

## INICIO

# Desarrollo de Microservicios con Spring Cloud Netflix OSS

ISC. Ivan Venor García Baños





+

**NETFLIX**  
**OSS**

# Agenda

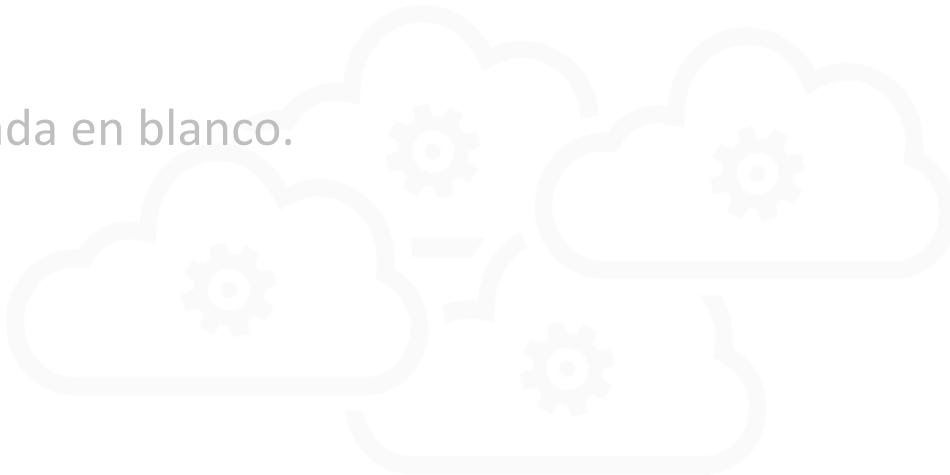
1. Presentación
2. Objetivos
3. Contenido
4. Despedida



Microservices



Esta página fue intencionalmente dejada en blanco.



Microservices



+

### **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**

### 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**

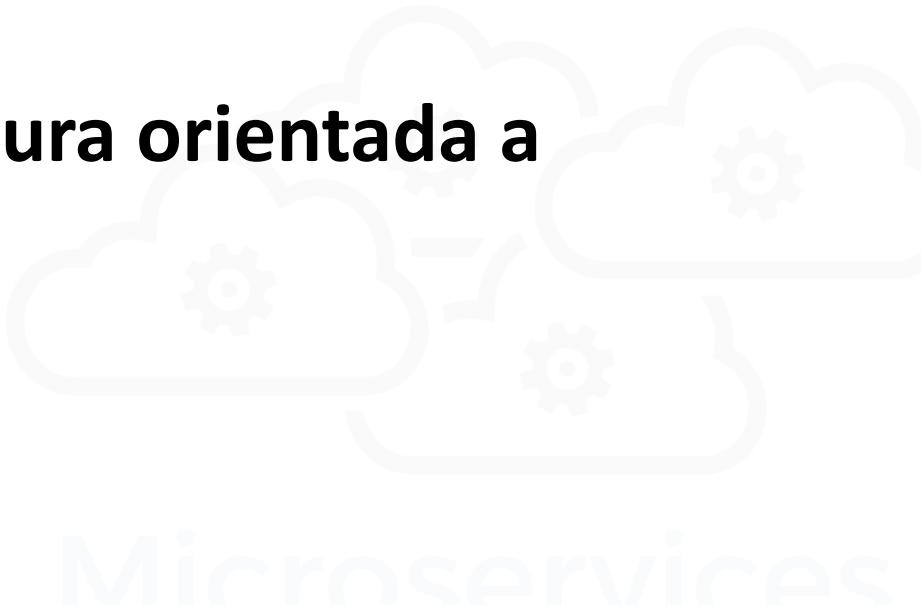
- iv.i ¿Qué es la arquitectura orientada a microservicios?**
- iv.ii Descomponiendo aplicaciones monolíticas.
- iv.iii Protocolos ligeros de comunicación para microservicios.
- iv.iv Aplicaciones “cloud-native”.
- iv.v Principios de diseño para aplicaciones en la nube.
- iv.vi Orquestación vs Coreografía.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



+

**NETFLIX**  
**OSS**

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



Microservices



+

**NETFLIX**  
**OSS**

## Objetivos de la lección

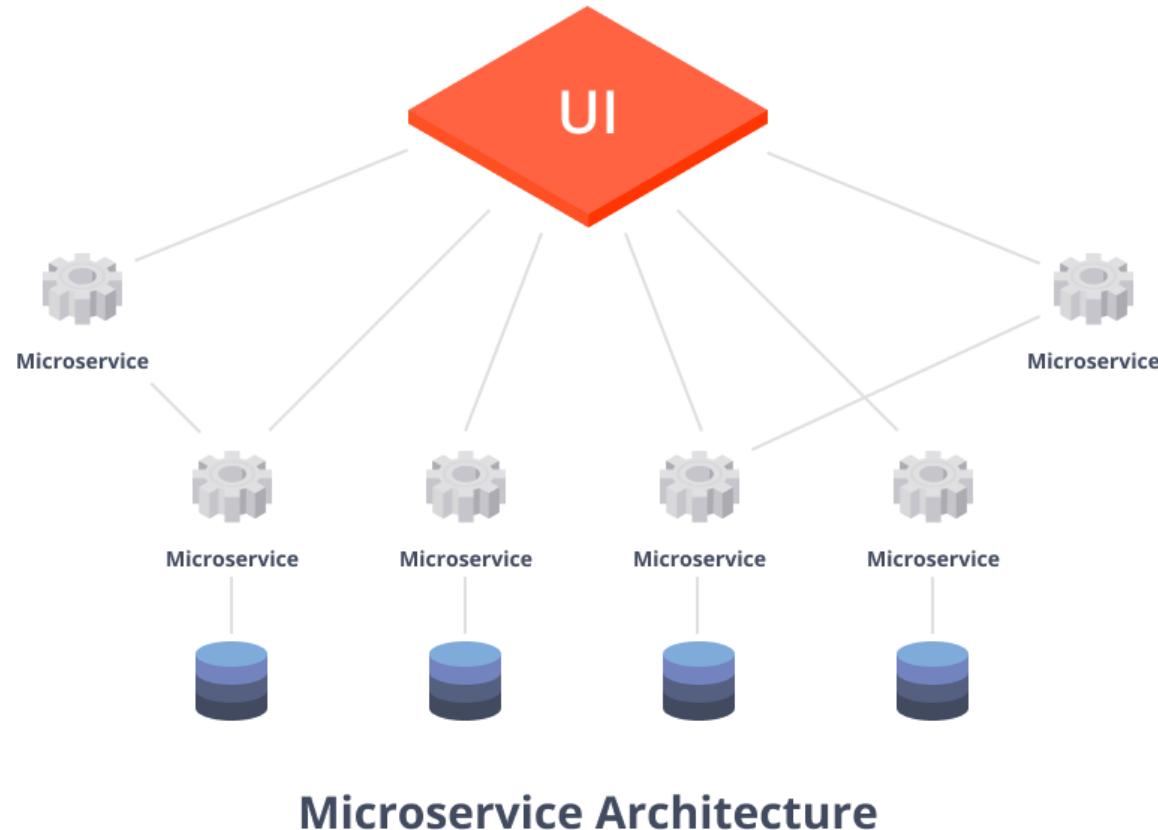
### iv.i ¿Qué es la arquitectura orientada a microservicios?

- 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)



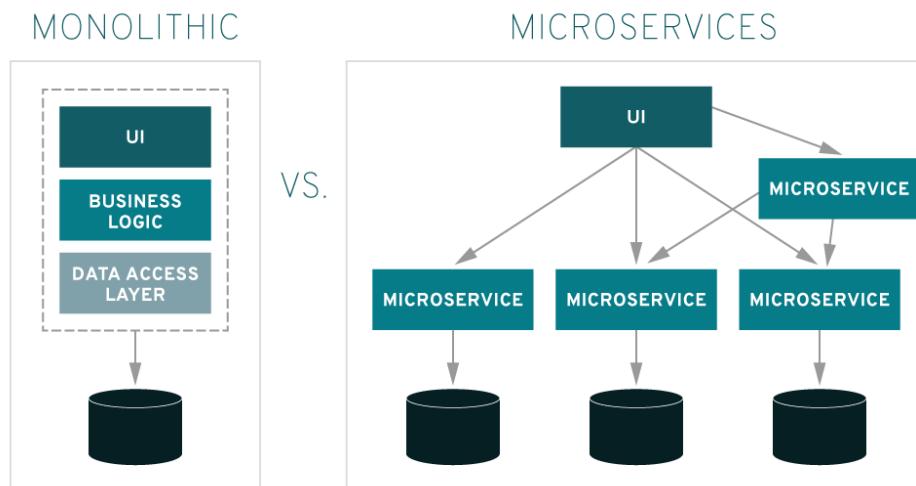


#### **iv.i ¿Qué es la arquitectura orientada a microservicios? (c)**

- 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





+

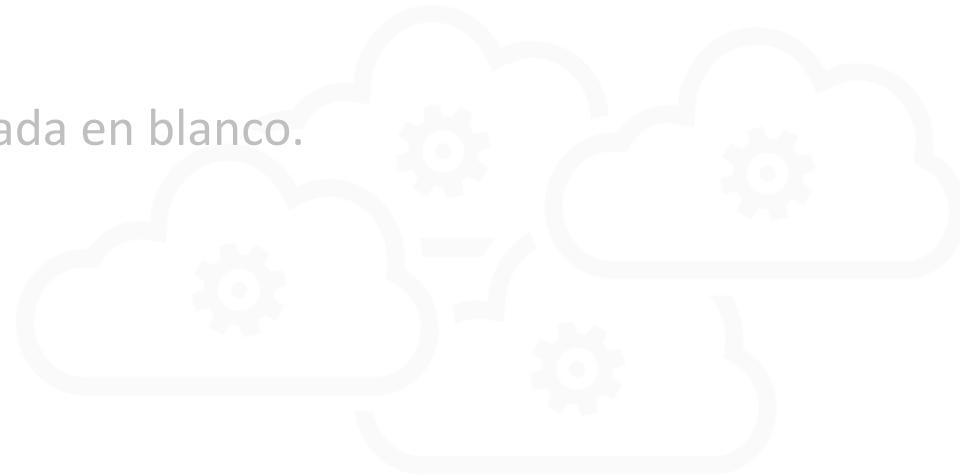
## Resumen de la lección

### 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



+

**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 Otros patrones de diseño para la nube.
- iv.ix 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 definan **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.



+

**NETFLIX**  
**OSS**

## 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?

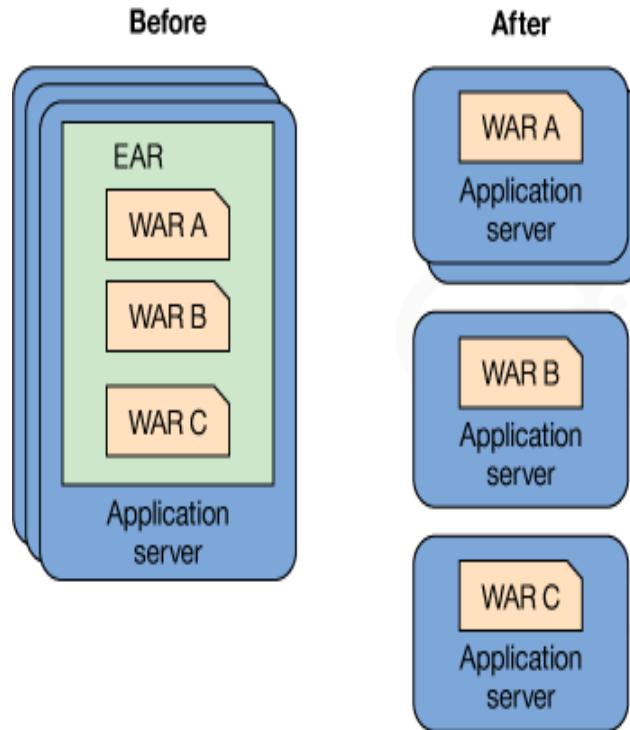


## iv.ii Descomponiendo aplicaciones monolíticas (e)

- ¿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.

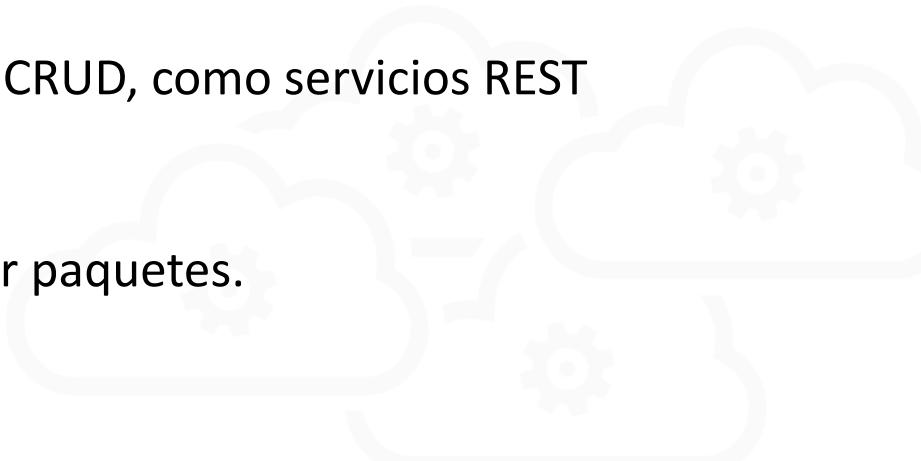


## iv.ii Descomponiendo aplicaciones monolíticas (g)

- 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



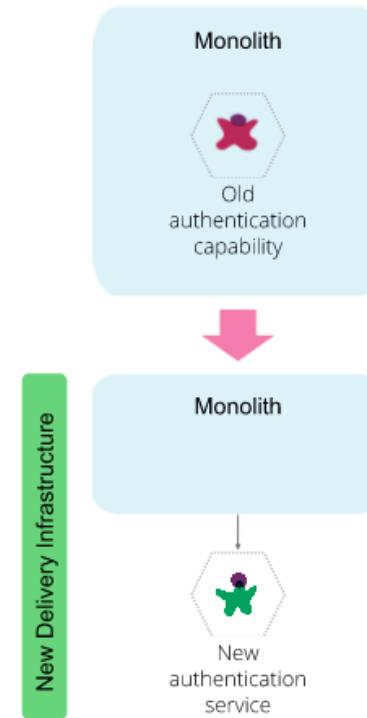
+

## 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.



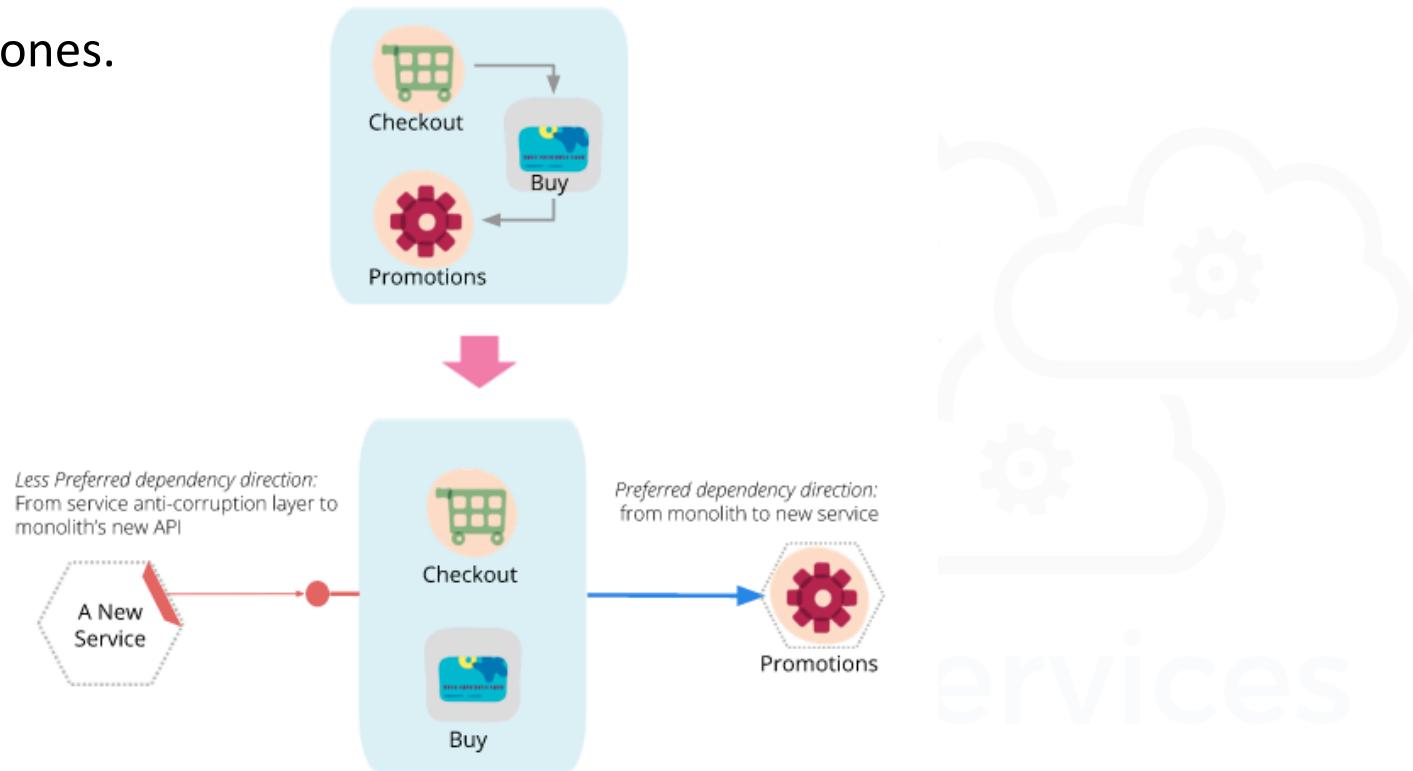


## **iv.ii Descomponiendo aplicaciones monolíticas (k)**

- 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.



## iv.ii Descomponiendo aplicaciones monolíticas (m)

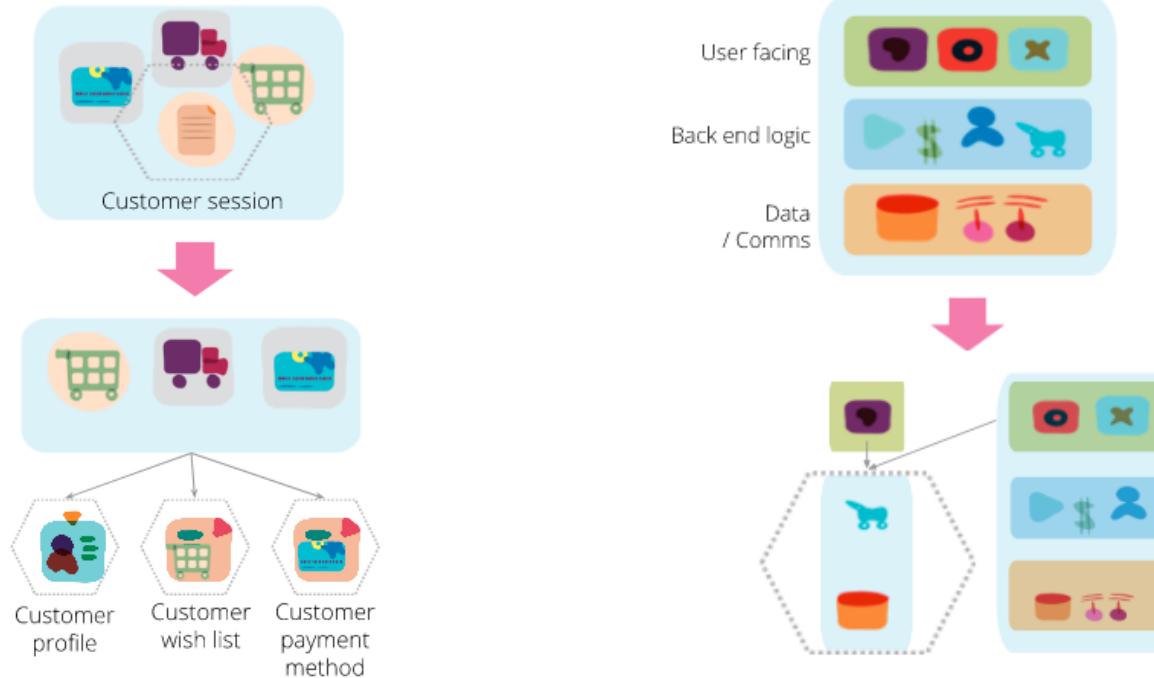
- Recomendaciones.
- Implementar cache distribuido en lugar de sesiones (web).
- Desacoplar grupos de funcionalidades por contexto de negocio (bounded-context).
- Desacoplar funcionalidades sin re-escribir código, es viable refactorizar utilizando interfaces, pero sin re-implementar lógica de negocio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



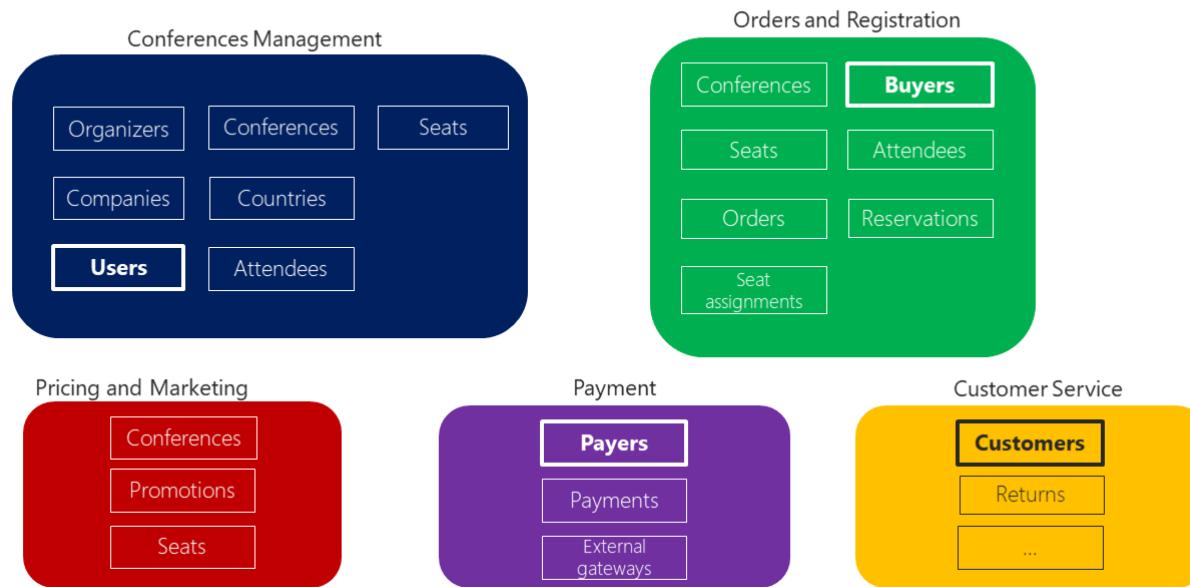
## iv.ii Descomponiendo aplicaciones monolíticas (o)

- 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





+

**NETFLIX**  
**OSS**

## 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.

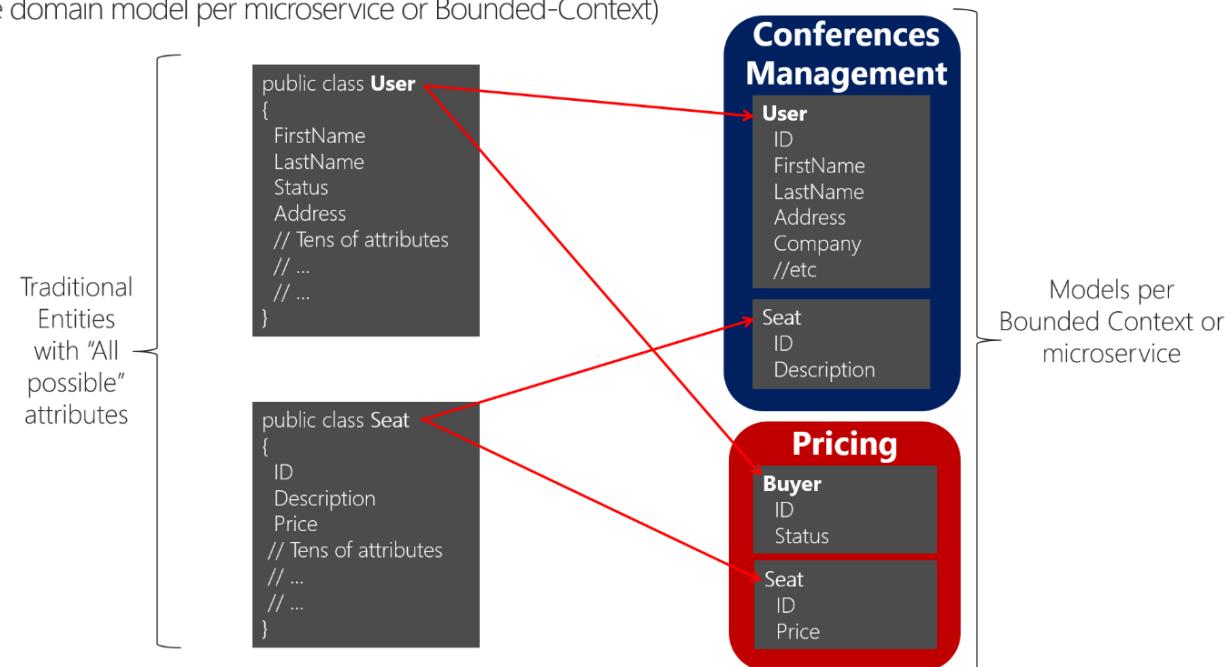
## iv.ii Descomponiendo aplicaciones monolíticas (r)

- 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

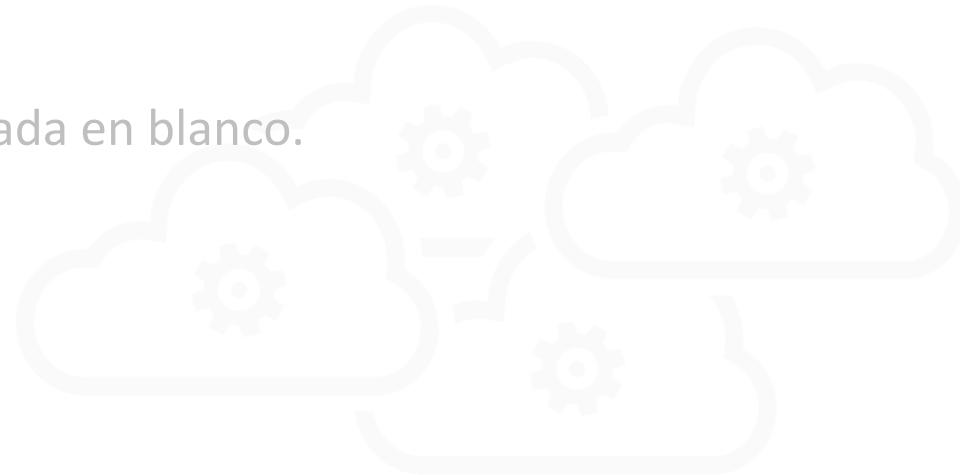
## Resumen de la lección

### 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



+

## **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 Otros patrones de diseño para la nube.
- iv.ix API Manager



+

**NETFLIX**  
OSS

## iv.iii Protocolos ligeros de comunicación para microservicios.



+

**NETFLIX**  
**OSS**

## 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)

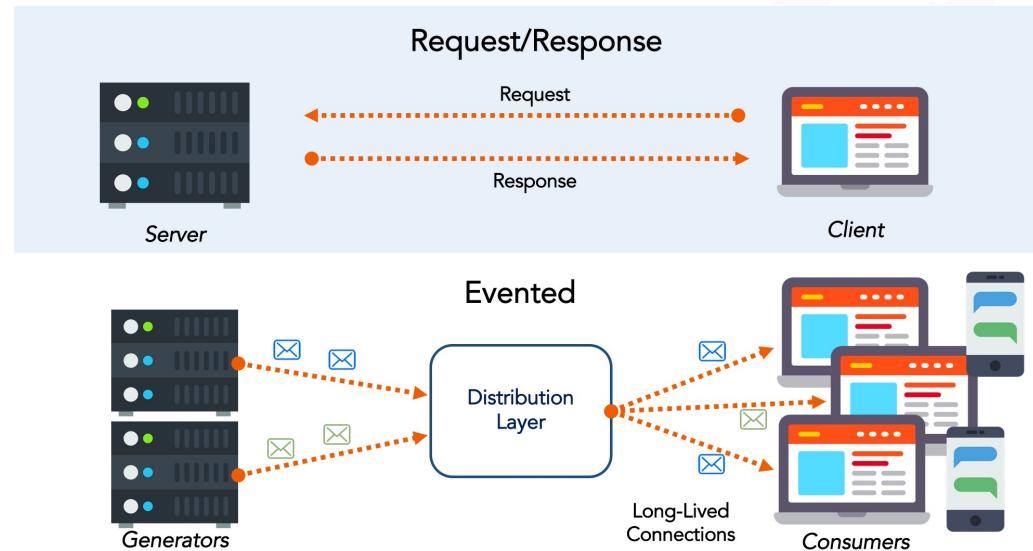
- Dado que cada instancia de un microservicio es un proceso independiente, debe implementar un protocolo de comunicación entre procesos como son HTTP, AMQP, gRPC o TCP/IP, dependiendo de la función de cada microservicio.
- Una sana implementación de microservicios promueve que la intercomunicación entre 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)

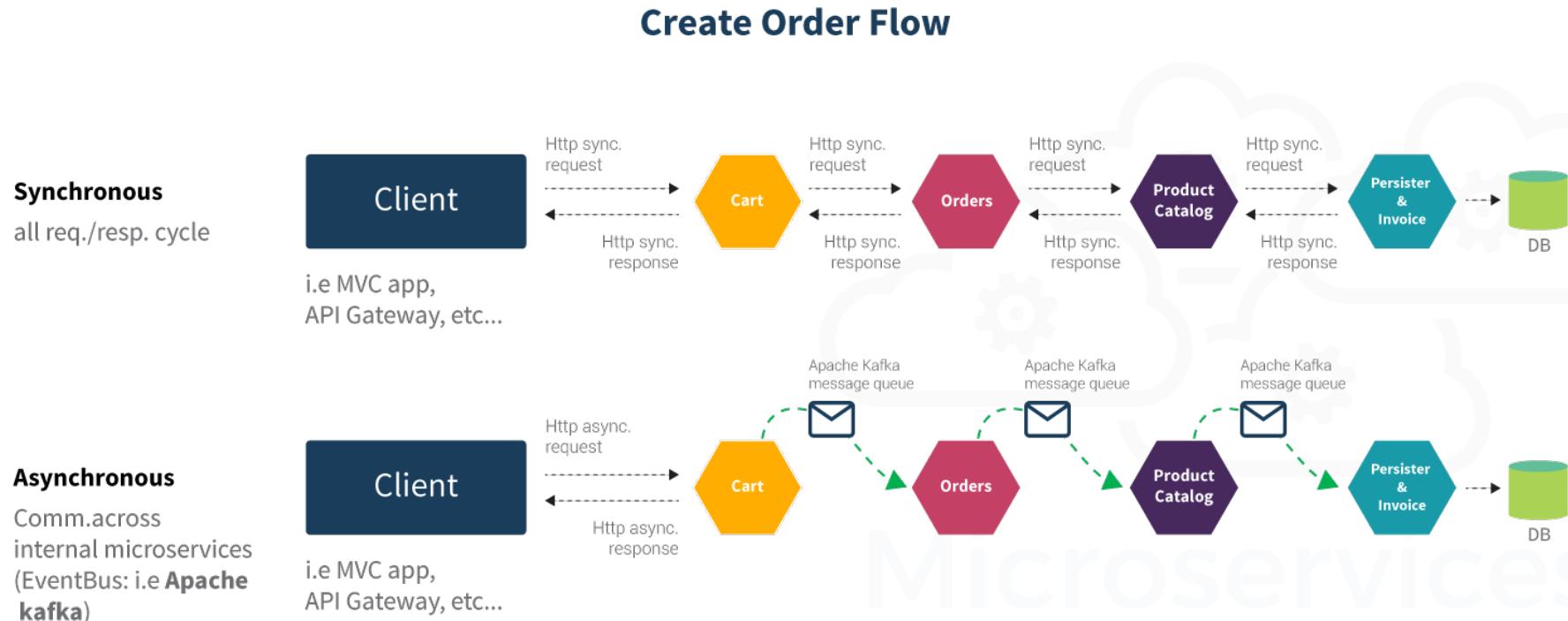
- Habitualmente se sugieren el protocolo HTTP para situaciones "**request-response**" mediante APIs REST y mensajería asíncrona ligera para comunicación de tipo "**fire-and-forget**".



### 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).

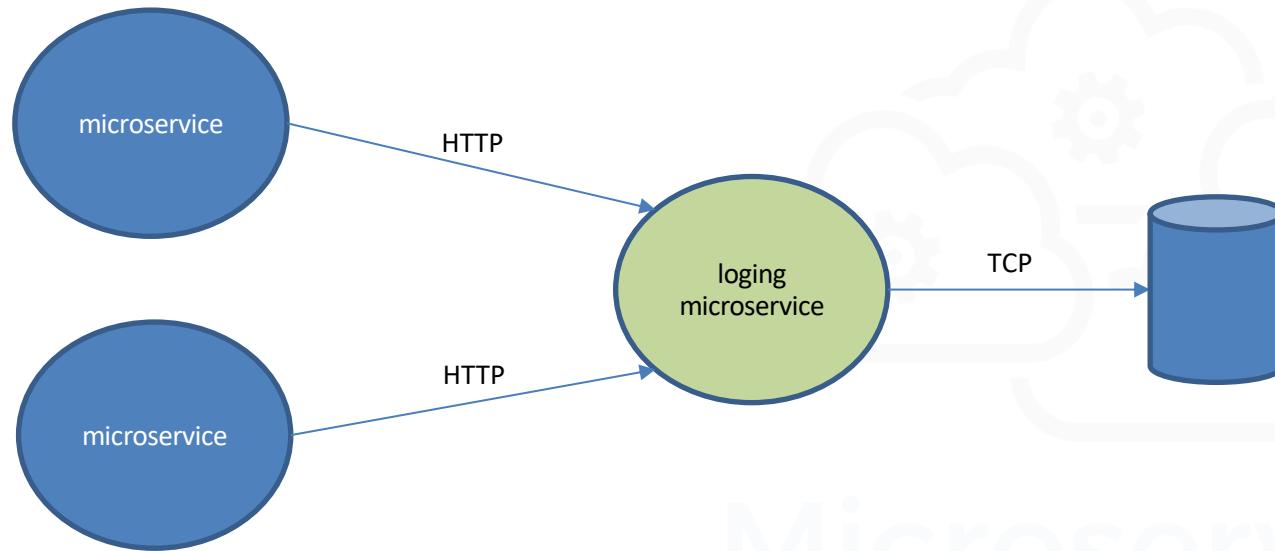


### **iv.iii Protocolos ligeros de comunicación para microservicios (j)**

- 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.

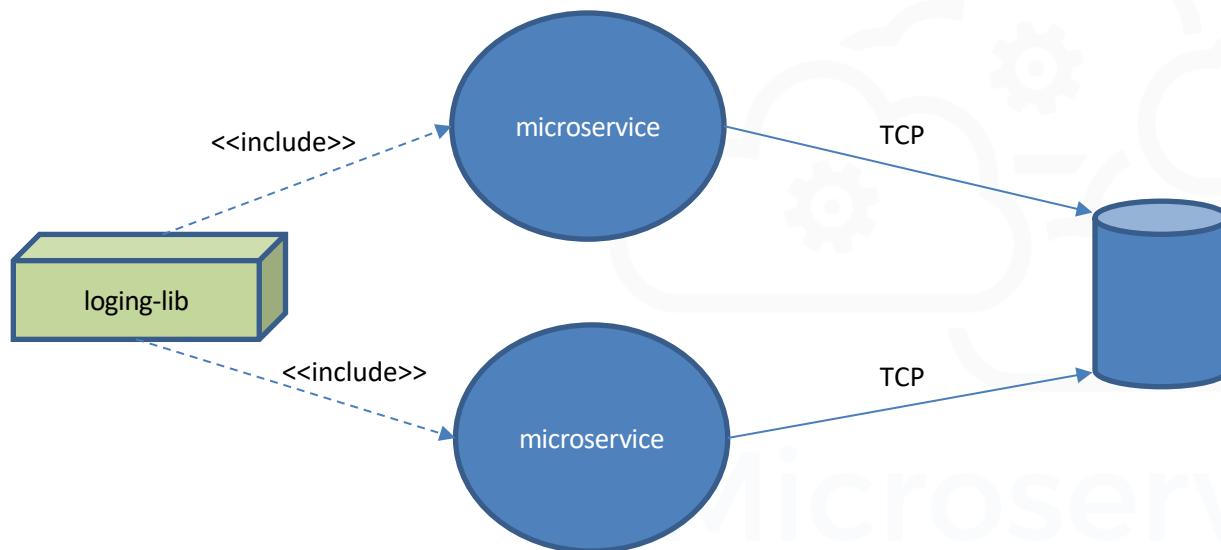
### iv.iii Protocolos ligeros de comunicación para microservicios (k)

- Autonomía de microservicios (b).



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

- Autonomía de microservicios (c).



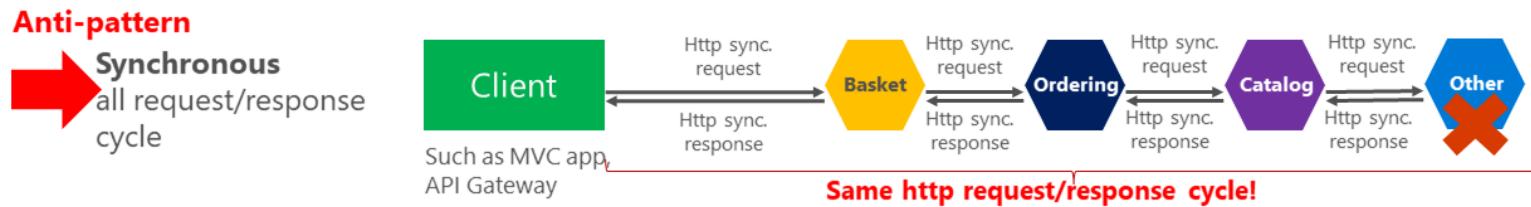
### **iv.iii Protocolos ligeros de comunicación para microservicios (m)**

- 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

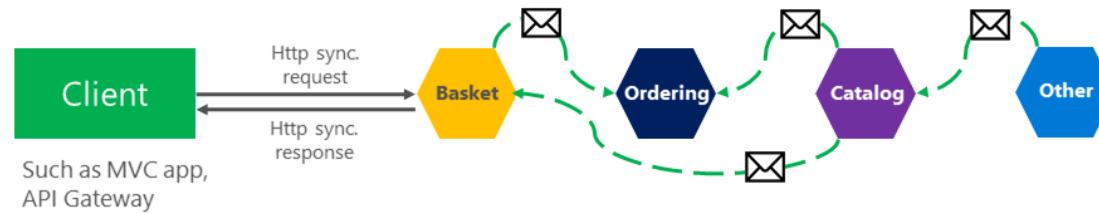


## iv.iii Protocolos ligeros de comunicación para microservicios (ñ)

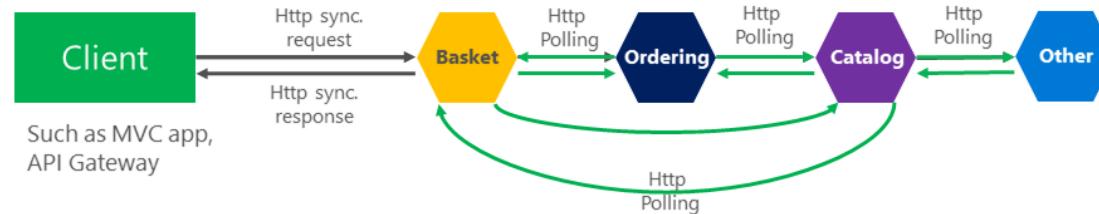
- Autonomía de microservicios (f).

### Comunicación síncrona vs asíncrona entre microservicios

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



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

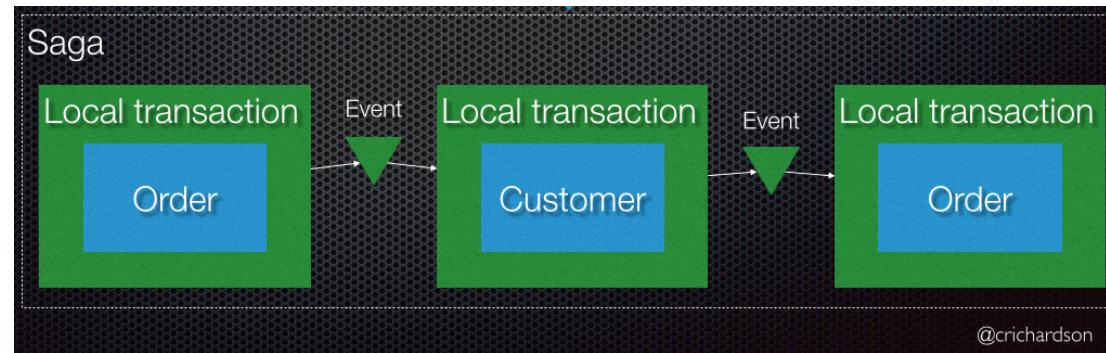




#### iv.iii Protocolos ligeros de comunicación para microservicios (o)

- 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.



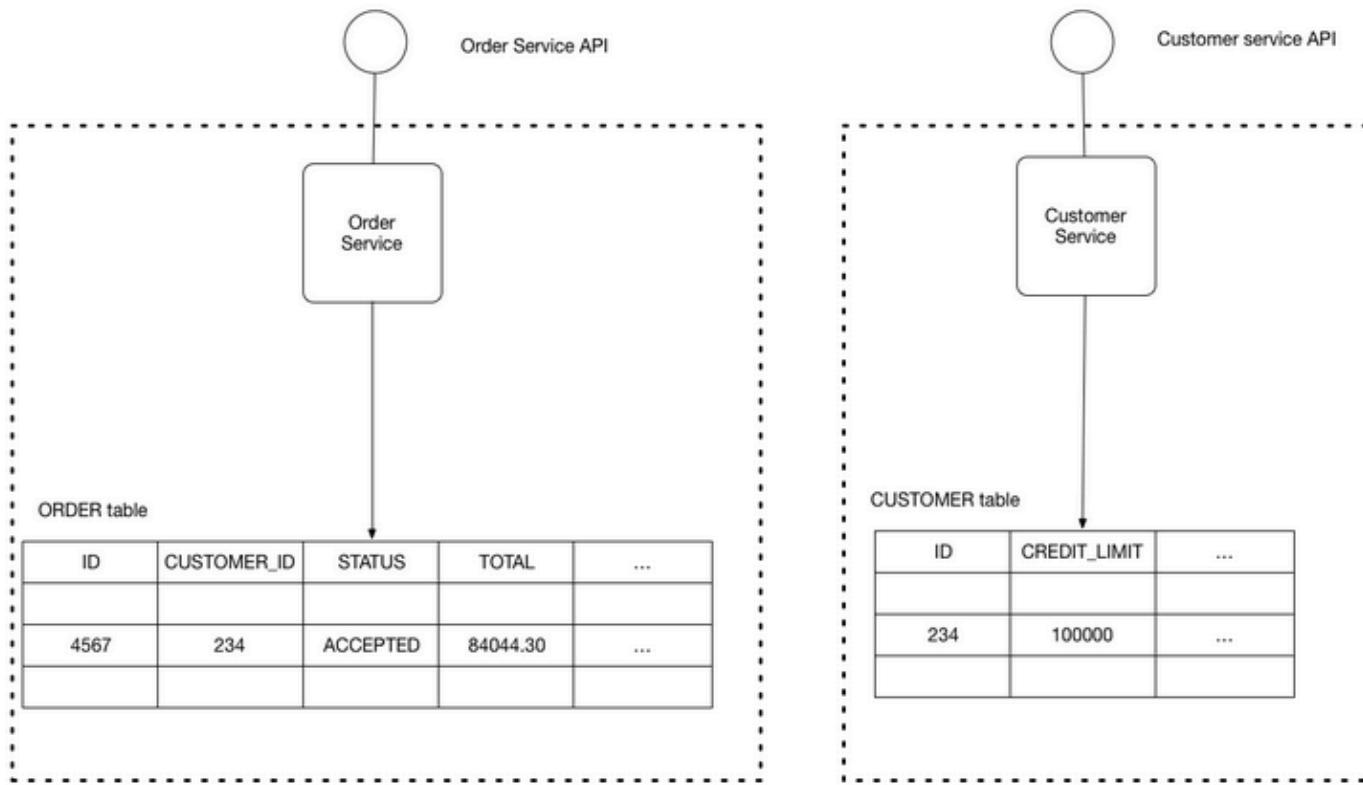
### iv.iii Protocolos ligeros de comunicación para microservicios (r)

- 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-micros
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**.



+

NETFLIX  
OSS

#### 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.



+

### 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**.



#### 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.



+

#### 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

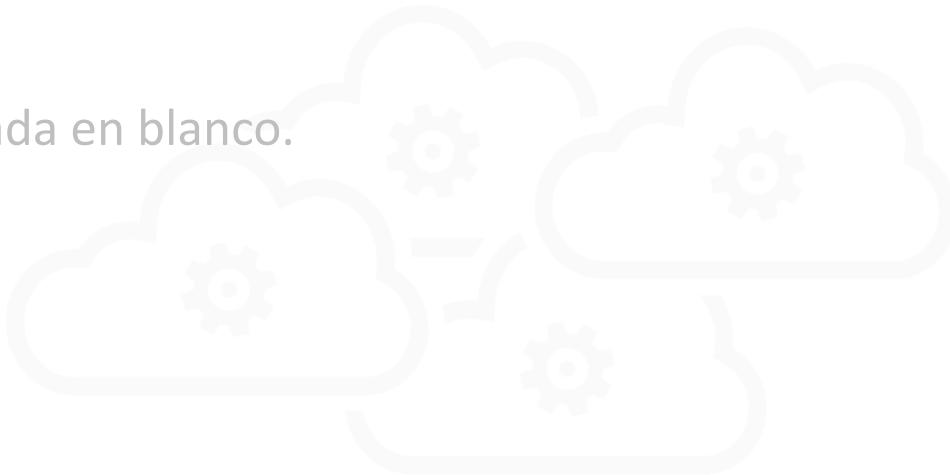
- Comprendemos los distintos mecanismos de comunicación entre microservicios diferenciando entre síncronos y asíncronos.
- Analizamos y discutimos las diferencias entre comunicación síncrona y asíncrona y cuál es la recomendación para los distintos casos de uso.
- Aplicamos comunicación asíncrona mediante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



+

**NETFLIX**  
**OSS**

Esta página fue intencionalmente dejada en blanco.



Microservices



+

**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 Otros patrones de diseño para la nube.
- iv.ix API Manager



+

**NETFLIX**  
**OSS**

#### iv.iv Aplicaciones “cloud-native”.



## Objetivos de la lección

### iv.iv Aplicaciones “cloud-native”.

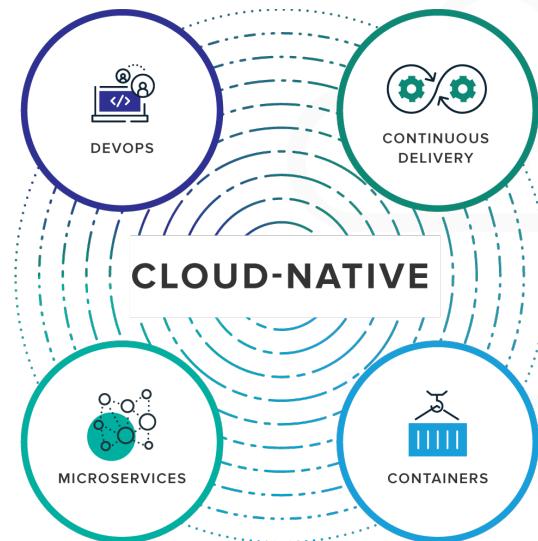
- Aprenderemos cuales son los pilares a considerar de las aplicaciones “cloud-native”.
- Analizaremos la metodología DevOps y sus beneficios.
- Revisaremos lo que son los contenedores y como aplican en una arquitectura de microservicios.
- Aprenderemos que es necesaria la integración continua en ambientes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



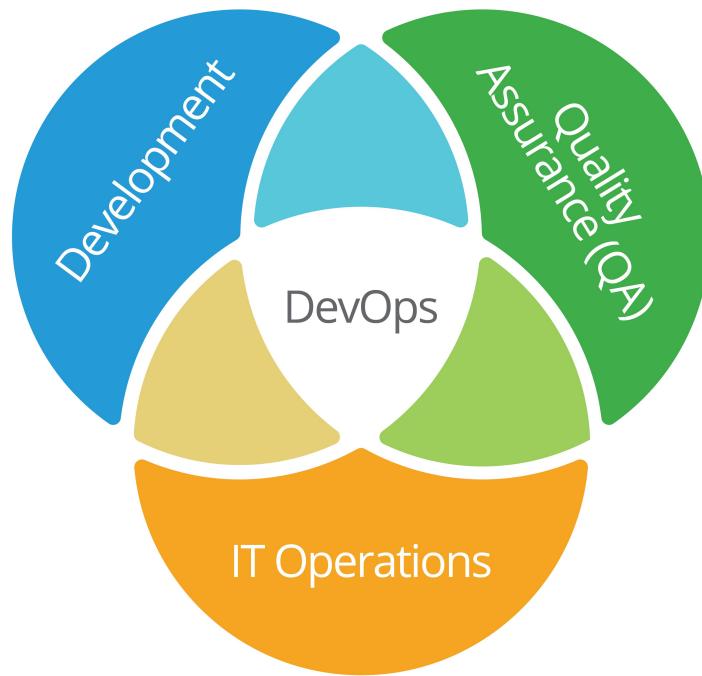


#### **iv.iv Aplicaciones “cloud-native” (c)**

- 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.



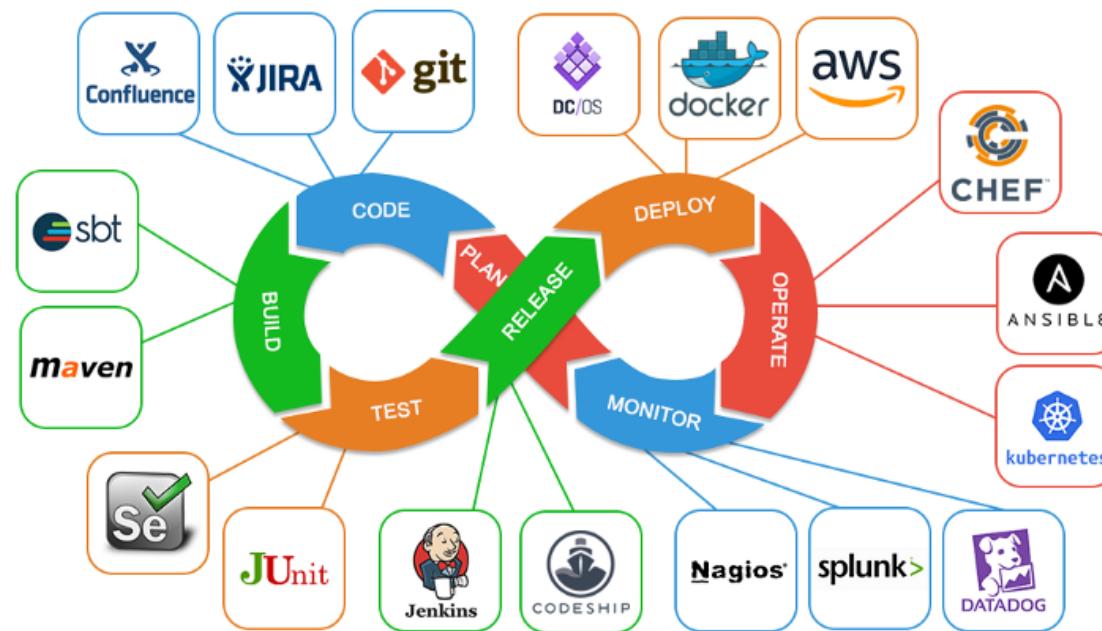


#### iv.iv Aplicaciones “cloud-native” (e)

- 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).

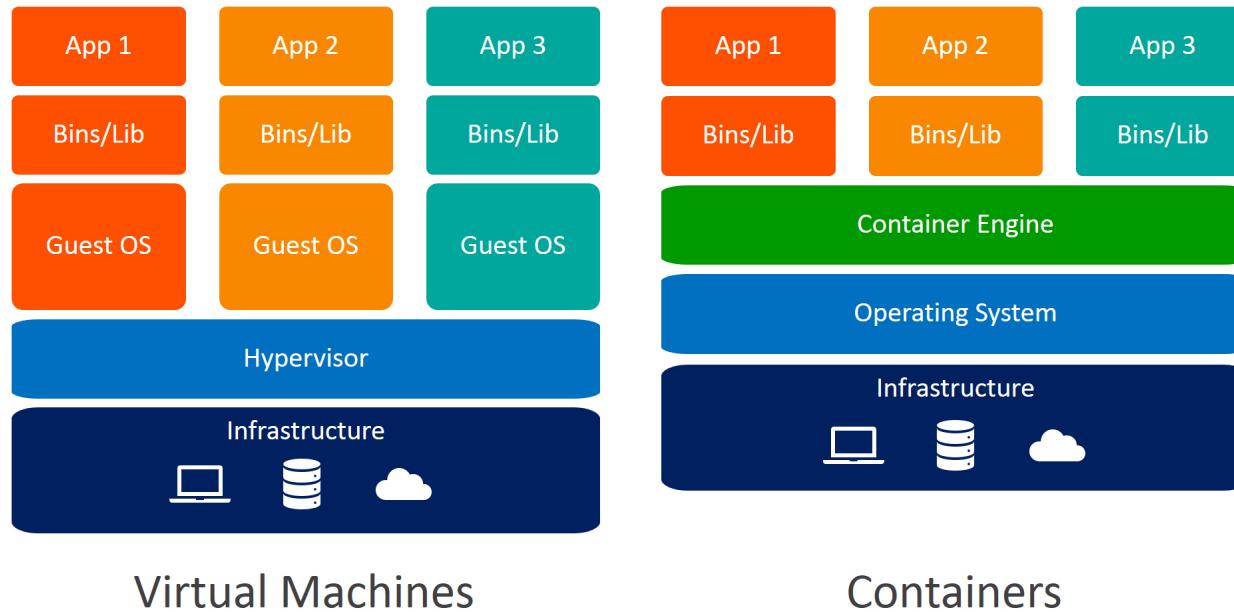


#### **iv.iv Aplicaciones “cloud-native” (g)**

- 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”

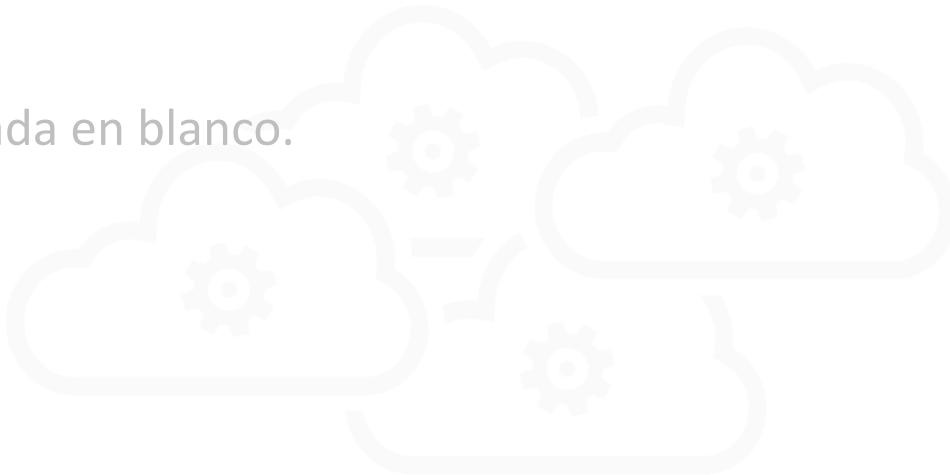
- Aprendimos que los microservicios, una cultura DevOps, contenedores y entrega continua (CI/CD) forman parte de los pilares de las aplicaciones “**cloud-native**”.
- Comprendimos las principales características que deben considerarse en la implementación de aplicaciones “**cloud-native**”.



+

**NETFLIX**  
**OSS**

Esta página fue intencionalmente dejada en blanco.



Microservices



+

**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 Otros patrones de diseño para la nube.
- iv.ix 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.



+

NETFLIX  
OSS

## 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)

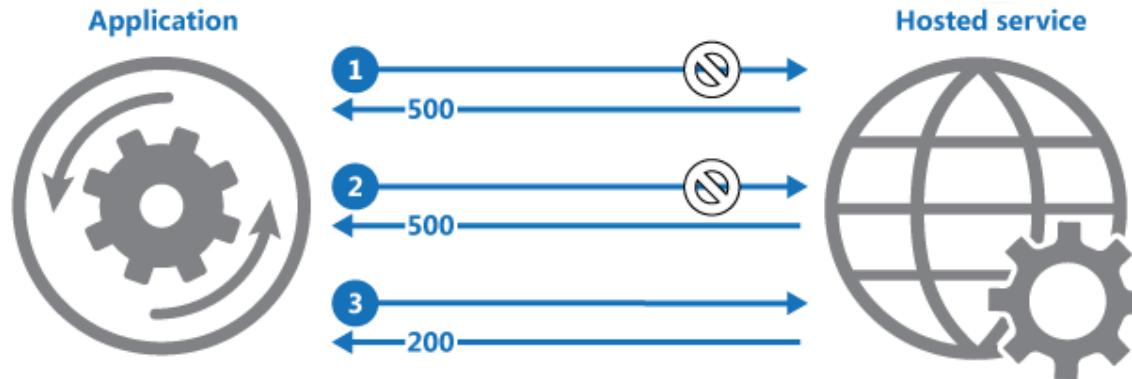
- Retry Pattern.
- **Ventajas:**
  - Resistencia.
- **Desventajas:**
  - Complejidad de implementación desacoplada.
  - Mantenimiento de operaciones.



Microservices

## 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}**”.



+

NETFLIX  
OSS

## 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**.



+

NETFLIX  
OSS

## 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.



## 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.



+

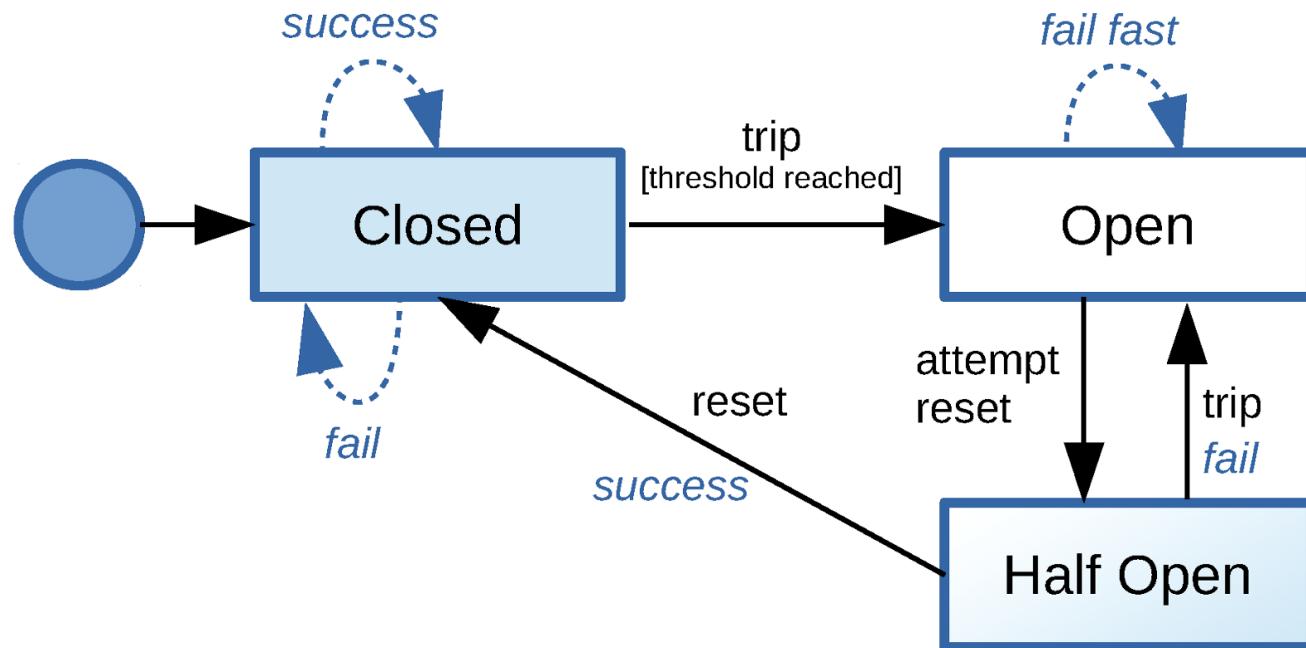
## iv.v Principios de diseño para aplicaciones en la nube. (a')

- 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"}**”.



## 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.



+

NETFLIX  
OSS

## 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”.



## 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.



+

**NETFLIX**  
**OSS**

## 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.



+

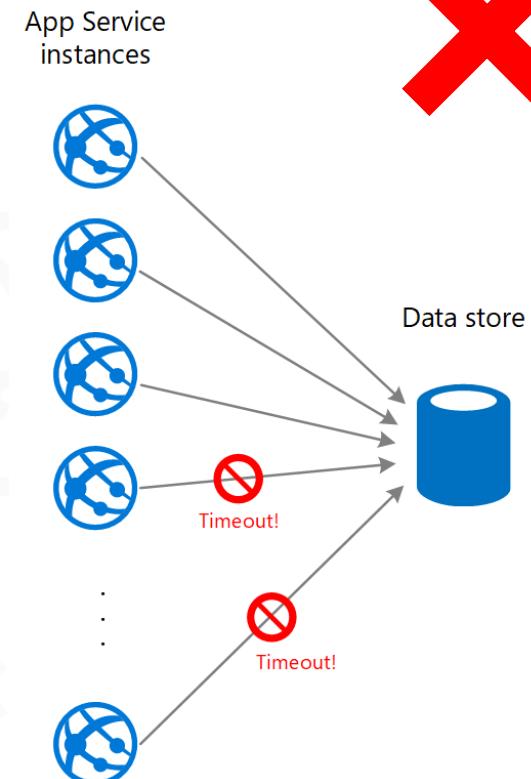
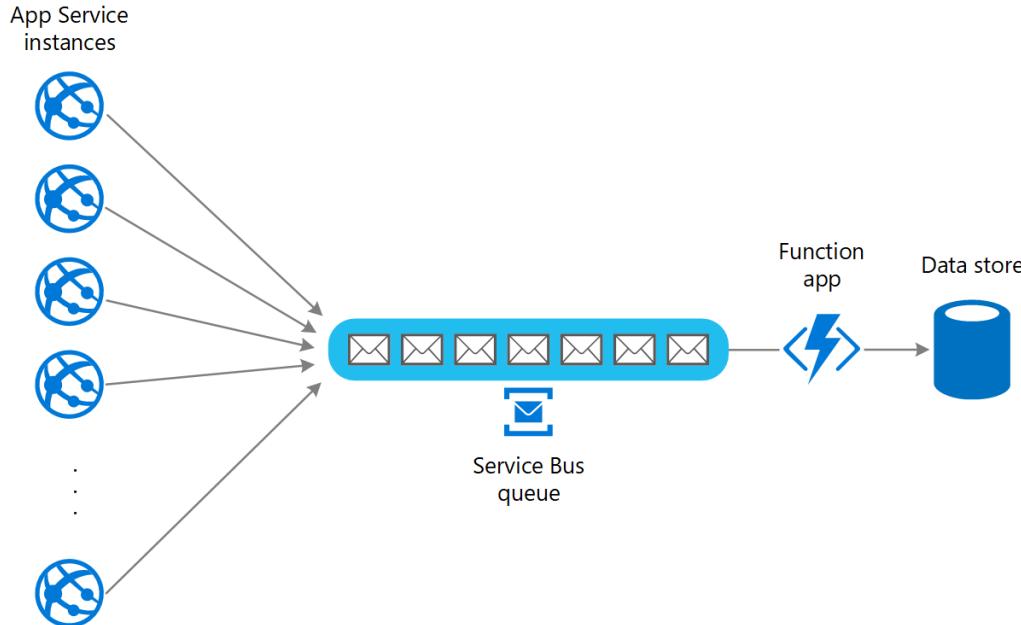
NETFLIX  
OSS

## 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”).

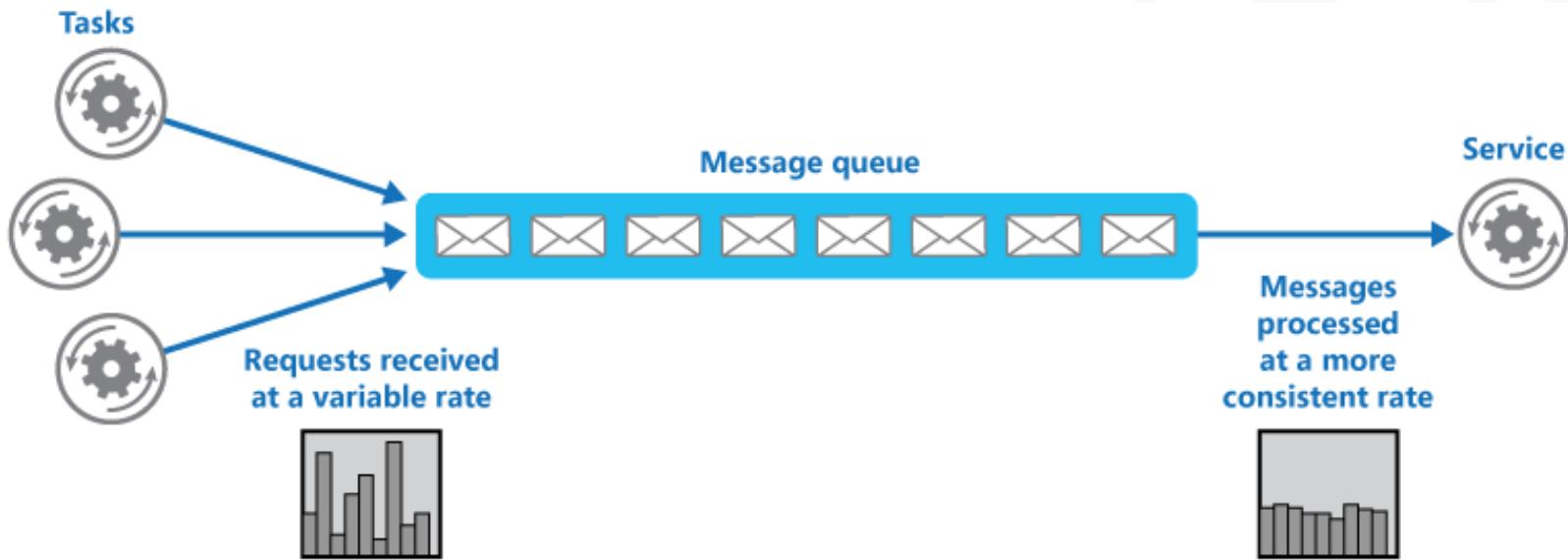
## iv.v Principios de diseño para aplicaciones en la nube. (x')

- Queue-Based Load Leveling Pattern.



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

- Queue-Based Load Leveling Pattern.





+

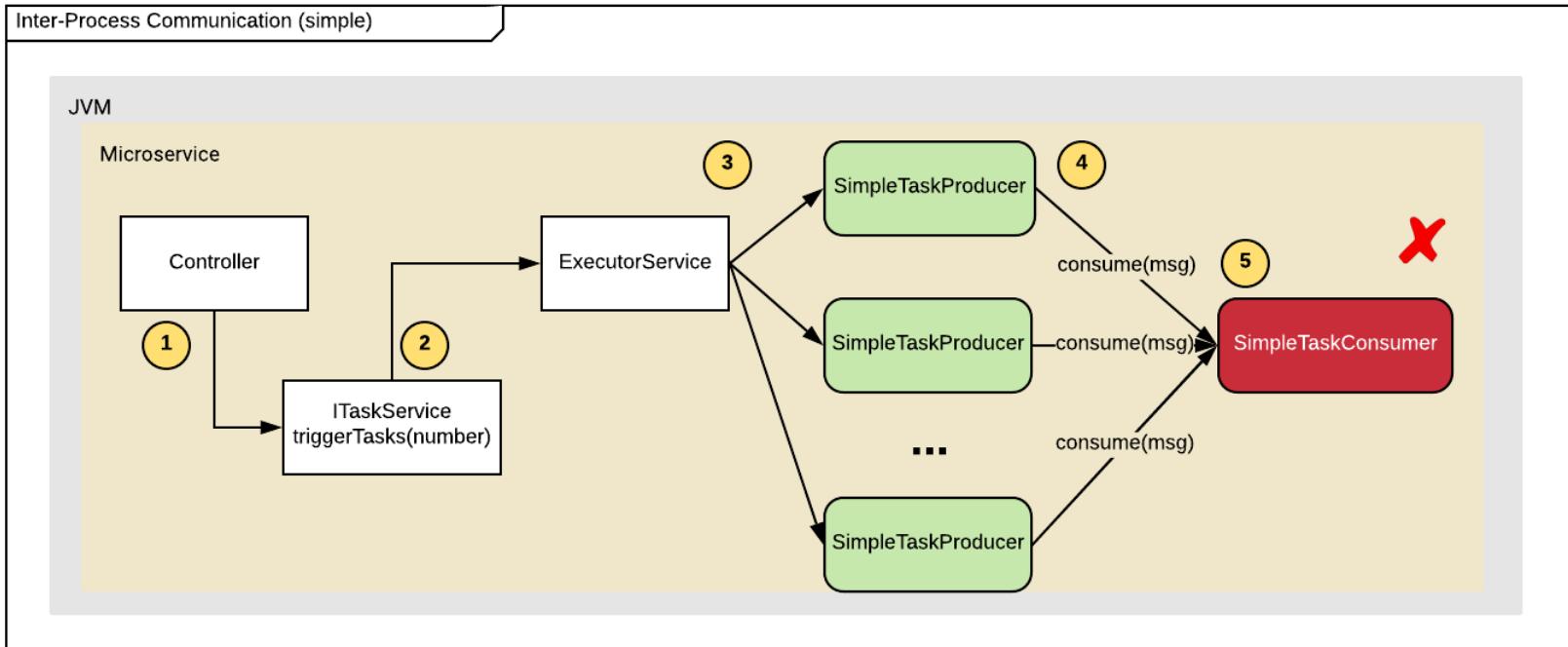
## iv.v Principios de diseño para aplicaciones en la nube. (z')

- 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.



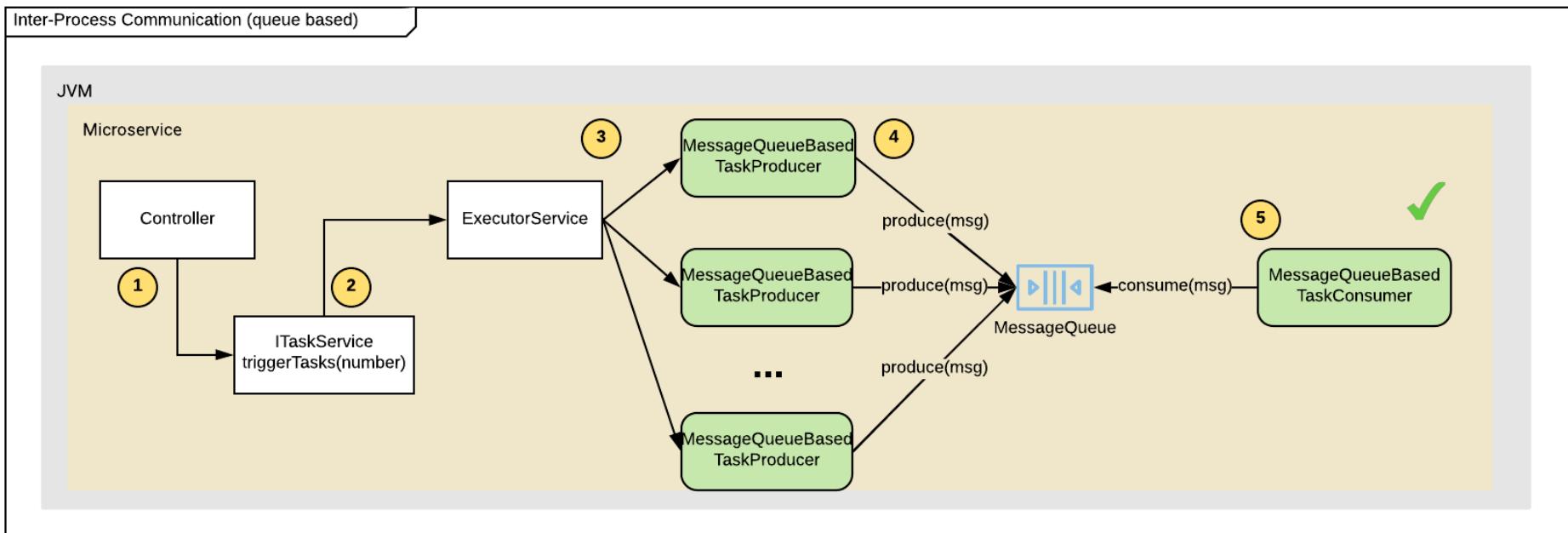
## iv.v Principios de diseño para aplicaciones en la nube. (a'')

- 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.



+

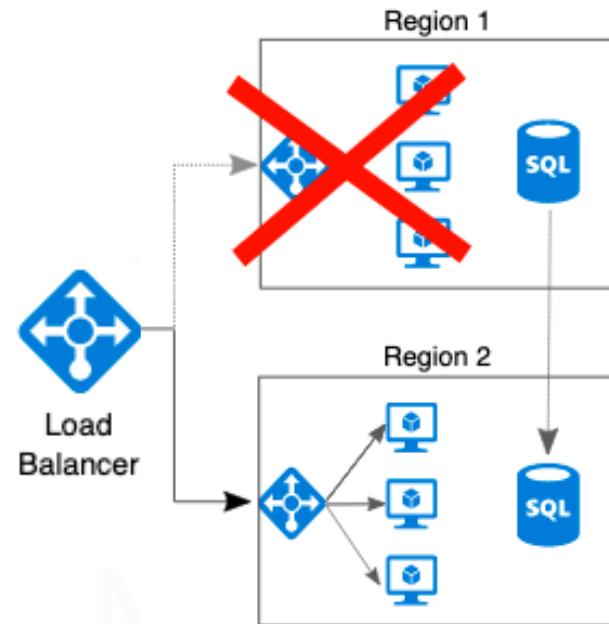
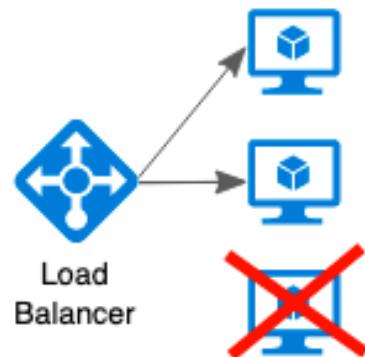
NETFLIX  
OSS

## 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.

## 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.



+

NETFLIX  
OSS

## 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.

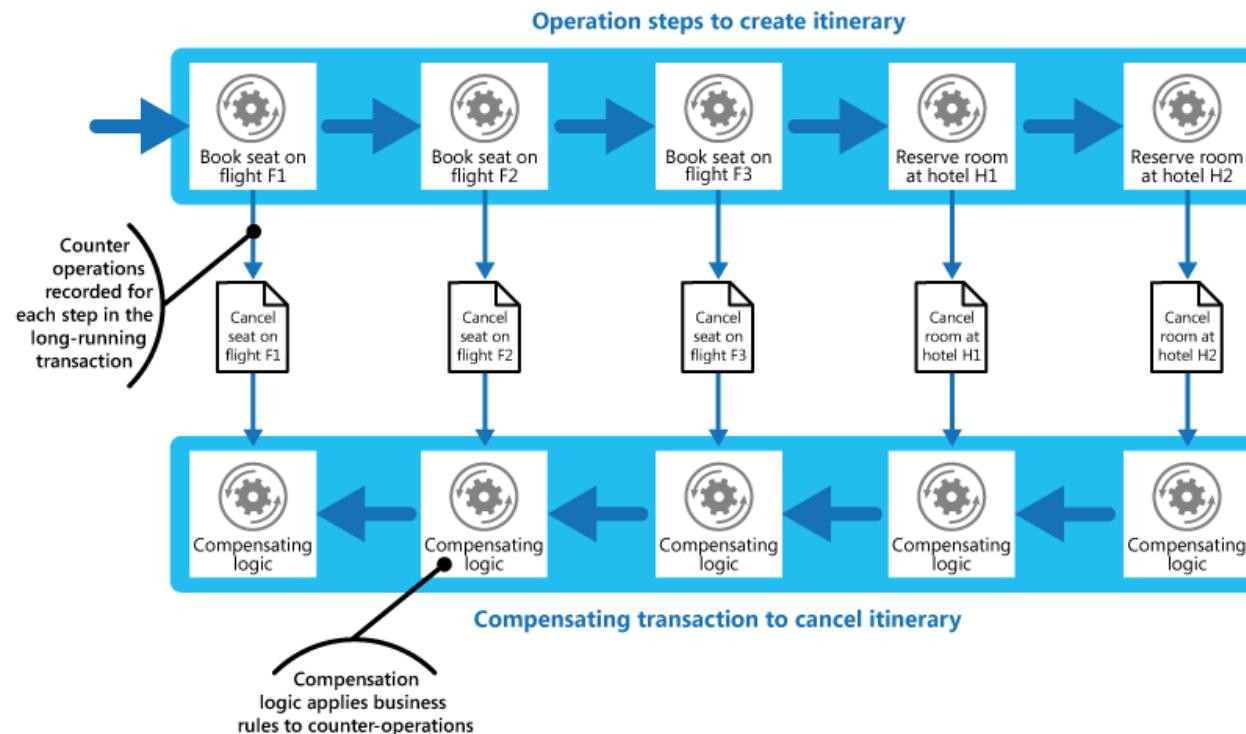


## iv.v Principios de diseño para aplicaciones en la nube. (I’')

- 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.





+

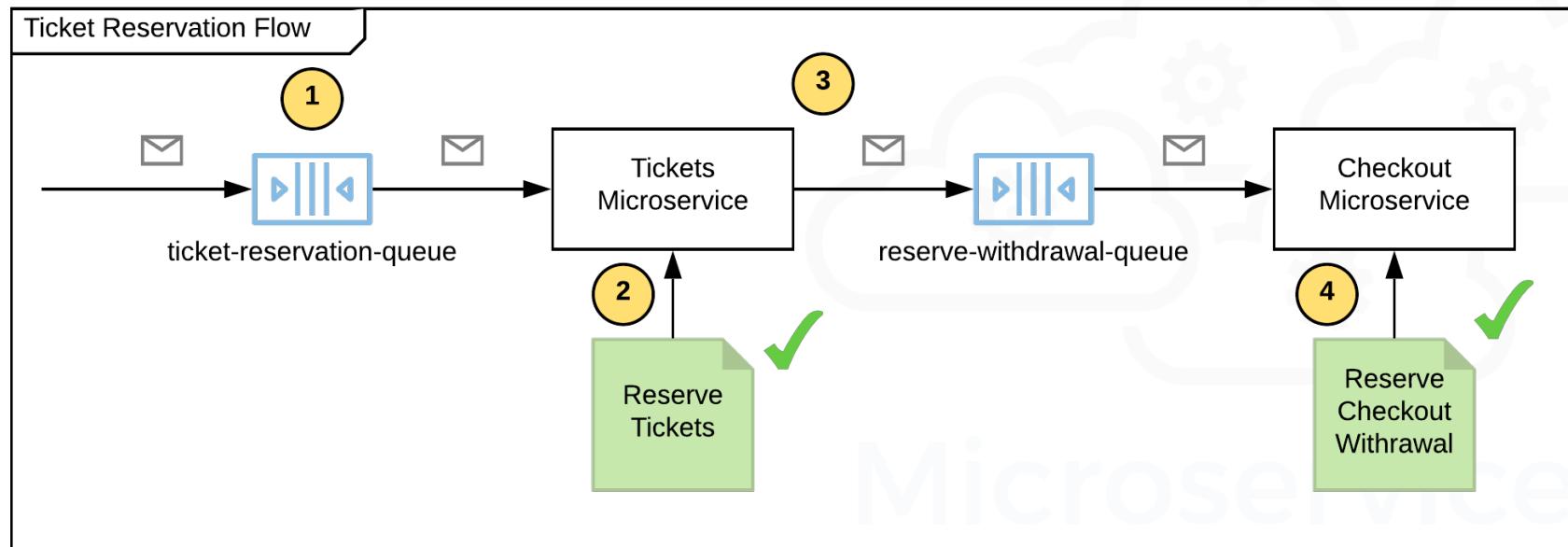
NETFLIX  
OSS

## 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”)

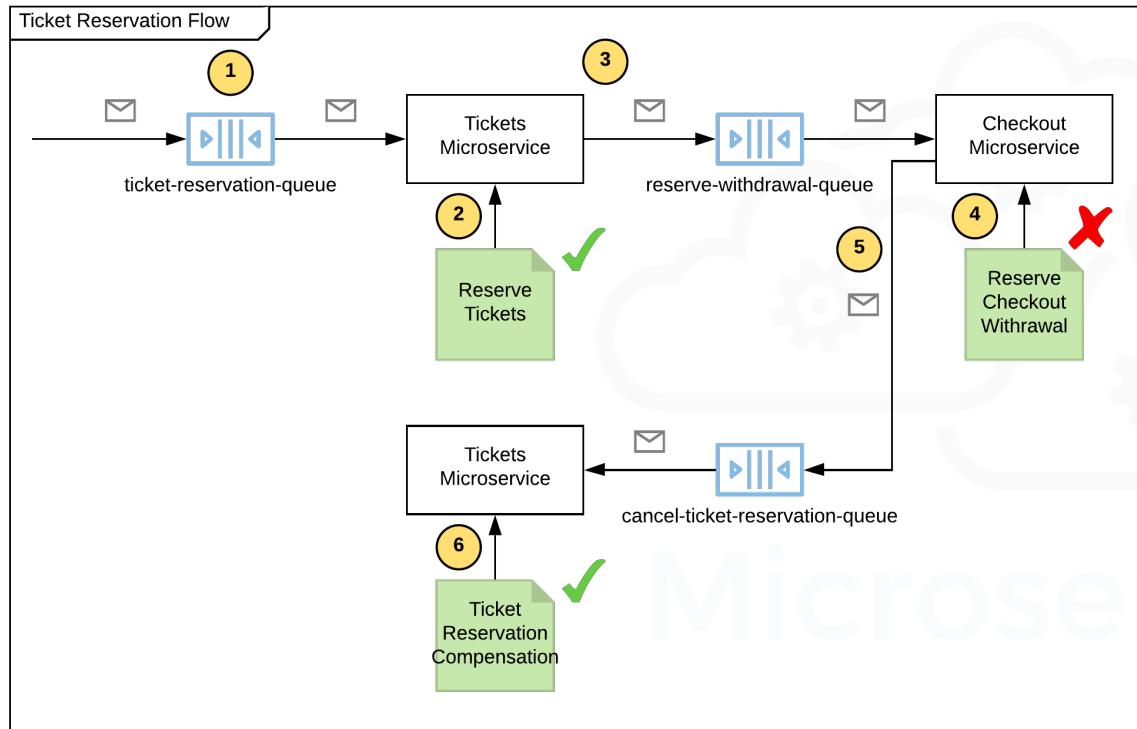
- Práctica 13. Compensation Transaction Pattern

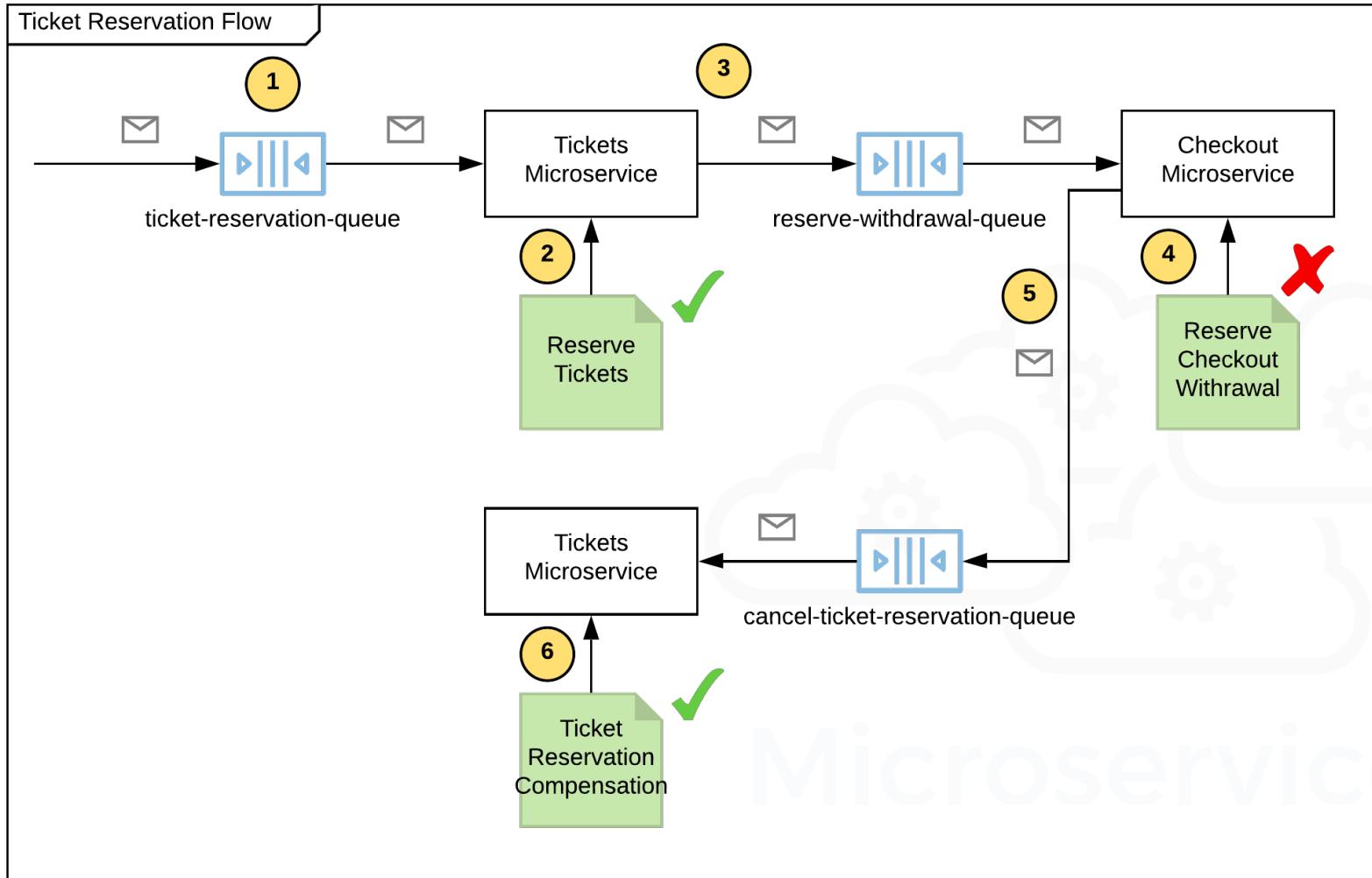




## iv.v Principios de diseño para aplicaciones en la nube. (o")

### - 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.

## 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.



+

NETFLIX  
OSS

## 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. (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.

