para el trabajo correcto.
- Recomendaciones:
 - **Elija disponibilidad sobre “solida” consistencia.**
 - Según el teorema CAP, en un sistema distribuido se debe compensar entre disponibilidad y consistencia.
 - Nunca se debe evitar la tolerancia al particionado debido a que habilita la tolerancia a fallos de un punto crítico.
 - Siempre es posible lograr mayor disponibilidad adoptando la eventual consistencia.



iv.v Principios de diseño para aplicaciones en la nube. (f)

- Teorema CAP (conjetura de Brewer).
- Enuncia que, en un sistema distribuido, nunca se puede garantizar a la vez las siguientes 3 características:
 - Consistencia (“**consistency**”).
 - Disponibilidad (“**availability**”).
 - Tolerancia al particionado (“**partition tolerance**”).



+

iv.v Principios de diseño para aplicaciones en la nube. (g)

- Teorema CAP (conjetura de Brewer).
- La consistencia: es decir, que todos los nodos deben garantizar la misma información al mismo tiempo, si insertamos datos (todos los nodos deben insertar los mismos datos), si actualizamos datos (todos los nodos deben actualizar los mismos datos) y si consultamos datos (todos los nodos deben devolver los mismos datos).
- Para lograr la consistencia, la comunicación entre los nodos debe ser de suma importancia; deben emitir confirmación de inserción de un dato, una vez dicho dato se almacenó en todos los nodos de la réplica.



iv.v Principios de diseño para aplicaciones en la nube. (h)

- Teorema CAP (conjetura de Brewer).
- La disponibilidad: Independientemente de si uno de los nodos, de la réplica, se ha caído o ha dejado de emitir respuestas, el sistema distribuido debe seguir en funcionamiento y aceptar peticiones tanto de escritura como de lectura.
- Una vez que se ha perdido la comunicación con un nodo, el sistema debe tener la capacidad de seguir operando mientras éste se restablece de forma automática y, una vez que lo hace, se debe sincronizar con los demás nodos.

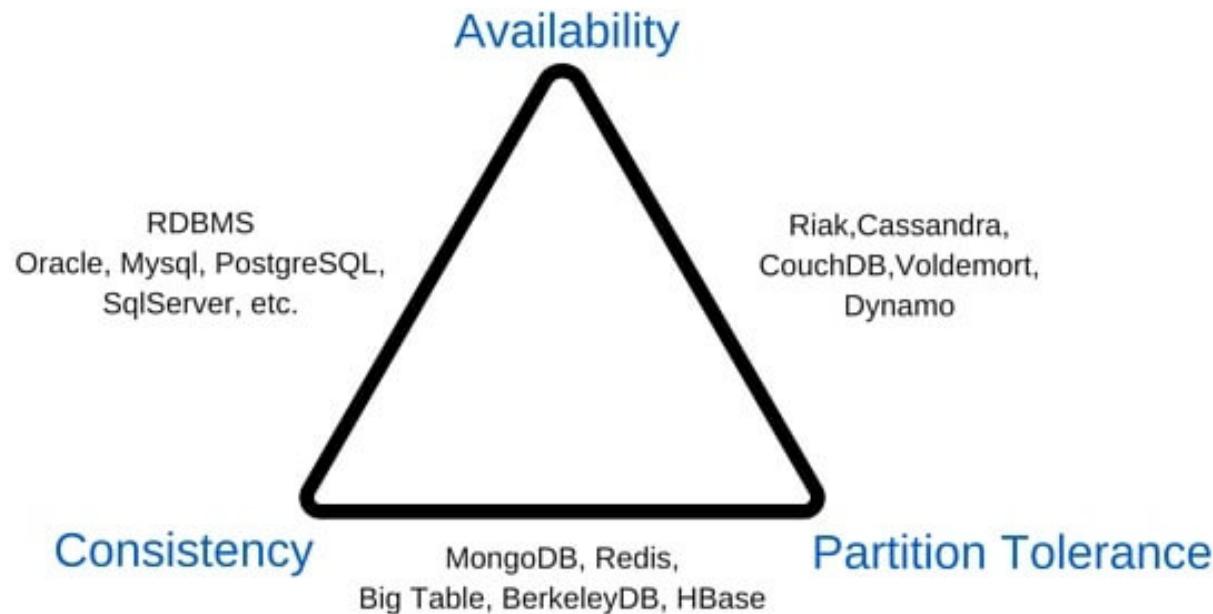


iv.v Principios de diseño para aplicaciones en la nube. (i)

- Teorema CAP (conjetura de Brewer).
- La tolerancia al particionado: El sistema debe estar disponible así existan problemas de comunicación entre los nodos, cortes de red que dificulten su comunicación o cualquier otro aspecto que genere su particionamiento.
- Por lo general, los sistemas distribuidos están divididos geográficamente, donde es normal que existan cortes de comunicación entre nodos, por lo cual, el sistema debe permitir seguir funcionando aunque existan fallas que dividan el sistema.

iv.v Principios de diseño para aplicaciones en la nube. (j)

- Teorema CAP (conjetura de Brewer).





+

iv.v Principios de diseño para aplicaciones en la nube. (k)

- Use el repositorio correcto para el trabajo correcto.
- Recomendaciones:
 - **Implemente compensación de transacciones.**
 - Si se está utilizando persistencia políglota, un efecto secundario y necesario al momento de implementar transacciones distribuidas es requerir compensación de transacciones forzosamente.
 - Recuerde que la compensación de transacciones no aplica un “**rollback**” sobre cambios comprometidos. Una compensación de transacciones aplica un “**commit**” para restaurar los cambios producidos por la transacción previamente comprometida.



iv.v Principios de diseño para aplicaciones en la nube.

- Diseñe para la recuperación automática.
- Haga todo redundante.
- Minimizar la coordinación.
- Diseñe para facilitar el escalamiento horizontal.
- Particione alrededor de límites.
- Diseñe para las operaciones.
- Use servicios administrados.
- Use el repositorio correcto para el trabajo correcto.
- **Diseñe para evolucionar el servicio.**
- Desarrolle considerando las necesidades del negocio.



iv.v Principios de diseño para aplicaciones en la nube. (a)

- Diseñe para evolucionar el servicio.
- Un diseño evolutivo es clave para la innovación continua.
- Toda aplicación exitosa evoluciona a través del tiempo, ya sea para corregir errores, agregar nuevas características funcionales, incorporar nuevas tecnologías o hacer que el sistema sea más resistente y escalable.
- Si las partes de una aplicación están altamente acopladas, resulta difícil introducir cambios en el sistema. Un cambio en una parte de la aplicación puede interrumpir el funcionamiento correcto de otra parte de la aplicación o provocar cambios que se propaguen por todo el código base.



iv.v Principios de diseño para aplicaciones en la nube. (b)

- Diseñe para evolucionar el servicio.
- Si las partes de una aplicación están altamente acopladas, resulta difícil. Cuando las partes de un sistema están diseñadas para evolucionar, los equipos pueden innovar y proporcionar continuamente nuevas características.
- Los microservicios se han convirtiendo en una manera popular de lograr un diseño evolutivo.



+

iv.v Principios de diseño para aplicaciones en la nube. (c)

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **Implemente alta cohesión y bajo acoplamiento.**
 - Siempre diseñe orientado a interfaces.
 - Respete los principios SOLID.
 - Alta cohesión en una clase significa que un cambio en alguna de sus funciones, requerirá posiblemente cambios en otras funciones de la misma clase.
 - Si un cambio en un servicio afecta o propaga cambios en otros servicios, significa que están altamente acoplados.

iv.v Principios de diseño para aplicaciones en la nube. (d)

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **Encapsule el conocimiento del dominio.**
 - Cuando un cliente consumidor de un servicio, la responsabilidad de implementar las reglas de negocio no deben recaer en el cliente.
 - El servicio debe encapsular el conocimiento del dominio (reglas de negocio) y mantener un lenguaje ubicuo.



iv.v Principios de diseño para aplicaciones en la nube. (e)

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **Implemente mensajería asíncrona.**
 - No existe mejor manera de desacoplar componentes que mediante mensajería asíncrona.
 - El productor del mensaje no dependerá de los consumidores (ni los conocerá).
 - Pueden haber más de un consumidor del mensaje que ejecutarán las acciones que tengan que dar lugar, dependiendo de su modelo de dominio.
 - Nuevos servicios pueden ser integrados a modo de “**plug-&-play**”.



+

iv.v Principios de diseño para aplicaciones en la nube. (f)

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **No utilizar conocimiento del dominio en el Gateway.**
 - Los “gateways” o API Gateway, es un patrón de diseño útil en arquitecturas de microservicios.
 - **Gateway Pattern**
 - Ruteador de “requests” (“request routing”).
 - Traducción de protocolos (“protocol translation”).
 - Balanceo de carga (“load balancing”).
 - Autenticación y Autorización.
 - Agregación.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (g)

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **No utilizar conocimiento del dominio en el Gateway.**
 - API Gateway sólo debe implementar los requerimientos antes descritos y jamás implementar cuestiones relativas al negocio, es decir, no debe implementar conocimiento del dominio debido a que ello ocasionaría una alta dependencia hacia el gateway por los servicios que hay detrás de él.
 - No debe implementar transformación de entidades de dominios diferentes en los objetos de entrada y salida del gateway.



iv.v Principios de diseño para aplicaciones en la nube. (h)

- API Gateway Pattern.
- **Categorías:** Administración, supervisión, diseño e implementación.
 - Implemente un servicio que sirva como único punto de entrada a la aplicación basada en microservicios desde solicitudes de clientes externos al API expuesta.



iv.v Principios de diseño para aplicaciones en la nube. (i)

- API Gateway Pattern.
- **Intención.**
 - Integrar un único punto de acceso, para los clientes, al consumo de las APIs expuestas por los servicios.
 - Habilitar el desacoplamiento entre el cliente / consumidor del API y los servicios expuestos.
 - Implementar “**request-routing**” mediante “**reverse-proxy**” para el consumo de los servicios (**Gateway Routing Pattern**).
 - Implementar balanceo de carga (“**load-balancer**”) para la distribución de carga entre los servicios expuestos.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (j)

- API Gateway Pattern.
- **Intención.**
 - Implementar requerimientos funcionales y/o no funcionales transversales (“**cross-cutting concerns**”) a todos los servicios (**Gateway Offloading Pattern**).
 - Implementar autenticación y autorización para el consumo de las APIs expuestas por el gateway (en conjunción con **Gatekeeper Pattern**).
 - Facilitar el consumo de las APIs a los clientes mediante la agregación (o composición) de llamadas a servicios desde el API Gateway (**Gateway Aggregation Pattern**).



iv.v Principios de diseño para aplicaciones en la nube. (k)

- API Gateway Pattern.
- **Aplicabilidad.**
 - Cuando se esté implementando una arquitectura de microservicios y sea requerido un único punto de acceso a las APIs de los servicios expuestos.
 - Cuando sea necesario minimizar las llamadas de los clientes a las APIs expuestas de los servicios mediante agregación a nivel del API Gateway.
 - Cuando la agregación de llamadas a APIs se da de forma distinta dependiendo del tipo de consumidor del API (clientes) (**Backend for Front-ends Pattern**).



iv.v Principios de diseño para aplicaciones en la nube. (I)

- API Gateway Pattern.
- **Aplicabilidad.**
 - Cuando se despliegan servicios en IaaS u “**onPremises**” y la exposición de puertos de los servicios es dinámica.
 - Cuando sea requerido aplicar cuestiones transversales como cifrado/descifrado, manejo de certificados SSL, autenticación, monitoreo, traducción de protocolos o limitación de clientes “**throttling**”.
 - Cuando sea necesario consumir servicios protegidos mediante propagación de tokens desde el Gateway asegurando el transito de la petición.



iv.v Principios de diseño para aplicaciones en la nube. (m)

- API Gateway Pattern.
- **Aplicabilidad.**
 - Cuando se requiera simplificar el consumo de las APIs al cliente mediante un único punto de acceso.
 - Cuando sea requerido enrutar un acceso externo, desde internet, a un servicio expuesto en una red virtual (VM) o red interna.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (n)

- API Gateway Pattern.
- **Ventajas:**
 - Habilitar un único punto de acceso al API.
 - Habilita minimizar las peticiones de los clientes mediante agregación.
 - Habilitando un único punto de acceso se maximiza los mecanismos de seguridad en peticiones provenientes desde el exterior.
 - Evita que el cliente conozca la ruta y puertos del servicio detrás del Gateway.
 - Permite habilitar patrones de diseño como **Throttling Pattern** para limitar a los clientes del consumo del API o integración de mecanismos de seguridad mediante **Gatekeeper Pattern**.



iv.v Principios de diseño para aplicaciones en la nube. (ñ)

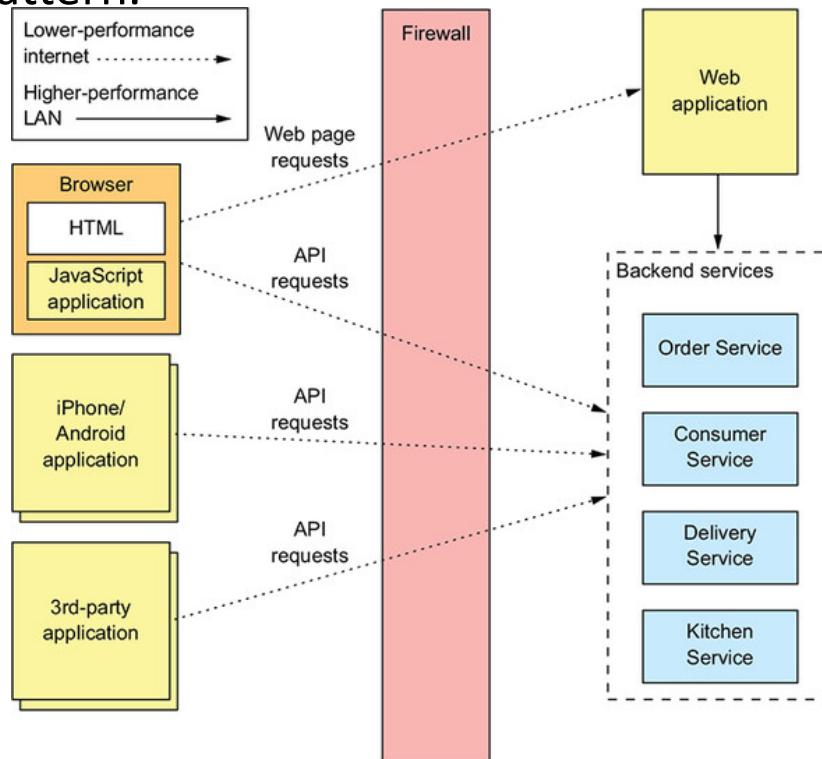
- API Gateway Pattern.
- **Ventajas:**
 - Integrar mecanismos de enruteamiento y balanceo de carga para las APIs de los servicios expuestos.
 - Implementar mecanismos no funcionales generales o transversales a las APIs expuestas por el Gateway.

iv.v Principios de diseño para aplicaciones en la nube. (o)

- API Gateway Pattern.
- **Desventajas:**
 - Único punto de quiebre (verificar escalabilidad). Verifique que el Gateway este diseñado para poder escalar horizontalmente sin inconvenientes.
 - Es fácil introducir acoplamiento entre el Gateway y los servicios “backend” (microservicios).
 - El Gateway agrega latencia en el tiempo de respuesta hacia el cliente.
 - Se puede llegar a generar cuellos de botella a nivel del API Gateway.
 - Nunca se debe implementar lógica de negocio a nivel del API Gateway.

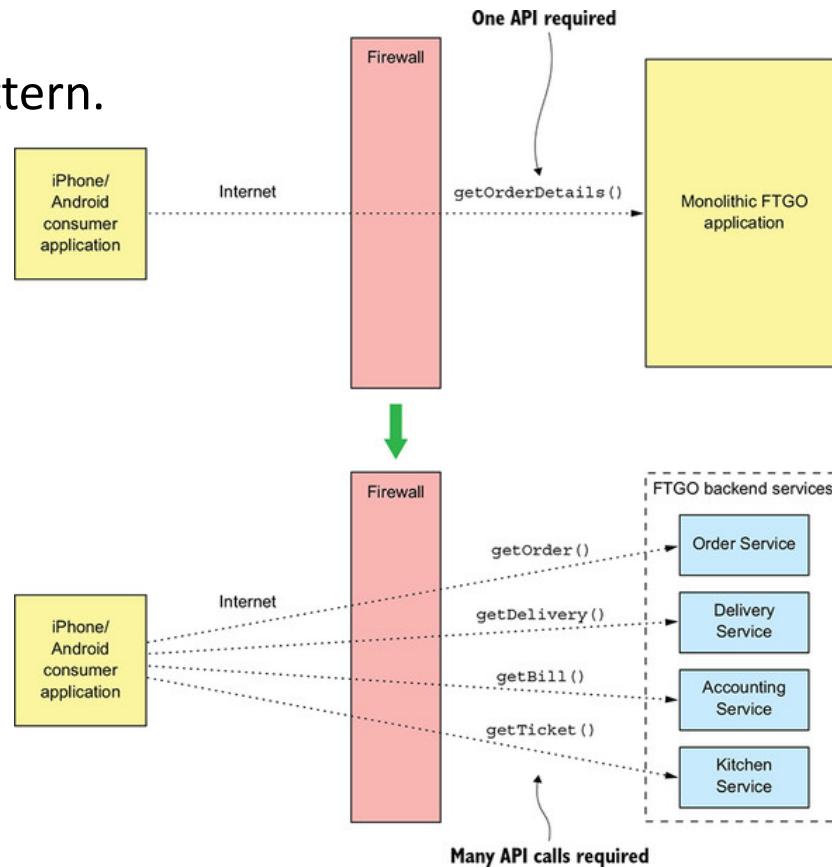
iv.v Principios de diseño para aplicaciones en la nube. (p)

- API Gateway Pattern.



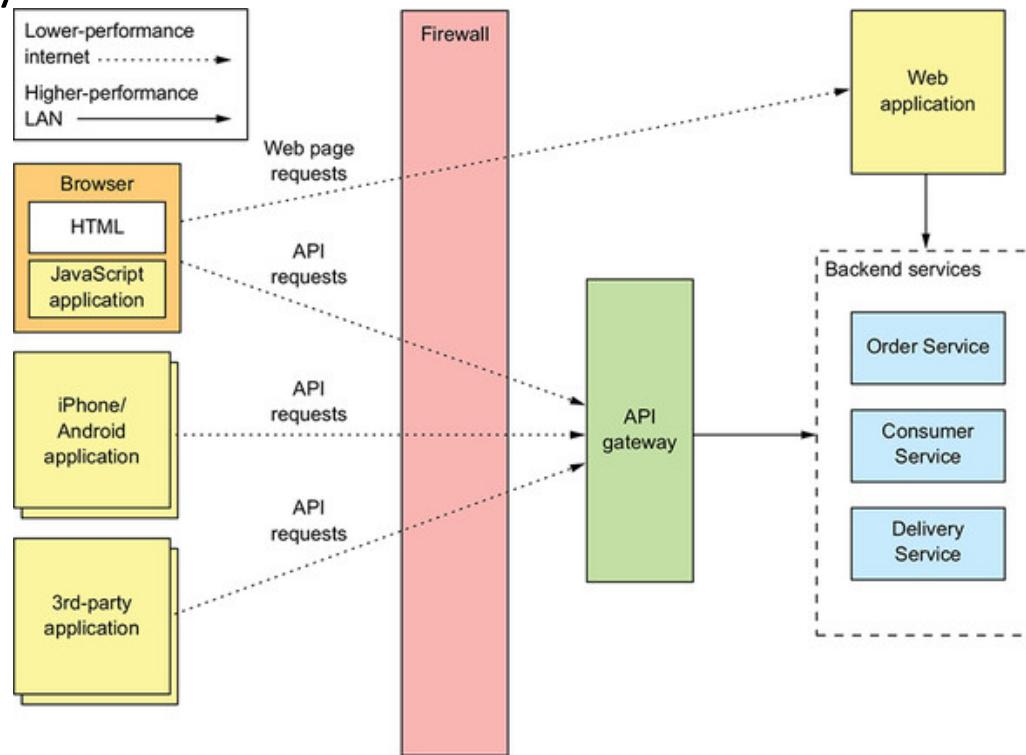
iv.v Principios de diseño para aplicaciones en la nube. (q)

- API Gateway Pattern.



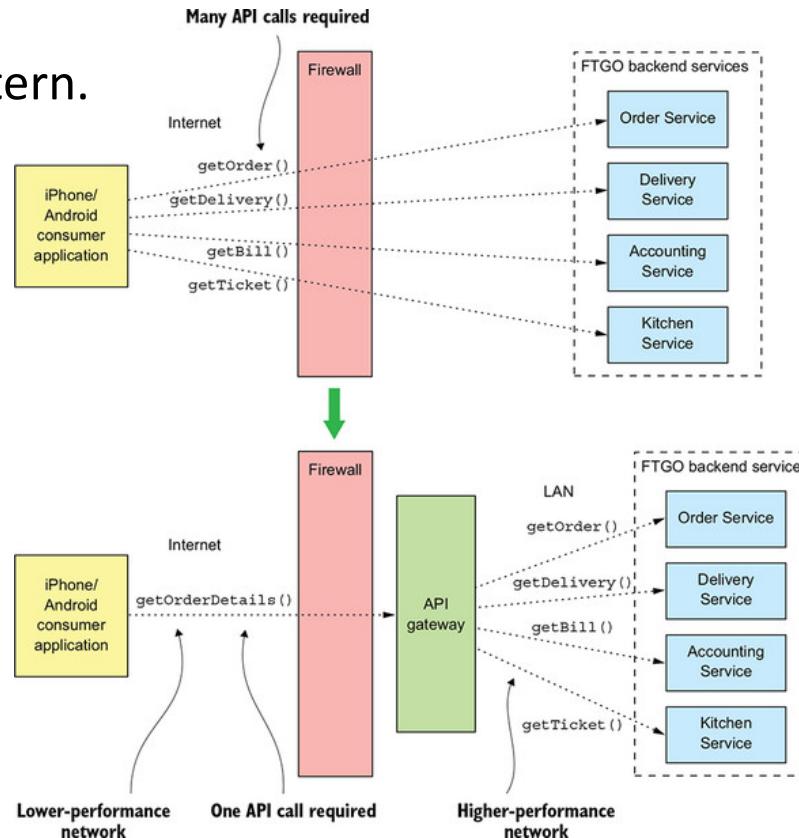
iv.v Principios de diseño para aplicaciones en la nube. (r)

- API Gateway Pattern.



iv.v Principios de diseño para aplicaciones en la nube. (s)

- API Gateway Pattern.





+

iv.v Principios de diseño para aplicaciones en la nube. (t)

- Práctica 18. Api Gateway Pattern
- Analiza la aplicación **18-Image-Microservice** y **18-Price-Microservice**.
- Ingresar a la ruta: **{tu-workspace}/18-Image-Microservice**
- Importar el proyecto **18-Image-Microservice** en STS.
- Analiza la aplicación:
 - ¿En que puerto levanta el servidor tomcat embebido?
 - ¿Cuál es el servlet context del aplicativo?
 - Ejecuta la aplicación y realiza una solicitud al servicio que expone.



iv.v Principios de diseño para aplicaciones en la nube. (u)

- Práctica 18. Api Gateway Pattern
- Ingresar a la ruta: **{tu-workspace}/18-Price-Microservice**
- Importar el proyecto **18-Price-Microservice** en STS.
- Analiza la aplicación:
 - ¿En que puerto levanta el servidor tomcat embebido?
 - ¿Cuál es el servlet context del aplicativo?
 - Ejecuta la aplicación y realiza una solicitud al servicio que expone.



+

iv.v Principios de diseño para aplicaciones en la nube. (v)

- Práctica 18. Api Gateway Pattern
- Analiza la aplicación **18-Gateway-Microservice**.
- Ingresar a la ruta: **{tu-workspace}/18-Gateway-Microservice**
- Importar el proyecto **18-Gateway-Microservice** en STS.
- Analizar las interfaces **ImageMicroserviceClient** y **PriceMicroserviceClient** del paquete **com.consulting.mgt.springboot.practica18.gateway.client**.
- Revisar las implementaciones **ImageMicroserviceClientImpl** y **PriceClientImpl** del paquete **com.consulting.mgt.springboot.practica18.gateway.client.impl**.



iv.v Principios de diseño para aplicaciones en la nube. (w)

- Práctica 18. Api Gateway Pattern
- Analizar la clase **GatewayMicroserviceConfig** del paquete **com.consulting.mgt.springboot.practica18.gateway._config**, en la cual únicamente deberá definir un bean **RestTemplate**.
- El proyecto **18-Gateway-Microservice** implementará un patrón Gateway por agregación, es decir, implementará y expondrá un servicio para devolver un producto dependiendo del “**front-end**” que lo invoque (Implementará “**Backends or Frontends Pattern**”).



+

iv.v Principios de diseño para aplicaciones en la nube. (x)

- **Práctica 18. Api Gateway Pattern**
- Si el “**front-end**” que lo invoca es un aplicativo móvil, devolverá únicamente el precio del producto. En caso de que el “**front-end**” que lo invoque sea un aplicativo de escritorio, devolverá el precio del producto y su imagen descriptiva.
- Analizar las clases **DesktopProductAggregate** y **MobileProductAggregate** del paquete **com.consulting.mgt.springboot.practica18.gateway.aggregates**.
- Para que el **API Gateway** a implementar devuelva el precio de un producto o, el precio y la imagen descriptiva de un producto, es necesario consumir los microservicios **18-Image-Microservice** y **18-Price-Microservice**, a través de sus implementaciones cliente, para realizar la agregación conforme sea requerido.



iv.v Principios de diseño para aplicaciones en la nube. (y)

- **Práctica 18. Api Gateway Pattern**
- Define los siguientes servicios sobre el **@RestController ApiGatewayController** del paquete **com.consulting.mgt.springboot.practica18.gateway.restcontroller**:
 - /desktop
 - Devolverá un producto visualizable para aplicaciones de escritorio el cual contenga precio e imagen descriptiva del producto.
 - /mobile
 - Devolverá un producto visualizable para aplicaciones móviles el cual contenga únicamente el precio del producto.



+

iv.v Principios de diseño para aplicaciones en la nube. (z)

- **Práctica 18. Api Gateway Pattern**
- Utilice **RestTemplate** para implementar las clases concretas **ImageMicroserviceClientImpl** y **PriceClientImpl**, del paquete **com.consulting.mgt.springboot.practica18.gateway.client.impl**, para completar su implementación.
- Realice las configuraciones necesarias para que funcione el proyecto **18-Gateway-Microservice**.
- Analice el archivo **application.properties**. No defina en código duro ninguna propiedad o variable, como puertos o URLs de conexión a los servicios **18-Image-Microservice** o **18-Price-Microservice**.



iv.v Principios de diseño para aplicaciones en la nube. (a')

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **Diseñe a partir de interfaces / contratos bien definidos.**
 - Exponga APIs mediante protocolos abiertos.
 - No implemente capas de traducción de protocolos o versiones de API entre servicios.
 - Implemente interfaces API bien definidas, incluya versionado para poder evolucionar fácilmente; de esta forma se mantiene compatibilidad hacia versiones anteriores.

iv.v Principios de diseño para aplicaciones en la nube. (b')

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **Diseñe a partir de interfaces / contratos bien definidos.**
 - Contratos o interfaces de APIs públicas deberán exponer APIs sobre HTTPS / RESTful mientras que, los servicios de “**back-end**” deberán utilizar protocolos de mensajería asíncrona o RPC para la comunicación entre ellos, preferentemente.
 - Desarrolle pruebas unitarias y de integración utilizando “mocks” de servicios a partir de las interfaces / contratos definidos.



iv.v Principios de diseño para aplicaciones en la nube. (c')

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **Delegue funcionalidades transversales en servicios separados.**
 - Implemente funcionalidades transversales, en servicios separados ya sea mediante el despliegue de nuevos microservicios o mediante la integración de librerías. Esto permite la evolución del servicio y su versionado mediante un API versionada o mediante versión de la librería.
 - No promueva la duplicidad de código.
 - **Gateway Offloading Pattern**



iv.v Principios de diseño para aplicaciones en la nube. (d')

- Diseñe para evolucionar el servicio.
- Recomendaciones:
 - **Despliegue servicios de forma independiente.**
 - Utilice mecanismos de CI / CD atómicos por servicio; de esta forma se permite la actualización de componentes de forma más rápida y segura, evitando el despliegue de todos los servicios de forma sincronizada.
 - La corrección de errores o agregación de nuevas funcionalidades pueden ejecutarse a un ritmo más rápido.
 - Aplique la metodología DevOps.



iv.v Principios de diseño para aplicaciones en la nube.

- Diseñe para la recuperación automática.
- Haga todo redundante.
- Minimizar la coordinación.
- Diseñe para facilitar el escalamiento horizontal.
- Particione alrededor de límites.
- Diseñe para las operaciones.
- Use servicios administrados.
- Use el repositorio correcto para el trabajo correcto.
- Diseñe para evolucionar el servicio.
- **Desarrolle considerando las necesidades del negocio.**



+

iv.v Principios de diseño para aplicaciones en la nube. (a)

- Desarrolle considerando las necesidades del negocio.
- Cada decisión de diseño debe ser justificada por un requisito de negocio.
- El aplicativo:
 - ¿Prevé tener millones de usuarios o unos pocos miles o cientos?
 - ¿Es aceptable una interrupción de la aplicación durante 1 hora?
 - ¿Espera grandes picos de tráfico o una carga de trabajo predecible?
 - Etcétera...
- Es altamente importante el levantamiento de requerimientos no funcionales.



+

iv.v Principios de diseño para aplicaciones en la nube. (b)

- Desarrolle considerando las necesidades del negocio.
- Recomendaciones:
 - **Definir objetivos de negocio.**
 - Siempre definir un Plan de Recuperación de Desastres o **DRP** (“**Disaster Recovery Plan**”).
 - Considerar **RTO** (Objetivo de Tiempo de Recuperación, “**Recovery Time Objective**”), **RPO** (Objetivo de Punto de Recuperación, “**Recovery Point Objective**”) y **MTO** (Interrupción Tolerable Máxima, “**Maximum Tolerable Outage**”).

iv.v Principios de diseño para aplicaciones en la nube. (c)

- Desarrolle considerando las necesidades del negocio.
- Recomendaciones:
 - **Definir objetivos de negocio.**
 - **RTO (“Recovery Time Objective”):** Tiempo que el negocio necesita para recuperar sus sistemas después de inactividad producida por un incidente.
 - **RPO (“Recovery Point Objective”):** Cantidad de datos que la empresa puede permitirse perder en caso de que sufriera un tiempo de inactividad.
 - **MTO (“Maximum Tolerable Outage”):** Es el tiempo máximo que el negocio tolera una interrupción antes de considerarse desastre.



iv.v Principios de diseño para aplicaciones en la nube. (d)

- Desarrolle considerando las necesidades del negocio.
- Recomendaciones:
 - **Modelar la aplicación en torno al dominio empresarial.**
 - Definir modelos de datos empresariales en base a los requisitos de negocio.
 - Considerar un diseño basado en el dominio del negocio (DDD, “**Domain Driven Design**”) para crear modelos de dominio, que reflejen los procesos empresariales y los casos de uso.
 - Utilice un lenguaje ubicuo.
 - Diseñar los microservicios entorno a contexto delimitado y por carga de trabajo (necesidades de escalabilidad).



iv.v Principios de diseño para aplicaciones en la nube. (e)

- Desarrolle considerando las necesidades del negocio.
- Recomendaciones:
 - **Considerar requisitos funcionales y no funcionales.**
 - Los requisitos funcionales permiten evaluar si la aplicación realiza las tareas u operaciones requeridas por el negocio.
 - Los requisitos no funcionales permiten evaluar si la aplicación realiza estas tareas correctamente con los SLAs establecidos, es decir, asegúrese de que sean comprendidos los requisitos de escalabilidad, disponibilidad y latencia.
 - Los requisitos no funcionales influirán en las decisiones de diseño y la elección de la tecnología para la implementación funcional.



+

NETFLIX
OSS

iv.v Principios de diseño para aplicaciones en la nube. (f)

- Desarrolle considerando las necesidades del negocio.
- Recomendaciones:
 - **Planear el crecimiento o evolución.**
 - Una aplicación podrá satisfacer los requerimientos actuales en términos de volumetrías, número de usuarios concurrentes, cantidad de datos almacenados, etc.
 - Implementando una arquitectura sólida siempre se puede controlar el crecimiento sin realizar cambios importantes a la arquitectura.
 - Diseñe para facilitar el escalamiento horizontal y Particione alrededor de límites aunque considere no ser necesario.



Resumen de la lección

iv.v Principios de diseño para aplicaciones en la nube

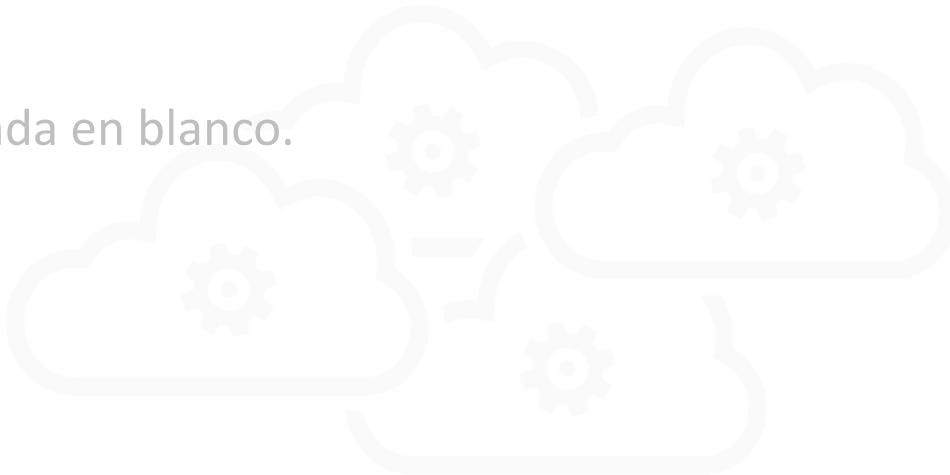
- Aprendimos 10 principios de diseño de aplicaciones en la nube para aplicar patrones de diseño que permiten implementar arquitecturas de microservicios resistentes, escalables, mantenibles, tolerantes a fallos y que promueban la alta disponibilidad del servicio.
- Implementamos diversos patrones de diseño especializados para la nube.
- Comprendimos diversos mecanismos para implementar transacciones distribuidas mediante compensación de transacciones.
- Analizamos múltiples patrones de diseño para la nube considerando sus ventajas y desventajas.



+

NETFLIX
OSS

Esta página fue intencionalmente dejada en blanco.



Microservices



+

iv. Arquitectura de Microservicios

- iv.i ¿Qué es la arquitectura orientada a microservicios?
- iv.ii Descomponiendo aplicaciones monolíticas.
- iv.iii Protocolos ligeros de comunicación para microservicios.
- iv.iv Aplicaciones “cloud-native”.
- iv.v Principios de diseño para aplicaciones en la nube.
- iv.vi **Orquestación vs Coreografía.**
- iv.vii Gestión de Transacciones ACID vs BASE.
- iv.viii API Manager



+

iv.vi Orquestación vs Coreografía.



+

NETFLIX
OSS

Objetivos de la lección

iv.vi Orquestación vs Coreografía.

- Analizar la diferencia entre orquestar servicios y coreografiar servicios.
- Comprender las dificultades de cada uno de los mecanismos de coordinación entre microservicios.
- Analizar las ventajas y desventajas de implementar orquestación y/o coreografía.

iv.vi Orquestación vs Coreografía. (a)

- En el diseño de una arquitectura orientada a microservicios existen dos aproximaciones para manejar la interacción entre microservicios:
 - **Orquestación:** Orquestar significa que hay una entidad central (idealmente un microservicio) que coordina la comunicación entre varios servicios (microservicios) para ejecutar un flujo de trabajo concreto.
 - **Coreografía:** La comunicación entre componentes se da mediante eventos, es decir, cada microservicio realiza su función y al terminar, realiza el envío de un evento para que los suscriptores a dicho evento continúen con su trabajo (**Publish/Subscribe Pattern**).



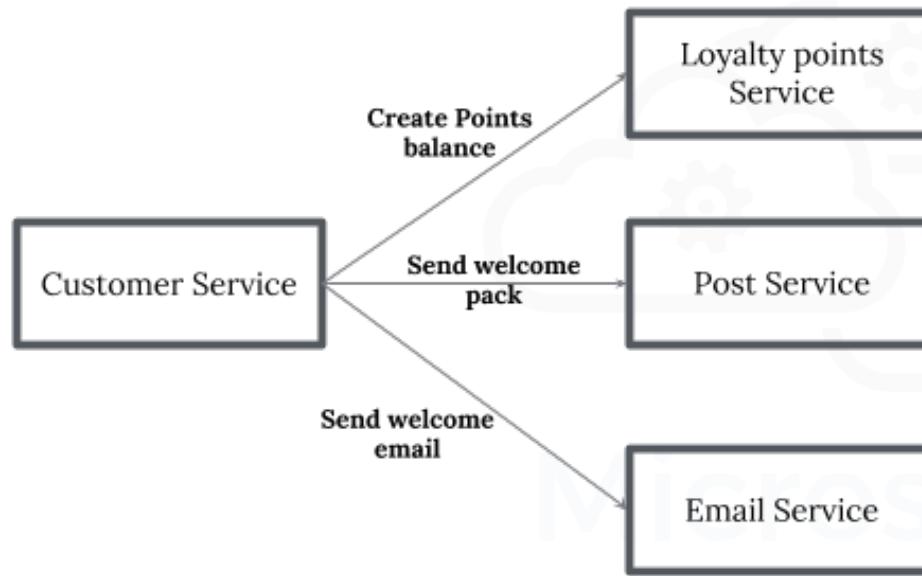
iv.vi Orquestación vs Coreografía. (b)

- Orquestación.
- La orquestación es la forma más confiable y aplicable de manejar las interacciones entre diferentes servicios en una Arquitectura Orientada a Servicios (SOA).
- En la orquestación, generalmente hay un controlador central que actúa como "**orquestador**" de las interacciones generales del servicio.
- Sigue un patrón de comunicación síncrono (solicitud / respuesta).
- Sólo el controlador central es responsable de todas las interacciones.

iv.vi Orquestación vs Coreografía. (c)

- Orquestación.

Orchestration Example





iv.vi Orquestación vs Coreografía. (d)

- Orquestación.
- Con el enfoque de comunicación mediante “**orquestación**”, el orquestador llama a un servicio y espera la respuesta del servicio antes de llamar al siguiente servicio.
- Una vez que sea recibida la respuesta del servicio llamado, será realizada la siguiente llamada, al siguiente servicio, para la interacción.
- En este enfoque, toda solicitud será gestionada/mediada/orquestada por el orquestador.



iv.vi Orquestación vs Coreografía. (e)

- Orquestación.
- Ventajas:
 - La orquestación proporciona una forma efectiva de coordinar el flujo de la aplicación cuando hay procesamiento síncrono, por ejemplo, si el Servicio A debe completarse con éxito antes de invocar el Servicio B.
- Desventajas:
 - Dependencia:
 - Alto acoplamiento del orquestador con los servicios que orquesta.
 - Alto acoplamiento de los servicios creando dependencias entre los demás servicios. Si un Servicio A está inactivo, nunca se llamará a otro Servicio B (ni a un supuesto Servicio C).



iv.vi Orquestación vs Coreografía. (f)

- Orquestación.
- Desventajas:
 - Dependencia:
 - La dependencia entre servicios es el punto de cuestión (crítico) en un entorno de ejecución distribuido.
 - Punto único de quiebre:
 - El orquestador es un punto único de control y coordinador único, siendo también un único punto de quiebre.
 - Si el orquestador se cae, todo el procesamiento se detiene y la aplicación falla por completo.



+

NETFLIX
OSS

iv.vi Orquestación vs Coreografía. (g)

- Orquestación.
- Desventajas:
 - Tiempo extra de ejecución:
 - El tiempo de procesamiento síncrono bloquea las solicitudes hasta sea recibida su respuesta.
 - Debido a la capa extra de orquestación, se incrementa el tiempo de ejecución y espera de las solicitudes entrantes, por ejemplo, el tiempo total de procesamiento de extremo a extremo es la suma de tiempo que toma el servicio A + Servicio B + Servicio C + el tiempo de orquestación y mediación por el orquestador.



+

NETFLIX
OSS

iv.vi Orquestación vs Coreografía. (h)

- Coreografía.
- En una arquitectura orientada a microservicios, el punto focal es evitar las dependencias entre microservicios.
- Un microservicio es un servicio hace una sola cosa bien y, que es independiente y puede ejecutarse de manera autónoma.
- Se promueve la implementación de arquitecturas reactivas.
- Una arquitectura “**reactiva**” se considera como un patrón de arquitectura impulsado por eventos aplicado a microservicios.



+

NETFLIX
OSS

iv.vi Orquestación vs Coreografía. (i)

- Coreografía.
- Los patrones de arquitectura “**reactiva**” resuelven algunos de los desafíos del enfoque de orquestación de servicios mediante composición.
- Cuando un servicio sabe qué hacer, cuando un determinado evento ocurra, se denomina coordinación o coreografía.
- Los servicios saben a qué reaccionar y cómo, antes de tiempo, mediante un enfoque autónomo.



+

iv.vi Orquestación vs Coreografía. (j)

- Coreografía.
- Los servicios utilizan un “**flujo de eventos**” para la comunicación asíncrona de eventos lo cual, permite que varios servicios pueden consumir los mismos eventos, realizar algún procesamiento y luego, producir sus propios eventos y, de nuevo, incluir dicho evento en la secuencia del flujo de eventos, todo al mismo tiempo.
- La naturaleza asíncrona de una arquitectura “**reactiva**” elimina el tiempo de bloqueo o de espera que ocurre con el procesamiento de tipo orquestación mediante el enfoque de comunicación síncrona (solicitud / respuesta).



+

NETFLIX
OSS

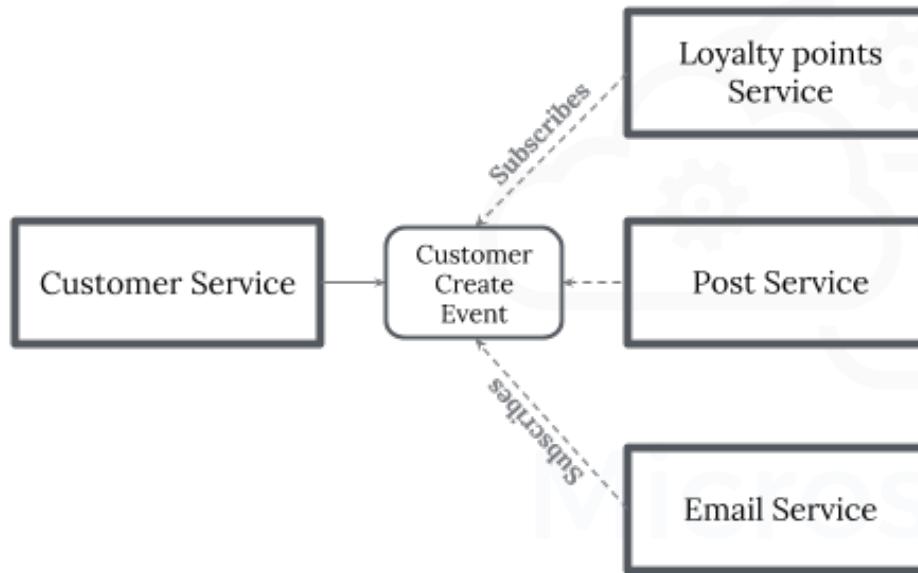
iv.vi Orquestación vs Coreografía. (k)

- Coreografía.
- Los servicios se vuelven “**reactivos**” debido a que “**reaccionan**” ante un evento originado en el tiempo lo cual, produce que dichos servicios ejecuten alguna lógica de negocio, disparen un nuevo evento y sigan escuchando eventos.
- El uso de un motor de “**flujo de eventos**”, como RabbitMQ, Kafka u otros, permite que la comunicación entre el productor y los consumidores se desacople habilitando que, el productor no necesite saber si el consumidor está activo y funcionando antes de producir un evento, o si el consumidor recibió el evento que se produjo.

iv.vi Orquestación vs Coreografía. (I)

- Coreografía.

Choreography Example

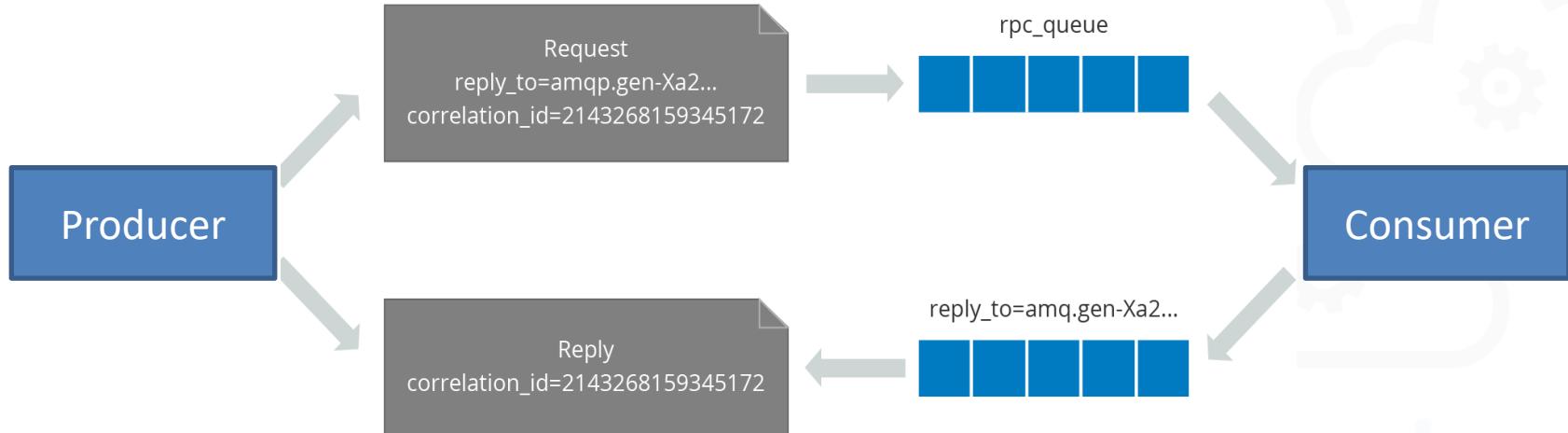


iv.vi Orquestación vs Coreografía. (m)

- Coreografía.
- La coreografía permite que los productores pueden querer dirigir los eventos a un servicio (o servicios) específico y recibir un reconocimiento de que el consumidor lo recibió.
- Por otro lado, los consumidores y productores pueden requerir consumir y producir eventos desde y hacia el “flujo de eventos”. Este es un patrón válido y frecuentemente utilizado en enfoques de comunicación en arquitecturas reactivas.

iv.vi Orquestación vs Coreografía. (n)

- Coreografía.





iv.vi Orquestación vs Coreografía. (ñ)

- Coreografía.
- Ventajas:
 - La coreografía habilita un procesamiento más rápido, ya que los servicios se pueden ejecutar de forma paralela sin depender del servicio de orquestación.
 - Es más fácil agregar y actualizar servicios, ya que se pueden conectar o desconectar fácilmente del flujo de eventos.
 - Se alinean bien con un modelo de entrega continua (metodologías agiles) ya que los equipos pueden centrarse en servicios particulares en lugar de centrarse en la aplicación completa.
 - El control del flujo de la aplicación se distribuye, por lo que no hay un único orquestador central.



+

iv.vi Orquestación vs Coreografía. (o)

- Coreografía.
- Ventajas:
 - Se pueden usar varios patrones con una arquitectura reactiva para proporcionar beneficios adicionales (Event Sourcing Pattern, Sagas Pattern, entre otros).
 - Se habilita el escalado horizontal de servicios más fácilmente.

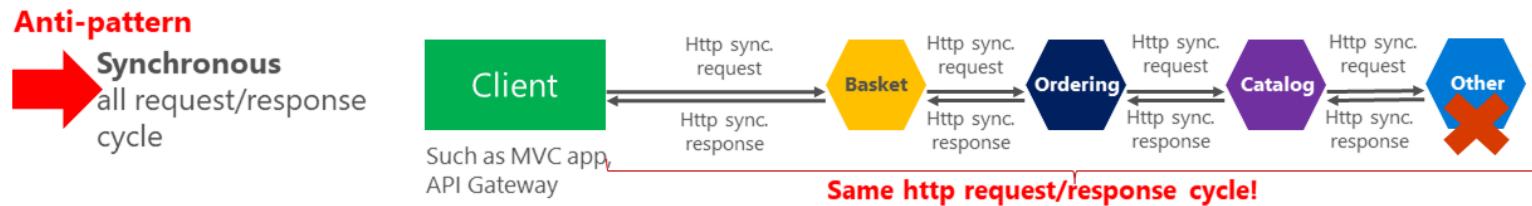


iv.vi Orquestación vs Coreografía. (p)

- Coreografía.
- Desventajas:
 - La programación asíncrona es a menudo un cambio de paradigma, lo cual requiere un cambio mental significativo para los desarrolladores en el modo de programación.
 - De forma particular, la coreografía se puede realizar de varias maneras.
 - Que cada servicio invoque al(los) servicio(s) que requiera de forma síncrona.
 - Que cada servicio invoque al(los) servicio(s) que requiera de forma asíncrona enviando un evento a cada uno de los servicios en cuestión, ó
 - Que cada servicio invoque al(los) servicio(s) que requiera de forma asíncrona enviando un evento y, que cada servicio interesado se suscriba a dicho evento.

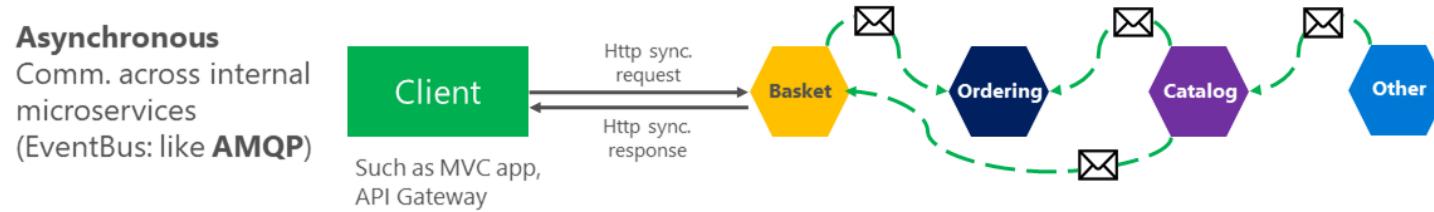
iv.vi Orquestación vs Coreografía. (q)

- Coreografía.
- Que cada servicio invoque al(los) servicio(s) que requiera de forma síncrona.



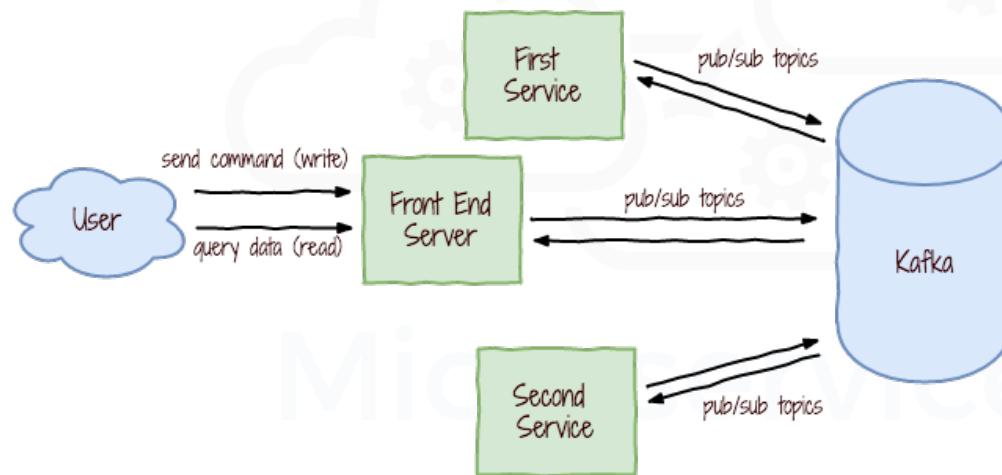
iv.vi Orquestación vs Coreografía. (r)

- Coreografía.
- Que cada servicio invoque al(los) servicio(s) que requiera de forma asíncrona enviando un evento a cada uno de los servicios en cuestión.



iv.vi Orquestación vs Coreografía. (s)

- Coreografía.
- Que cada servicio invoque al(los) servicio(s) que requiera de forma asíncrona enviando un evento y, que cada servicio interesado se suscriba a dicho evento.





iv.vi Orquestación vs Coreografía. (t)

- Coreografía.
- Desventajas:
 - La complejidad del flujo de comunicaciones entre microservicios es nuevamente un punto de preocupación ya que, en lugar de tener el control de flujo centralizado en un orquestador, el control del flujo ahora se divide y distribuye a través de los servicios individuales.
 - Cada servicio tendrá su propia lógica del flujo, y esta lógica identificará cuándo y cómo debe reaccionar el servicio basándose en los datos específicos del evento al cual está suscrito en el flujo de eventos.



+

NETFLIX
OSS

iv.vi Orquestación vs Coreografía. (u)

- Orquestación vs Coreografía.
- La orquestación es poco recomendable, ya que requiere de un mecanismo que se encargue de distribuir peticiones y enviar datos a un servicio cuando otro ya haya acabado su trabajo, como si se tratara de un director de orquesta. Posiblemente sea recomendable en ambientes transaccionales que dependan del patrón 2PC o que su migración a un esquema de compensación de transacciones sea difícil.
- La orquestación no es recomendable en la mayoría de las casuísticas.



iv.vi Orquestación vs Coreografía. (v)

- Orquestación vs Coreografía.
- Es cierto que un servicio que ofrezca una funcionalidad atómica, como un CRUD, no sirve de mucho, sin embargo un CRUD, no es un microservicio.
- Lo que da valor al negocio no es un único microservicio atómico sino un API que componga múltiples microservicios y ofrezca una funcionalidad de negocio. ¿Entonces podríamos recurrir a orquestación? No.
- La orquestación es recomendable en aplicaciones consumidoras o UI de cliente ligero.



iv.vi Orquestación vs Coreografía. (w)

- Orquestación vs Coreografía.
- La coreografía es la aproximación más recomendada para arquitecturas de microservicios, donde cada componente conoce perfectamente su función, sabe en qué momento debe actuar y a dónde enviar sus datos al terminar. Es decir, cada microservicio es autónomo y no necesita de una directriz externa.
- La coreografía elimina el punto de tener un único punto de fallo.
- La coreografía aumenta la complejidad de flujos de negocio, pero habilita su escalabilidad.



+

iv.vi Orquestación vs Coreografía. (x)

- Orquestación vs Coreografía.
- ¿Quién dijo que hacer microservicios hiba a ser fácil?
 - Los objetivos de las arquitecturas de microservicios son:
 - Escalabilidad
 - Independencia
 - Robustez
 - Disponibilidad
 - Facilidad de mantenimiento
 - Facilidad de despliegue
 - Entre otros...
 - Lo anterior, no se obtiene con una arquitectura simple al contrario, los sistemas se vuelven más complejos en su diseño y organización.



iv.vi Orquestación vs Coreografía. (y)

- Publisher / Subscriber Pattern
- **Categorías:** Rendimiento, escalabilidad, mensajería.
 - Permite a una aplicación enviar eventos de forma asíncrona a varios consumidores interesados sin necesidad de acoplar los productores de los consumidores.



iv.vi Orquestación vs Coreografía. (z)

- Publisher / Subscriber Pattern
- **Intención.**
 - Transmitir los mensajes del remitente a todos los receptores interesados.
 - Desacoplar productores de consumidores.
 - Evitar el bloqueo inherente de enviar una solicitud (request) y esperar su respuesta (response).
 - Habilitar un mecanismo asíncrono de comunicación uno a muchos.

iv.vi Orquestación vs Coreografía. (a')

- Publisher / Subscriber Pattern
- **Aplicabilidad.**
 - Cuando una aplicación necesita difundir información a un número significativo de consumidores.
 - Cuando una aplicación necesita comunicarse con una o más aplicaciones o servicios desarrollados de forma independiente, que pueden utilizar diferentes plataformas, lenguajes de programación y protocolos de comunicación.
 - Cuando una aplicación puede enviar información a los consumidores sin requerir respuestas en tiempo real de los consumidores.



+

NETFLIX
OSS

iv.vi Orquestación vs Coreografía. (b')

- Publisher / Subscriber Pattern
- **Aplicabilidad.**
 - Cuando los sistemas que se están integrando están diseñados para admitir un modelo eventual de consistencia para el resguardo de sus datos.
 - Cuando una aplicación necesita comunicar información a varios consumidores, que pueden tener diferentes requisitos de disponibilidad o distintos momentos de actividad en comparación con el remitente / productor del mensaje.



+

NETFLIX
OSS

iv.vi Orquestación vs Coreografía. (c')

- Publisher / Subscriber Pattern
- **Ventajas:**
 - Permite implementar de forma granular los servicios de las aplicaciones en partes más pequeñas, menos acopladas, más modulares, más manejables y mejora su mantenibilidad.
 - Permite la flexibilidad debido a que pueden existir relaciones direccionales entre publicadores y suscriptores.
 - Representa la mejor solución para sistemas distribuidos y para arquitecturas orientadas a microservicios.



+

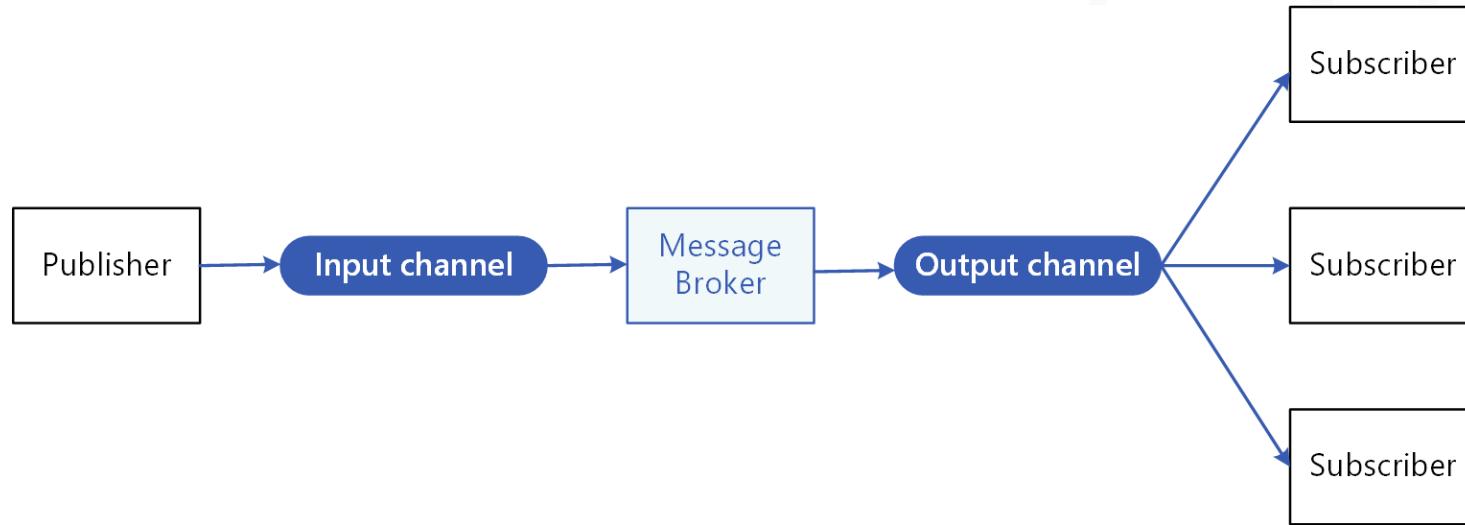
NETFLIX
OSS

iv.vi Orquestación vs Coreografía. (d')

- Publisher / Subscriber Pattern
- **Desventajas:**
 - Dificultad para el rastreo o debug de errores.
 - Requiere de una amplia documentación de productores y suscriptores de eventos, para su mantenimiento.
 - No se recomienda en aplicaciones donde se requiera información de vuelta, por parte de los suscriptores, en tiempo real.
 - Se debe implementar servicios idempotentes para asegurar la alta disponibilidad del servicio.
 - No recomendable para comunicación entre pocos servicios intercomunicados.

iv.vi Orquestación vs Coreografía. (e')

- Publisher / Subscriber Pattern



iv.vi Orquestación vs Coreografía. (f')

- Publisher / Subscriber Pattern

