

Introducción a la gestión de Servicios Web con Spring Cloud y Netflix OSS

1. [Introducción](#)
2. [Entorno](#)
3. [Spring Cloud Config](#)
4. [Spring Cloud Netflix Eureka](#)
5. [Spring Cloud Netflix Zuul](#)
6. [Servicios Rest de ejemplo](#)
7. [Conclusiones](#)
8. [Repositorios](#)
9. [Referencias](#)

En este tutorial vamos a ver una introducción a Spring Cloud y Netflix OSS como herramientas para implementar algunos de los patrones más comunes utilizados en sistemas distribuidos.

1. Introducción

Ante modelos distribuidos es muy común toparnos con aplicaciones que frente a un creciente número de servicios afronten nuevos retos en lo referente a su administración y sincronización. En tal sentido, algunas de las dudas más comunes con las que nos podemos topar suelen ser:

- ¿Cómo me puedo asegurar que todo mis servicios están configurados correctamente?
- ¿Como puedo saber qué servicios se despliegan y dónde?
- ¿Cómo puedo mantener actualiza la información del enrutamiento de todos los servicios?
- ¿Cómo puedo prevenir las fallas por servicios caídos?
- ¿Cómo puedo verificar que todos los servicios están en funcionamiento?
- ¿Cómo puedo asegurarme cuales son los servicios expuestos externamente?

Para brindar una solución a estos escenarios **Spring.io** nos ofrece un conjunto de componentes que integrados con las herramientas de Netflix OSS nos permite desarrollar de una manera fácil y rápida aplicaciones que implementen algunos de los patrones más comúnmente usados en sistemas distribuidos.

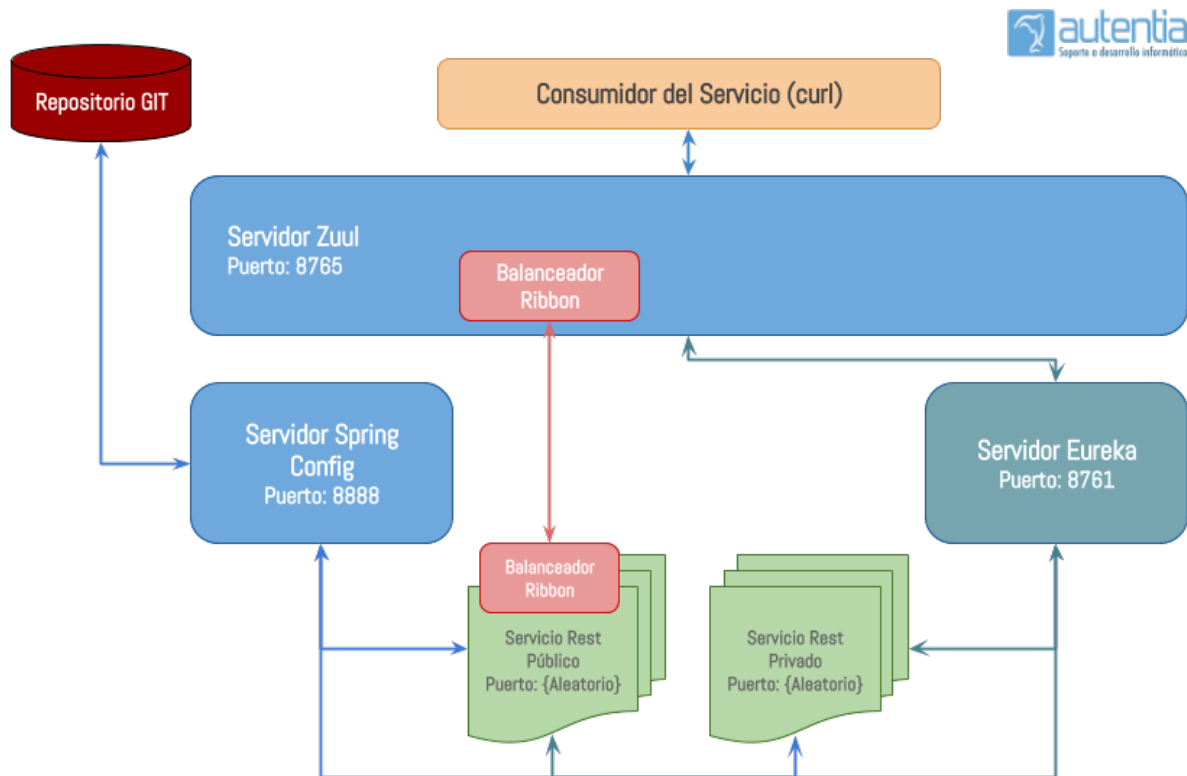
Algunos de estos patrones suelen ser:

- La configuración distribuida.
- El registro y auto-reconocimiento de servicios.
- Enrutamiento.
- Llamadas servicio a servicio.
- Balanceo de carga.
- Control de ruptura de comunicación con los servicios.
- Clusterización.
- Mensajería distribuida.

Dado que son muchas las posibilidades que nos brindan Spring y Netflix, en este tutorial nos enfocaremos en un ejemplo con los siguientes componentes:

- **Spring Cloud Config:** Para gestionar de manera centralizada la configuración de nuestros servicios utilizando un repositorio git.
- **Netflix Eureka:** Para registrar los servicios en tiempo de ejecución en un repositorio compartido.
- **Netflix Zuul:** Como servidor de enrutamiento y filtrado de peticiones externas. Zuul utiliza Ribbon para buscar servicios disponibles y enruta las solicitudes externa a una instancia apropiada de los servicios.

Para visualizar como se vinculan estos componentes podemos fijarnos en la siguiente imagen:



Para construir un ejemplo sencillo que involucre a todos estos componentes vamos a configurar un servidor con cada uno de ellos y un par de servicios “cliente” que comprueben su funcionamiento.

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- **Hardware:** Portátil MacBook Pro Retina 15' (2.5 Ghz Intel Core I7, 16GB DDR3).
- **Sistema Operativo:** Mac OS Sierra 10.12.3
- **Maven** – Versión: 3.3.9
- **Java** – Versión: 1.8.0_112
- **Spring Boot** – Versión: 1.5.1.RELEASE
- **Spring Cloud** – Versión: 1.3.0.M1

3. Spring Cloud Config

Lo primero que vamos a configurar será nuestro servidor Spring Cloud Config desde el que nuestros servicios obtendrán un archivo de propiedades. Para crear nuestro servidor nos aprovecharemos de las bondades de maven para importar las dependencias necesarias y las de Spring Boot para tener un servidor autocontenido.

En este orden de ideas, lo primero que haremos será crear nuestro proyecto maven donde nuestro **pom.xml** debería tener la siguiente forma:

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
3   >
4   <modelVersion>4.0.0</modelVersion>
5   <groupId>com.autentia.spring.cloud</groupId>
6   <artifactId>SpringCloudConfig-Server</artifactId>
7   <version>0.0.1-SNAPSHOT</version>
8   <parent>
9     <groupId>org.springframework.boot</groupId>
10    <artifactId>spring-boot-starter-parent</artifactId>
11    <version>1.5.1.RELEASE</version>
12  </parent>
13
14  <properties>
15    <java.version>1.8</java.version>
16  </properties>
17
18  <dependencyManagement>
19    <dependencies>
20      <dependency>
21        <groupId>org.springframework.cloud</groupId>
22        <artifactId>spring-cloud-config</artifactId>
23        <version>1.3.0.M1</version>
24        <type>pom</type>
25        <scope>import</scope>
26      </dependency>
27    </dependencies>
28  </dependencyManagement>
29
30  <dependencies>
31    <dependency>
32      <groupId>org.springframework.cloud</groupId>
33      <artifactId>spring-cloud-config-server</artifactId>
34    </dependency>
35  </dependencies>
36
37  <repositories>
38    <repository>
39      <id>spring-milestones</id>
40      <name>Spring Milestones</name>
41      <url>https://repo.spring.io/libs-milestone</url>
42      <snapshots>
43        <enabled>false</enabled>
44      </snapshots>
45    </repository>
46  </repositories>
47
48  </project>
```

Dado que nuestro Servidor de configuración es una aplicación Spring Boot necesitaremos un Main y en él incluiremos la anotación **@EnableConfigServer** que lo habilitará como servidor Spring Cloud Config.

```

1 package com.autentia.spring.cloud.config.server;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
5 import org.springframework.cloud.config.server.EnableConfigServer;
6
7 @EnableAutoConfiguration
8 @EnableConfigServer
9 public class ApplicationConfigServer {
10
11     public static void main(String[] args) {
12         SpringApplication.run(ApplicationConfigServer.class, args);
13     }
14
15 }

```

El último paso que debemos tener en cuenta es incluir un archivo de configuración llamado **application.yml** que ubicaremos en la carpeta **/src/main/resources/** y en el que configuraremos el nombre de la aplicación, el puerto en el que se ejecutará y el repositorio git donde se almacenarán los archivos de configuración de nuestros servicios.

```

1 # Component Info
2 info:
3   component: SpringConfig-Server
4
5 # HTTP Server
6 server:
7   port: 8888
8
9 spring:
10 # Spring Cloud Config Server Repository
11 cloud:
12   config:
13     server:
14       git:
15         uri: https://github.com/jmangialomini/Spring.Cloud.Config.Server
16
17 # Spring properties
18 profiles:
19   active: dev

```

Para probar nuestra aplicación vamos a la consola y en la carpeta de nuestro servidor Spring Config ejecutamos

```
1 mvn spring-boot:run
```

Una vez iniciado nuestro servidor podemos probar que todo esté funcionando correctamente con nuestro browser <http://localhost:8888/health> la cual nos debe reportar el estado actual de nuestro servidor.

```

1 {
2   "status": "UP"
3 }

```

4. Spring Cloud Netflix Eureka

El segundo servidor que incorporaremos para el soporte de nuestros servicios será el de **Eureka**, con el incorporaremos la capacidad de descubrir automáticamente los servicios e instancias que vayamos incorporando en tiempo de ejecución. Para ello y al igual que el servidor anterior utilizaremos Maven y Spring Boot.

En tal sentido, nuestro **pom.xml** debería quedar de la siguiente manera:

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
2 nce" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
3 >
4 <modelVersion>4.0.0</modelVersion>
5 <groupId>com.autentia.spring.cloud</groupId>
6 <artifactId>SpringCloudNetflix-EurekaServer</artifactId>
7 <version>0.0.1-SNAPSHOT</version>
8 <parent>
9 <groupId>org.springframework.boot</groupId>
10 <artifactId>spring-boot-starter-parent</artifactId>
11 <version>1.5.1.RELEASE</version>
12 </parent>
13
14 <properties>
15 <java.version>1.8</java.version>
16 </properties>
17
18 <dependencyManagement>
19 <dependencies>
20 <dependency>
21 <groupId>org.springframework.cloud</groupId>
22 <artifactId>spring-cloud-netflix</artifactId>
23 <version>1.3.0.M1</version>
24 <type>pom</type>
25 <scope>import</scope>
26 </dependency>
27 </dependencies>
28 </dependencyManagement>
29
30 <dependencies>
31 <dependency>
32 <groupId>org.springframework.cloud</groupId>
33 <artifactId>spring-cloud-starter-eureka-server</artifactId>
34 </dependency>
35
36 <dependency>
37 <groupId>org.springframework.boot</groupId>
38 <artifactId>spring-boot-starter-web</artifactId>
39 </dependency>
40
41 <dependency>
42 <groupId>org.springframework.boot</groupId>
43 <artifactId>spring-boot-starter-actuator</artifactId>
44 </dependency>
45
46 <dependency>
47 <groupId>org.springframework.boot</groupId>
48 <artifactId>spring-boot-starter-test</artifactId>
49 </dependency>
50 </dependencies>
51
52 <repositories>
53 <repository>
54 <id>spring-milestones</id>
55 <name>Spring Milestones</name>
56 <url>https://repo.spring.io/libs-milestone</url>
57 <snapshots>
58 <enabled>false</enabled>
59 </snapshots>
60 </repository>
61 </repositories>
</project>
```

Paso siguiente a la inclusión de las dependencias necesarias para habilitar nuestro servidor Eureka, debemos crear nuestro `ApplicationEurekaServer.java` e incluir la anotación **@EnableEurekaServer** .

```
1 package com.autentia.spring.cloud.netflix.eureka;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
6 import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
7
8 @SpringBootApplication
9 @EnableEurekaServer
10 public class ApplicationEurekaServer {
11
12     public static void main(String[] args) throws Exception {
13         SpringApplication.run(ApplicationEurekaServer.class, args);
14     }
15
16 }
```

Para configurar nuestro servidor incluiremos un archivo **application.yml** dentro de la carpeta `/src/main/resources/`, y en el que configuraremos el nombre de la aplicación, el puerto donde se ejecutará y le indicaremos que no debe publicarse en otro servidor eureka con la propiedad `registerWithEureka` en `false`.

```
1 # HTTP Server
2 server:
3   port: 8761
4
5 # Eureka Configuration Properties
6 eureka:
7   client:
8     registerWithEureka: false
9     fetchRegistry: false
10  server:
11    waitTimeInMsWhenSyncEmpty: 0
```


Para probar nuestra aplicación vamos a la consola y en la carpeta de nuestro servidor Eureka ejecutamos:

```
1 mvn spring-boot:run
```

Una vez iniciado nuestro servidor podemos probar que todo esté funcionando correctamente con nuestro browser `http://localhost:8761/health` la cual nos debe reportar el estado actual de nuestro servidor.

```
1 {
2   "description": "Spring Cloud Eureka Discovery Client",
3   "status": "UP"
4 }
```

También podemos navegar la interfaz gráfica de Eureka (`http://localhost:8761/`) y validar los servicios que se vayan incorporando.


HOME
LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2017-03-24T16:21:24+0100
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

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
No instances available			

5. Spring Cloud Netflix Zuul

Como el último de los servidores de soporte para nuestros servicios web vamos a configurar los servicios de Zuul como puerta de enlace para los servicios que queramos publicar.

Para ello y al igual que en los anteriores utilizaremos Maven y Spring Boot donde el **pom.xml** debería lucir de la siguiente manera:

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
3 >
4 <modelVersion>4.0.0</modelVersion>
5 <groupId>com.autentia.spring.cloud</groupId>
6 <artifactId>SpringCloudNetflix-ZuulServer</artifactId>
7 <version>0.0.1-SNAPSHOT</version>
8
9 <parent>
10 <groupId>org.springframework.boot</groupId>
11 <artifactId>spring-boot-starter-parent</artifactId>
12 <version>1.5.1.RELEASE</version>
13 </parent>
14
15 <properties>
16 <java.version>1.8</java.version>
17 </properties>
18
19 <dependencyManagement>
20 <dependencies>
21 <dependency>
22 <groupId>org.springframework.cloud</groupId>
23 <artifactId>spring-cloud-netflix</artifactId>
24 <version>1.3.0.M1</version>
25 <type>pom</type>
26 <scope>import</scope>
27 </dependency>
28 </dependencies>
29 </dependencyManagement>
30
31 <dependencies>
32 <dependency>
33 <groupId>org.springframework.cloud</groupId>
34 <artifactId>spring-cloud-starter-eureka</artifactId>
35 </dependency>
36
37 <dependency>
38 <groupId>org.springframework.cloud</groupId>

```

```

39 <artifactId>spring-cloud-starter-zuul</artifactId>
40 </dependency>
41
42 <dependency>
43 <groupId>org.springframework.boot</groupId>
44 <artifactId>spring-boot-starter-web</artifactId>
45 </dependency>
46
47 <dependency>
48 <groupId>org.springframework.boot</groupId>
49 <artifactId>spring-boot-starter-actuator</artifactId>
50 </dependency>
51
52 </dependencies>
53 <repositories>
54 <repository>
55 <id>spring-milestones</id>
56 <name>Spring Milestones</name>
57 <url>https://repo.spring.io/libs-milestone</url>
58 <snapshots>
59 <enabled>false</enabled>
60 </snapshots>
61 </repository>
</repositories>
</project>

```

Para habilitar el servicio Zuul en nuestra clase Main incluiremos la anotación **@EnableZuulProxy**.

```

1 package com.autentia.spring.cloud.netflix.zuul;
2
3 import org.springframework.boot.autoconfigure.SpringBootApplication;
4 import org.springframework.boot.builder.SpringApplicationBuilder;
5 import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
6 import org.springframework.stereotype.Controller;
7
8 @SpringBootApplication
9 @Controller
10 @EnableZuulProxy
11 public class ApplicationZuul {
12
13     public static void main(String[] args) {
14         new SpringApplicationBuilder(ApplicationZuul.class).web(true).run(args);
15     }
16 }

```

Finalmente y al igual que los otros servidores incluiremos un **application.yml** para configurar nuestro servidor en el que debemos destacar el enrutado para las peticiones a nuestro servicio de público de ejemplo que identificamos con su Service Id: **public-restservice**.

```

1 #Component Info
2 info:
3   component: Zuul-Server
4
5 #Spring Application Name
6 spring:
7   application:
8     name: Zuul-Server
9
10 #Server Port
11 server:
12   port: 8765
13
14 #Endpoints
15 endpoints:
16   restart:

```



```

17 enabled: true
18 shutdown:
19 enabled: true
20 health:
21 sensitive: false
22
23 #Zuul routes active
24 zuul:
25 routes:
26 public-restservice:
27 path: /public/**
28 serviceId: public-restservice
29
30 #Eureka Instance ID
31 eureka:
32 instance:
33 instanceId: ${spring.application.name}:${server.port}
34
35 #Ribbon Activation
36 ribbon:
37 eureka:
38 enabled: true

```

Para probar nuestra aplicación vamos a la consola y en la carpeta de nuestro servidor Zuul ejecutamos:

```
1 mvn spring-boot:run
```

Una vez iniciado nuestro servidor podemos probar que todo esté funcionando validando que se haya registrado contra nuestro servidor Eureka anteriormente iniciado.

6. Servicios Rest de prueba con Spring Boot

Finalmente y para poder probar cómo se sincroniza toda nuestra solución vamos a crear un par de servicios web que tomen un puerto aleatorio y se registren automáticamente contra nuestro servidor Eureka.

Al igual que con los servidores utilizaremos Maven y Spring Boot para facilitar nuestra implementación. Entre las dependencias que incluiremos para los servicios estarán las del Starter de Eureka como cliente y el Starter Web para habilitarlos como controladores REST.

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
2 nce" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
3 >
4 <modelVersion>4.0.0</modelVersion>
5 <groupId>com.autentia.spring</groupId>
6 <artifactId>SpringCloudNetflix-Public-RestService</artifactId>
7 <version>0.0.1-SNAPSHOT</version>
8
9 <parent>
10 <groupId>org.springframework.boot</groupId>
11 <artifactId>spring-boot-starter-parent</artifactId>
12 <version>1.5.1.RELEASE</version>
13 </parent>
14
15 <properties>
16 <java.version>1.8</java.version>
17 </properties>
18

```

```

19 <dependencyManagement>
20 <dependencies>
21 <dependency>
22 <groupId>org.springframework.cloud</groupId>
23 <artifactId>spring-cloud-netflix</artifactId>
24 <version>1.3.0.M1</version>
25 <type>pom</type>
26 <scope>import</scope>
27 </dependency>
28
29 <dependency>
30 <groupId>org.springframework.cloud</groupId>
31 <artifactId>spring-cloud-config</artifactId>
32 <version>1.3.0.M1</version>
33 <type>pom</type>
34 <scope>import</scope>
35 </dependency>
36 </dependencies>
37 </dependencyManagement>
38
39 <dependencies>
40 <dependency>
41 <groupId>org.springframework.cloud</groupId>
42 <artifactId>spring-cloud-starter-config</artifactId>
43 </dependency>
44
45 <dependency>
46 <groupId>org.springframework.cloud</groupId>
47 <artifactId>spring-cloud-starter-eureka</artifactId>
48 </dependency>
49
50 <dependency>
51 <groupId>org.springframework.boot</groupId>
52 <artifactId>spring-boot-starter-web</artifactId>
53 </dependency>
54
55 <dependency>
56 <groupId>org.springframework.boot</groupId>
57 <artifactId>spring-boot-starter-actuator</artifactId>
58 </dependency>
59
60 <dependency>
61 <groupId>org.springframework.boot</groupId>
62 <artifactId>spring-boot-starter-test</artifactId>
63 </dependency>
64 </dependencies>
65
66 <repositories>
67 <repository>
68 <id>spring-milestones</id>
69 <name>Spring Milestones</name>
70 <url>https://repo.spring.io/libs-milestone</url>
71 <snapshots>
72 <enabled>false</enabled>
73 </snapshots>
74 </repository>
75 </repositories>

</project>

```

Dado que es una aplicación Spring Boot necesitaremos un Main con la variante de incluir la anotación **@EnableDiscoveryClient** que permitirá que nuestros servicios se suscriban automáticamente en el servidor Eureka.

```

1 package com.autentia.spring.cloud.netflix;
2

```

```

3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
6
7 @EnableDiscoveryClient
8 @SpringBootApplication
9 public class ApplicationRestService {
10
11     public static void main(String[] args) {
12         SpringApplication.run(ApplicationRestService.class, args);
13     }
14 }

```

Para habilitar un ejemplo para nuestros servicios incluiremos una clase que responda al mapeo **/example**:

```

1 package com.autentia.spring.cloud.netflix;
2
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5 import org.springframework.beans.factory.annotation.Value;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RestController;
8
9 @RestController
10 public class ServiceExample {
11
12     @Value("${rest.service.cloud.config.example}")
13     String valueExample = null;
14
15     private static Logger log = LoggerFactory.getLogger(ServiceExample.class);
16
17     @RequestMapping(value = "/example")
18     public String example() {
19
20         String result = "{Empty Value}";
21         if(valueExample.equals(null)){
22
23             log.error("PublicRestService - Called with errors property rest.service.cloud.config.example is empty");
24
25         }else{
26             log.info("PublicRestService - Called with this property: (rest.service.cloud.config.example:"+valueExample+"");
27             result = valueExample;
28         }
29         return result;
30     }
31 }

```

Finalmente y para diferenciar a nuestros servicios incluiremos un archivo de configuración para cada uno de ellos. **Application.yml** para el servicio público:

```

1 #Application Name
2 spring:
3   application:
4     name: Public-RestService
5
6 #Component Info
7 info:
8   component: Public-RestService
9
10 #Port - If 0 get random port
11 server:
12   port: 0
13
14 #Eureka Instance ID
15 eureka:
16   instance:

```

```

17 instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:{random
18 .value}}}
19
20 #Service Registration Method
21 cloud:
22   services:
23     registrationMethod: route
24
25 #Disable HTTP Basic Authentication
26 security:
27   basic:
28     enabled: false

```

Y el **Application.yml** para el servicio privado:

```

1 #Application Name
2 spring:
3   application:
4     name: Private-RestService
5
6 #Component Info
7 info:
8   component: Private-RestService
9
10 #Port - If 0 get random port
11 server:
12   port: 0
13
14 #Eureka Instance ID
15 eureka:
16   instance:
17     instanceId: ${spring.application.name}:${vcap.application.instance_id:${spring.application.instance_id:{random
18 .value}}}
19
20 #Service Registration Method
21 cloud:
22   services:
23     registrationMethod: route
24
25 #Disable HTTP Basic Authentication
26 security:
27   basic:
28     enabled: false

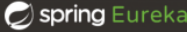
```

Para probar nuestros servicios vamos a la consola y en la carpeta de cada uno de ellos ejecutamos:

```
1 mvn spring-boot:run
```

Una vez iniciados los servicios podremos comprobar que todo esté funcionando en nuestro servidor Eureka, en él podremos visualizar cuántas instancias de nuestros servicios se han registrado y bajo qué puerto podemos invocarlos.

Nota: Para hacer más rigurosa nuestra prueba podemos iniciar una segunda instancia de alguno de nuestros servicios y comprobar la cantidad de instancias que están disponibles del mismo servicio.


HOME
LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2017-03-24T16:51:06 +0100
Data center	default	Uptime	00:03
		Lease expiration enabled	false
		Renews threshold	8
		Renews (last min)	0

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
PRIVATE-RESTSERVICE	n/a (1)	(1)	UP (1) - Private-RestService:195b19362758e7083959ac547d195031
PUBLIC-RESTSERVICE	n/a (2)	(2)	UP (2) - Public-RestService:98e01675f24089dbb9804163bbea5ad , Public-RestService:093e309d1ea1bd69902e0bda5270f1db
ZUUL-SERVER	n/a (1)	(1)	UP (1) - Zuul-Server:8765

Finalmente y para probar que el servicio público sea accesible desde el exterior a través de nuestro servidor Zuul podemos visitar <http://localhost:8765/public/example> y comprobar que nuestra configuración de servicios esté respondiendo correctamente.

7. Conclusiones

Hemos visto cómo se pueden usar los componentes de Spring Cloud y Netflix OSS para simplificar la gestión y sincronización de servicios web.

8. Repositorios

- Servidor Spring Cloud Config: <https://github.com/jmangialomini/SpringCloudConfig-Server>
- Repositorio Git Spring Cloud Config: <https://github.com/jmangialomini/SpringCloudConfig-GitRepository>
- Servidor Netflix Eureka: <https://github.com/jmangialomini/SpringCloudNetflix-EurekaServer>
- Servidor Netflix Zuul: <https://github.com/jmangialomini/SpringCloudNetflix-ZuulServer>
- Servicio Rest Privado: <https://github.com/jmangialomini/SpringCloudNetflix-Private-RestService>
- Servicio Rest Público: <https://github.com/jmangialomini/SpringCloudNetflix-Public-RestService>

Spring Cloud Stream:event-driven microservices

Existen muchas opciones para conectar microservicios entre sí, las comunicaciones no tienen porque ser todas síncronas mediante invocaciones directas haciendo uso del servicio de descubrimiento; también podemos realizar invocaciones asíncronas haciendo uso de brokers de mensajería.

Índice de contenidos.

- [1. Introducción.](#)
- [2. Entorno.](#)
- [3. Configuración.](#)
- [4. Publicación de un evento.](#)
- [5. Consumo de un evento.](#)
- [6. Integración del productor y el consumidor.](#)
- [7. Referencias.](#)
- [8. Conclusiones.](#)

1. Introducción.

Spring Cloud Stream es un proyecto de Spring Cloud construido sobre la base de Spring Boot y Spring Integration que nos facilita la creación de microservicios bajo los patrones de message-driven y event-driven.

El hecho de estar construido sobre la base de Spring Boot hace que sea muy sencilla su configuración mediante el soporte de starters, anotaciones y propiedades de configuración susceptibles de elevarse a un servicio de configuración centralizada y es el soporte de Spring Integration el que proporciona la conectividad con brokers de mensajería.

El patrón message-driven, en el entorno de microservicios, introduce el concepto de comunicación asíncrona entre microservicios dirigida por el envío de mensajes; un mensaje es un dato que se envía a un destino específico.

El patrón event-driven, sobre la base del patrón anterior y usando el mismo canal de comunicaciones, se orienta al envío de eventos; un evento es una señal emitida por un componente al alcanzar un determinado estado.

Sobre la base de estos principios se construyen patrones como CQRS y Sagas que, en un entorno de microservicios nos ayudan a mantener, eventualmente, consistente la información, mediante la generación de eventos de dominio (conceptos de DDD).

En este tutorial haremos una introducción al proyecto Spring Cloud Stream, como candidato para implementar dichos patrones en un entorno de microservicios, abstrayéndonos totalmente de la capa de transporte y sin necesidad de conocer detalles sobre el protocolo de comunicación.

Se presuponen conocimientos de Spring Boot y Spring Cloud, puesto que no se generará ninguno de los proyectos en los que se basan las pruebas desde cero.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15' (2.5 GHz Intel Core i7, 16GB DDR3).
- Sistema Operativo: Mac OS Sierra 10.12.5
- Oracle Java: 1.8.0_25
- Spring Cloud Ditmars.SR3
- Spring Boot 1.5.9.RELEASE

3. Configuración.

Se recomienda la configuración mediante el sistema de gestión de dependencias de maven y, para ello, lo primero es incluir las siguientes dependencias en nuestro pom.xml

```
1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>org.springframework.cloud</groupId>
5       <artifactId>spring-cloud-stream-dependencies</artifactId>
6       <version>Ditmars.SR3</version>
7       <type>pom</type>
8       <scope>import</scope>
9     </dependency>
10  </dependencies>
11 </dependencyManagement>
12 <dependencies>
13   <dependency>
14     <groupId>org.springframework.cloud</groupId>
15     <artifactId>spring-cloud-stream</artifactId>
16   </dependency>
17   <dependency>
18     <groupId>org.springframework.cloud</groupId>
19     <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
20   </dependency>
21 </dependencies>
```

En cuanto incluimos la dependencia de Spring Cloud Stream, que arrastra de forma transitiva la de Spring Cloud Stream Binder, cualquier aplicación Spring Boot con la anotación **@EnableBinding** intentará engancharse al broker de mensajería externo que encuentre en el classpath (Rabbit MQ, Apache Kafka,...).

A continuación definiremos los canales de comunicaciones en el fichero de configuración de nuestras aplicaciones (application.yml) y aquí vamos a hacer una distinción entre el productor de los eventos (command) y el consumidor de los mismos (query).

Esta sería la configuración del productor (command):

```
1 spring:
2   cloud:
3     stream:
4       bindings:
5         output:
6           destination: queue.orders.events
7         binder: local_rabbit
```

```

8   binders:
9     local_rabbit:
10    type: rabbit
11    environment:
12      spring:
13        rabbitmq:
14          host: localhost
15          port: 5672
16          username: guest
17          password: guest
18          virtual-host: /

```

A continuación la configuración del consumer (query):

```

1  spring:
2    cloud:
3      stream:
4        bindings:
5          input:
6            destination: queue.orders.events
7            binder: local_rabbit
8        binders:
9          local_rabbit:
10         type: rabbit
11         environment:
12           spring:
13             rabbitmq:
14               host: localhost
15               port: 5672
16               username: guest
17               password: guest
18               virtual-host: /

```

La diferencia está en la semántica del canal, en este segundo usamos “input” en vez de “output”.

Por último, antes de comenzar con nuestra emisión de eventos, vamos a incluir las dependencias necesarias para poder ejecutar tests de integración, como sigue:

```

1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-stream-test-support</artifactId>
4   <scope>test</scope>
5 </dependency>

```

4. Publicación de un evento (command).

Para publicar un evento lo primero es definir la estructura de datos del evento a emitir, en nuestro caso vamos a publicar un evento de generación de un pedido, con la información del pedido creado:

```

1 @Builder
2 @Getter
3 public class OrderCreationEvent implements Serializable{
4
5     private Order order;

```


6
7 }

En este punto debemos tomar una decisión, ¿compartimos estructuras de los datos de los eventos entre microservicios?. Si añadimos a la estructura de los eventos tipos complejos como en este ejemplo, se convertirá en un problema de dependencias y no queremos hacer que nuestros microservicios dependan de una única estructura de dominio con lo que, pensadlo primero, la estructura de datos de los eventos debería tener su propia estructura de datos independiente del dominio de negocio de cada microservicio. No deberíamos incluir en el evento objetos específicos de dominio como, en el ejemplo, el pedido.

A continuación, vamos a añadir a nuestra configuración la anotación **@EnableBinding(Processor.class)** para realizar un binding con el stream correspondiente (por defecto con Processor.class tendremos los canales “output” e “input” que ya hemos definido en el application.yml).

```
1 package com.autentia.training.microservices.spring.cloud;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
6 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
7 import org.springframework.cloud.netflix.feign.EnableFeignClients;
8 import org.springframework.cloud.stream.annotation.EnableBinding;
9 import org.springframework.cloud.stream.messaging.Source;
10 import org.springframework.context.annotation.Bean;
11 import org.springframework.context.annotation.Configuration;
12
13 import com.autentia.training.microservices.spring.cloud.order.repository.OrderRepository;
14 import com.autentia.training.microservices.spring.cloud.product.consumer.ProductConsumerFallBack;
15 import com.autentia.training.microservices.spring.cloud.service.OrderResourceService;
16
17 @SpringBootApplication
18 @EnableDiscoveryClient
19 @EnableCircuitBreaker
20 @EnableFeignClients
21 @Configuration
22 @EnableBinding(Processor.class)
23 public class OrdersCommandApplication {
24
25     public static void main(String[] args) {
26         SpringApplication.run(OrdersApplication.class, args);
27     }
28
29     @Bean
30     public OrderResourceService orderService(OrderRepository orderRepository, Source pipe){
31         return new OrderResourceService(orderRepository, pipe);
32     }
33
34
35 }
```

Podemos configurar una fuente (Source.class), un destino (Sink.class) o ambos (Producer.class), lo único que estamos haciendo es declarar como canales “input” y/o “output”, según el caso.

Lo siguiente es publicar el evento y lo vamos a enviar en nuestro servicio de negocio justo después de persistir la entidad; para ello debemos recibir la inyección del repositorio (Spring Data) y la fuente para poder realizar el envío.

```
1 package com.autentia.training.microservices.spring.cloud.service;
```

```

2
3 import java.time.LocalDateTime;
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 import javax.transaction.Transactional;
8
9 import org.springframework.cloud.stream.messaging.Source;
10 import org.springframework.integration.support.MessageBuilder;
11
12 import com.autentia.training.microservices.spring.cloud.event.OrderCreationEvent;
13 import com.autentia.training.microservices.spring.cloud.order.domain.Order;
14 import com.autentia.training.microservices.spring.cloud.order.domain.OrderLine;
15 import com.autentia.training.microservices.spring.cloud.order.repository.OrderRepository;
16 import com.autentia.training.microservices.spring.cloud.order.vo.OrderResource;
17
18 @Transactional
19 public class OrderResourceService {
20
21     private OrderRepository orderRepository;
22
23     private Source pipe;
24
25     public OrderResourceService(OrderRepository orderRepository, Source pipe) {
26         this.orderRepository = orderRepository;
27         this.pipe = pipe;
28     }
29
30     public Order create(OrderResource orderResource) {
31         final Order orderCreated = orderRepository.save(mapOrder(orderResource));
32         pipe.output().send(MessageBuilder.withPayload(OrderCreationEvent.builder().order(orderCr
33 eated).build()).build());
34         return orderCreated;
35     }
36
37     private Order mapOrder(OrderResource orderResource) {
38         final List<OrderLine> lines = orderResource.getLines().stream()
39             .map(line -> OrderLine.builder().build())
40             .collect(Collectors.toList());
41         return Order.builder().createdAt(LocalDateTime.now()).lines(lines).build();
42     }
43 }

```

Una vez hecho esto vamos a comprobar que realmente se envía escribiendo el siguiente test de integración:

```

1 package com.autentia.training.microservices.spring.cloud.service;
2
3 import static org.junit.Assert.assertThat;
4 import static org.junit.Assert.assertTrue;
5
6 import java.util.concurrent.BlockingQueue;
7
8 import org.hamcrest.Matchers;
9 import org.junit.Test;
10 import org.junit.runner.RunWith;
11 import org.springframework.beans.factory.annotation.Autowired;
12 import org.springframework.boot.test.context.SpringBootTest;
13 import org.springframework.cloud.stream.annotation.EnableBinding;
14 import org.springframework.cloud.stream.messaging.Sink;
15 import org.springframework.cloud.stream.messaging.Source;
16 import org.springframework.cloud.stream.test.binder.MessageCollector;
17 import org.springframework.messaging.Message;
18 import org.springframework.test.annotation.DirtiesContext;

```

```

19 import org.springframework.test.context.ActiveProfiles;
20 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
21
22 import com.autentia.training.microservices.spring.cloud.OrdersApplication;
23 import com.autentia.training.microservices.spring.cloud.event.OrderCreationEvent;
24 import com.autentia.training.microservices.spring.cloud.order.domain.Order;
25 import com.autentia.training.microservices.spring.cloud.order.vo.OrderResource;
26
27 @RunWith(SpringJUnit4ClassRunner.class)
28 @SpringBootTest(classes = OrdersApplication.class, webEnvironment = SpringBootTest.WebEnvironment.NONE
29 )
30 @DirtiesContext
31 @ActiveProfiles("development")
32 public class OrderResourceServiceIT {
33
34     @Autowired
35     private Source channels;
36
37     @Autowired
38     private MessageCollector collector;
39
40     @Autowired
41     private OrderResourceService orderService;
42
43     @Test
44     public void shouldPropagateOrderCreationEvents() {
45
46         // given
47         final BlockingQueue<Message<?>> messages = this.collector.forChannel(channels.output());
48
49         // when
50         final Order order = orderService.create(OrderResource.builder().build());
51
52         // then
53         assertThat(messages, Matchers.hasSize(1));
54
55         assertTrue(messages.stream().filter(o -> ((OrderCreationEvent) o.getPayload()).getOrder().getId() == order.getId()).findFirst().isPresent());
56
57     }
58 }

```

Declaramos por inyección de dependencias un recolector de mensajes y el canal fuente definido en el binder; con este último configuramos el recolector y comprobamos que una vez invocado el servicio de negocio podemos consumir un evento almacenado en el recolector que tiene como payload la estructura de datos del evento enviado.

En este punto deberíamos estar en verde.

5. Consumo de un evento (query).

Consumir el evento desde un test de integración esta más que bien, pero el objetivo es consumirlo desde otro microservicio, para lo cuál debemos realizar la misma configuración en cuanto a dependencias de maven en nuestro segundo microservicio y crear la estructura de datos a la que se mapeará la recepción del evento.

Lo siguiente es configurar el binder y un método de listener que escuchará los eventos:

```
1 package com.autentia.training.microservices.spring.cloud;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
6 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
7 import org.springframework.cloud.netflix.feign.EnableFeignClients;
8 import org.springframework.cloud.stream.annotation.EnableBinding;
9 import org.springframework.cloud.stream.annotation.StreamListener;
10 import org.springframework.cloud.stream.messaging.Processor;
11 import org.springframework.cloud.stream.messaging.Sink;
12 import org.springframework.context.annotation.Bean;
13 import org.springframework.context.annotation.Configuration;
14
15 import com.autentia.training.microservices.spring.cloud.event.OrderCreationEvent;
16
17 import lombok.extern.slf4j.Slf4j;
18
19 @SpringBootApplication
20 @EnableDiscoveryClient
21 @EnableCircuitBreaker
22 @EnableFeignClients
23 @Configuration
24 @EnableBinding(Sink.class)
25 @Slf4j
26 public class OrdersApplication {
27     public static void main(String[] args) {
28         SpringApplication.run(OrdersApplication.class, args);
29     }
30
31     @StreamListener(Processor.INPUT)
32     public void transform(OrderCreationEvent payload) {
33         log.info("order received {}", payload.getOrder().getId());
34     }
35
36 }
```

El listener solo traza la recepción del evento con el sistema de logging estándar (Simple Logging Facade For Java).

Y, como no, vamos a probar el envío y la recepción con un test de integración dentro del contexto del propio microservicio:

```
1 package com.autentia.training.microservices.spring.cloud.service;
2
3
4 import static org.awaitility.Awaitility.await;
5 import static org.hamcrest.CoreMatchers.containsString;
6
7 import java.time.LocalDateTime;
8 import java.util.concurrent.Callable;
9 import java.util.concurrent.TimeUnit;
10
11 import org.junit.Before;
12 import org.junit.Rule;
13 import org.junit.Test;
14 import org.junit.runner.RunWith;
15 import org.springframework.beans.factory.annotation.Autowired;
16 import org.springframework.boot.test.context.SpringBootTest;
17 import org.springframework.boot.test.rule.OutputCapture;
18 import org.springframework.cloud.stream.messaging.Sink;
```

```

19 import org.springframework.integration.support.MessageBuilder;
20 import org.springframework.test.annotation.DirtiesContext;
21 import org.springframework.test.context.ActiveProfiles;
22 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
23
24 import com.autentia.training.microservices.spring.cloud.OrdersApplication;
25 import com.autentia.training.microservices.spring.cloud.event.OrderCreationEvent;
26
27 @RunWith(SpringJUnit4ClassRunner.class)
28 @SpringBootTest(classes = OrdersApplication.class, webEnvironment = SpringBootTest.WebEnvironment.NONE
29 )
30 @DirtiesContext
31 @ActiveProfiles("development")
32 public class OrderCreationEventIT {
33
34     @Rule
35     public OutputCapture capture = new OutputCapture();
36
37     @Autowired
38     private Sink channels;
39
40     @Before
41     public void setup() {
42         capture.flush();
43     }
44
45     @Test
46     public void shouldConsumeOrderCreationEvents() {
47
48         // when
49         channels.input().send(MessageBuilder.withPayload(OrderCreationEvent.builder().id(1L).createAt(LocalDateTime.now()).build()).build());
50
51         // then
52         await().atMost(10, TimeUnit.SECONDS).until(getCapturedContentAsString(), containsString("
53 order received 1"));
54
55     }
56
57     private Callable<String> getCapturedContentAsString() {
58         return new Callable<String>() {
59             public String call() throws Exception {
60                 return capture.toString();
61             }
62         };
63     }
64 }

```

Simulamos el servicio que emite un evento generándolo nosotros manualmente y como el listener simplemente traza un mensaje y en el entorno de test la salida es por consola, la capturamos y esperamos 10 segundos a que se imprima.

En este punto deberíamos estar también en verde.

6. Integración del productor (command) y el consumidor (query).

Tenemos ambos tests de integración funcionando y ahora vamos a realizar una prueba integrada de los dos microservicios para lo cuál necesariamente tendremos que levantar el broker de mensajería en local para comprobar el envío y la recepción.

Para levantar en local un Rabbit MQ, que mejor que docker y con el siguiente comando lo tendremos funcionando en un solo paso, exponiendo los mismos puertos que ya tenemos configurados en el application.yml de ambos microservicios (en entornos productivos en nuestro servicio de configuración centralizada).

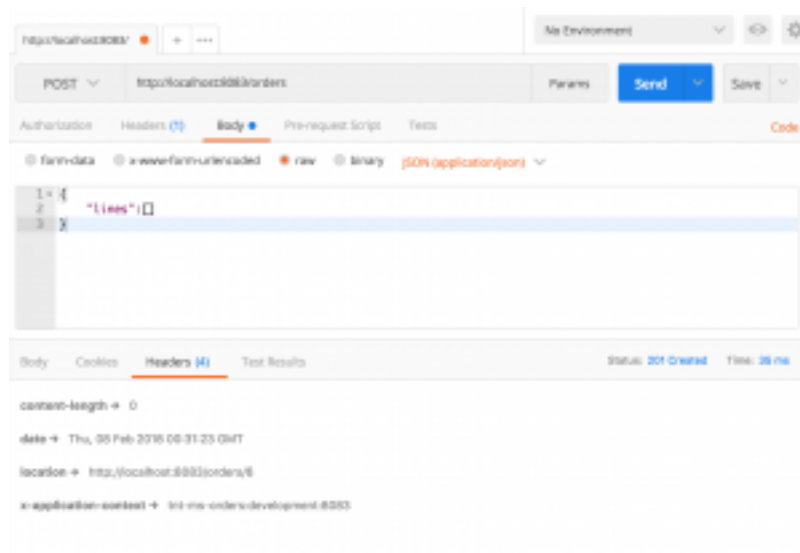
```
1 docker run -d --hostname localrabbit --name demo-rmq -p 15672:15672 -p 5672:5672 rabbitmq:3.6.11-management
```

Al finalizar el comando anterior, con el siguiente **docker ps** deberíamos tener un contenedor levantado como el que se muestra a continuación:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORT
1 S		NAMES			
2 ad2002c9ca3c	rabbitmq:3.6.11-management	"docker-entrypoint.s..."	16 seconds ago	Up 16 seconds	43 69/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp, 15671/tcp, 25672/tcp, 0.0.0.0:15672->15672/tcp demo-rmq

Si levantamos los dos microservicios con un perfil de desarrollo, sin necesidad de depender de los servicios estructurales de Spring Cloud y probamos a lanzar una operación contra el API que termina generando un pedido podremos comprobar como se emite el evento desde el microservicio de command y podemos consumirlo desde el microservicio de query.

A continuación os muestro un ejemplo de invocación desde postman con la confirmación de la creación del recurso con la cabecera de location:



Si accedemos a la consola del segundo microservicio podremos comprobar cómo se imprime por consola la traza del evento aunque también podemos comprobar, desde la consola de administración del propio Rabbit MQ, las estadísticas de envío de mensajes en el canal que hemos definido.

Exchange: queue.orders.events

Overview

Message rates (chart: last minute) (7)



Publish (In) 0.00/s
Publish (Out) 0.00/s

Details

Type: topic
Features: durable: true
Policy:

Bindings

This exchange



To	Routing key	Arguments
queue.orders.events.anonymous.YH4iqXhZQ0uioKhGEsD6Pw	#	unbind

Para ello no tenemos más que acceder a través de un navegador a <http://localhost:15672/>

Queue queue.orders.events.anonymous.YH4iqXhZQ0uioKhGEsD6Pw

Overview

Spooled messages (chart: last minute) (0)



Ready 0
Unacked 0
Total 0

Message rates (chart: last minute) (0)



Publish 0.00/s
Unacked (manual ack) 0.00/s
Unacked (auto ack) 0.00/s

Consumer ack 0.00/s
Unacked 0.00/s
Ack (manual ack) 0.00/s
Ack (auto ack) 0.00/s

Details

Feature	Value	Unit	State	Unit	Message (T)	Total	Ready	Unacked	In memory	Pending	Transient, Paged Out
Features	exclusive: true		Consumers	1							
Policy	Consumer activation (T)	0/s			Message body bytes (T)	0	0	0	0	0	0
					Message body bytes (T)	0	0	0	0	0	0
					Message body bytes (T)	0	0	0	0	0	0

7. Referencias.

- <http://projects.spring.io/spring-cloud/>
- <https://cloud.spring.io/spring-cloud-stream/>
- <http://martinfowler.com/eaaDev/EventSourcing.html>
- <http://martinfowler.com/bliki/CQRS.html>

8. Conclusiones.

Continuamos explotando las posibilidades del stack de Spring Cloud y nos sorprendemos de la sencillez; cada vez más nos olvidamos de la configuración y uso de plantillas para acercarnos a los conceptos de los patrones que queremos implementar.

Un saludo.

Jose

REST, el principio HATEOAS y Spring HATEOAS.

0. Índice de contenidos.

- [1. Introducción.](#)
- [2. Entorno.](#)
- [3. ¿Qué es HATEOAS?](#)
- [4. Añadiendo links a la representación de nuestros recursos con Spring HATEOAS.](#)
- [5. ¿Vemos un ejemplo?](#)
 - [5.1. Respuestas sin links.](#)
 - [5.2. Respuestas que siguen el principio HATEOAS](#)
- [6. Referencias.](#)
- [7. Conclusiones.](#)

1. Introducción

Poco podemos decir de REST que no hayamos comentado ya. Su simplicidad hace que muchos desarrolladores opten por esta solución por encima de SOAP a la hora de exponer las diferentes funcionalidades (mejor dicho, recursos) de sus plataformas. La **simplicidad** es, sin lugar a dudas, su gran arma pero, **¿abusamos de ella?**, ¿diseñamos verdaderas APIs REST o simples interfaces sobre HTTP?, ¿conocemos realmente los principios de diseño de servicios RESTful?.

En este tutorial intentaremos explicar qué es el principio HATEOAS, de obligado cumplimiento para cualquier API REST que se enorgullezca de serlo y veremos, mediante un ejemplo, el soporte que nos proporciona Spring para conseguirlo gracias a Spring HATEOAS.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15' (2.2 Ghz Intel Core I7, 8GB DDR3).
- Sistema Operativo: Mac OS Mountain Lion 10.8

- Entorno de desarrollo: [IntelliJ Idea 11.1 Ultimate.](#)
- Apache Tomcat 7.0.47
- Maven 3.0.3
- Java 1.7.0_45
- Spring 3.2.4.RELEASE
- Spring HATEOAS 0.8.0.RELEASE
- H2 database 1.3.170

3. ¿Qué es HATEOAS?.

HATEOAS es un acrónimo de **Hypermedia As The Engine Of Application State** (hipermedia como motor del estado de la aplicación). Significa algo así como que, dado un punto de entrada genérico de nuestra API REST, podemos ser capaces de descubrir sus recursos basándonos únicamente en las respuestas del servidor. Dicho de otro modo, cuando el servidor nos devuelva la representación de un recurso (JSON, XML...) parte de la información devuelta serán identificadores únicos **en forma de hipervínculos** a otros recursos asociados.

Lo vamos a entender más fácilmente con este ejemplo. Imaginemos que tenemos un API de un concesionario de coches donde nuestros clientes, evidentemente, compran coches. Supongamos que queremos obtener los datos del cliente con **id** igual a 78. Haríamos una petición de este estilo:

```
1 Request URL: http://miservidor/concesionario/api/v1/clientes/78
2 Request Method: GET
3 Status Code: 200 OK
```

Y obtendríamos algo como:

```
1 {
2   "id": 78,
3   "nombre": "Juan",
4   "apellido": "García",
5   "coches": [
6     {
7       "id": 1033
8     },
9     {
10      "id": 3889
11    }
12  ]
13 }
```

Con esto ya sabemos que nuestro cliente compró dos coches pero, **¿cómo accedemos a la representación de esos dos recursos?**. Sin consultar la documentación del API no tenemos forma de obtener la URL que identifique de forma única a cada uno de los coches. Además, aunque supiésemos conformar la URL de acceso a los recursos, cualquier cliente que quisiese consumir los recursos debería tener la **responsabilidad de construir dicha URL**. Por último, **¿qué ocurriría si la URL cambiase?**, habría que cambiar todos los clientes que consumen los recursos.

Siguiendo el principio HATEOAS la respuesta sería algo como:

```
1 {
2   "id": 78,
3   "nombre": "Juan",
4   "apellido": "García",
```

```

5  "coches": [
6      {
7          "coche": "http://miservidor/concesionario/api/v1/clientes/78/coches/1033"
8      },
9      {
10         "coche": "http://miservidor/concesionario/api/v1/clientes/78/coches/3889"
11     }
12 ]
13 }

```

De esta forma, ya sabemos dónde debemos ir a buscar los recursos relacionados (coches) con nuestro recurso original (cliente) gracias a la respuesta del servidor (hypertext-driven). Sin seguir este principio, nuestra API nunca seguirá el verdadero estilo arquitectónico REST. Y no lo digo solo yo, [lo dice su principal promotor Roy Fielding](#).

Spring HATEOAS es un pequeño módulo perteneciente al “ecosistema Spring” que nos ayudará a crear APIs REST que respeten el principio HATEOAS.

4. Añadiendo links a la representación de nuestros recursos con Spring HATEOAS.

Añadir links a nuestros recursos (mejor dicho, a sus representaciones) es muy sencillo con Spring HATEOAS. La única condición de que debemos cumplir es que el objeto u objetos que devolvamos en la respuesta extiendan de **org.springframework.hateoas.ResourceSupport**. Una vez que cumplan con esto ya podremos añadir a nuestro recurso los links que apunten a las URLs de otros recursos de nuestra API con los que queramos relacionarlo.

Supongamos que tenemos una clase `PersonWrapper` que representará un recurso “persona”:

```

1  @XmlElement(name = "person")
2  public class PersonWrapper extends ResourceSupport {
3
4      private String name;
5
6      private int age;
7
8      private PersonWrapper() {}
9
10     public PersonWrapper(Person person) {
11         this.name = person.getName();
12         this.age = person.getAge();
13     }
14
15     public String getName() {
16         return name;
17     }
18
19     public void setName(String name) {
20         this.name = name;
21     }
22
23     public int getAge() {
24         return age;
25     }
26
27     public void setAge(int age) {
28         this.age = age;

```

```
29 }
30 }
```

En nuestro controlador **PersonController** tendremos un método que devolverá un recurso persona que coincida con el nombre solicitado. Algo como lo siguiente:

```
1 @Controller
2 public class PersonController {
3
4     @RequestMapping(value = "/persons/{name}", method = RequestMethod.GET)
5     public @ResponseBody PersonWrapper getPerson(@PathVariable String name) {
6         final Person person = getPersonByName(name);
7         final PersonWrapper wrapper = new PersonWrapper(person);
8         return wrapper;
9     }
10
11 }
```

A través de la URI **/persons/pepe** obtendríamos la representación (XML, JSON o lo que sea) de la persona con nombre igual a pepe.

Ahora supongamos que queremos añadir un link a nuestro recurso que apunte a sí mismo. La forma de hacerlo es muy sencilla. La clase `ResourceSupport` de la que heredamos viene con un método **add** que recibe como parámetro un objeto de tipo **org.springframework.hateoas.Link** que representará un enlace a cualquier recurso. Dicho enlace sigue el [estandar Atom](#) para links.

Añadir el link a nuestro recurso que apunte a sí mismo es muy fácil. Tan fácil como invocar a este nuevo método antes de devolver el recurso:

```
1 private void addSelfLink(PersonWrapper resource){
2     final PersonWrapper person = methodOn(PersonController.class).getPerson(resource.getName());
3     final Link link = linkTo(person).withSelfRel();
4     resource.add(link);
5 }
```

Creo que el código habla por sí solo, pero por si no queda claro vamos a explicarlo. Lo que hacemos es crear un link que apunta recurso que daría como resultado la invocación del método correspondiente en el controlador correspondiente. Nótese que tanto *methodOn* como *linkTo* son métodos estáticos de la clase [org.springframework.hateoas.mvc.ControllerLinkBuilder](#). El resultado tras añadir el link nos daría una respuesta como la que sigue:

Respuesta en formato JSON:

```
1 {
2   "links": [{
3     "rel": "self",
4     "href": "http://localhost:8080/myapi/v1/persons/pepe"
5   }],
6   "name": "pepe",
7   "age": 25
8 }
```

Respuesta en formato XML:

```
1     <atom:link rel="self" href="http://localhost:8080/myapi/v1/persons/pepe"/>
2     25
3     luis
```

¿Existen otras formas de añadir links a nuestros recursos?

Pues la respuesta es sí y no. No porque todas pasan irremediabilmente (al menos que yo sepa) por devolver un elemento del tipo `ResourceSupport`. Y sí porque Spring

HATEOAS nos proporciona mecanismos para devolver estos elementos de tipo `ResourceSupport` de una manera **más elegante**. Al menos, existen estas dos alternativas:

- Hacer uso de la clase **`org.springframework.hateoas.Resource`** que ya extiende de `ResourceSupport` y que nos permite no tener que crear nosotros los envoltorios puesto que `Resource` ya es, en sí mismo, un envoltorio. Yo, personalmente, tuve una mala experiencia usando esta clase con representaciones XML. Aquí os dejo un [enlace para que que quiera saber más](#).
- Hacer uso de la clase **`org.springframework.hateoas.mvc.ResourceAssemblerSupport`**, que básicamente es un conversor de POJOS que usemos en nuestro modelo de datos a objetos del tipo `ResourceSupport`. Probablemente esta **es la mejor solución** porque nos permite separar el proceso de conversión e inserción de links en una clase independiente ([principio de responsabilidad única](#)).

5. ¿Vemos un ejemplo?.

A continuación veremos un ejemplo que nos ayudará a asimilar mejor todos estos conceptos. Para ello crearemos un **API REST muy futbolera** :-). Estará compuesta de tres recursos distintos: **equipos, estadios y jugadores**, que estarán relacionados de la siguiente forma:

- Un equipo juega en un estadio.
- Un equipo está compuesto por un conjunto de jugadores.
- Cada jugador solo puede jugar en un equipo.
- En un estadio solo juega un equipo.

Resource	Descripción
GET /teams	Devuelve el listado de equipos Da de alta un equipo. Es necesario enviar en el cuerpo de la petición los siguientes campos: <i>name, foundationYear</i> y <i>rankingPosition</i> .
POST /teams	
GET /teams/:id	Devuelve el equipo cuyo identificador coincida con <i>:id</i> Devuelve el estadio del equipo cuyo identificador coincida con <i>:id</i>
GET /teams/:id/stadium	Da de alta el estadio del equipo cuyo identificador coincida con <i>:id</i> . Es necesario enviar en el cuerpo de la petición los siguientes campos: <i>capacity, name</i> y <i>city</i> .
POST /teams/:id/stadium	Devuelve los jugadores del equipo cuyo identificador coincida con <i>:id</i>
GET /teams/:id/players	Da de alta un jugador perteneciente al equipo cuyo identificador coincida con <i>:id</i> . Es necesario enviar en el cuerpo de la petición los siguientes campos: <i>name, goals, country</i> y <i>age</i> .
POST /teams/:id/players	Devuelve los datos del jugador que juegue en un equipo cuyo identificador coincida con <i>:id</i> y con un
GET /teams/:id/players/:id_player	identificador de jugador que coincida con <i>:id_player</i> .

Además, todas las peticiones y respuestas (body) soportarán **json y xml** (application/json y application/xml media types).

5.1. Respuestas sin links.

Si no seguimos el principio HATEOAS tendríamos el siguiente método en nuestro controlador que se encargaría de recibir las peticiones GET solicitando un equipo con un identificador dado.

```
1 @RequestMapping(value = "/teams/{id}", method = RequestMethod.GET)
2 public @ResponseBody Team getById(@PathVariable int id) {
3     LOG.trace("Recibida solicitud para devolver el equipo con id {}", id);
4     final Team team = teamDao.getById(id);
5     if (team == null) {
6         LOG.warn("El equipo con id {} no existe", id);
7         throw new ResourceNotFoundException();
8     }
9
10    return team;
11 }
```

Tras realizar una la petición solicitando el equipo con id = 5000

```
1 Request URL: http://localhost:8080/soccer/api/teams/5000
2 Request Method: GET
3 Status Code: 200 OK
```

Obtendríamos un respuesta de este tipo:

```
1 {
2   "teamId": 5000,
3   "name": "Real Madrid C.F.",
4   "foundationYear": 1902,
5   "rankingPosition": 1
6 }
```

Como dijimos anteriormente, un equipo tiene asociado un **estadio** y un conjunto de **jugadores**, pero como cliente del API ¿cómo puedo saber cómo acceder a estos elementos asociados?. Sin seguir el principio HATEOAS, la única forma posible es **consultar la documentación** para saber que un equipo tiene un estadio y un conjunto de jugadores además de las URL's donde residen los mismos. Además, esas URLs las deberíamos construir nosotros.

5.2. Respuestas que siguen el principio HATEOAS.

Vamos a modificar ligeramente nuestro código para que, haciendo uso de Spring-HATEOAS, podamos devolver los links que apuntan tanto al estadio como al listado de jugadores asociados a un equipo. Haremos lo siguiente:

- Nuestra clase team ahora extenderá de **ResourceSupport** y heredará el soporte para que podamos añadir links asociados al equipo.
- Añadimos los links del estadio y del listado de jugadores al equipo apuntando directamente a los métodos de sus correspondientes controladores, de esta forma, si cambiase la URL del recurso o su método HTTP no tendríamos que tocar nada en nuestro código.

OJO!!! probablemente ésta no sería la mejor forma en lo que a términos de diseño se refiere puesto que estamos mapeando directamente la salida de un DAO (Team) a la representación del recurso (se acabó el bajo acoplamiento). Lo ideal sería hacer uso

de **ResourceAssemblerSupport** y devolver un wrapper. Sin embargo optamos por esta otra opción por simplicidad en el código.

Quedaría algo así (recordemos que ahora Team extiende de ResourceSupport):

```
1  ]
2  @RequestMapping(value = "/teams/{id}", method = RequestMethod.GET)
3  public @ResponseBody Team getById(@PathVariable int id) {
4      LOG.trace("Recibida solicitud para devolver el equipo con id {}", id);
5      final Team team = teamDao.getById(id);
6      if (team == null) {
7          LOG.warn("El equipo con id {} no existe", id);
8          throw new ResourceNotFoundException();
9      }
10     addTeamLinks(team);
11     return team;
12 }
13
14 private void addTeamLinks(Team team) {
15     addSelfLink(team);
16     addStadiumLink(team);
17     addPlayerLink(team);
18 }
19
20 private void addSelfLink(Team team) {
21     team.add(linkTo(methodOn(TeamController.class).getById(team.getTeamId())).withSelfRel());
22 }
23
24 private void addStadiumLink(Team team) {
25     team.add(linkTo(methodOn(StadiumController.class).getByTeamId(team.getTeamId())).withRel("stadium"));
26 }
27
28 private void addPlayerLink(Team team) {
29     team.add(linkTo(methodOn(PlayerController.class).getTeamPlayers(team.getTeamId())).withRel("players"));
30 }
```

Y, tras la misma petición que hicimos en el punto anterior, el resultado sería este:

```
1  ]
2  {
3      "links": [{
4          "rel": "self",
5          "href": "http://localhost:8080/soccer/api/teams/5000"
6      }, {
7          "rel": "stadium",
8          "href": "http://localhost:8080/soccer/api/teams/5000/stadium"
9      }, {
10         "rel": "players",
11         "href": "http://localhost:8080/soccer/api/teams/5000/players"
12     }],
13     "teamId": 5000,
14     "name": "Real Madrid C.F.",
15     "foundationYear": 1902,
16     "rankingPosition": 1
17 }
```

Como se puede apreciar, con una simple petición para obtener un equipo, podemos saber dónde residen sus recursos asociados. Si quisiésemos que la respuesta fuese XML, el resultado sería:

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2
3 <atom:link rel="self" href="http://localhost:8080/soccer/api/teams/5000" />
4 <atom:link rel="stadium" href="http://localhost:8080/soccer/api/teams/5000/stadium" />
```

```
5 <atom:link rel="players" href="http://localhost:8080/soccer/api/teams/5000/players" />
6 5000
7 Real Madrid C.F.
8 1902
9 1
```

Y, evidentemente, accediendo a alguno de los recursos marcados por los enlaces, obtendríamos respuestas válidas:

```
1 Request URL: http://localhost:8080/soccer/api/teams/5000/players
2 Request Method: GET
3 Status Code: 200 OK
4
5 {
6   "players": [{
7     "links": [{
8       "rel": "self",
9       "href": "http://localhost:8080/soccer/api/teams/5000/players/7000"
10    }, {
11      "rel": "team",
12      "href": "http://localhost:8080/soccer/api/teams/5000"
13    }],
14    "playerId": 7000,
15    "name": "Cristiano Ronaldo",
16    "goals": 172,
17    "age": 28,
18    "country": "Portugal",
19    "teamId": 5000
20  }, {
21    "links": [{
22      "rel": "self",
23      "href": "http://localhost:8080/soccer/api/teams/5000/players/7001"
24    }, {
25      "rel": "team",
26      "href": "http://localhost:8080/soccer/api/teams/5000"
27    }],
28    "playerId": 7001,
29    "name": "Xabi Alonso",
30    "goals": 12,
31    "age": 32,
32    "country": "España",
33    "teamId": 5000
34  }],
35  // etc, etc, etc...
36 }]
```

Podéis ver y descargar el [CÓDIGO FUENTE COMPLETO DEL EJEMPLO AQUÍ](#).

6. Referencias.

- [Roy Fielding: REST APIs must be hypertext-driven](#)
- [Spring HATEOAS: Quick start](#)
- [Código fuente del ejemplo](#)

7. Conclusiones.

En este tutorial hemos visto que, aunque REST destaca por su sencillez, es necesario que tengamos presentes ciertos principios de diseño a la hora de modelar nuestras APIs. HATEOAS es uno de los principios que debemos seguir.

Gracias al principio HATEOAS facilitamos el descubrimiento de los recursos que componen nuestra API delegando en el servidor la manera en la que enlazaremos con ellos, una gran ventaja a la hora de mantener diferentes versiones de nuestra API y

evitando problemas a nuestros clientes a la hora de solicitar recursos

Desarrollo de microservicios con Spring Boot y Docker

junio 7, 2016 Jorge Pacheco Mengual 5 comentarios Tutoriales 2377 visitas

Siguiendo con la serie de tutoriales dedicados a Docker, vamos a ver cómo desplegar un microservicio desarrollado con Spring Boot en un contenedor Docker. Posteriormente veremos como escalar y balancear este microservicio a través de HAProxy.

0. Índice de contenidos

- [1. Introducción](#)
- [2. Entorno](#)
- [3. El microservicio](#)
- [4. Dockerizar el microservicio](#)
- [5. Escalando la solución](#)
- [6. Conclusiones](#)
- [7. Referencias](#)

1. Introducción

En el tutorial [Introducción a los microservicios](#) del gran José Luis vimos una introducción al concepto de microservicios, cuales son sus ventajas e inconvenientes y cuando podemos utilizarlos.

El objetivo que perseguimos con el presente tutorial es desarrollar un microservicio con Spring Boot, empaquetarlo dentro de una imagen Docker, dentro de la fase de construcción de maven, y una vez podamos levantarlo, ver una posibilidad de escalabilidad gracias a **Docker Compose** y **HAProxy**.

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: MacBook Pro 15' (2.3 GHz Intel Core i7, 16GB DDR3 SDRAM)

- Sistema Operativo: Mac OS X El Capitan 10.11
- Software: Docker 1.11.1, Docker Machine 0.7.0, Docker Compose 1.7.1
- Software: Spring Boot 1.4.0.M3

3. El Microservicio

El objetivo del tutorial no es tanto el desarrollo de microservicios con Spring Boot, sino su empaquetamiento y despliegue, por tanto vamos a implementar un microservicio 'tonto' cuya única funcionalidad es devolvernos un mensaje de hola.

El código lo podéis encontrar en mi cuenta de github [aquí](#), lo primero que deberíamos implementar es un **@RestController** como el que describimos a continuación:

```

1 package com.autentia;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 public class MicroServiceController {
9
10
11     private final AddressService service;
12
13     @Autowired
14     public MicroServiceController(AddressService service) {
15         this.service = service;
16     }
17
18     @RequestMapping(value = "/micro-service")
19     public String hello() throws Exception {
20
21         String serverAddress = service.getServerAddress();
22         return new StringBuilder().append("Hello from IP address: ").append(serverAddress).toString();
23     }
24
25
26 }
```

Como podemos observar es un ejemplo muy sencillo, hemos declarado un controlador rest, al cual le hemos inyectado un servicio que recupera la IP del servidor, y devuelve un string del tipo **"Hello from, IP address xx.xx.xx.xx"**

La clase principal de nuestro microservicio encargada levantar un tomcat embebido con nuestro microservicio tendría un aspecto parecido a este:

```

1 package com.autentia;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class MicroServiceSpringBootApplication {
8
9     public static void main(String[] args) {
10
11         SpringApplication.run(MicroServiceSpringBootApplication.class, args);
12     }
13 }
```

```
1 mvn clean spring-boot run
```

Una vez levantado el microservicio podemos invocarlo de la siguiente manera:

```
jorgepacheco at MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker
$ curl http://localhost:8080/micro-service
Hello from IP address: 192.168.168.68
```

4. Dockerizar el microservicio

Antes de meternos de lleno en el uso de este plugin, vamos a generar un Dockerfile de nuestro microservicio, para ello nos creamos un directorio **src/main/docker** y creamos nuestro Dockerfile de la siguiente manera:

- ```
1 FROM frovlad/alpine-oraclejdk8:slim
2 MAINTAINER jpacheco@autentia.com
3 ADD micro-service-spring-boot-0.0.1-SNAPSHOT.jar app.jar
4 EXPOSE 8080
5 ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

## Repasemos el Dockerfile:

- **FROM:** Tomamos como imagen base [frolvlad/alpine-oraclejdk8](#) esta imagen está basada en Alpine Linux que es una distribución Linux de sólo 5 MB, a la cual se le ha añadido la OracleJDK 8.
- **ADD:** Le estamos indicando que copie el fichero micro-service-spring-boot-0.0.1-SNAPSHOT.jar al contenedor con el nombre app.jar
- **EXPOSE:** Exponemos el puerto 8080 hacia fuera (es el puerto por defecto en el que escuchará el tomcat embebido de nuestro microservicio)

- **ENTRYPOINT:** Le indicamos el comando a ejecutar cuando se levante el contenedor, como podemos ver es la ejecución de nuestro jar

El siguiente paso es añadir el plugin a nuestro pom.xml de la siguiente manera

```

1 <properties>
2 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3 <java.version>1.8</java.version>
4 <docker.image.prefix>autentia</docker.image.prefix>
5 </properties>
6
7
8 <plugins>
9 <plugin>
10 <groupId>org.springframework.boot</groupId>
11 <artifactId>spring-boot-maven-plugin</artifactId>
12 </plugin>
13 <plugin>
14 <groupId>com.spotify</groupId>
15 <artifactId>docker-maven-plugin</artifactId>
16 <version>0.4.10</version>
17 <configuration>
18 <imageName>${docker.image.prefix}/${project.artifactId}</imageName>
19 <dockerDirectory>src/main/docker</dockerDirectory>
20 <serverId>docker-hub</serverId>
21 <registryUrl>https://index.docker.io/v1/</registryUrl>
22 <forceTags>true</forceTags>
23 <imageTags>
24 <imageTag>${project.version}</imageTag>
25 </imageTags>
26 <resources>
27 <resource>
28 <targetPath></targetPath>
29 <directory>${project.build.directory}</directory>
30 <include>${project.build.finalName}.jar</include>
31 </resource>
32 </resources>
33 </configuration>
34 <executions>
35 <execution>
36 <id>build-image</id>
37 <phase>package</phase>
38 <goals>
39 <goal>build</goal>
40 </goals>
41 </execution>
42 <execution>
43 <id>push-image</id>
44 <phase>install</phase>
45 <goals>
46 <goal>push</goal>
47 </goals>
48 <configuration>
49 <imageName>${docker.image.prefix}/${project.artifactId}:${project.version}</imageName>
50 </configuration>
51 </execution>
52 </executions>
53 </plugin>

```

En el apartado **properties** definimos:

- **docker.image.prefix:** que indica el prefijo de la imagen a generar

En el apartado **configuration** de la sección de plugins definimos los siguiente parámetros :

- **imageName:** Nombre de la imagen (prefijo + artifactId del proyecto)

- **dockerDirectory:** Directorio en el que se encuentra el Dockerfile definido anteriormente
- **serverId:** Identificador del registry de Docker (opcional: si queremos realizar un docker push a nuestro registry)
- **registryUrl:** URL del registry de Docker (opcional: si queremos realizar un docker push a nuestro registry)
- **imageTag:** Definimos las tags de nuestra imagen
- **resource:** Le indicamos el recurso que vamos a empaquetar dentro de la imagen ('targetPath' path base de los recursos, 'directory' directorio de los recursos, 'include' incluimos el jar resultante )

En el apartado **executions** vinculamos los goals del plugin a las fases de maven:

- **build-image:** Vinculamos a la fase package de maven, el goal docker:build que construye la imagen con el microservicio
- **build-image:** Vinculamos a la fase de install de maven, el goal docker:push que sube nuestra imagen al registro de docker

Una vez configurado podemos ejecutar alguno de los goals del plugin:

#### 1 mvn clean package

```
[INFO] Building image jpacheco/micro-service-spring-boot
Step 1 : FROM frolvlad/alpine-oraclejdk8:slim
--> 7f321b30ec66
Step 2 : MAINTAINER jpacheco@autentia.com
--> Running in afea6b316da8
--> ac603658e8ce
Removing intermediate container afea6b316da8
Step 3 : ADD micro-service-spring-boot-0.0.1-SNAPSHOT.jar app.jar
--> 0ec77aba1beb
Removing intermediate container 701720a3929a
Step 4 : EXPOSE 8080
--> Running in cc4e8310e47b
--> 196f7135ca0d
Removing intermediate container cc4e8310e47b
Step 5 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom -jar /app.jar
--> Running in 77cec2b3a7d3
--> 635454a48c9c
Removing intermediate container 77cec2b3a7d3
Successfully built 635454a48c9c
[INFO] Built jpacheco/micro-service-spring-boot
[INFO] Tagging jpacheco/micro-service-spring-boot with 0.0.1-SNAPSHOT
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 10.729 s
[INFO] Finished at: 2016-05-30T17:09:03+02:00
[INFO] Final Memory: 40M/375M
[INFO]
```

Como se puede ver en los logs, después de realizar el empaquetado se construye la imagen. Comprobamos si se han generado la imagen:

#### 1 docker images

```
jorgepacheco at MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker
$ docker images
```

| REPOSITORY                         | TAG            | IMAGE ID     | CREATED       | SIZE     |
|------------------------------------|----------------|--------------|---------------|----------|
| jpacheco/micro-service-spring-boot | 0.0.1-SNAPSHOT | 635454a48c9c | 3 minutes ago | 181.3 MB |
| jpacheco/micro-service-spring-boot | latest         | 635454a48c9c | 3 minutes ago | 181.3 MB |
| frolvlad/alpine-oraclejdk8         | slim           | 7f321b30ec66 | 3 days ago    | 167.4 MB |

```
jorgepacheco at MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker
$
```

Podemos observar que en nuestro registro local, están disponibles tanto la imagen base de la que hemos partido **frolvlad/alpine-oraclejdk8:slim**, como nuestra imagen con los tags **0.0.1-SNAPSHOT** y **latest**. El siguiente paso es arrancar un contenedor a partir de nuestra imagen

1 `docker run -d -p 8080:8080 --name microservicio jpacheco/micro-service-spring-boot:0.0.1-SNAPSHOT`

Con esto arrancamos nuestro contenedor, podemos comprobarlo ejecutando

1 docker ps

```
jorgepacheco@jorgepacheco:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
jorgepacheco@jorgepacheco:~$
```

Una vez levantado el contenedor accedemos a nuestro servicio de manera análoga a la anterior sustituyendo 'localhost' por la IP de nuestro docker-machine

1 curl http://192.168.99.100:8080/micro-service

```
jorgepacheco@MacBook-Pro-de-Jorge:~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker$ curl http://192.168.99.100:8080/micro-service
Hello from IP address: 172.17.0.2
jorgepacheco@MacBook-Pro-de-Jorge:~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker$
```

Podemos observar que la IP que devuelve es la IP interna del contenedor 172.17.0.2.

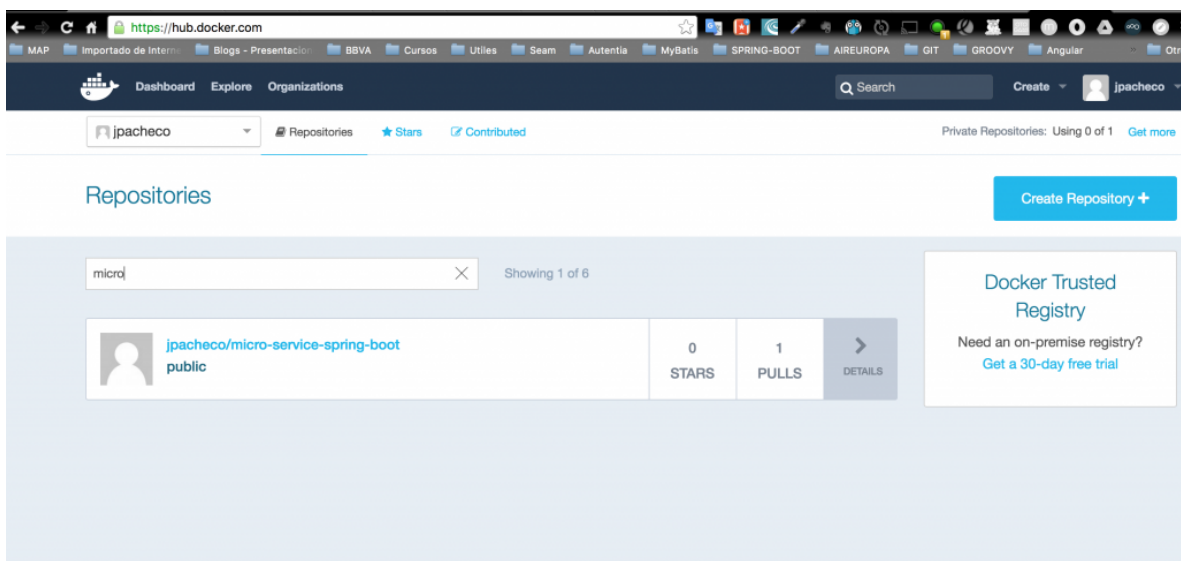
El último paso que nos quedaría para completar el ciclo sería realizar el 'push' de nuestra imagen a nuestro docker registry (es este ejemplo usaremos docker-hub, como hemos definido en el pom.xml con los parámetros: serverId, registryUrl), nos faltaría añadir nuestras credenciales de docker-hub en el settings.xml de maven

```
1 <server>
2 <id>docker-hub</id>
3 <username>myuser</username>
4 <password>mypassword</password>
5 <configuration>
6 <email>user@company.com</email>
7 </configuration>
8 </server>
```

Ya estamos listos para realizar un 'push' de nuestra imagen. Recordar que hemos vinculado el push a la tarea maven 'install'

1 mvn install

Podemos acceder a nuestra cuenta de docker-hub y comprobar que se ha creado nuestra imagen



La cosa se empieza a poner interesante ... ya tenemos nuestro microservicio empaquetado en un contenedor y disponible en nuestro registry

## 5. Escalando la solución

El siguiente paso que vamos a estudiar, es como podemos lanzar varias instancias de nuestro microservicio y como podemos balancear el trafico entre ellas.

Para esto vamos a usar **HAProxy** que es una herramienta open source que actúa como balanceador de carga (load balancer) ofreciendo alta disponibilidad, balanceo de carga y proxy para comunicaciones TCP y HTTP.

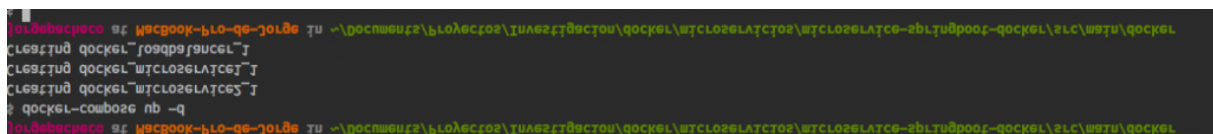
Como no podía ser de otra manera, vamos a usar Docker para levantar el servicio [HAProxy](#) Para ello vamos a usar docker-compose para definir tanto el microservicio, como el balanceador

```
1 microservice1:
2 image: 'jpacheco/micro-service-spring-boot:latest'
3 expose:
4 - "8080"
5 microservice2:
6 image: 'jpacheco/micro-service-spring-boot:latest'
7 expose:
8 - "8080"
9 loadbalancer:
10 image: 'dockercloud/haproxy:latest'
11 links:
12 - microservice1
13 - microservice2
14 ports:
15 - '80:80'
```

Como podemos ver en el fichero docker-compose.yml, hemos definido 2 instancias de nuestro microservicio (microservice1, microservice2) y un balanceador (loadbalancer) con enlaces a los microservicios definidos anteriormente. Lo que conseguimos con esta imagen de HAProxy es exponer el puerto 80 y redirigir a los 2 microservicios expuestos en el 8080 usando una estrategia round-robin. Levantamos nuestro entorno con:

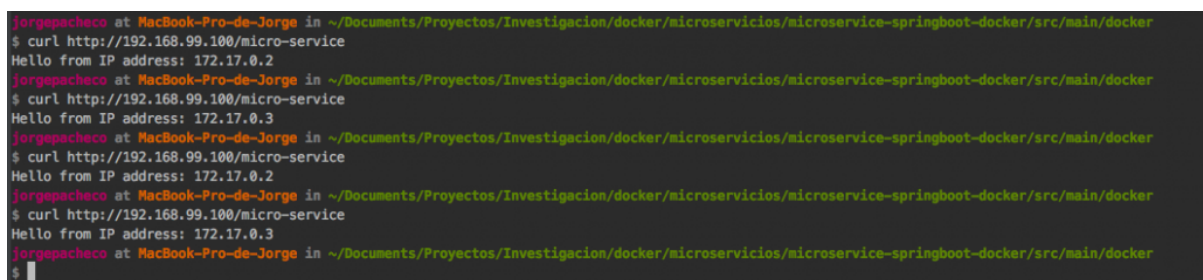
```
1 docker-compose up -d
```

y podemos observar como se levantan los 3 contenedores



```
jorgepacheco@MacBook-Pro-de-Jorge: ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$ docker-compose up -d
Creating microservice1...
Creating microservice2...
Creating loadbalancer...
$ docker-compose ps
$
```

Vamos a invocar a nuestro microservicio a través del balanceador:



```
jorgepacheco@MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$ curl http://192.168.99.100/micro-service
Hello from IP address: 172.17.0.2
jorgepacheco@MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$ curl http://192.168.99.100/micro-service
Hello from IP address: 172.17.0.3
jorgepacheco@MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$ curl http://192.168.99.100/micro-service
Hello from IP address: 172.17.0.2
jorgepacheco@MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$ curl http://192.168.99.100/micro-service
Hello from IP address: 172.17.0.3
jorgepacheco@MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$
```

Como podemos observar en la consola, el balanceador va accediendo cada vez a una instancia del microservicio, logrando el balanceo de carga que íbamos buscando ... No está mal no?, el ejemplo va tomando 'cuerpo'.

Pero.. ¿y si queremos levantar más instancias de nuestro microservicio? ¿Tenemos que modificar el docker-compose.yml, y añadir 'microservice3...microserviceN'?

Revisando la documentación de la imagen [HAProxy](#) encontramos una solución a esta problemática, la idea es levantar una primera instancia del balanceador y del microservicio y posteriormente en función de las necesidades, levantar más instancias del microservicio y que el balanceador se reconfigure para añadirlas. Veamos como quedaría el docker-compose.yml

```
1 version: '2'
2 services:
3 microservice:
4 image: 'jpacheco/micro-service-spring-boot:latest'
5 expose:
6 - '8080'
7 loadbalancer:
8 image: 'dockercloud/haproxy:latest'
9 links:
10 - microservice
11 volumes:
12 - /var/run/docker.sock:/var/run/docker.sock
13 ports:
14 - '80:80'
```

Repasemos las lineas más destacadas:

- **version: '2'** estamos indicando que use la v2 de docker-compose (necesaria para este ejemplo)
- **service:** Tag raíz del que cuelgan nuestros contenedores
- **microservice: loadbalancer** Definición de nuestros contenedores
- **volumes:** El contenedor de HAProxy necesita acceder al 'docker socket' para poder detectar nuevas instancias y reconfigurarse

Vamos a levantar nuestros contenedores:

```
1 docker-compose -f docker-composeV2.yml up -d
```

```
jorgepacheco at MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$ docker-compose -f docker-composeV2.yml up -d
Creating docker_microservice_1
Creating docker_loadbalancer_1
jorgepacheco at MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$
```

Vemos como se han levantado una instancia del balanceador y otra del microservicio. Ahora vamos a escalar nuestro microservicio añadiendo 2 instancias más:

```
1 docker-compose -f docker-composeV2.yml scale microservice=3
```

```
jorgepacheco at MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$ docker-compose -f docker-composeV2.yml scale microservice=3
Creating docker_microservice_2
Creating docker_microservice_3
jorgepacheco at MacBook-Pro-de-Jorge in ~/Documents/Proyectos/Investigacion/docker/microservicios/microservice-springboot-docker/src/main/docker
$
```

Comprobamos que se han creado 2 nuevas instancias de nuestro microservicio, ahora vamos a probar que estas instancias se han añadido al balanceador:



```

$ curl -X GET http://localhost:8080/api/v1/health
{"status": "UP", "timestamp": "2017-07-07T10:10:10.101Z"}
$ curl -X GET http://localhost:8080/api/v1/health
{"status": "UP", "timestamp": "2017-07-07T10:10:10.101Z"}
$ curl -X GET http://localhost:8080/api/v1/health
{"status": "UP", "timestamp": "2017-07-07T10:10:10.101Z"}
$ curl -X GET http://localhost:8080/api/v1/health
{"status": "UP", "timestamp": "2017-07-07T10:10:10.101Z"}
$ curl -X GET http://localhost:8080/api/v1/health
{"status": "UP", "timestamp": "2017-07-07T10:10:10.101Z"}

```

Como podemos ver, cada petición es atendida por una instancia distinta .... podríamos ir añadiendo instancias según vayamos necesitando, bastaría con ejecutar **docker-compose -f docker-composeV2.yml scale microservice=<Instancias\_vivas+Nuevas>**

## 6. Conclusiones

Como hemos podido ver a lo largo del tutorial, la combinación de Spring Boot y Docker nos permite desarrollar fácilmente microservicios, incluso montar infraestructuras que permitan su escalabilidad. El siguiente paso sería investigar la posibilidad de escalarlo a través de un cluster de máquinas usando herramientas como **Docker Swarn** o **Kubernetes**, pero eso os

lo dejo a vosotros

Un saludo.