

INICIO

Desarrollo de Microservicios con Spring Cloud Netflix OSS

ISC. Ivan Venor García Baños





+

NETFLIX
OSS

Agenda

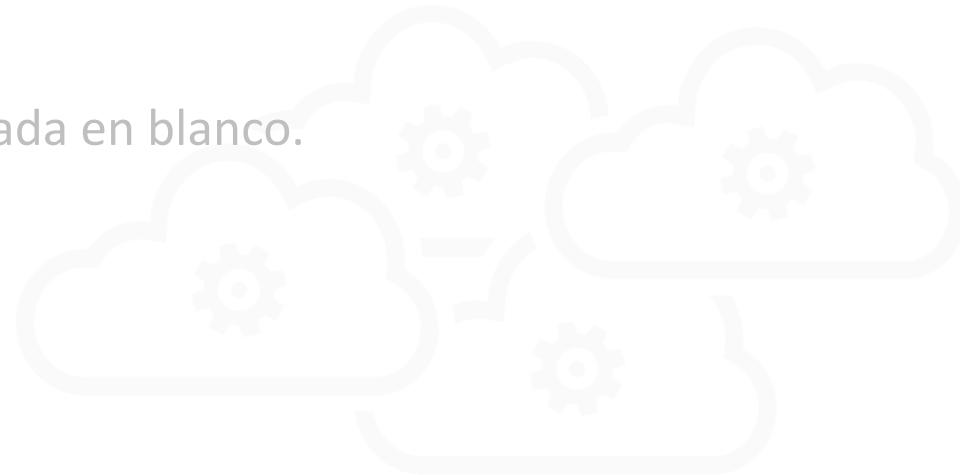
1. Presentación
2. Objetivos
3. Contenido
4. Despedida



Microservices



Esta página fue intencionalmente dejada en blanco.



Microservices



+

NETFLIX
OSS

3. Contenido

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



+

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

v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS

Microservices



v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS

- v.i **Twelve-Factor Apps.**
- v.ii Spring Cloud y Spring Cloud Netflix OSS
- v.iii Configuración externalizada con Spring Cloud Config y Spring Cloud Bus.
- v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka.
- v.v Balanceo de carga del lado del cliente con Ribbon.
- v.vi Clientes REST declarativos con Feign.
- v.vii Implementación de corto-circuito con Hystrix.
- v.viii Visualización de corto-circuitos con Turbine.
- v.ix API Gateway con Spring Cloud Zuul.



+

NETFLIX
OSS

v.i Twelve-Factor Apps.



Microservices



+

NETFLIX
OSS

Objetivos de la lección

v.i Twelve-Factor Apps.

- Conocer la metodología "twelve-factor apps" para implementar arquitecturas de microservicios exitosas.
- Comprender cada uno de los doce factores de la metodología "twelve-factor apps".

v.i Twelve-Factor Apps. (a)

- Para diseñar arquitecturas orientadas a microservicios satisfactorias, no sólo es requerido implementar patrones de diseño para la nube que permitan diseñar aplicaciones resistentes, tolerantes a fallos, distribuidas, confiables, etc.
- Tampoco es suficiente diseñar microservicios en contextos limitados, ni ser políglotas o diseñar centrado en APIs ligeras, entre otros atributos de aplicaciones “cloud-native”.
- También es necesario conocer y diseñar aplicaciones basandonos en la metodología de aplicaciones de 12 factores o “**“twelve-factor apps”**”.



v.i Twelve-Factor Apps. (b)

- ¿Qué son los “twelve-factor apps”?
- Es una metodología para construir aplicaciones basadas en la nube.
- Es un conjunto de principios para trasladar satisfactoriamente aplicaciones a la nube sin importar el ambiente en el que se despliegan.
- Inicialmente fueron propuestos para construir aplicaciones SaaS sobre la plataforma-como-servicio (PaaS) de Heroku.



+

v.i Twelve-Factor Apps. (c)

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (d)

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.





+

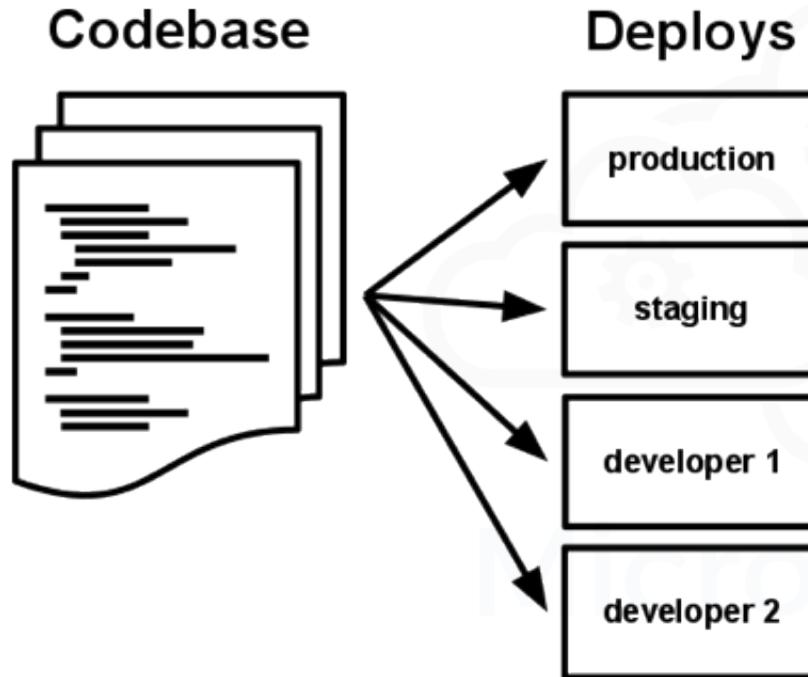
NETFLIX
OSS

v.i Twelve-Factor Apps. (e)

- 1. Código Case:
- El código base de una aplicación debe ser restreado por alguna herramienta de control de versiones, como git o svn, y permitir múltiples despliegues en múltiples ambientes.
- Si hay múltiples códigos base en los repositorios de una organización, no existe sólo una única aplicación, sino un sistema distribuido.

v.i Twelve-Factor Apps. (f)

- 1. Código Case:





+

v.i Twelve-Factor Apps. (g)

- Principios twelve-factor apps:
 - Código Base.
 - **Dependencias.**
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.



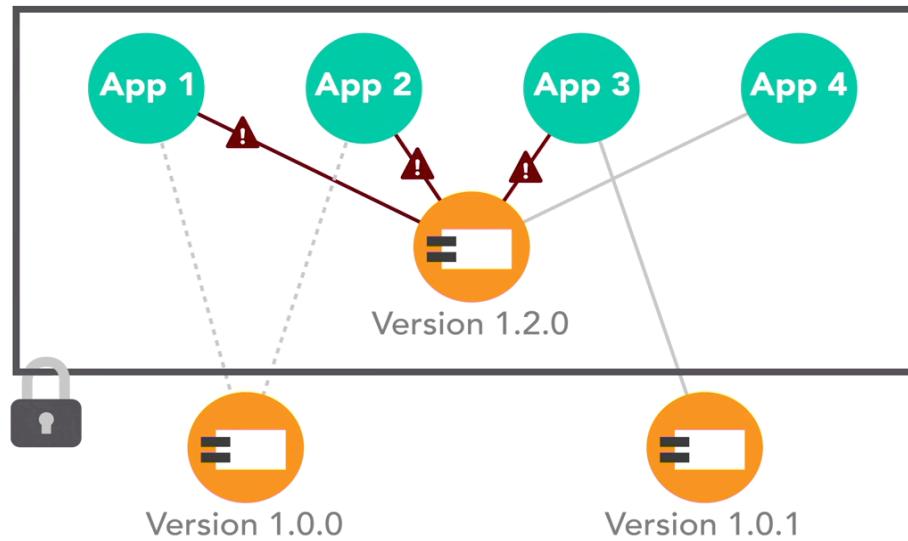


v.i Twelve-Factor Apps. (h)

- 2. Dependencias:
- Las dependencias utilizadas por la aplicación deben ser explicitamente declaradas y aisladas, es decir, accesibles y utilizadas únicamente por la aplicación que las declara.
- No reutilizar librerías o dependencias desplegadas en un ambiente de ejecución.
- Utilizar herramientas como Maven o Gradle para el manejo de dependencias.

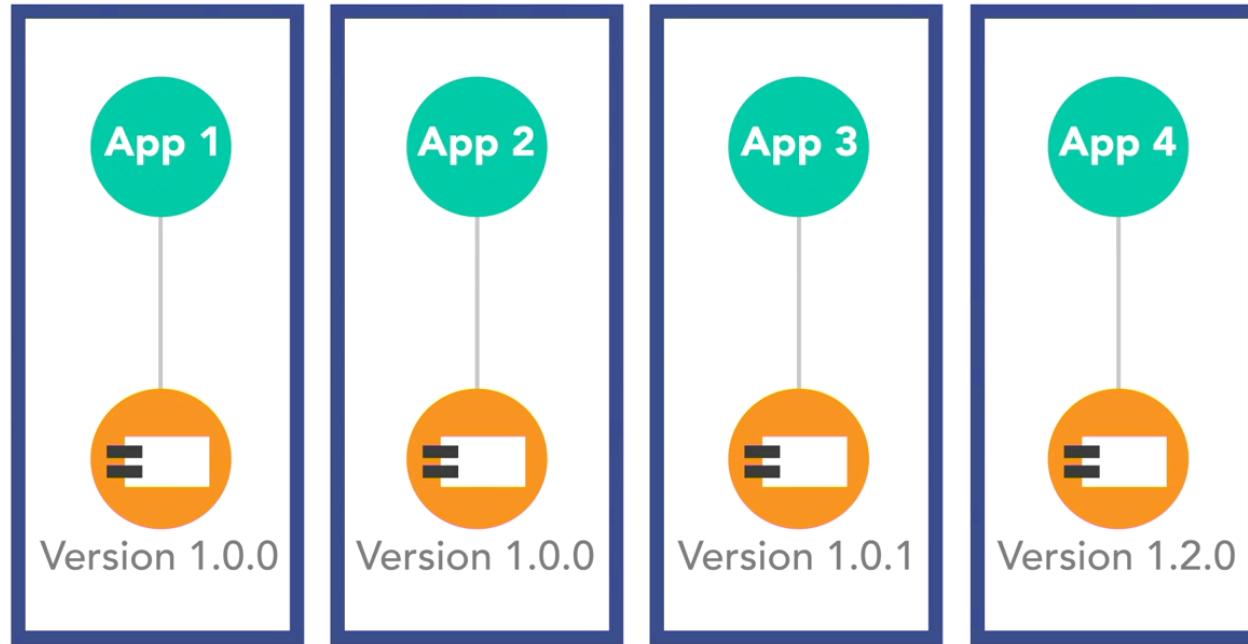
v.i Twelve-Factor Apps. (i)

- 2. Dependencias:
- Prevenir el “vendor-locking” al desplegar aplicaciones en la nube.



v.i Twelve-Factor Apps. (j)

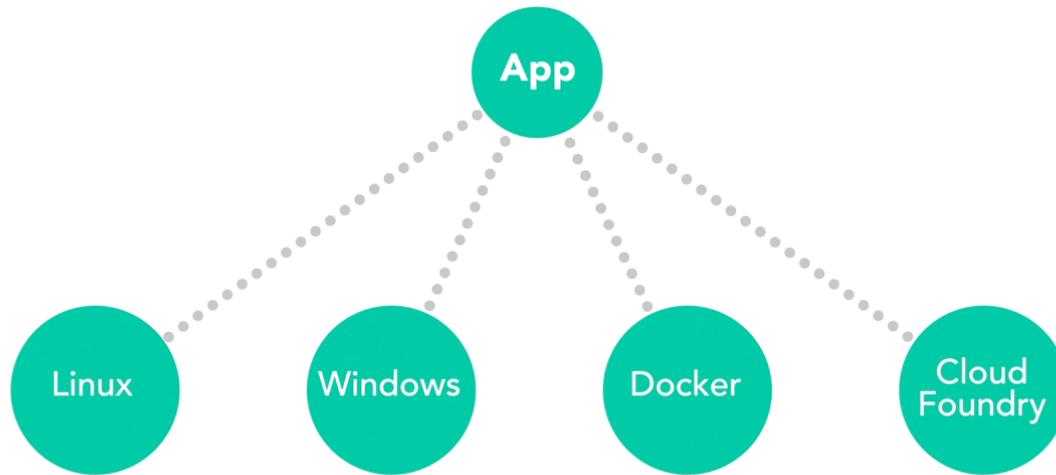
- 2. Dependencias:





v.i Twelve-Factor Apps. (k)

- 2. Dependencias:
- Una de las mayores ventajas de declarar explicitamente y aislar las dependencias de la aplicación es portabilidad.





+

v.i Twelve-Factor Apps. (I)

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - **Configuraciones.**
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.





v.i Twelve-Factor Apps. (m)

- 3. Configuraciones:
- Las configuraciones son lo único que deben variar en cada despliegue entre ambientes; son todos aquellos valores que cambian entre ambientes.
- Todas las configuraciones deben almacenarse o ser dependientes del ambiente de despliegue y ser recuperadas mediante:
 - Servicios externos
 - Variables de entorno/ambiente
 - Archivos de configuración
 - Servicios customizables (externos a la aplicación).



+

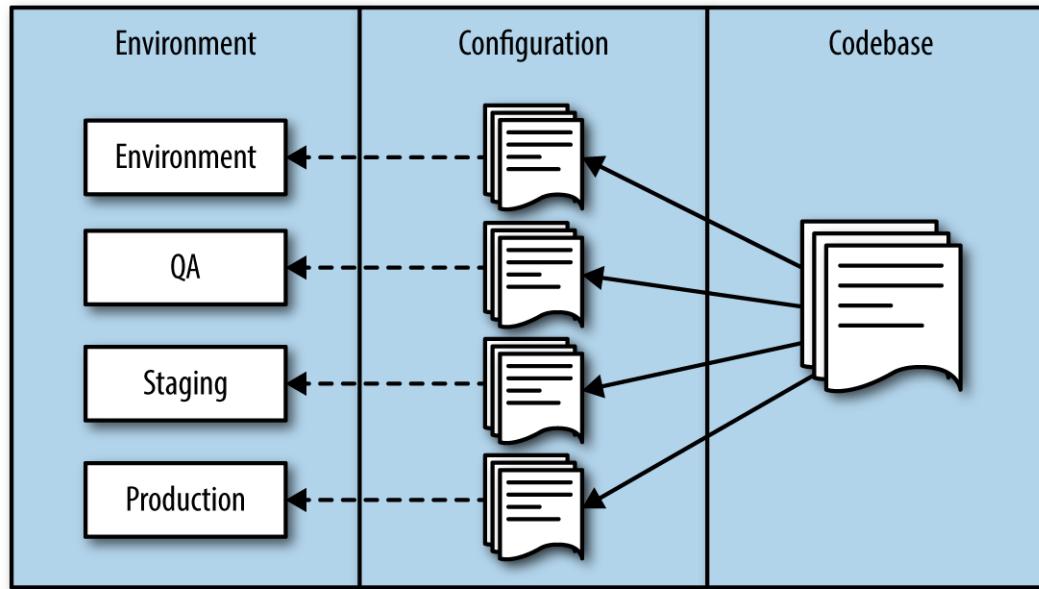
NETFLIX
OSS

v.i Twelve-Factor Apps. (n)

- 3. Configuraciones:
- Evitar configuraciones embebidas en el código.
- Verificar qué es una configuración:
 - Credenciales de acceso a recursos.
 - Información interna referente al ambiente de ejecución.
 - URLs de conexión a recursos externos a la aplicación.
- Mensajes de internacionalización o cadenas (“**strings**”) y/o valores que no cambian entre ambientes, no son parte de la configuración.

v.i Twelve-Factor Apps. (ñ)

- 3. Configuraciones:





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (o)

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - **Servicios back-end.**
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.





+

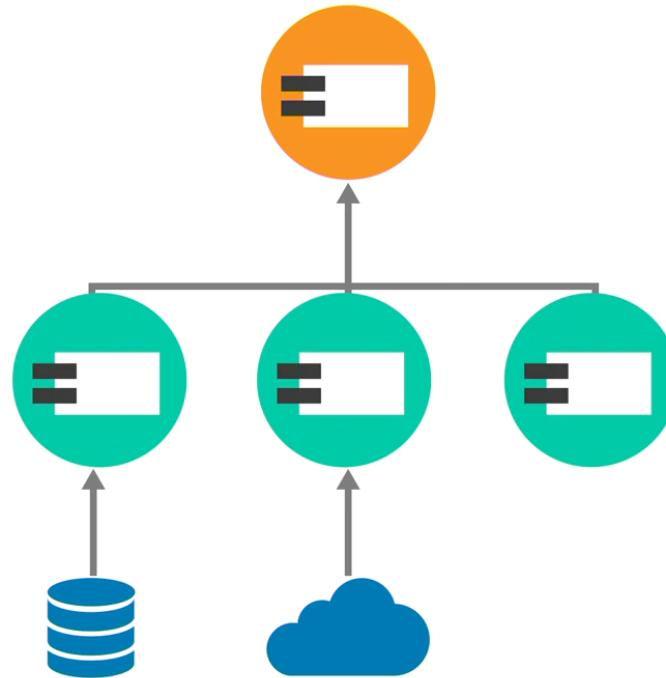
NETFLIX
OSS

v.i Twelve-Factor Apps. (p)

- 4. Servicios back-end:
- Tratar a los servicios “back-end” como recursos conectables.
- Cualquier recurso que ocupe la aplicación y que sea conectable a través de la red es un servicio “back-end”.
 - Bases de Datos SQL
 - Bases de Datos NoSQL
 - Broker de mensajes
 - Servicios de email SMTP
 - etc.
- URLs de conexión a servicios “back-end” son parte de la configuración.

v.i Twelve-Factor Apps. (q)

- 4. Servicios back-end:





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (r)

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - **Build, Deploy and run.**
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.





+

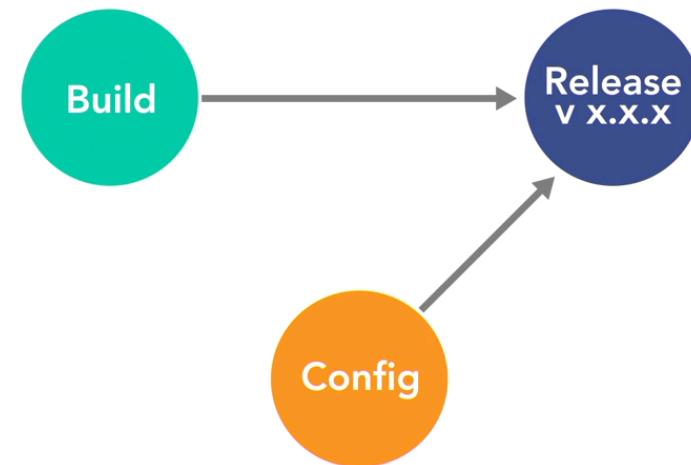
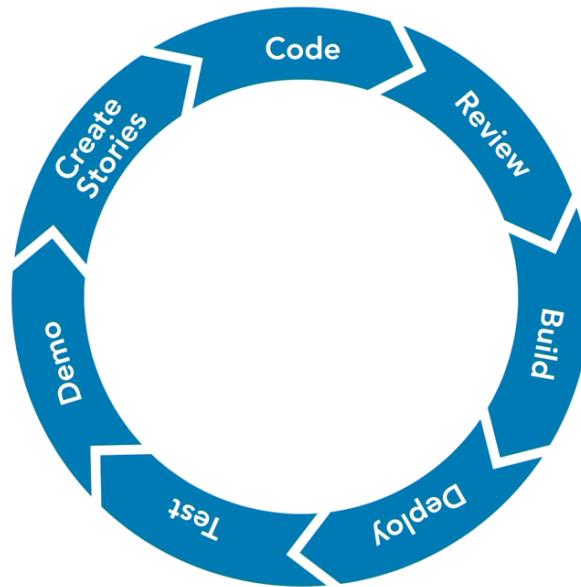
NETFLIX
OSS

v.i Twelve-Factor Apps. (s)

- 5. Build, Deploy and run:
- Implementa "pipe-lines" de CI/CD para compilar, desplegar y ejecutar.
- Automatiza la implementación de CI/CD.
- Implementa estrategias para mantener compilaciones repetibles, versionamiento de empaquetados y habilita regresar el aplicativo a una compilación anterior ("rollback").
- El código base será convertido en un compilado y después en un despliegue.

v.i Twelve-Factor Apps. (t)

- 5. Build, Deploy and run:





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (u)

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - **Procesos**
 - Asignación de puertos.
 - Concurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (v)

- 6. Procesos:
- Ejecutar la aplicación como uno o más procesos sin estado, en el entorno de ejecución.
- Las aplicaciones, al ser ejecutadas, se visualizan como procesos que no tienen estado y comparten la filosofía “**share-nothing**”.
- Cualquier información que requiera ser almacenada por la aplicación, debe ser delegada a un servicio “**back-end**” con estado como lo es una base de datos o un sistema de cache persistente.



v.i Twelve-Factor Apps. (w)

- 6. Procesos:
- La aplicación distribuida se ve como un conjunto de procesos para el personal de operaciones.
- “**Sticky-sessions**” o afinidad de sesión debe ser evitada. En aplicaciones legadas, la afinidad de sesión debe ser revisada e incluso reimplementada.



+

v.i Twelve-Factor Apps. (x)

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - **Asignación de puertos.**
 - Concurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.





v.i Twelve-Factor Apps. (y)

- 7. Asignación de puertos:
- Un aplicación basada en los 12 factores, no se incluye o despliega en un servidor de aplicaciones o servidor web.
- Las aplicaciones basadas en 12 factores son auto-contenidas, es decir no dependen de características específicas del entorno para su ejecución.
- Es posible que aplicaciones basadas en 12 factores incluyan un servidor web embebido para exponer sus servicios a través de HTTP/HTTPS.



+

NETFLIX
OSS

v.i Twelve-Factor Apps. (z)

- 7. Asignación de puertos:
- Exponer servicios mediante asignación de puertos o “**port-binding**” significa que la aplicación al ejecutarse, es decir el proceso, escucha peticiones en un puerto arbitrario y, será la configuración de sus aplicativos consumidores quienes establezcan a qué puerto debe conectarse para consumir el servicio.
- En ambiente de desarrollo el “**port-binding**” se realiza de forma manual, mediante variables de entorno o línea de comandos.



v.i Twelve-Factor Apps. (a')

- 7. Asignación de puertos:
- En ambientes de despliegue distintos a desarrollo, un framework o elemento de la topología de red se encargará de routear las comunicaciones entre servicios. De esta forma, el puerto que esuchen los aplicativos será comúnmente aleatorio.
- ¿Cómo se establecerá el puerto de los servicios productores a los consumidores?
 - Mediante descubrimiento de servicios.



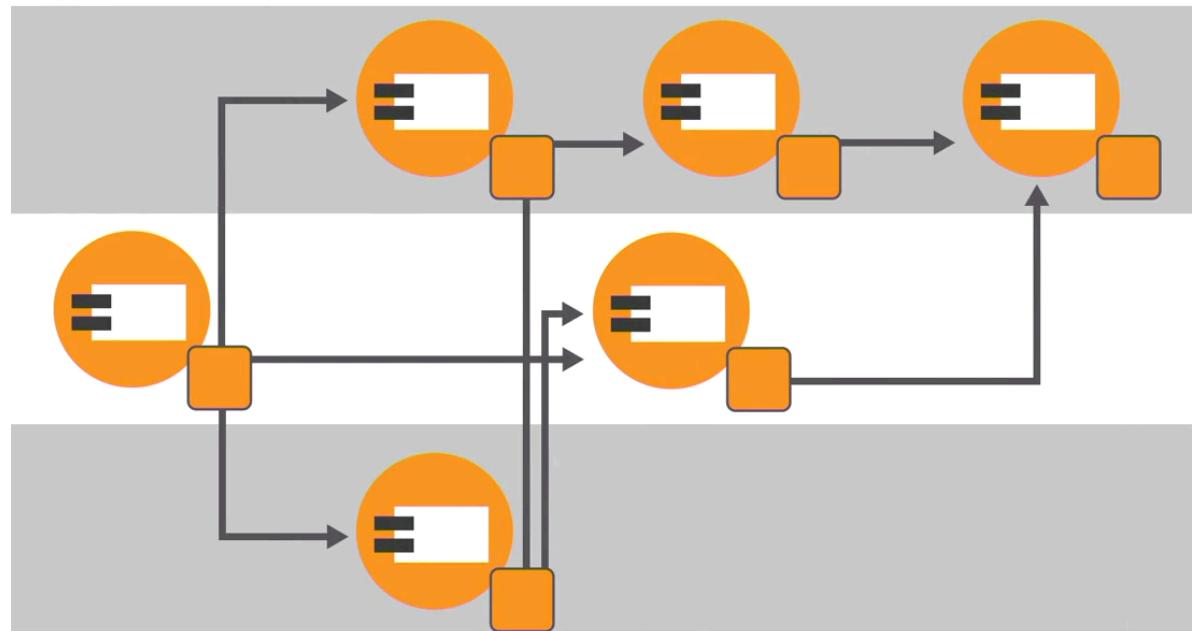
v.i Twelve-Factor Apps. (b')

- 7. Asignación de puertos:
- Comunmente el productor del servicio notifica a los consumidores sobre que puerto escucha peticiones y ello se lleva a cabo mediante un framework o contenedor en el ambiente de ejecución que maneje esta asignación de puertos o "**port-binding**" por lo tanto no nos debemos de preocupar por ello.
- Debido a que los servicios se ejecutan de forma aislada y son autocontenido, comunmente los aplicativos, si utilizan HTTP para el transito de mensajes, utilizan todos el puerto **80** para las comunicaciones.



v.i Twelve-Factor Apps. (c')

- 7. Asignación de puertos:





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (d')

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - **Concurrencia.**
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.



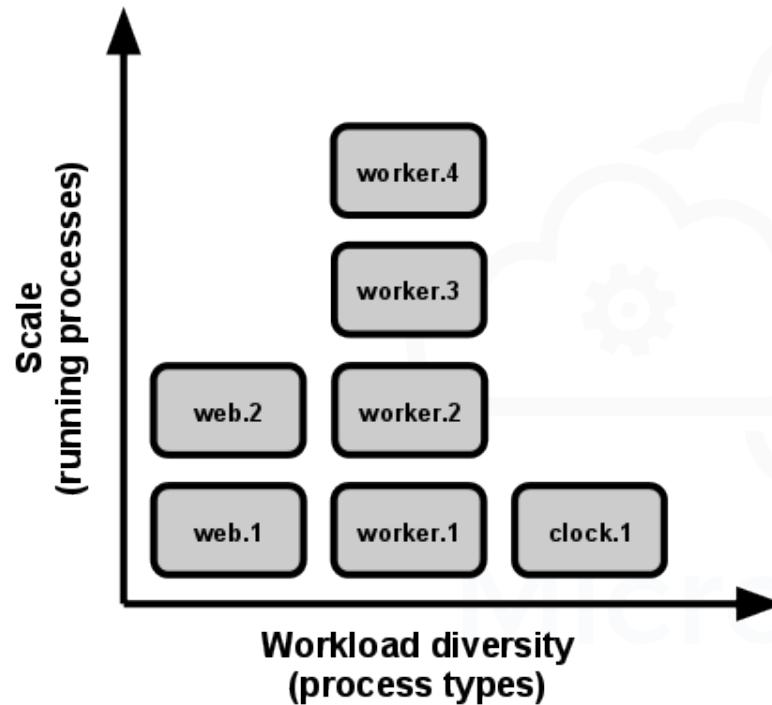


v.i Twelve-Factor Apps. (e')

- 8. Conurrencia:
- Escalar aplicaciones mediante el modelo de escalamiento horizontal.
- Ejecutar una o más replicas por servicio de forma horizontal habilitará mejorar la concurrencia de los servicios expuestos.
- Comunmente los servicios “back-end” con estado, como las bases de datos, son los elementos más difíciles de escalar, por tanto es necesario implementar patrones asíncronos para trabajar con ellos.
- Considere siempre diseñar para la escalabilidad horizontal.

v.i Twelve-Factor Apps. (f')

- 8. Conurrencia:





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (g')

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - **Desechabilidad.**
 - Paridad de ambientes.
 - Logs.
 - Procesos Administrativos.





v.i Twelve-Factor Apps. (h')

- 9. Desechabilidad:
- Considerar las aplicaciones de 12 factores como aplicaciones desecharables, es decir que implementen inicios (o reinicios) rápidos y terminaciones seguras (y en cualquier momento).
- La desechabilidad permite conseguir minimizar el tiempo de arranque lo que implica escalado más rápido y flexible, despliegues más rápidos y robustos.
- En sistemas distribuidos todo puede fallar, por tanto todos los servicios deben ser desecharables, iniciar y terminar lo más rápido posible.



v.i Twelve-Factor Apps. (i')

- 9. Desechabilidad:
- La desechabilidad beneficia a:
 - **La seguridad:** debido a que favorece los contextos de vida cortos (“short-lived-contexts”) evitando el tiempo de vida de un componente si éste ha sido comprometido.
 - **La escalabilidad:** permite escalar aumentando o disminuyendo réplicas conforme sea requerido de forma fácil, ágil y flexible.
 - **La tolerancia a fallos:** los fallos ocurren, por tanto, ofrecer mecanismos de inicio y terminación rápido permite una mejor respuesta ante los fallos, logrando aplicaciones más resistentes.



+

NETFLIX
OSS

v.i Twelve-Factor Apps. (j')

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - **Paridad de ambientes.**
 - Logs.
 - Procesos Administrativos.





v.i Twelve-Factor Apps. (k')

- 10. Paridad de ambientes:
- Mantener los diferentes ambientes (dev, pre, pro) tan parecidos como sea posible.
- Las aplicaciones basadas en los 12 factores están diseñadas para realizar despliegues continuos que reduzcan la brecha entre los entornos de desarrollo y producción lo cual habilita mayores despliegues de nuevos “releases” disminuyendo la puesta a producción.

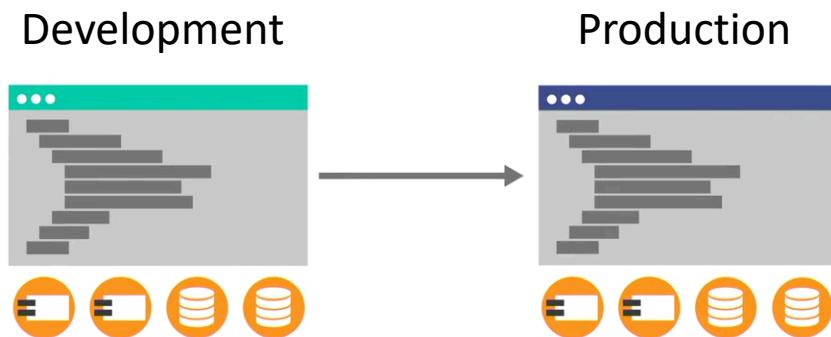


v.i Twelve-Factor Apps. (I')

- 10. Paridad de ambientes:
- Mantener paridad entre ambientes también reduce los problemas al arreglar un “**bug**” en el código.
- Conjuntar múltiples ramas de trabajo en un único release o despliegue, dificulta el rastreo de “**bugs**” y su posible arreglo.
- Entre mayor sea el tiempo en encontrar un “**bug**”, más costoso es su arreglo. Entre mayor sea la diferencia entre ambientes de dev y prod, mayor será la probabilidad de encontrar errores en el código.

v.i Twelve-Factor Apps. (m')

- 10. Paridad de ambientes:



Same people





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (n')

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - **Logs.**
 - Procesos Administrativos.





v.i Twelve-Factor Apps. (ñ')

- 11. Logs:
- Los “**logs**” o historiales son muy importantes para las operaciones, principalmente para encontrar y arreglar errores en la aplicación.
- Tradicionalmente los “**logs**” son mostrados en consola o en archivos de texto y de esa forma son analizados.
- Para sistemas distribuidos, revisar “**logs**” en este tipo de medios no es práctico ni apoya a las operaciones.



+

NETFLIX
OSS

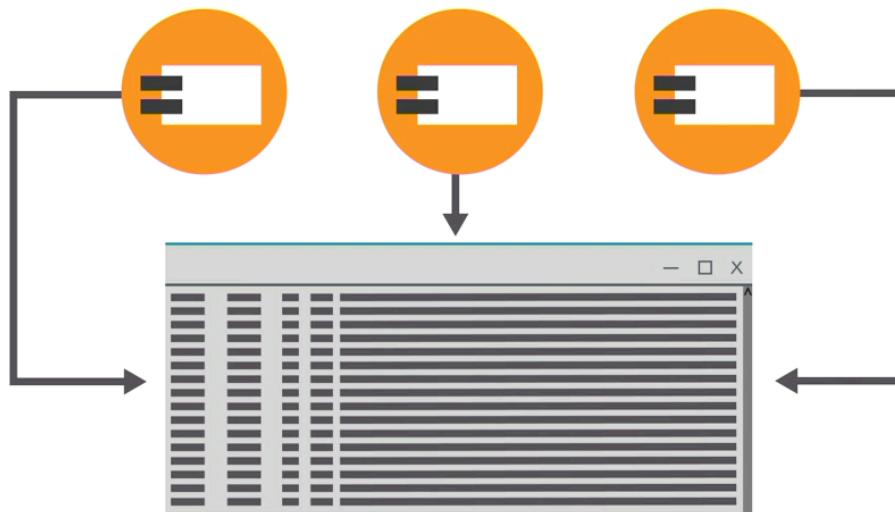
v.i Twelve-Factor Apps. (o')

- 11. Logs:
- Una aplicación basada en los 12 factores nunca se preocupa del direccionamiento o el almacenamiento del flujo de salida de sus “**logs**”.
- Los “**logs**” deben ser tratados como un flujo de eventos el cuál es compartido por toda la aplicación.
- Existen herramientas como ELK para tratar logs como eventos y almacenarlos para su futura consulta y referencia.



v.i Twelve-Factor Apps. (p')

- 11. Logs:





+

NETFLIX
OSS

v.i Twelve-Factor Apps. (q')

- Principios twelve-factor apps:
 - Código Base.
 - Dependencias.
 - Configuraciones.
 - Servicios back-end.
 - Build, Deploy and run.
 - Procesos
 - Asignación de puertos.
 - Conurrencia.
 - Desechabilidad.
 - Paridad de ambientes.
 - Logs.
 - **Procesos Administrativos.**





v.i Twelve-Factor Apps. (r')

- 12. Procesos Administrativos:
- Implementa procesos administrativos como migración de base de datos, eliminación de registros que han expirado o eliminación de datos en cache, como otro servicio o aplicación la cuál cumpla con sus propios principios de 12 factores.
- La metodología de 12 factores recomienda el uso de lenguajes que proporcionen consola de tipo REPL para la ejecución de comandos desde Shell.



v.i Twelve-Factor Apps. (s')

- 12. Procesos Administrativos:
- Las aplicaciones administrativas deben poder ser iniciadas rápido, ejecutadas y terminadas al final de su uso.
- No se recomienda mantener el aplicativo del proceso administrativo ejecutándose mientras no se encuentra operando debido a que ello consumirá recursos.
- Se recomienda implementar algún disparador o “trigger” que ejecute el proceso administrativo y evitar su despliegue manual.



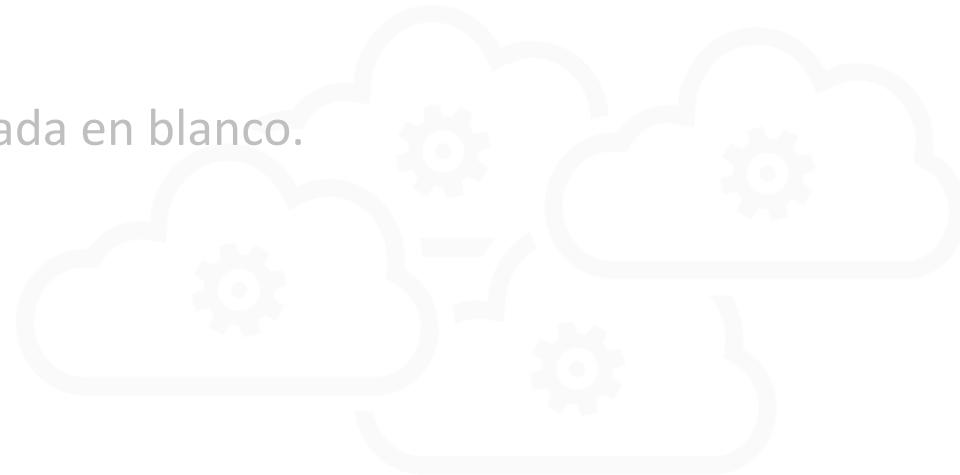
Resumen de la lección

v.i Twelve-Factor Apps.

- Comprendemos la metodología "twelve-factor apps".
- Analizamos cada uno de los doce factores de la metodología "twelve-factor apps".



Esta página fue intencionalmente dejada en blanco.



Microservices



+

v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS

- v.i Twelve-Factor Apps.
- v.ii Spring Cloud y Spring Cloud Netflix OSS
- v.iii Configuración externalizada con Spring Cloud Config y Spring Cloud Bus.
- v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka.
- v.v Balanceo de carga del lado del cliente con Ribbon.
- v.vi Clientes REST declarativos con Feign.
- v.vii Implementación de corto-circuito con Hystrix.
- v.viii Visualización de corto-circuitos con Turbine.
- v.ix API Gateway con Spring Cloud Zuul.



+

NETFLIX
OSS

v.ii Spring Cloud y Spring Cloud Netflix OSS



Objetivos de la lección

v.ii Spring Cloud y Spring Cloud Netflix OSS

- Conocer a grandes rasgos los beneficios de utilizar Spring Cloud en aplicaciones de microservicios mediante Spring Boot.
- Conocer las implementaciones principales que provee Spring Cloud Netflix OSS para el desarrollo de aplicaciones de microservicios con Spring Cloud.



v.ii Spring Cloud y Spring Cloud Netflix OSS (a)

- Spring Cloud es un proyecto construido sobre Spring Boot que facilita el desarrollo de aplicaciones nativas para la nube o “**cloud-native**”.
- Es un framework para el desarrollo de aplicaciones en la nube, basado en Java y Spring Boot, el cual proporciona implementaciones “**out-of-the-box**” para desarrollar aplicaciones basadas en la metodología “**twelve-factor apps**”.



v.ii Spring Cloud y Spring Cloud Netflix OSS (b)

- Spring Cloud ofrece un conjunto de herramientas, librerías y buenas prácticas para construir aplicaciones distribuidas rápidamente.
- Integra patrones comunes como administración de configuraciones (“**configuration-management**”), servicio de registro y descubrimiento de servicios (“**service-discovery**” y “**service-registry**”), corto circuito (“**circuit-breakers**”), enrutamiento inteligente (“**routing**”), bus de control (“**control-bus**”), “**leader-election**”, sesiones distribuidas, entre otros.

v.ii Spring Cloud y Spring Cloud Netflix OSS (c)

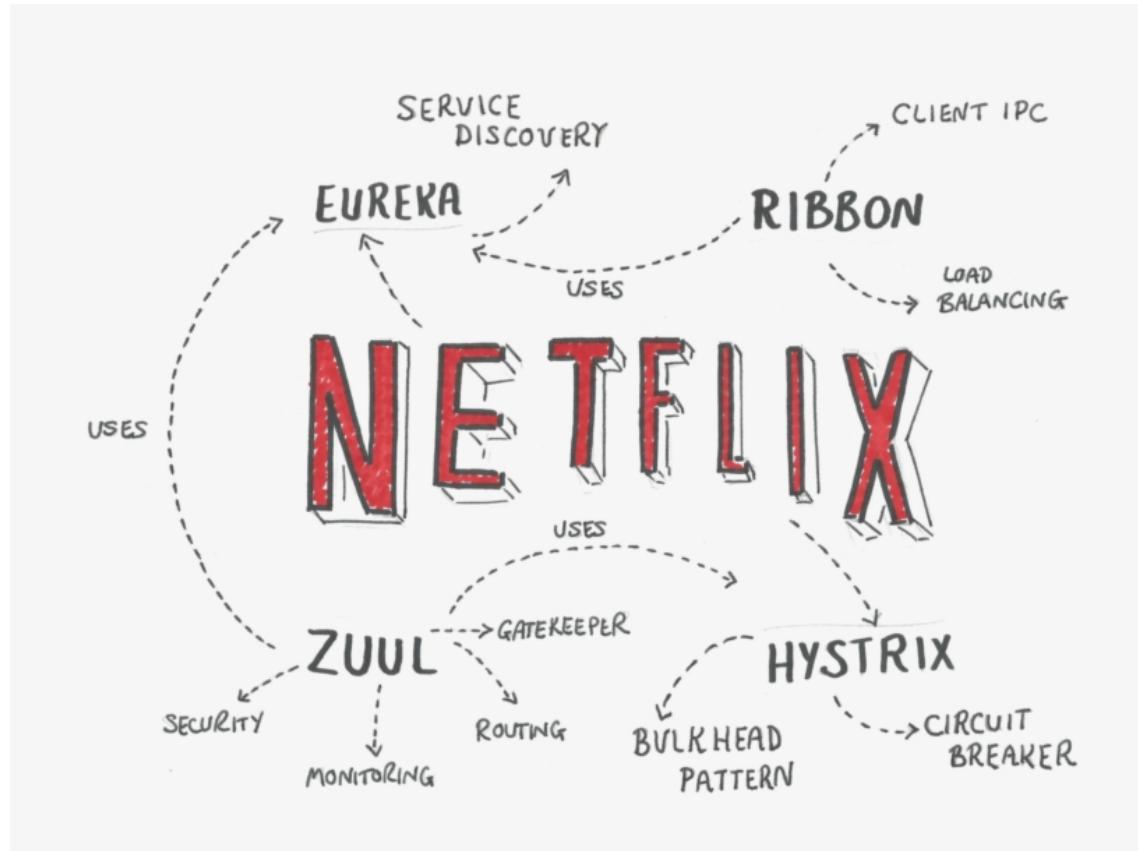
- Spring Cloud implementa código de plomería (“**plumbing-code**” o “**boilerplate-code**”) que facilita la coordinación entre microservicios para cuestiones relativas como por ejemplo, la configuración externalizada o la implementación de “**circuit-breakers**”.
- Spring Cloud habilita mayor velocidad a los desarrolladores y permite implementar “**pair-programming**” con el equipo de Spring Cloud y Spring Boot.



v.ii Spring Cloud y Spring Cloud Netflix OSS (d)

- Spring Cloud Netflix OSS (Open-Source Software) provee integraciones para Spring Cloud mediante implementaciones de patrones ampliamente probados por Netflix (**"well-battle-tested-components"**) para aplicaciones distribuidas tales como:
 - **Eureka**: Servicio de descubrimiento y registro de servicios.
 - **Hystrix**: Implementación de **"Circuit Breaker Pattern"**.
 - **Zuul**: Servidor de borde o **"Edge Server"** implementando funcionalidades de **API Gateway** y enrutamiento inteligente.
 - **Ribbon**: Balanceo de carga del lado del cliente.
 - **Feign**: Implementación de clientes REST declarativos.
 - **Archaius**: Servicio de configuración externa.
 - entre otros: <https://netflix.github.io/>

v.ii Spring Cloud y Spring Cloud Netflix OSS (e)



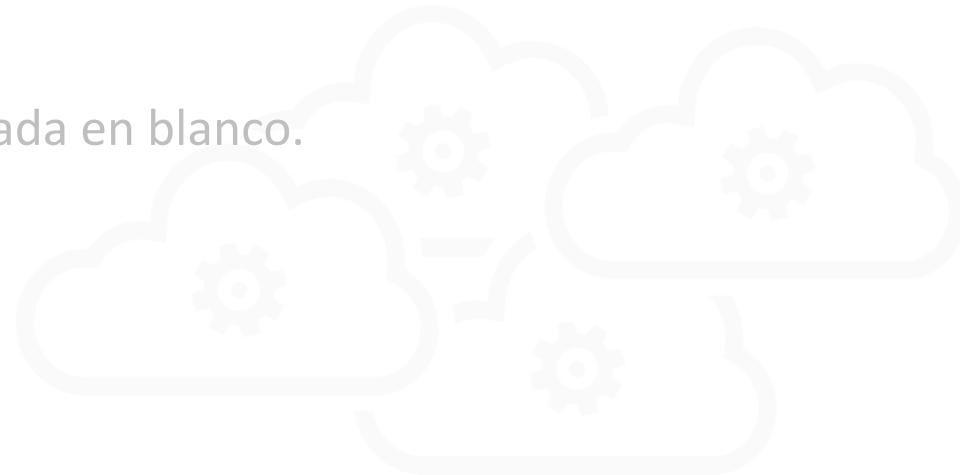
Resumen de la lección

v.ii Spring Cloud y Spring Cloud Netflix OSS

- Conocimos los principales proyectos relativos a Spring Cloud para implementar aplicaciones de microservicios mediante Spring Boot.
- Conocimos las principales implementaciones que provee Spring Cloud Netflix OSS para el desarrollo de aplicaciones de microservicios con Spring Cloud.



Esta página fue intencionalmente dejada en blanco.



Microservices



v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS

- v.i Twelve-Factor Apps.
- v.ii Spring Cloud y Spring Cloud Netflix OSS
- v.iii Configuración externalizada con Spring Cloud Config y Spring Cloud Bus.
- v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka.
- v.v Balanceo de carga del lado del cliente con Ribbon.
- v.vi Clientes REST declarativos con Feign.
- v.vii Implementación de corto-circuito con Hystrix.
- v.viii Visualización de corto-circuitos con Turbine.
- v.ix API Gateway con Spring Cloud Zuul.



+

NETFLIX
OSS

v.iii Configuración externalizada con Spring Cloud Config y Spring Cloud Bus.



+

NETFLIX
OSS

Objetivos de la lección

v.iii Configuración externalizada con Spring Cloud Config y Spring Cloud Bus.

- Conocer el servidor de configuración externalizada Spring Cloud Config Server.
- Analizar por qué es necesario implementar un servidor de configuración externo.
- Implementar aplicaciones de microservicios mediante configuración externalizada con Spring Cloud Config Server.
- Implementar Spring Cloud Bus para "reiniciar" los microservicios y "recargar" su configuración externalizada.

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (a)

- Spring Cloud Config es un administrador de configuración versionada y centralizada para aplicaciones distribuidas.
- Spring Cloud Config ofrece soporte para aplicaciones cliente-servidor el cual ofrece un mecanismo para implementar configuración externalizada para sistemas distribuidos.
- Mediante un servidor de configuración se habilita un servicio central que administre la configuración de los aplicativos (microservicios) de forma centralizada el cual puede ser accesible para todos los servicios y ambientes de despliegue.

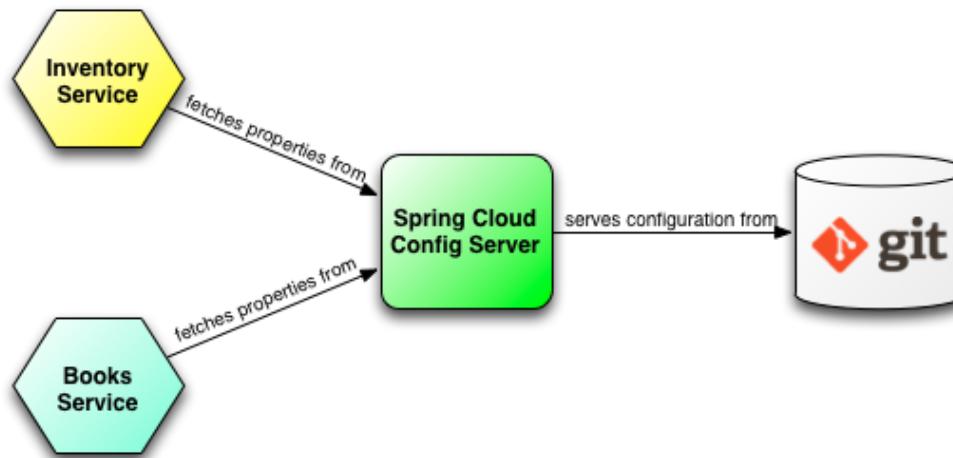


v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (b)

- Cliente vs Servidor
- En microservicios o arquitecturas distribuidas, el proveedor de un servicio, comúnmente es llamado “servidor” sin embargo el servidor también puede ser “cliente” de otro servicio o proveedor.
- Ser cliente o servidor es relativo y sólo indica el rol de una relación de consumo cliente-servidor.

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (c)

- Spring Cloud Config Server.
- Spring Cloud Config Server puede obtener la configuración de los microservicios desde “**file-system**” o desde un sistema de control de versiones como Git.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (d)

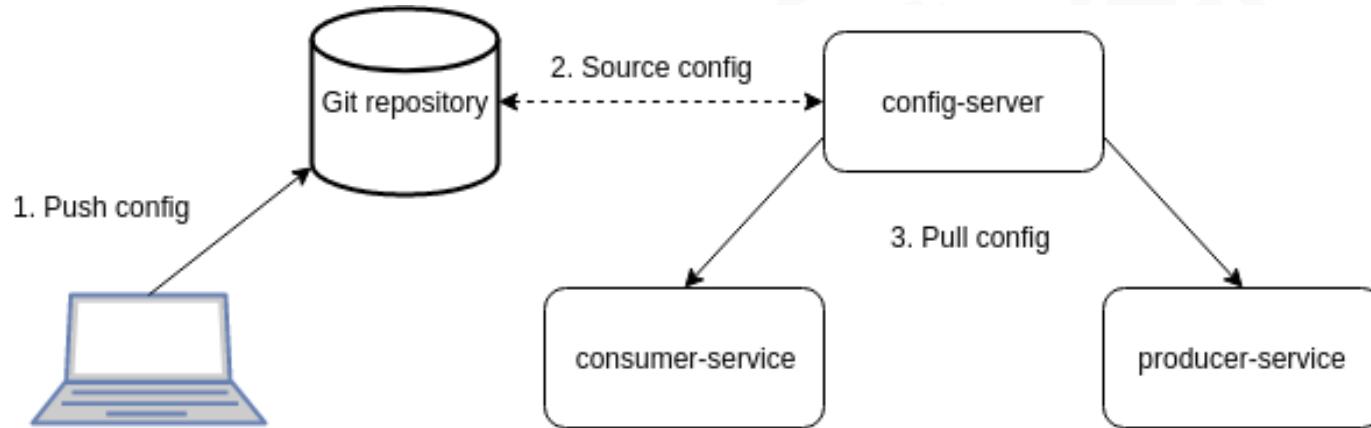
- ¿Por qué necesitamos un servidor de configuración como Spring Cloud Config Server?
 - Empaquetar las archivos de configuración dentro del aplicativo.
 - Requiere re-construir, re-desplegar y re-iniciar el aplicativo.
 - Archivos de configuración en un file system “común”.
 - Mala idea, no disponible en ambientes “**cloud-aware**”.
 - Variables de entorno/ambiente.
 - Se realiza de forma diferente en diversas plataformas
 - Extenso número de configuraciones por realizar.
 - Utilizar alguna solución específica del proveedor “**cloud**”. No.

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (e)

- ¿Por qué necesitamos un servidor de configuración como Spring Cloud Config Server?
 - Cantidad de Microservicios a desplegar conlleva un gran número de configuraciones que manejar de forma manual.
 - Actualizaciones dinámicas en la configuración del microservicio requerirán reinicio manual del mismo.
 - ¿Cómo versionamos la configuración? Utiliza configuración como código habilita versiónado y permite trazabilidad.

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (f)

- Designar a un servidor de configuración basado en Spring Cloud Config Server para servir la configuración de los microservicios conectados a través de HTTP.
- Utilizar Git o “**file system**” como fuente de configuraciones.





+

NETFLIX
OSS

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (g)

- Spring Cloud Config Server.
- Para generar un servidor de configuración es necesario agregar las dependencias requeridas:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<properties>
  <java.version>1.8</java.version>
  <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
</properties>
```



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (h)

- Spring Cloud Config Server.
- Ningun proyecto basado en Spring Cloud se recomiendo configurar sin el apoyo de Spring Boot, por tanto el <parent> del pom.xml sigue siendo spring-boot-starter-parent.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (i)

- Spring Cloud Config Server.
- Para habilitar un Spring Cloud Config Server, agregar la dependencia:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    ...
</dependencies>
```



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (j)

- Spring Cloud Config Server.
- Agregar la anotación **@EnableConfigServer** a la clase principal del proyecto.

@EnableConfigServer

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (k)

- Spring Cloud Config Server.
- Por último, agregar las propiedades correspondientes a la configuración sobre el archivo “**application.properties**” o “**application.yml**”.
- Se implementarán dos tipos de configuración para el servidor Spring Cloud Config Server.
 - Implementando el perfil “**native**”.
 - Implementando el perfil “**git**”.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (I)

- Spring Cloud Config Server, native configuration.
- La configuración “**native**” indica que el servidor de configuración utilizará una ruta local en el “**file system**” para servir los archivos de configuración.
- Implementar en el proyecto del servidor de configuración las siguientes propiedades sobre el archivo “**application.properties**”:
 - # native or git
 - **spring.profiles.active=native**
 - **spring.application.name=MyConfigServer**



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (m)

- Spring Cloud Config Server, native configuration.
- Al habilitar el perfil activo “**native**”, podemos especificar propiedades específicas al perfil “**native**”.
- Especificar las siguientes propiedades sobre el archivo “**application-native.properties**”:
 - # For native
 - **spring.cloud.config.server.native.searchLocations=<ruta>**



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (n)

- Spring Cloud Config Server, native configuration.
- Spring Cloud Config Server utilizará la propiedad **spring.cloud.config.server.native.searchLocations**, para encontrar las configuraciones y servirlas a los clientes.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (ñ)

- Spring Cloud Config Server, git configuration.
- La configuración “git” indica que el servidor de configuración utilizará el sistema de control de versiones “git” para servir los archivos de configuración.
- Implementar en el proyecto del servidor de configuración las siguientes propiedades sobre el archivo “**application.properties**”:
 - # native or git
 - **spring.profiles.active=git**
 - **spring.application.name=MyConfigServer**



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (o)

- Spring Cloud Config Server, git configuration.
- Al habilitar el perfil activo “git”, podemos especificar propiedades específicas al perfil “git” las cuales servirán para obtener las configuraciones desde un repositorio, ya sea local o remoto.
- Especificar las siguientes propiedades sobre el archivo “**application-native.properties**”:
 - # For git
 - **spring.cloud.config.server.git.uri=<git url>**
 - # only necessary if private repository
 - **spring.cloud.config.server.git.username=<username>**
 - **spring.cloud.config.server.git.password=<password>**



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (p)

- Spring Cloud Config Client.
- Cualquier microservicio requerirá configuración.
- Difícilmente un microservicio se desplegará exactamente igual en los diferentes ambientes.
- Spring Cloud Config habilita el registro automático del microservicio hacia el servidor de configuración y extraer su configuración basandose en propiedades del microservicio definidas sobre el archivo **“bootstrap.properties”** o en su versión YAML.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (q)

- Spring Cloud Config Client.
- Para habilitar la configuración de Spring Cloud Config Client sobre cualquier microservicio para obtener su configuración desde el servidor de configuraciones será requerido agregar las dependencias de Spring Cloud correspondientes (revisar siguiente slide).

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (r)

- Spring Cloud Config Client.
- Dependencias de Spring Cloud:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```



```
<properties>
  <java.version>1.8</java.version>
  <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
</properties>
```



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (s)

- Spring Cloud Config Client.
- De igual forma que en la configuración de Spring Cloud Config Server, ningun proyecto basado en Spring Cloud se recomiendo configurar sin el apoyo de Spring Boot, por tanto es requerido definir el <parent> del pom.xml mediante la dependencia spring-booot-starter-parent.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.6.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (t)

- Spring Cloud Config Client.
- Para habilitar un Spring Cloud Config Client, agregar la dependencia:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    ...
</dependencies>
```



+

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (u)

- Spring Cloud Config Client.
- Cualquier aplicación o microservicio que requiera consumir el servidor de configuración basado en Spring Cloud Config Server requerirá definir propiedades sobre el archivo “**bootstrap.properties**”.
 - # bootstrap.properties
 - **spring.cloud.config.uri=<config server URL>**
 - **spring.cloud.config.name=\${spring.application.name}**
 - **spring.cloud.config.profile=\${spring.profiles.active}**
 - # branch to get config from remote, defaults “master”
 - **spring.cloud.config.label=release1**



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (v)

- Spring Cloud Config Client.
- Antes de que Spring Boot comience a configurar beans requerirá satisfacer todas las propiedades requeridas, por tanto, las propiedades definidas en el archivo “**bootstrap.properties**” indican a Spring que deberá obtener dichas configuraciones (propiedades) desde el servidor de configuración.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (w)

- Spring Cloud Config Client.
- Spring Cloud Config Client realizará la búsqueda de las propiedades (configuración) de la aplicación/microservicio de la siguiente manera:
 - `{context}/{application-name}/{profile}[/{label}]`
 - `{context}/{application-name}-{profile}.yml`
 - `{context}/{label}/{application}-{profile}.yml`
 - `{context}/{application}-{profile}.properties`
 - `{context}/{label}/{application}-{profile}.properties`
- Donde `{context}` es la URI definida por la propiedad `spring.cloud.config.uri`.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (x)

- Spring Cloud Config Client.
- Al utilizar las propiedades **spring.application.name** y **spring.profiles.active** sobre el archivo “**bootstrap.properties**”, dichas propiedades deberán ser, también, definidas sobre el archivo “**application.properties**”.
 - # application.properties
 - **spring.application.name=<application-name>**
 - **spring.profiles.active=<active-profile>**
- ¡ Listo !, es posible empezar a utilizar propiedades desde el servidor de configuración en los microservicios cliente.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (y)

- Práctica 21. Spring Cloud Config - Parte 1
- Analiza la aplicación **21-Spring-Cloud-Config-Server**, **21-Hello-Spring-Cloud-Config-Client** y **21-Users-Spring-Cloud-Config-Client**.
- Ingresar a la ruta: **{tu-workspace}/21-Spring-Cloud-Config-Server**
- Importar el proyecto **21-Spring-Cloud-Config-Server** en STS.
- Agregar la dependencia **org.springframework.cloud:spring-cloud-config-server** en el archivo pom.xml. Revisar el apartado `<dependencyManagement>`.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (z)

- **Práctica 21. Spring Cloud Config - Parte 1**
- Sobre la clase **Application**, clase principal del proyecto del paquete **com.consulting.mgt.springboot.practica21.configserver**, habilitar la auto- configuración de Spring Cloud Config Server mediante la anotación **@EnableConfigServer**.
- Sobre el archivo “**application.properties**” define las propiedades para:
 - Habilitar el perfil activo “**native**”.
 - Asignar el nombre de la aplicación como “**MyConfigServer**”.
 - Habilita el servlet context-path “**/my-config-server**”.
 - Define el puerto **9090** para el “**web-server**” del tomcat embebido de la aplicación.

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (a')

- **Práctica 21. Spring Cloud Config - Parte 1**
- En otro archivo de configuración, particular para el perfil “native”, define la propiedad **spring.cloud.config.server.native.searchLocations** donde se asigne la ruta **classpath:/development-config-files**, la cual contiene los archivos de configuración (propiedades) de los microservicios los cuales servirá.
- Por otro lado, sobre otro archivo de configuración, particular para el perfil “git”, define la propiedad **spring.cloud.config.server.git.uri** donde se asigne la **URI** del repositorio **git** a utilizar para servir la configuración (propiedades) de los microservicios los cuales servirá.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (b')

- **Práctica 21. Spring Cloud Config - Parte 1**
- Compile y ejecute la aplicación desde línea de comandos y habilite el perfil activo “**native**”:
 - mvn clean package
 - java -jar target/21-Spring-Cloud-Config-Server-0.0.1-SNAPSHOT.jar --spring.profiles.active=native
- Pruebe las diferentes formas de obtener configuraciones de los microservicios “**MyConfigServer**” y “**HelloService**”.
 - <http://localhost:9090/my-config-server/MyConfigServer/pre>
 - <http://localhost:9090/my-config-server/HelloService-dev.properties>
 - <http://localhost:9090/my-config-server/master/HelloService-dev.yml>



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (c')

- Práctica 21. Spring Cloud Config - Parte 1
- Ingresar a la ruta: {tu-workspace}/21-Hello-Spring-Cloud-Config-Client
- Importar el proyecto 21-Hello-Spring-Cloud-Config-Client en STS.
- Agregar la dependencia **org.springframework.cloud:spring-cloud-starter-config** en el archivo pom.xml. Revisar el apartado <dependencyManagement>.
- Analizar la clase principal **Application**, del paquete **com.consulting.mgt.springboot.practica21.hello.configclient**.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (d')

- Práctica 21. Spring Cloud Config - Parte 1
- Definir en el archivo **bootstrap.properties** las siguientes propiedades:
 - # Spring Cloud Config Client configuration
 - **spring.cloud.config.uri=http://localhost:9090/my-config-server**
 - **spring.cloud.config.name=\${spring.application.name}**
 - **spring.cloud.config.profile=\${spring.profiles.active}**
 - # branch to get config from remote
 - **spring.cloud.config.label=release1**



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (e')

- **Práctica 21. Spring Cloud Config - Parte 1**
- Definir en el archivo **application.properties** las siguientes propiedades:
 - # Spring application properties
 - **spring.application.name=HelloService**
 - **server.servlet.context-path=/hello-config-client**
 - **server.port=9091**
- Compile y ejecute la aplicación desde línea de comandos y habilite el perfil activo “**dev**”:
 - mvn clean package
 - java -jar target/21-Hello-Spring-Cloud-Config-Client-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev



+

NETFLIX
OSS

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (f')

- **Práctica 21. Spring Cloud Config - Parte 1**
- Pruebe la aplicación accediendo a la URL <http://localhost:9091/hello-config-client/> desde el navegador. ¿Cuál es el mensaje que se despliega?
- Intente el mismo ejercicio habilitando los diferentes perfiles, el mensaje debe ser diferente para cada uno de los casos.
- Compruebe cuál deberá de ser el mensaje que despliegue el aplicativo al ejecutar los perfiles **dev**, **pre**, **pro** y el perfil **default** (sin activar perfil).



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (g')

- Práctica 21. Spring Cloud Config - Parte 1
- Ingresar a la ruta: {tu-workspace}/21-Users-Spring-Cloud-Config-Client
- Importar el proyecto 21-Users-Spring-Cloud-Config-Client en STS.
- Agregar la dependencia **org.springframework.cloud:spring-cloud-starter-config** en el archivo pom.xml. Revisar el apartado <dependencyManagement>.
- Analizar la clase principal **Application**, del paquete **com.consulting.mgt.springboot.practica21.hello.configclient**. Para acceder a la configuración centralizada que proporciona Spring Cloud Config Server, no es necesario que el cliente sea una aplicación web.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (h')

- **Práctica 21. Spring Cloud Config - Parte 1**
- Define las propiedades requeridas, sobre el archivo “**bootstrap.properties**” para acceder al servidor Spring Cloud Config Server y obtener la configuración requerida para éste microservicio llamado **UserService**.
- Define en el archivo “**application.properties**” la propiedad **spring.application.name**.
- Puedes habilitar por default un perfil sobre el archivo “**application.properties**” sin embargo la recomendación es que se habilite en tiempo de ejecución, es decir a través de línea de comando.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (i')

- **Práctica 21. Spring Cloud Config - Parte 1**
- La configuración del microservicio **UsersService** no se encuentra en la ubicación **“classpath:/development-config-files”** del servidor de configuración. Recuerde que el servidor de configuración está ejecutándose con el perfil **“native”**.
- Vuelva a ejecutar el servidor de configuración cambiando el perfil activo del a **“git”** mediante línea de comando.
- Verifique que en la URI del repositorio git se encuentre la configuración requerida para el microservicio **UsersService**.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (j')

- **Práctica 21. Spring Cloud Config - Parte 1**
- Compile y ejecute la aplicación desde línea de comandos y habilite el perfil activo “**dev**”:
 - mvn clean package
 - java -jar target/21-Users-Spring-Cloud-Config-Client-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev
- En caso de no cambiar el perfil activo a “**git**” del servidor de configuración, la salida en consola será:
 - no message
 - dev



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (k')

- **Práctica 21. Spring Cloud Config - Parte 1**
- En caso de si haber cambiado el perfil activo a “git” del servidor de configuración, la salida en consola será:
 - Hi users service dev from git!
 - dev
- Analice los resultados.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (I')

- Spring Cloud Bus.
- Es un proyecto basado en **Spring Cloud** y **RabbitMQ** que permite enlazar ("link") todos los nodos de un sistema distribuido mediante un "broker" de mensajes.
- El "broker" de mensajes es utilizado para emitir cambios de estado o cualquier otro evento de administración para todos los servicios, nodos o instancias que estén escuchando dicho evento.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (m')

- Spring Cloud Bus.
- La idea principal de Spring Cloud Bus es implementar un actuador distribuido para todos los componentes o microservicios interconectados a través del “**broker**”.
- Por el momento únicamente son soportados **RabbitMQ** o **Kafka** como medios de transporte para enlazar los microservicios.



+

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (n')

- Spring Cloud Bus.
- Para iniciar a interconectar microservicios a través del “**broker**” RabbitMQ para comunicar cambios de estado entre microservicios es necesario agregar la dependencia:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

Verificar que “actuator” esté en el classpath:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (ñ')

- Spring Cloud Bus.
- Una vez agregada la dependencia `spring-cloud-starter-bus-amqp` agregar las propiedades requeridas, para conectar la aplicación con el "broker" RabbitMQ, típicas de Spring Boot.
 - # RabbitMQ configuration
 - `spring.rabbitmq.host=localhost`
 - `spring.rabbitmq.port=5672`
 - `spring.rabbitmq.username=guest`
 - `spring.rabbitmq.password=guest`



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (o')

- Spring Cloud Bus.
- Spring Cloud Bus habilitará el reinicio o “refresh” de los beans anotados con la anotación **@RefreshScope** cuyas propiedades o configuración sean injectadas a través de **@Value**, **@ConfigurationProperties** o accedidas directamente desde el objeto **Environment**.
- Para habilitar el reinicio o “refresh” de los beans anotados con **@RefreshScope**, es necesario también habilitar la exposición web del actuador “bus-refresh” en todos los microservicios interconectados.
 - **management.endpoints.web.exposure.include=bus-refresh**

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (p')

- Spring Cloud Bus.
- Una vez habilitado la exposición web del actuador “**bus-refresh**” en todos los microservicios interconectados. Para habilitar el reinicio o “**refresh**” automático de todos los microservicios en el “**cluster**” será necesario ejecutar una petición **POST** a cualquiera de los microservicios interconectados.
 - # Mediante httpie
 - **http POST http://localhost:9092/users-config-client/actuator/bus-refresh**
 - # Mediante cURL
 - **curl -X POST http://localhost:9092/users-config-client/actuator/bus-refresh**
- ¿Para que casos necesitaremos reiniciar los microservicios?



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (q')

- Spring Cloud Bus.
- A su vez, es posible agregar pares clave/valor (propiedades) al objeto **Environment** de las aplicaciones Spring Boot mediante otro “**endpoint**” actuador, habilitando la exposición del actuador “**bus-env**” a través de la propiedad **management.endpoints.web.exposure.include=bus-env**
- El actuador “**bus-env**” espera un **JSON** con los atributos “**name**” y “**value**”, donde “**name**” hace referencia al nombre de la propiedad a asignar y “**value**” hace referencia al valor de la propiedad a asignar.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (r')

- Spring Cloud Bus.
- Para asociar un par clave/valor al objeto **Environment** de todos los microservicios en el “**cluster**” será necesario realizar una petición **POST** al actuador “**bus-env**” y después ejecutar otra petición **POST** al actuador “**bus-refresh**”.
 - # Mediante httpie
 - **http POST http://localhost:9092/users-config-client/actuator/bus-env name=someKey value=someValue**
 - # Mediante cURL
 - **curl -X POST -H "Content-Type: application/json"
http://localhost:9092/users-config-client/actuator/bus-env -d
'{"name":"someKey", "value":"someOtherValue"}'**



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (s')

- **Práctica 21. Spring Cloud Config - Parte 2**
- Analiza la aplicación **21-Spring-Cloud-Config-Server**, **21-Hello-Spring-Cloud-Config-Client** y **21-Users-Spring-Cloud-Config-Client**.
- La práctica 21 – Spring Cloud Config – Parte 2 es continuación de la práctica previa.
- El servidor de configuración Spring Cloud Config Server no tiene cambio alguno.
- Se realizarán adecuaciones a la configuración y agregaciones de funcionalidad a los proyectos **21-Hello-Spring-Cloud-Config-Client** y **21-Users-Spring-Cloud-Config-Client**.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (t')

- Práctica 21. Spring Cloud Config - Parte 2
- Ingresar a la ruta: {tu-workspace}/21-Hello-Spring-Cloud-Config-Client
- Importar el proyecto 21-Hello-Spring-Cloud-Config-Client en STS.
- Agregar las dependencias **org.springframework.cloud:spring-cloud-starter-bus-amqp** y **org.springframework.boot:spring-boot-starter-actuator** en el archivo pom.xml.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (u')

- **Práctica 21. Spring Cloud Config - Parte 2**
- Agregar, sobre el archivo “**application.properties**”, la configuración de Spring Boot AMQP para habilitar la conectividad hacia el “**broker**” RabbitMQ definiendo **host=localhost**, **port=5672**, **username** y **password** con valor “**guest**” para ambos casos.
- De igual forma, sobre el archivo “**application.properties**”, agregar la exposición web de los actuadores “**bus-refresh**” y “**bus-env**”.

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (v')

- Práctica 21. Spring Cloud Config - Parte 2
- Sobre la clase principal **Application**, del paquete **com.consulting.mgt.springboot.practica21.hello.configclient**, misma que contiene la definición de **@RestController**, anotar la clase principal con **@RefreshScope** y, defina una nuevo propiedad “**String key**” la cual le sea injectado el valor de la propiedad “**someKey**” (mediante **@Value**) y, por último, defina un nuevo “**handler method**” que, a través del **path “/key”** devuelva el valor de la propiedad “**String key**”.
- Desde línea de comando compile y ejecute la aplicación habilitando el perfil “**dev**”.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (w')

- Práctica 21. Spring Cloud Config - Parte 2
- Ingresar a la ruta: {tu-workspace}/21-Users-Spring-Cloud-Config-Client
- Importar el proyecto 21-Users-Spring-Cloud-Config-Client en STS.
- Agregar las dependencias **org.springframework.boot:spring-boot-starter-web**,
org.springframework.cloud:spring-cloud-starter-bus-amqp y
org.springframework.boot:spring-boot-starter-actuator en el archivo pom.xml.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (x')

- **Práctica 21. Spring Cloud Config - Parte 2**
- Agregar, sobre el archivo “**application.properties**”, la configuración de Spring Boot AMQP para habilitar la conectividad hacia el “**broker**” RabbitMQ definiendo **host=localhost**, **port=5672**, **username** y **password** con valor “**guest**” para ambos casos.
- De igual forma, sobre el archivo “**application.properties**”, agregar la exposición web de los actuadores “**bus-refresh**” y “**bus-env**”.



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (y')

- **Práctica 21. Spring Cloud Config - Parte 2**
- Sobre la clase principal **Application**, del paquete **com.consulting.mgt.springboot.practica21.users.configclient**, misma que contiene la definición de **@Bean CommandLineRunner**, anotar la clase principal con **@RefreshScope** y definirla, también como un **@RestController**.
- Defina una nuevo propiedad “**String key**” la cual le sea injectado el valor de la propiedad “**someKey**” (mediante **@Value**) y, por último, defina un nuevo “**handler method**” que, a través del path “**/key**” devuelva el valor de la propiedad “**String key**”.
- Desde línea de comando compile y ejecute la aplicación habilitando el perfil “**dev**”.



+

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (z')

- **Práctica 21. Spring Cloud Config - Parte 2**
- Pruebe el aplicativo accediendo desde el navegador a las URLs:
 - <http://localhost:9091/hello-config-client/> y
 - <http://localhost:9092/users-config-client/>
- Analice la respuesta de cada uno de los endpoints anteriores.
- Realice cambios sobre las configuraciones de ambos microservicios sobre el perfil “**dev**” y súbalos al repositorio remoto.
- Al acceder nuevamente a los endpoints previos, ¿se visualizan los cambios generados en la configuración?

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (a’)

- Práctica 21. Spring Cloud Config - Parte 2
- Ejecute una petición **POST** al endpoint actuador “**bus-refresh**” de cualquiera de los dos microservicios. El bus se encargará de propagarlo a los demás microservicios que están escuchando el evento y refrescará la configuración de los Bean **@RefreshScope**.
 - # Mediante httpie
 - **http POST http://localhost:9092/users-config-client/actuator/bus-refresh**
 - # Mediante cURL
 - **curl -X POST http://localhost:9092/users-config-client/actuator/bus-refresh**



v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (b'')

- **Práctica 21. Spring Cloud Config - Parte 2**
- Pruebe nuevamente el aplicativo accediendo desde el navegador a las URLs:
 - <http://localhost:9091/hello-config-client/> y
 - <http://localhost:9092/users-config-client/>
- ¿Se visualizan los cambios en la configuración?

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (c'')

- Práctica 21. Spring Cloud Config - Parte 2
- Envíe a través del “bus” de eventos el par clave/valor **someKey=someValue** a través del endpoint del actuador “**bus-env**”, de ser necesario vuelva a refreshar los microservicios en el “cluster” mediante el actuador “**bus-refresh**”.
 - # Mediante httpie
 - **http POST http://localhost:9092/users-config-client/actuator/bus-env name=someKey value=someValue**
 - # Mediante cURL
 - **curl -X POST http://localhost:9092/users-config-client/actuator/bus-env -d '{"name":"someKey", "value":"someOtherValue"}'**



+

NETFLIX
OSS

v.iii Conf. externalizada con Spring Cloud Config y Spring Cloud Bus. (d’)

- Práctica 21. Spring Cloud Config - Parte 2
- Acceda desde el navegador a las URLs:
 - <http://localhost:9091/hello-config-client/key>
 - <http://localhost:9092/users-config-client/key>
- Al acceder a los endpoints previos, ¿Qué información despliega cada uno de ellos?
- Si ejecutamos nuevamente otra petición POST al actuador “bus-env” con clave/valor diferentes y, accedemos nuevamente a los endpoints previos, ¿Qué información despliega cada uno de los endpoints previos?
- ¡ Practique !



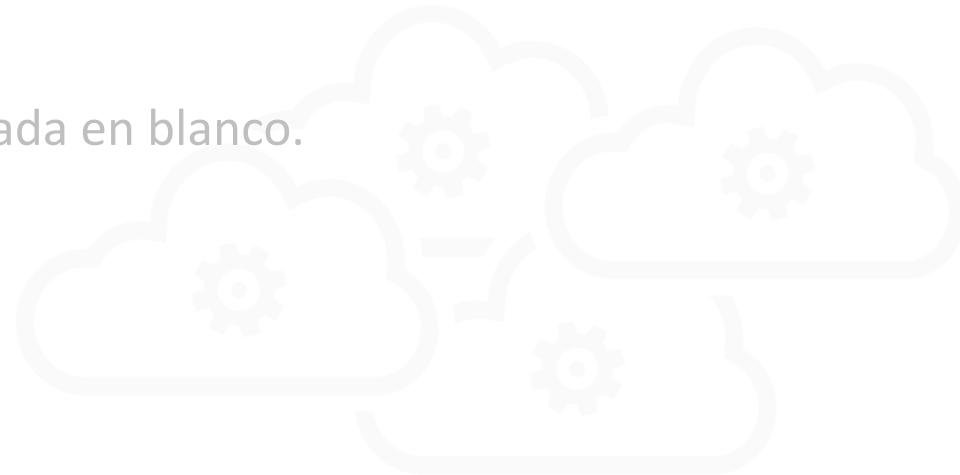
Resumen de la lección

v.iii Configuración externalizada con Spring Cloud Config y Spring Cloud Bus.

- Conocimos las funcionalidades y formas de configuración del servidor de configuración externalizada Spring Cloud Config Server.
- Comprendimos como debe de ser configurado el servidor y el cliente para implementar configuración externalizada mediante Spring Cloud Config Server.
- Implementamos aplicaciones de microservicios mediante configuración externalizada con Spring Cloud Config Server.
- Implementamos Spring Cloud Bus para "recargar" su configuración externalizada.



Esta página fue intencionalmente dejada en blanco.



Microservices



+

v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS

- v.i Twelve-Factor Apps.
- v.ii Spring Cloud y Spring Cloud Netflix OSS
- v.iii Configuración externalizada con Spring Cloud Config y Spring Cloud Bus.
- v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka.
- v.v Balanceo de carga del lado del cliente con Ribbon.
- v.vi Clientes REST declarativos con Feign.
- v.vii Implementación de corto-circuito con Hystrix.
- v.viii Visualización de corto-circuitos con Turbine.
- v.ix API Gateway con Spring Cloud Zuul.



+

NETFLIX
OSS

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka.

Microservices



Objetivos de la lección

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka.

- Conocer el servidor de "**service discovery**" y "**service registry**" de Spring Cloud Netflix Eureka.
- Comprender como debe configurarse el servidor Eureka en ambientes de pruebas y producción.
- Analizar por qué es necesario implementar un servidor de "**service discovery**" y "**service registry**" en arquitecturas de microservicios.
- Aprender a configurar las aplicaciones cliente para conectarse al servidor de Eureka.
- Implementar Spring Cloud Netflix Eureka para registrar y descubrir microservicios.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (a)

- ¿Qué es descubrimiento de servicios o “**service discovery**”?
- En arquitecturas orientadas a microservicios típicamente las aplicaciones resultan en un gran número de microservicios operativos.
- Éstos microservicios deben ser interconectados entre sí para realizar llamadas entre servicios.
- No utilizar un “**directorío**” para encontrar las coordenadas (URLs) de los servicios a encontrar (o descubrir) puede resultar en una configuración sumamente difícil para cada microservicio en cuestión.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (b)

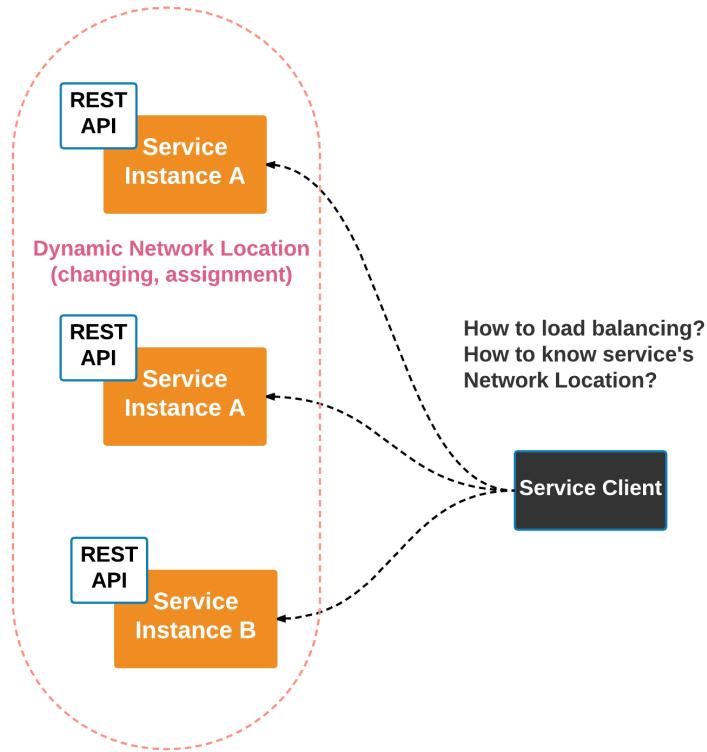
- ¿Qué es descubrimiento de servicios o “**service discovery**”?
- En una arquitectura de microservicios, los servicios son “**servidores**” o “**proveedores**” del servicio que implementan sin embargo, a su vez pueden ser “**clientes**” o “**consumidores**” de servicios que otros microservicios exponen.
- ¿Cómo puede saber el cliente de un servicio saber la ubicación/URI de la instancia del proveedor del servicio que requiere?
 - Múltiples hostnames en una arquitectura distribuida.
 - Múltiples IPs/puertos en los cuales los servicios pueden estar desplegados.
 - Múltiples instancias/replicas por servicio.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (c)

- ¿Qué es descubrimiento de servicios o “**service discovery**”?
- Un servicio de “**descubrimiento de servicios**” o “**service discovery**” es un componente fundamental en arquitecturas distribuidas que permite “**encontrar**” o “**descubrir**” la ubicación en la red de los servicios desplegados.
- Los servicios o microservicios pueden desplegarse en múltiples IPs, hostnames y puertos, lo que ocasiona que realizar el seguimiento de conocer cuál es la ubicación en la red de cada servicio sea difícil.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (d)

- ¿Qué es descubrimiento de servicios o “service discovery”?



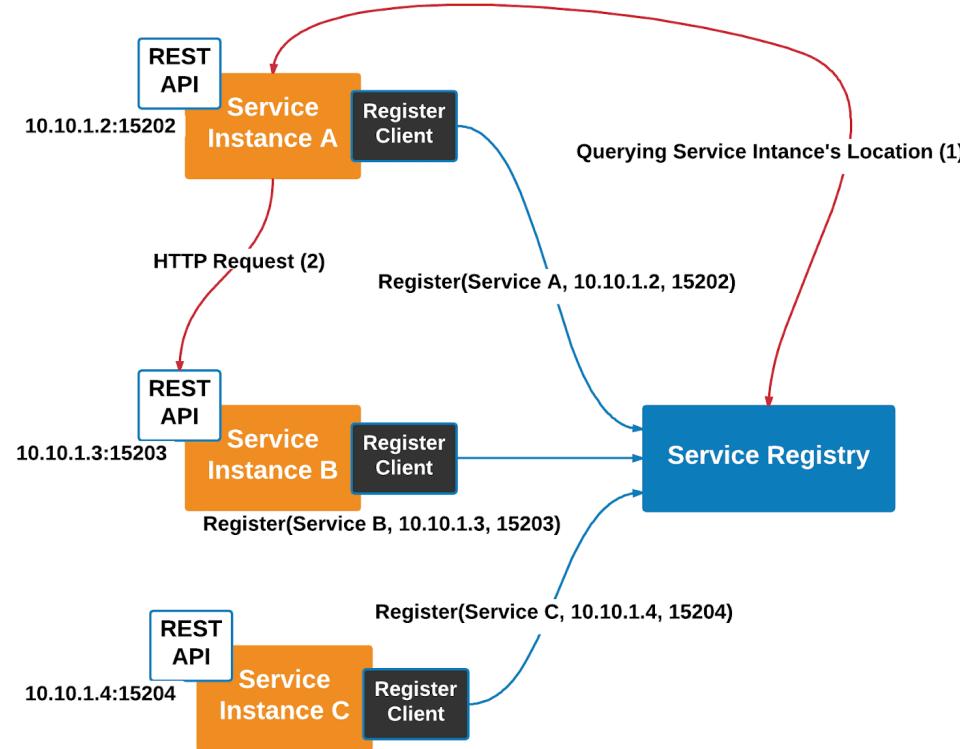


v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (e)

- ¿Qué es registro de servicios o “**service registry**”?
- Para que un aplicativo cliente pueda encontrar un servicio en el componente de “**descubrimiento de servicios**” o “**service discovery**”, cada proveedor de servicios debe auto-registrarse en el servicio de descubrimiento; lo cuál genera el servicio de “**registro de servicios**” o “**service registry**”.
- Existen múltiples implementaciones de “**service discovery / service registry**” como: Eureka, Consul, Zookeeper, Etcd, entre otros.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (f)

- ¿Qué es registro de servicios o “service registry”?





v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (g)

- Spring Cloud Eureka.
- Componente de registro y descubrimiento de servicios parte del stack tecnológico de Spring Cloud Netflix OSS.
- Eureka provee el servicio de registro y descubrimiento o “**lookup**” de servicios.
 - Permite mantener alta disponibilidad del servicio mediante ejecutar múltiples instancias de Eureka.
 - Permite replicar el estado de los servicios registrados entre las múltiples instancias de Eureka (cluster).

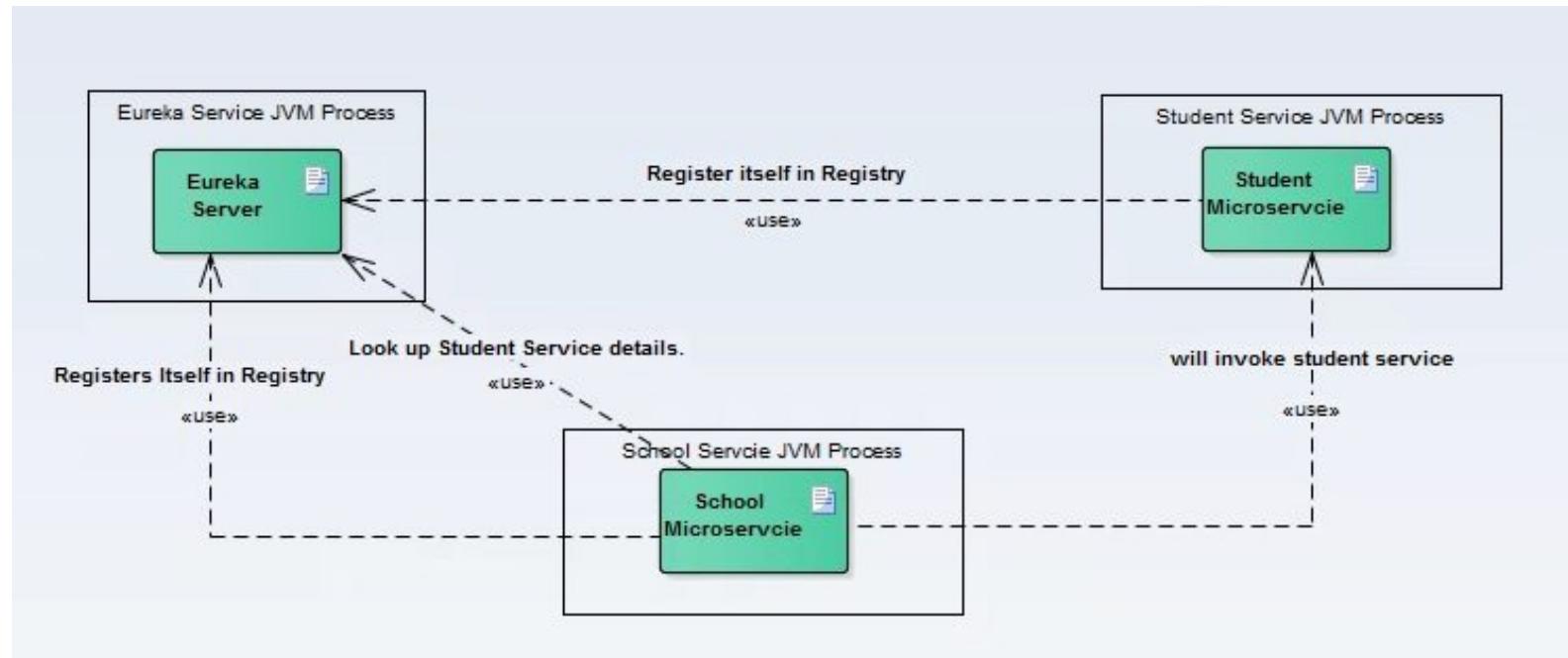


v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (h)

- Spring Cloud Eureka.
- Los servicios/microservicios “**servidores**” se registran en Eureka para poder ser “**descubiertos**” por otros servicios “**consumidores**” o “**clientes**” de los servicios que exponen.
 - Envian metadatos como hostname, puerto, URI para verificación de “**healt check**”, entre otros parámetros útiles para su monitoreo. Por convención utilizan las métricas y actuadores de Spring Boot Actuator.
- Los servicios envian “**hearthbeats**” al servidor Eureka para verificar su “**vitalidad**” o “**liveness**”.

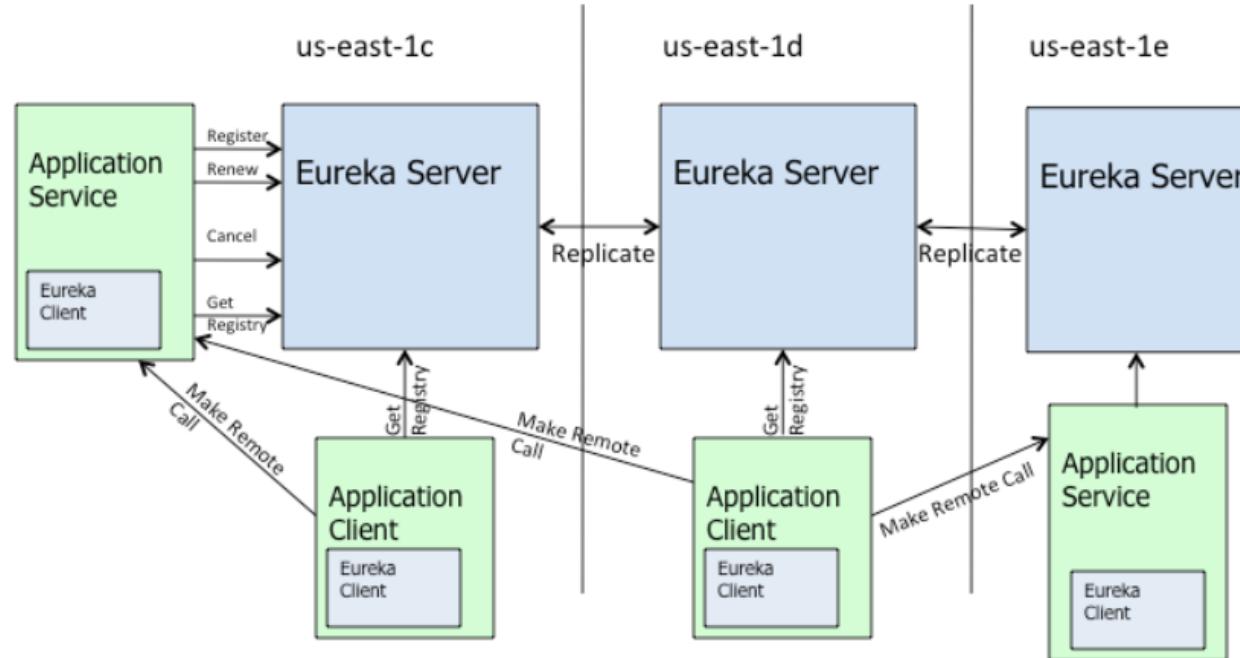
v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (i)

- Spring Cloud Eureka.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (j)

- Spring Cloud Eureka.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (k)

- Spring Cloud Eureka.
- “**Battle tested**” y actualmente utilizado en producción por Netflix.



NETFLIX



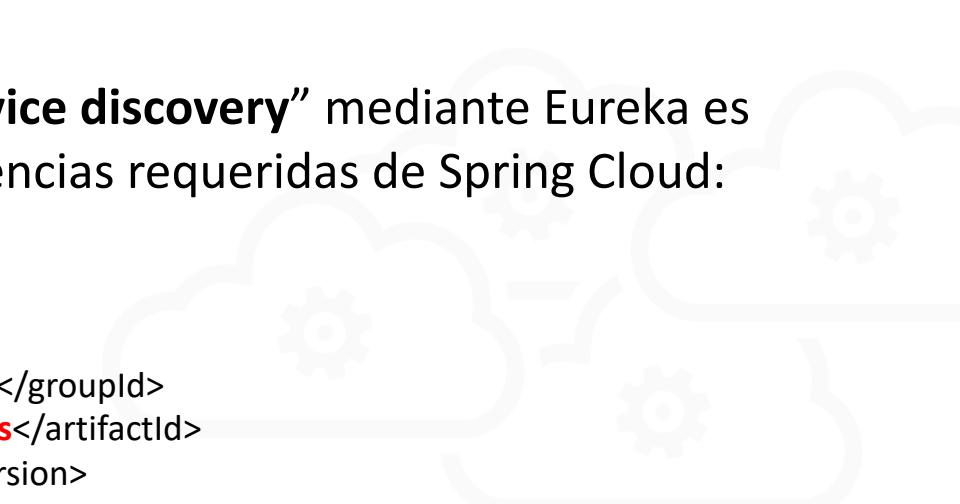
+

NETFLIX
OSS

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (I)

- Spring Cloud Eureka Server.
- Para generar un servidor “**service discovery**” mediante Eureka es necesario agregar las dependencias requeridas de Spring Cloud:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```



```
<properties>
  <java.version>1.8</java.version>
  <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
</properties>
```



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (m)

- Spring Cloud Eureka Server.
- Ningun proyecto basado en Spring Cloud se recomiendo configurar sin el apoyo de Spring Boot, por tanto el <parent> del pom.xml sigue siendo spring-boot-starter-parent.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (n)

- Spring Cloud Eureka Server.
- Para habilitar un servidor Spring Cloud Eureka, agregar la dependencia:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-netflix-eureka-server</artifactId>
    </dependency>
    ...
</dependencies>
```



+

NETFLIX
OSS

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (ñ)

- Spring Cloud Eureka Server.
- Agregar la anotación **@EnableEurekaServer** a la clase principal del proyecto.

@EnableEurekaServer

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (o)

- Alta disponibilidad en Spring Cloud Eureka Server.
- El servicio de “**service registry**” de Eureka no persiste el estado o registro de los servicios auto-registrados.
 - Todos los servicios auto-registrados deben enviar “**heartbeats**” para mantener vivo y al día su registro; por tanto dicho registro puede mantenerse en memoria.
- Los clientes que “**descubren**” los servicios auto-registrados sobre el servicio de “**service discovery**” de Eureka mantienen el registro en memoria para evitar obtener el registro en cada petición que deban ejecutar hacia algún servicio.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (p)

- Alta disponibilidad en Spring Cloud Eureka Server.
- Por default, cada instancia de Eureka es un servidor y también un cliente que requiere al menos una URL, de un servidor Eureka, para localizar a un “**peer**” (par o nodo “igual”).
- Es posible no definir una URL de un servidor Eureka par (“**peer**”), sin embargo, la consola enviará muchos errores debido a que Eureka está diseñado para desplegarse en alta disponibilidad, auto-registrándose y sincronizando el estado de los servicios auto-registrados con los demás “**peers**” en el cluster.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (q)

- Spring Cloud Eureka Server con única instancia (standalone mode).
- Para efectos de pruebas, no será necesario desplegar más de una instancia del servidor Eureka (sin “peers”); por tanto se sugiere la siguiente configuración:
 - # Eureka properties
 - # Eureka default port 8761
 - **server.port=9099**
 - **eureka.client.register-with-eureka=false**
 - **eureka.client.fetch-registry=false**
 - **eureka.client.service-url.defaultZone=http://localhost:9099/eureka**

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (r)

- Spring Cloud Eureka Server con única instancia (standalone mode).

```
# Eureka YAML properties
# Eureka default port 8761
server:
  port: 9099
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
  service-url:
    defaultZone: http://localhost:9099/eureka
```

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (s)

- Spring Cloud Eureka Server en réplica (peer awareness mode).
- En ambientes productivos o de pruebas de carga y rendimiento es preferible desplegar el servidor Eureka en alta disponibilidad, es decir, en replica; por tanto se sugiere la siguiente configuración:

```
---  
spring:  
  profiles: peer1  
eureka:  
  instance:  
    hostname: peer1  
  client:  
    serviceUrl:  
      defaultZone: http://peer2/eureka/
```

Same YAML File

```
---  
spring:  
  profiles: peer2  
eureka:  
  instance:  
    hostname: peer2  
  client:  
    serviceUrl:  
      defaultZone: http://peer1/eureka/
```



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (t)

- Spring Cloud Eureka Client.
- Para generar un cliente que se auto-registre en el servidor de “**service registry**” (Eureka Server) es necesario agregar las dependencias requeridas de Spring Cloud:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

```
          <properties>
            <java.version>1.8</java.version>
            <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
          </properties>
```

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (u)

- Spring Cloud Eureka Client.
- Ningun proyecto basado en Spring Cloud se recomiendo configurar sin el apoyo de Spring Boot, por tanto el <parent> del pom.xml sigue siendo spring-boot-starter-parent.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (v)

- Spring Cloud Eureka Client.
- Para habilitar un servidor Spring Cloud Eureka, agregar la dependencia:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    ...
</dependencies>
```



+

NETFLIX
OSS

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (w)

- Spring Cloud Eureka Client.
- Agregar la anotación **@EnableDiscoveryClient** a la clase principal del proyecto.

```
@EnableDiscoveryClient
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (x)

- Spring Cloud Eureka Client.
- Un aplicativo que se auto-registra al servidor Eureka debe registrar en su configuración, al menos una URL de conexión a alguno de los nodos de la replica (cluster) de Eureka mediante la propiedad `eureka.client.service-url.defaultZone=http://localhost:9099/eureka`.
- Cada instancia, aplicativo o microservicio que se auto-registra en el servidor Eureka debe enviar “**heartbeats**” para mantener vivo y al día su registro.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (y)

- Spring Cloud Eureka Client.
- Por default Spring Cloud Eureka utiliza Spring Boot Actuator para exponer el actuador “**health**”.
- Es necesario habilitar el actuador “**health**” y exponer los “**actuators**” vía web.

```
# Spring Cloud Eureka Client
management:
endpoint:
  health:
    enabled: true
  shutdown:
    enabled: true
endpoints:
  web:
    exposure: expose
    include: '*'
```



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (z)

- **@EnableDiscoveryClient**
- La anotación **@EnableDiscoveryClient** automáticamente registra el aplicativo/microservicio cliente contra el servicio de “**service registry**” del servidor Eureka.
- Registra la aplicación identificandola mediante nombre, hostname y puerto.
 - Los valores los obtiene a través del objeto **Environment**, es decir, a través de sus propiedades los cuales pueden ser configurados mediante archivo **“application.properties”** o **YAML**.
 - Es posible utilizar Spring Cloud Config para asignar los valores de las propiedades de su configuración.



+

NETFLIX
OSS

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (a')

- Spring Cloud Eureka Client.
- Inyectar la interface **DiscoveryClient**, para “descubrir” los servicios (instancias) registrados en el servidor Eureka, mediante su nombre.

```
@Autowired
```

```
private DiscoveryClient discoveryClient;
```

```
public URI serviceUrl(String appName) {  
    List<ServiceInstance> list = discoveryClient.getInstances(appName);  
    if (list != null && list.size() > 0)  
        return list.get(0).getUri();  
    return null;  
}
```

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (b')

- Spring Cloud Eureka Client.
- Es posible utilizar la anotación **@EnableEurekaClient** en lugar de **@EnableDiscoveryClient** sobre la clase principal del microservicio cliente.
 - Es posible utilizar múltiples proveedores de servicios de “**service registry**” y “**service discovery**” como **Eureka**, **Consul**, **Zookeeper**, **Etcd**, entre otros. Para desacoplar la implementación del servicio de descubrimiento y registro de la implementación del cliente, es necesario utilizar la anotación **@EnableDiscoveryClient**.
 - Para el caso de utilizar Eureka como servicio de “**service registry**” y “**service discovery**” es posible utilizar la anotación **@EnableEurekaClient** o la anotación **@EnableDiscoveryClient** indistintamente.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (c')

- Spring Cloud Eureka Client.
- La anotación **@EnableEurekaClient** esta acoplada a la implementación de Eureka como servicio de registro y descubrimiento de servicios y pertenece al módulo **spring-cloud-netflix**.
- La anotación **@EnableDiscoveryClient** esta desacoplada de cualquier implementación de servicio de registro y descubrimiento de servicios y pertenece al módulo **spring-cloud-commons**.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (d')

- Spring Cloud Eureka Client.
- De forma análoga al uso de las anotaciones **@EnableEurekaClient** o **@EnableDiscoveryClient**, es posible utilizar las clases **DiscoveryClient** o **EurekaClient** como implementación cliente para obtener las instancias de los microservicios registrados en el servicio de registro y descubrimiento de servicios.
- El uso de la clase **EurekaClient** funciona únicamente si el servidor del servicio de registro y descubrimiento de servicios es **Eureka**.
 - Se recomienda utilizar **DiscoveryClient** para desacoplar la implementación del servicio de “service registry” y “service discovery” del lado del cliente.



+

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (e')

- Spring Cloud Eureka Client.
- Inyectar la interface **EurekaClient**, para “descubrir” los servicios (instancias) registrados en el servidor Eureka, mediante su nombre.

```
@Autowired
```

```
private EurekaClient eurekaClient;
```

```
public URI serviceUrl(String appName) {
```

```
    InstanceInfo instance = eurekaClient.getNextServerFromEureka(  
        appName, false);
```

```
    return new URI(instance.getHomePageUrl());
```

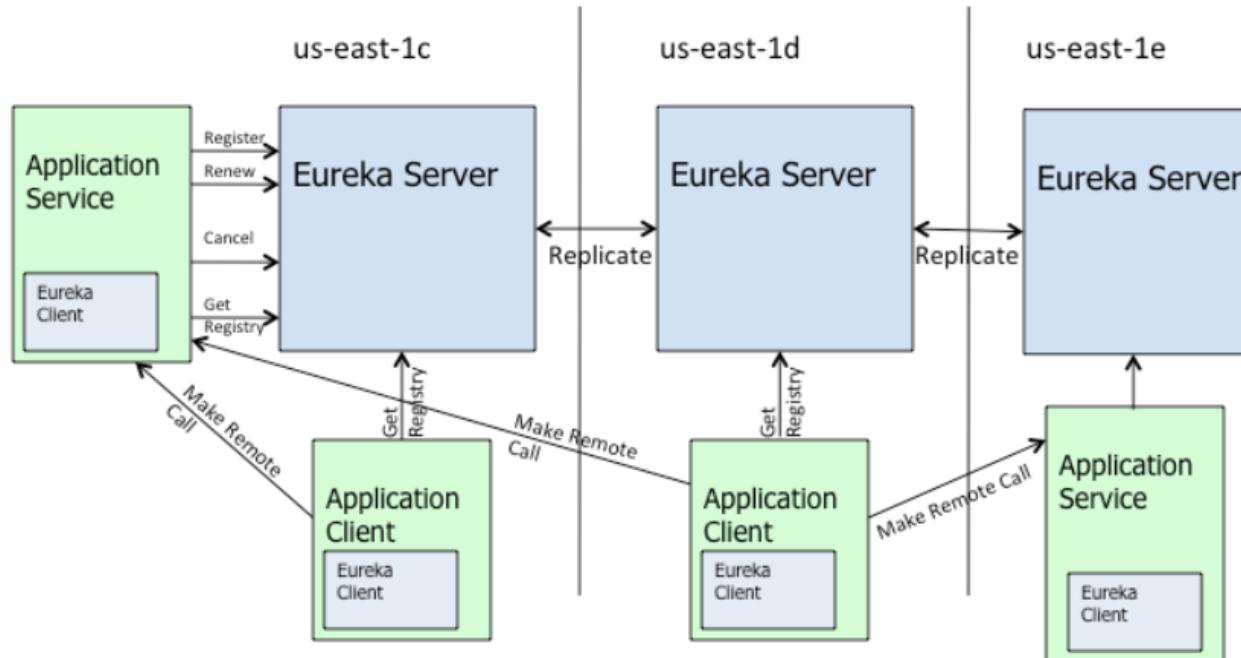
```
}
```

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (f')

- Consideraciones en ambientes productivos para Spring Cloud Eureka.
- El servidor **Eureka** fue diseñado para usar en replica (“**peer awareness mode**”).
 - En caso de utilizar el “**standalone mode**” (sin replica), **Eureka** advierte, mediante logs, que se está ejecutando sin “**peers**”.
- **Eureka** no persiste registro de servicios, los mantiene en memoria.
- En producción, típicamente, se despliegan múltiples instancias del servidor **Eureka** en diferentes **zonas/regiones** de despliegue en la nube.
 - Cada instancia de Eureka se conecta con las demás como pares (“**peers**”).

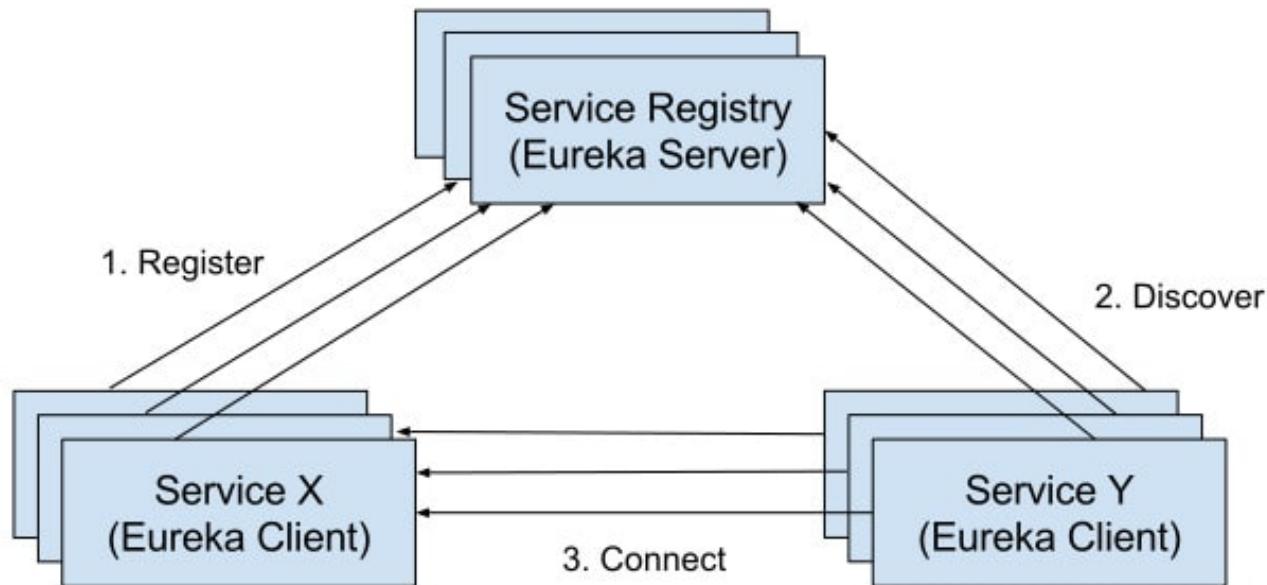
v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (g')

- Consideraciones en ambientes productivos para Spring Cloud Eureka.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (h')

- Spring Cloud Eureka.





v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (i')

- **Práctica 22. Spring Cloud Eureka**
- Analiza la aplicación **22-Spring-Cloud-Eureka-Server** y **22-Spring-Cloud-Eureka-Client**.
- Ingresar a la ruta: **{tu-workspace}/22-Spring-Cloud-Eureka-Server**
- Importar el proyecto **22-Spring-Cloud-Eureka-Server** en STS.
- Agregar la dependencia **org.springframework.cloud:spring-cloud-netflix-eureka-server** y **org.springframework.cloud:spring-cloud-starter-netflix-eureka-client** en el archivo pom.xml. Revisar el apartado **<dependencyManagement>**.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (j')

- **Práctica 22. Spring Cloud Eureka**
- Sobre la clase **Application**, clase principal del proyecto del paquete **com.consulting.mgt.springboot.practica22.eurekaserver**, habilitar la auto- configuración de Spring Cloud Eureka Server mediante la anotación **@EnableEurekaServer**.
- Revisa el archivo “**application.properties**” y analiza los perfiles “**single**”, “**peer-1**” y “**peer-2**”.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (k')

- **Práctica 22. Spring Cloud Eureka**
- Sobre el archivo “**application-single.properties**”, el cuál define la configuración del perfil “**single**”, define las siguientes propiedades:
 - Puerto del servidor embebido Tomcat **9099**.
 - Evitar que la instancia del servidor Eureka se auto-registre consigo misma, debido a que el perfil “**single**” levantará una única instancia del servidor Eureka (**register-with-eureka=false**).
 - Evitar que la instancia del servidor Eureka recupere el registro de servicios de otros “**peers**”, debido a que el perfil “**single**” levantará una única instancia del servidor Eureka (**fetch-registry=false**).
 - Registrar la URL de la zona default, de la instancia del servidor Eureka, al endpoint “**/eureka**” de la instancia misma. Registrar la URL de la zona default consigo misma.



+

NETFLIX
OSS

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (I')

- **Práctica 22. Spring Cloud Eureka**
- Configura el archivo “**hosts**” (dependiendo del sistema operativo utilizado) de la siguiente manera:
 - 127.0.0.1 eureka-host1
 - 127.0.0.1 eureka-host2
- Ubicación de archivo “**hosts**”.
 - Windows 10 / 8: **c:\Windows\System32\Drivers\etc\hosts**
 - Linux: **sudo nano /etc/hosts**
 - MacOS X 10.6+: **sudo nano /private/etc/hosts**
- Analiza las propiedades “**customizadas**” actualmente definidas sobre el archivo “**application.yml**”.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (m')

- **Práctica 22. Spring Cloud Eureka**
- En el archivo “**application.yml**” se configuran los perfiles “**peer-1**” y “**peer-2**”, correspondientes a dos réplicas del servidor Eureka (“**peer awareness mode**”).
- Sobre el archivo “**application.yml**” define el perfil “**peer-1**” con las siguientes configuraciones (en formato YAML) (a):
 - La propiedad **eureka.instance.hostname** con el valor del “**host peer-1**” definido en las propiedades “**customizadas**” sobre el archivo “**application.yml**”.
 - La propiedad **eureka.instance.appname** con el valor de la propiedad “**spring.application.name**”, definido en las propiedades “**customizadas**” sobre el archivo “**application.yml**”, concatenando con la cadena “**-in-cluster**”.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (n')

- **Práctica 22. Spring Cloud Eureka**
- Sobre el archivo “**application.yml**” define el perfil “**peer-1**” con las siguientes configuraciones (en formato YAML) (b):
 - La propiedad **eureka.client.service-url.defaultZone** con el valor del endpoint “/eureka” del “host, puerto y servlet-context del peer-2 de Eureka” definido en las propiedades “**customizadas**” sobre el archivo “**application.yml**”.
 - Habilitar que la instancia del servidor Eureka “**peer-1**” se auto-registre con los demás nodos del ”**cluster**” de servidores Eureka (“**peers**”), debido a que el perfil “**peer-1**” tendrá como “**peer**” la instancia “**peer-2**” (**eureka.client.register-with-eureka=true**).



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (ñ')

- **Práctica 22. Spring Cloud Eureka**
- Sobre el archivo “**application.yml**” define el perfil “**peer-1**” con las siguientes configuraciones (en formato YAML) (c):
 - Habilitar que la instancia del servidor Eureka “**peer-1**” recupere el registro de servicios de los demás nodos del “**cluster**” de servidores Eureka (“**peers**”), debido a que el perfil “**peer-1**” tendrá como “**peer**” la instancia “**peer-2**” (**eureka.client.fetch-registry=true**).
 - Define el puerto donde el servidor Tomcat embebido atenderá peticiones al “**puerto del peer-1**”.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (o')

- **Práctica 22. Spring Cloud Eureka**
- De forma análoga, sobre el archivo “**application.yml**” define el perfil “**peer-2**” con las mismas configuraciones definidas para el perfil “**peer-1**” con la diferencia de apuntar a los “**hosts**” y puertos correspondientes para la configuración de la instancia “**peer-2**” (en formato YAML):
 - La URL de la zona default a donde la instancia “**peer-2**” apuntará debe ser hacia la instancia “**peer-1**”, debido a que la instancia “**peer-1**” será la réplica de la instancia “**peer-2**”.
- La resolución de URLs de réplicas de instancias del servidor Eureka desplegadas sobre el mismo “**host físico**” (misma máquina) no funciona mediante IP, es forzoso utilizar distintos “**hostnames**” (eureka-host1 y eureka-host2).



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (p')

- Práctica 22. Spring Cloud Eureka
- Ingresar a la ruta: {tu-workspace}/22-Spring-Cloud-Eureka-Client
- Importar el proyecto 22-Spring-Cloud-Eureka-Client en STS.
- Agregar la dependencia **org.springframework.cloud:spring-cloud-starter-netflix-eureka-client** en el archivo pom.xml. Revisar el apartado <dependencyManagement>.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (q')

- **Práctica 22. Spring Cloud Eureka**
- Sobre la clase **Application**, clase principal del proyecto del paquete **com.consulting.mgt.springboot.practica22.eurekaclient**, habilitar la auto-configuration de Spring Cloud Eureka Client mediante la anotación **@EnableDiscoveryClient**.
 - Podrá utilizarse indistintamente la anotación **@EnableEurekaClient**.
- Analiza la clase **HelloRestController**, del paquete **com.consulting.mgt.springboot.practica22.eurekaclient.restcontroller**, es requerida la inyección de los beans **DiscoveryClientHelper** y **EurekaClientHelper**.
 - Elimina el atributo “**required=false**” de la inyección de éstos beans.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (r')

- Práctica 22. Spring Cloud Eureka
- Implementa la clase **DiscoveryClientHelper**, del paquete **com.consulting.mgt.springboot.practica22.eurekaclient.helper**, para obtener las instancias registradas sobre el servidor Eureka.
 - Inyecta la clase **DiscoveryClient**.
 - La clase **DiscoveryClient** no implementa balanceo de carga. Una vez obtenidas las instancias, utiliza un algoritmo “**random**” para obtener la URI de una de las instancias registradas.



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (s')

- **Práctica 22. Spring Cloud Eureka**
- Implementa la clase **EurekaClientHelper**, del paquete **com.consulting.mgt.springboot.practica22.eurekaclient.helper**, para obtener las instancias registradas sobre el servidor **Eureka**.
 - Inyecta la clase **EurekaClient**.
 - La clase **EurekaClient** implementa balanceo de carga mediante un algoritmo simple de tipo “**round-robin**”. Una vez obtenida la instancia “**siguiente**” del algoritmo “**round-robin**”, obtén su URI.
 - También es posible obtener todas las instancias registradas.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (t')

- **Práctica 22. Spring Cloud Eureka**
- Analiza la configuración “default” sobre el archivo “**application.yml**”.
- Define 2 perfiles para ejecutar la aplicación cliente (a):
 - Define el perfil “**client-in-single-eureka**” donde se definan las siguientes propiedades:
 - **eureka.client.service-url.defaultZone** la cual defina el valor de la URL del servidor Eureka ejecutándose en el perfil “**single**”.
 - **eureka.instance.instanceId** la cual defina un identificador único para la instancia registrada. Por default Spring Cloud Eureka utiliza el formato **hostname:application-name:port**.
 - **server.port** la cual defina la propiedad **\${PORT:\${SERVER_PORT:0}}**

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (u')

- **Práctica 22. Spring Cloud Eureka**
- Define 2 perfiles para ejecutar la aplicación cliente (b):
 - Define el perfil “**client-in-eureka-cluster**” donde se definan las siguientes propiedades:
 - **eureka.client.service-url.defaultZone** la cual defina el valor de las URLs de los servidores Eureka ejecutandose en los perfiles “**peer-1**” y “**peer-2**”.
 - **eureka.instance.instanceId** la cual defina un identificador único para la instancia registrada. Por default Spring Cloud Eureka utiliza el formato **hostname:application-name:port**.
 - **server.port** la cual defina la propiedad **\${PORT:\${SERVER_PORT:0}}**



v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (v')

- **Práctica 22. Spring Cloud Eureka**
- Ejecuta 3 servidores Eureka.
 - Dos en replica, mediante el perfil “**peer-1**” y “**peer-2**”.
 - Uno sin replica, mediante el perfil “**single**”.
 - Utiliza el comando:
mvn spring-boot:run -Dspring-boot.run.profiles=<perfil>
- Verifica en que puertos se están ejecutando los servidores Eureka y, en el navegador, abre la URL principal del proyecto:
 - <http://eureka-host1:9091/my-eureka-server/>
 - <http://eureka-host2:9092/my-eureka-server/>
 - <http://localhost:9099/my-eureka-server/>
- Puedes utilizar la opción: **-Dspring-boot.run.arguments=--server.port=55991** para definir el puerto desde línea de comandos.

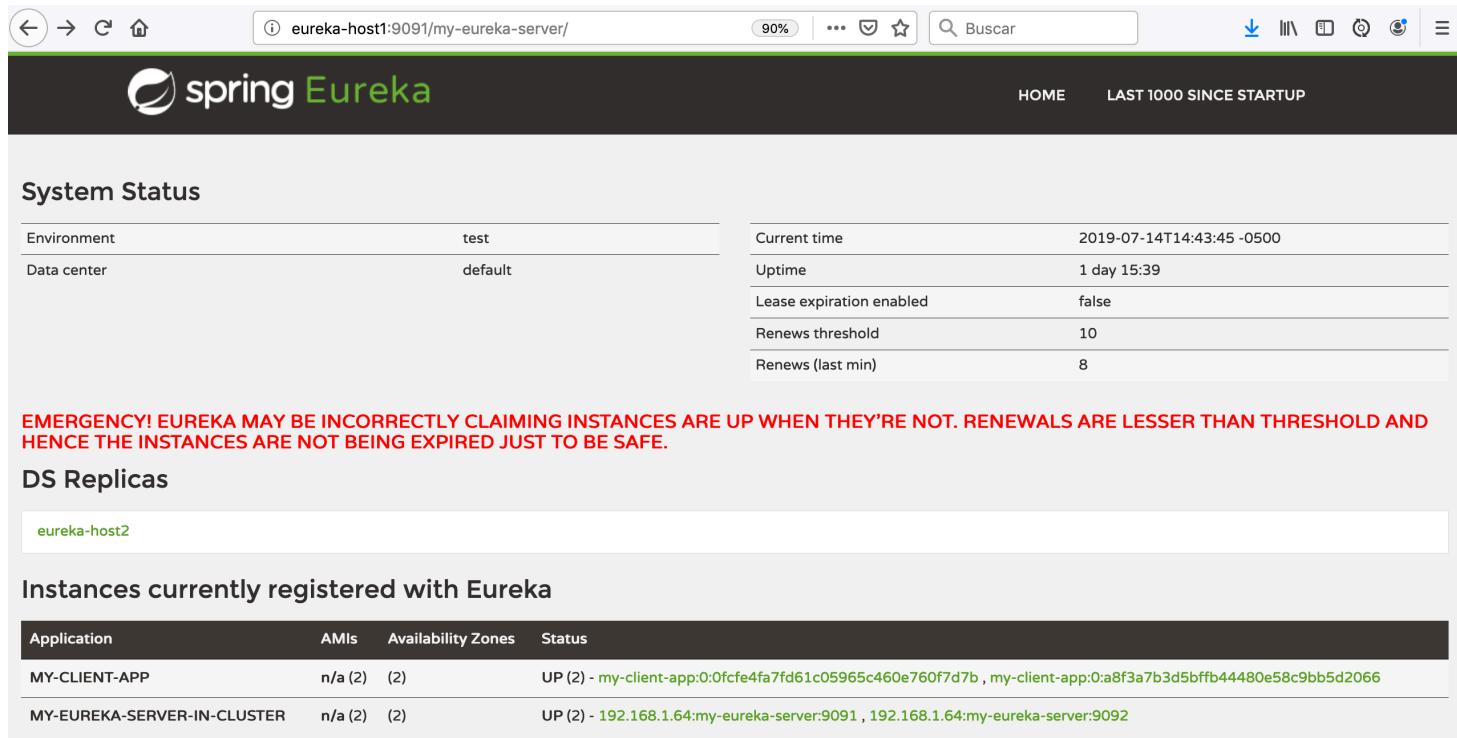


v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (w')

- **Práctica 22. Spring Cloud Eureka**
- Ejecuta 3 instancias del microservicio cliente.
 - Dos en replica, mediante el perfil “**client-in-eureka-cluster**”.
 - Uno sin replica, mediante el perfil “**client-in-single-eureka**”.
 - Utiliza el comando:
mvn spring-boot:run -Dspring-boot.run.profiles=<perfil>
- Verifica en que puertos se están ejecutando los microservicios clientes. Puedes verificarlo en consola o accediendo al “**dashboard**” de los servidores Eureka correspondientes.
 - <http://eureka-host1:9091/my-eureka-server/>
 - <http://eureka-host2:9092/my-eureka-server/>
 - <http://localhost:9099/my-eureka-server/>
- No se sugiere especificar puerto en los microservicios cliente.

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (x')

- Práctica 22. Spring Cloud Eureka



The screenshot shows the Spring Cloud Eureka dashboard at `eureka-host1:9091/my-eureka-server/`. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP.

System Status:

Environment	test	Current time	2019-07-14T14:43:45 -0500
Data center	default	Uptime	1 day 15:39
		Lease expiration enabled	false
		Renews threshold	10
		Renews (last min)	8

DS Replicas:

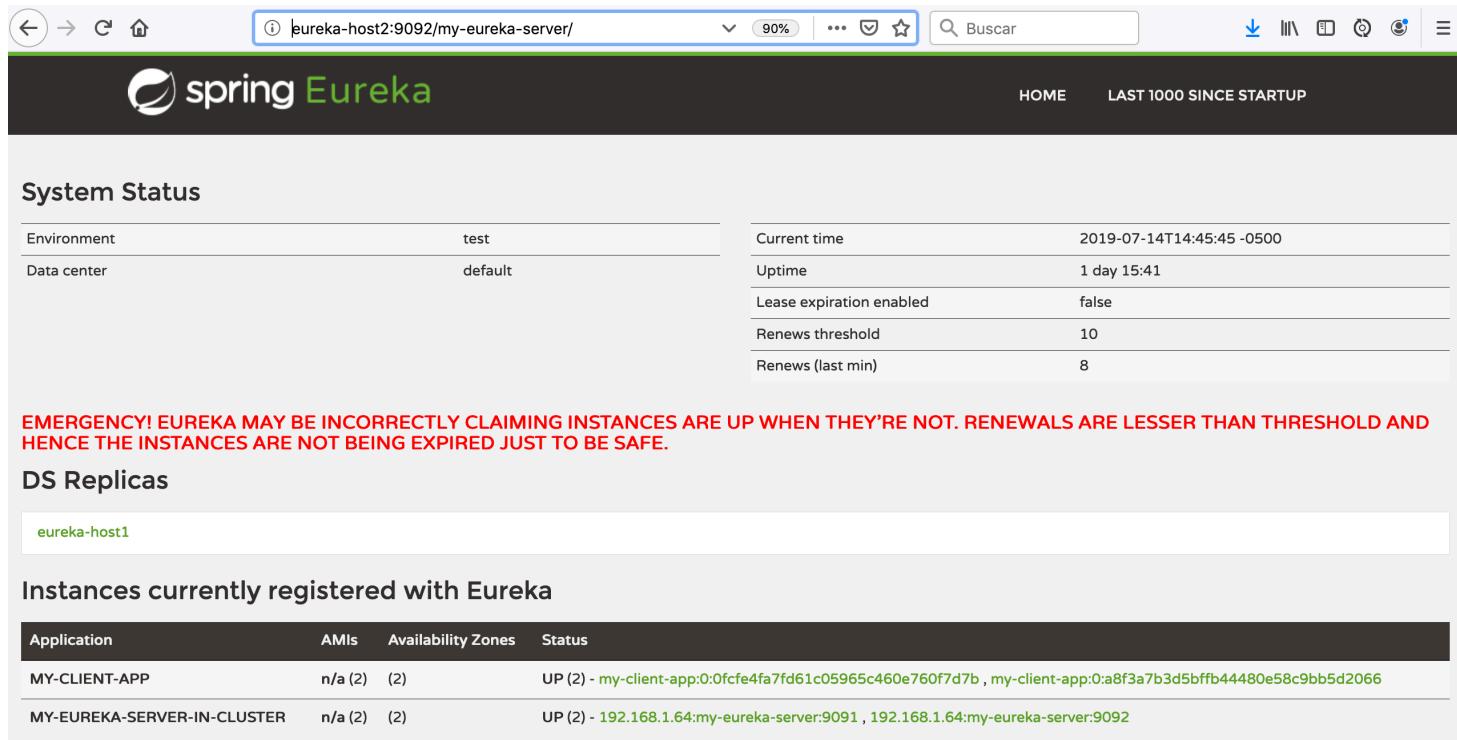
eureka-host2

Instances currently registered with Eureka:

Application	AMIs	Availability Zones	Status
MY-CLIENT-APP	n/a (2)	(2)	UP (2) - my-client-app:0:0fcfe4fa7fd61c05965c460e760f7d7b , my-client-app:0:a8f3a7b3d5bffb44480e58c9bb5d2066
MY-EUREKA-SERVER-IN-CLUSTER	n/a (2)	(2)	UP (2) - 192.168.1.64:my-eureka-server:9091 , 192.168.1.64:my-eureka-server:9092

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (y')

- Práctica 22. Spring Cloud Eureka



The screenshot shows the Spring Cloud Eureka dashboard at eureka-host2:9092/my-eureka-server/. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP.

System Status:

Environment	test	Current time	2019-07-14T14:45:45 -0500
Data center	default	Uptime	1 day 15:41
		Lease expiration enabled	false
		Renews threshold	10
		Renews (last min)	8

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas:

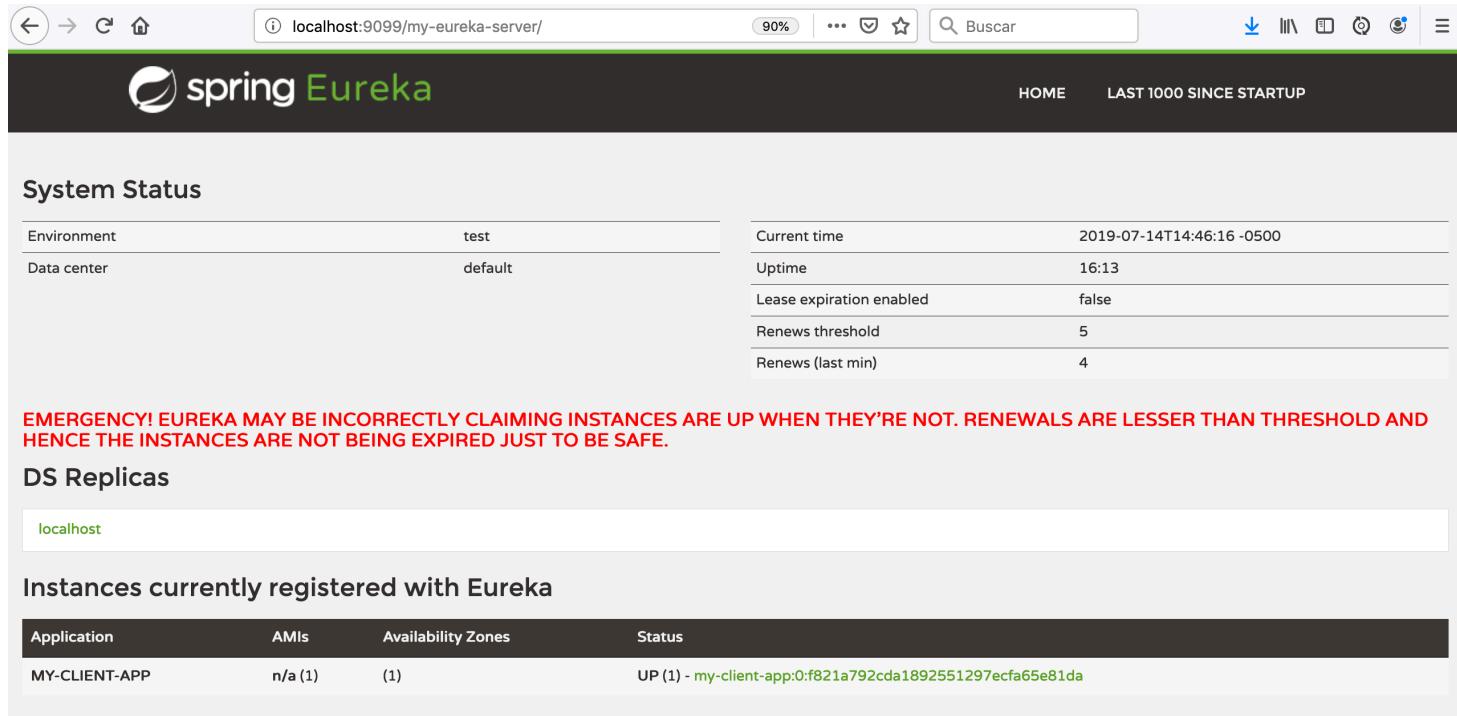
eureka-host1

Instances currently registered with Eureka:

Application	AMIs	Availability Zones	Status
MY-CLIENT-APP	n/a (2)	(2)	UP (2) - my-client-app:0:0fcfe4fa7fd61c05965c460e760f7d7b , my-client-app:0:a8f3a7b3d5bffb44480e58c9bb5d2066
MY-EUREKA-SERVER-IN-CLUSTER	n/a (2)	(2)	UP (2) - 192.168.1.64:my-eureka-server:9091 , 192.168.1.64:my-eureka-server:9092

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (z')

- Práctica 22. Spring Cloud Eureka



The screenshot shows the Spring Cloud Eureka UI at `localhost:9099/my-eureka-server/`. The top navigation bar includes links for Home, Help, Logout, and a search bar. The main content area has a dark header with the Spring Eureka logo and navigation links for Home and Last 1000 since startup.

System Status

Environment	test
Data center	default

Current time	2019-07-14T14:46:16 -0500
Uptime	16:13
Lease expiration enabled	false
Renews threshold	5
Renews (last min)	4

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

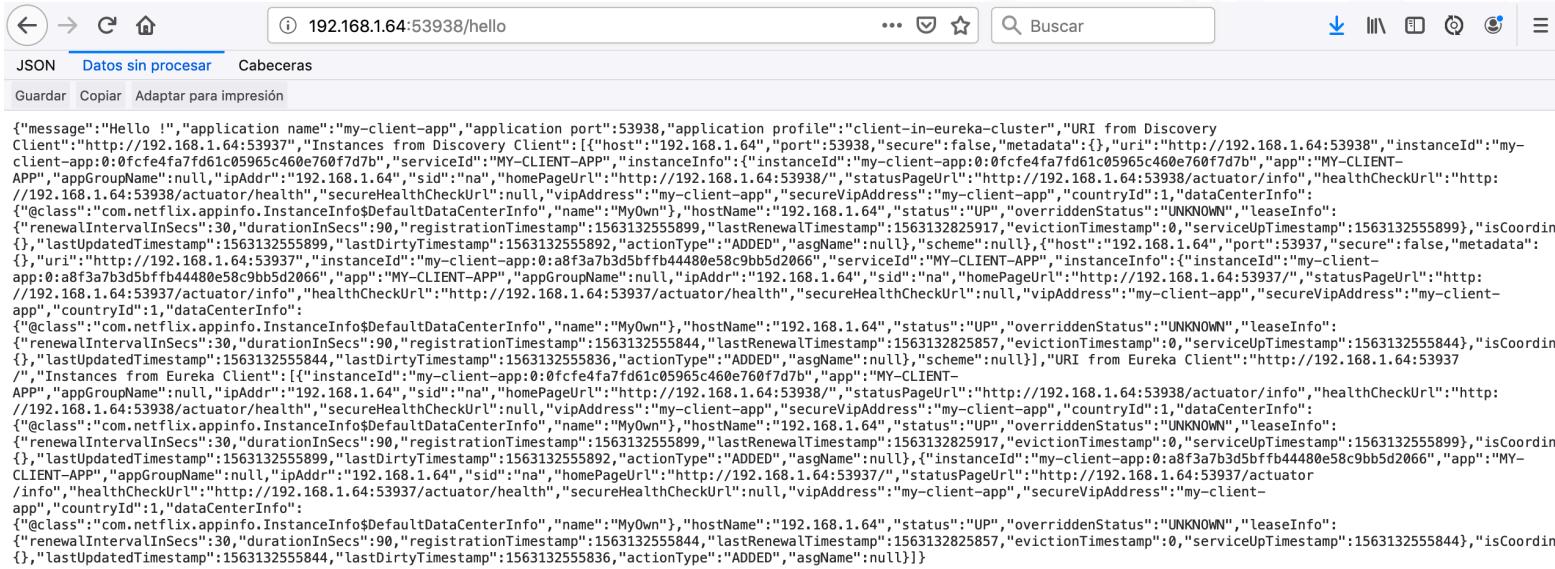
localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MY-CLIENT-APP	n/a (1)	(1)	UP (1) - my-client-app:0:f821a792cda1892551297ecfa65e81da

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka. (a")

- **Práctica 22. Spring Cloud Eureka**
- Accede a cada uno de los microservicios cliente y ejecuta el endpoint “/hello”.
 - ¿Cuál es el resultado de la ejecución?



```

{
  "message": "Hello !",
  "application_name": "my-client-app",
  "application_port": 53938,
  "application_profile": "client-in-eureka-cluster",
  "URI from Discovery Client": "http://192.168.1.64:53937",
  "Instances from Discovery Client": [
    {
      "host": "192.168.1.64",
      "port": 53938,
      "secure": false,
      "metadata": {},
      "uri": "http://192.168.1.64:53938",
      "instanceId": "my-client-app:0:0fcf4fa7fd61c05965c460e760f7d7b",
      "serviceId": "MY-CLIENT-APP",
      "instanceInfo": {
        "instanceId": "my-client-app:0:0fcf4fa7fd61c05965c460e760f7d7b",
        "app": "MY-CLIENT-APP",
        "appGroupName": null,
        "ipAddr": "192.168.1.64",
        "sid": "na",
        "homePageUrl": "http://192.168.1.64:53938",
        "statusPageUrl": "http://192.168.1.64:53938/actuator/info",
        "healthCheckUrl": "http://192.168.1.64:53938/actuator/health",
        "secureHealthCheckUrl": "http://192.168.1.64:53938/actuator/health",
        "vipAddress": "my-client-app",
        "countryId": 1,
        "dataCenterInfo": {
          "eClass": "com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo",
          "name": "MyOwn",
          "hostName": "192.168.1.64",
          "status": "UP",
          "overriddenStatus": "UNKNOWN",
          "leaseInfo": {
            "renewalIntervalInSecs": 30,
            "durationInSecs": 90,
            "registrationTimestamp": 1563132555899,
            "lastRenewalTimestamp": 1563132825917,
            "evictionTimestamp": 0,
            "serviceUpTimestamp": 1563132555899
          },
          "isCoordinator": false,
          "lastUpdatedTimestamp": 1563132555899,
          "lastDirtyTimestamp": 1563132825917,
          "scheme": null
        },
        "secure": false,
        "metadata": {}
      }
    }
  ],
  "cabeceras": {
    "Content-Type": "application/json"
  }
}
  
```



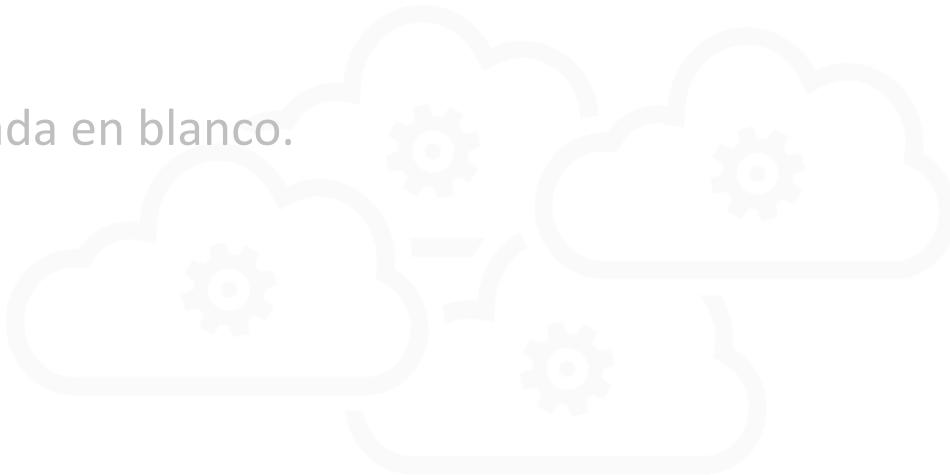
Resumen de la lección

v.iv Registro y descubrimiento de servicios con Spring Cloud Eureka.

- Comocimos el servidor de "**service discovery**" y "**service registry**" de Spring Cloud Netflix Eureka.
- Analizamos como debe realizarse la configuración del servidor Eureka en sus diferentes modalidades. ("**standalone**" y "**peer awareness**")
- Aprendimos como configurar los múltiples clientes de Eureka.
- Analizamos por qué es necesario implementar un servidor de "**service discovery**" y "**service registry**" en arquitecturas de microservicios.
- Implementamos una arquitectura distribuida mediante Spring Cloud Netflix Eureka para registrar y descubrir microservicios.



Esta página fue intencionalmente dejada en blanco.



Microservices