Construyendo Microservicios Usando Spring Boot y Docker - Parte 1

por Shamik Mitra MVB · **24 y 18 de febrero · Zona de microservicios**

Conozca los beneficios y principios de la arquitectura de microservicios para la empresa

En este tutorial, le mostraré cómo construir microservicios usando Spring Boot y sus diferentes componentes, y en la última sección, le mostraré la implementación usando contenedores Docker.

Aprenderemos sobre:

1. Implementando diferentes componentes de Microservicios.

2. Despliegue de servicios mediante contenedorización.

Antes de comenzar, solo tocaré base en los componentes importantes de la arquitectura de microservicio.

Al implementar Microservicios, los siguientes componentes son el pilar de la arquitectura:

1. Servidores de configuración : para mantener el archivo de Propiedades centralizado y compartido por todos los Microservicios, crearemos un servidor de configuración que es a su vez un Microservicio, y administra todos los archivos de propiedades de microservicios y esos archivos están controlados por

microservicios y esos archivos están controlados por versiones; cualquier cambio en las propiedades se publicará automáticamente en todos los microservicios sin reiniciar los servicios. Una cosa para recordar es que cada microservicio se comunica con el servidor de configuración para obtener valores de propiedades, por lo que el servidor de configuración debe ser un componente altamente disponible; si falla, todos los microservicios fallan porque no puede resolver los valores de las propiedades. Por lo tanto, debemos ocuparnos de la situación: el servidor de configuración no debe ser un SPF (punto único de falla), por lo que activaremos más de un contenedor para el servidor de configuración.

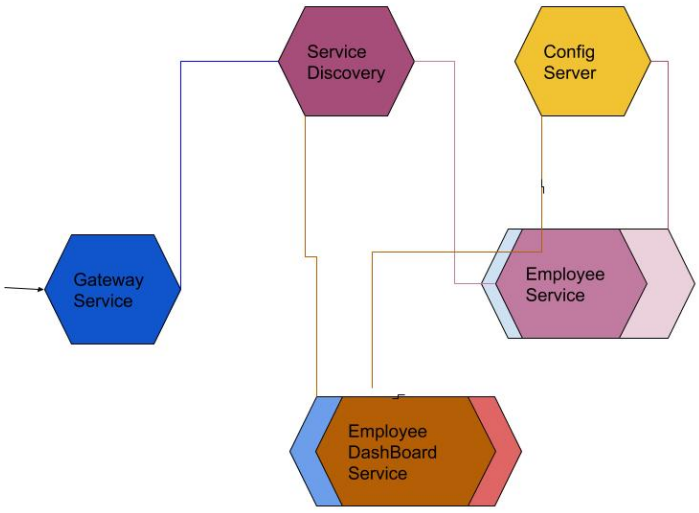
2. **Servidor Eureka Discovery**: El objetivo principal de Microservices es descentralizar los diferentes componentes en función de las características del negocio, de modo que cada componente, también conocido como microservicio, pueda escalarse según las necesidades, por lo que para un microservicio particular, hay múltiples instancias y podemos agregar y eliminar instancias como según la necesidad, entonces la forma en que los monolitos cargan el blanqueamiento no va a funcionar en un paradigma de microservicio. Como genera contenedores sobre la marcha, los contenedores tienen direcciones IP dinámicas, por lo que para rastrear todas las instancias de un servicio, se necesitará un servicio de administrador, de modo que cuando los contenedores se generan, se registra ante el administrador y el administrador mantiene la pista de las instancias; si se elimina un servicio, el administrador lo elimina del registro de servicios del administrador. Si otros servicios necesitan comunicarse entre ellos, se pone en contacto con un servicio de descubrimiento para obtener la instancia de otro servicio. De nuevo, este es un componente altamente disponible; si el servicio de descubrimiento está inactivo, los microservicios no pueden comunicarse entre sí, por lo que el servicio de descubrimiento debe tener varias instancias.

3. **Componentes, también conocidos como servicios** : los componentes son los ingredientes clave en la arquitectura de Microservicio. Por

componente, me refiero a una utilidad o función comercial que puede administrarse o actualizarse de manera independiente. Tiene un límite predefinido y expone una API mediante la cual otros componentes pueden comunicarse con este servicio. La idea de los microservicios es descomponer una funcionalidad comercial completa en varias características independientes pequeñas que se comunicarán entre sí para producir la funcionalidad comercial total. Si alguna parte de la funcionalidad cambia en el futuro, podemos actualizar o eliminar ese componente y agregar un nuevo componente a la arquitectura. Por lo tanto, la arquitectura de Microservice produce una arquitectura modular adecuada con una encapsulación adecuada y límites bien definidos.

4. **Servicio de puerta de enlace :** un microservicio es una colección de servicios independientes que colectivamente produce una funcionalidad comercial. Cada microservicio publica una API, generalmente una API REST, por lo que, como cliente, es engorroso administrar tantas URL de punto final para comunicarse. Además, piense en otra perspectiva: si alguna aplicación desea construir un marco de autenticación o comprobación de seguridad, debe implementarla en todos los servicios, de modo que se repita contra DRY. Si tenemos un servicio Gateway, que está orientado a Internet, el cliente llamará solo a un punto final y delegará la llamada en un microservicio real, y toda la autenticación o verificación de seguridad se realizará en el servicio de puerta de enlace.

Ahora tenemos una comprensión básica de cómo las diferentes partes de un microservicio trabajan juntas. En este tutorial, crearé un servicio de búsqueda de empleados que devolverá información del empleado, un servicio EmployeeDashBoard que invocará el servicio de búsqueda y mostrará los resultados, un servidor Eureka para que estos servicios puedan registrarse y un servicio de puerta de enlace para llegar a estos servicios desde afuera. Luego desplegaremos nuestros servicios en un contenedor acoplable y usaremos DockerCompose para generar los contenedores Docker. Usaré Spring Boot para este tutorial.



Microservice Architecture

Comencemos por construir nuestro proyecto de microservicio, ya que tenemos que crear cinco servicios individuales:

1. Servidor de configuración

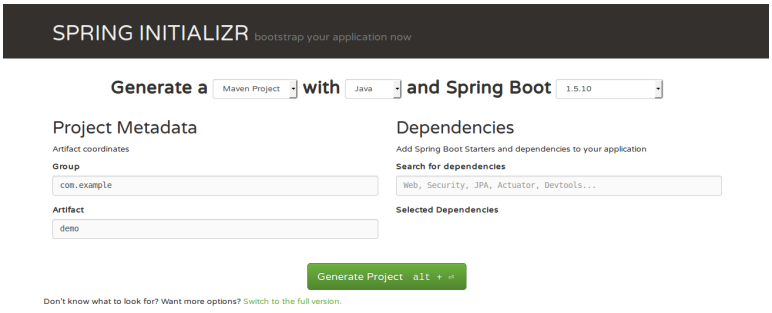
2. Servidor Eureka

3. Servicio del empleado

4. Servicio del tablero de empleados

5. Zuul Proxy

The best place to start is going to http://start.spring.io/., shopping the requires modules, and hitting "generate project."



For this tutorial, we will use Spring Boot 1.5.4.

Creating the Config Server

To create the config server, first we need to check the config server module from starts.spring.io, and also check the actuator to see the endpoints. Then, download the zip file and open it in Eclipse.

The pom file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
xsi:schemaLocation="http://maven.apache.org/POM
<modelVersion>4.0.0</modelVersion>
<groupId>com.example</groupId>
<artifactId>MicroserviceConfigServer</artifactI
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>MicroserviceConfigServer</name>
<description>Demo project for Spring Boot</desc
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifac
<version>1.5.4.RELEASE</version>
<relativePath/>
<!-- lookup parent from repository -->
</parent>
<properties>
<project.build.sourceEncoding>UTF-8</project.bu
<project.reporting.outputEncoding>UTF-8</projec
<java.version>1.8</java.version>
<spring-cloud.version>Dalston.SR1</spring-cloud
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artif
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-config-server</artifac
</dependency>
<dependency>
```

```
35  <dependency>
    <groupId>org.springframework.boot</groupId>
36
    <artifactId>spring-boot-starter-test</artifactI
37
38  <scope>test</scope>
39  </dependency>
40  </dependencies>
41  <dependencyManagement>
42  <dependencies>
43  <dependency>
    <groupId>org.springframework.cloud</groupId>
44
    <artifactId>spring-cloud-dependencies</artifact
45
46  <version>${spring-cloud.version}</version>
47  <type>pom</type>
48  <scope>import</scope>
49  </dependency>
50  </dependencies>
51  </dependencyManagement>
52  <build>
53  <plugins>
54  <plugin>
    <groupId>org.springframework.boot</groupId>
55
    <artifactId>spring-boot-maven-plugin</artifactI
56
57  </plugin>
58  </plugins>
59  </build>
60  </project>
```

It downloads the spring-cloud-config-server artifacts.

Next, we have to create a bootstrap.properties file where we mention from which location the config server reads the property. In production mode, it should be the URL of the Git repository, but as this is a demo, I'll use my local disk. All properties files will be placed there, and the config server reads those property files.

Let's see the **bootstrap.properties** file:

```
1  server.port=9090
   spring.cloud.config.server.native.searchLocatic
2
```

```
SPRING_PROFILES_ACTIVE=native
```

Here, I instruct Spring Boot to spawn the embedded server in port 9090 and use a centralProperties folder as a folder to search all properties files. Note that in our Docker container, you have to create a central properties folder and place all the properties files there.

Now let's see the Java part:

```
package com.example.MicroserviceConfigServer;

import org.springframework.boot.SpringApplicati

import org.springframework.boot.autoconfigure.S

import org.springframework.cloud.config.server.


@EnableConfigServer
@SpringBootApplication
public class ConfigServer {

 public static void main(String[] args) {
  SpringApplication.run(ConfigServer.class, arg

 }
}
```

Here I use the @EnableConfigserver annotation, with which we instruct Spring Boot to consider this service as a config server application.

Now place few test properties files in centralProperties folder.

We are all set for the config server now. If we run this service and hit the http://localhost:9090/config/default URL, we see the following response:

```
{
   "name": "config",
   "profiles": [
      "default"
   ],
   "label": null,
```

```
   "version": null,
   "state": null,
   "propertySources": [
      {
         "name": "file:///home/shamik/MicroServ
         "source": {
            "application.message": "Hello Shami
         }
      },
      {
         "name": "file:///home/shamik/MicroServ
         "source": {
            "welcome.message": "Hello Spring Cl
         }
      }
   ]
}
```

It shows the all file names and key and values I placed in the centralProperties folder.

The next step is to create a Eureka server for service discovery.

Implementing Service Discovery

We will use Netflix's Eureka server for service discovery. To do that I am selecting the Eureka server module from start.spring.io and downloading the project.

The pom.xml looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
xsi:schemaLocation="http://maven.apache.org/POM
<modelVersion>4.0.0</modelVersion>
<groupId>com.example</groupId>
<artifactId>EmployeeEurekaServer</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>EmployeeEurekaServer</name>
<description>Demo project for Spring Boot</desc
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifac
<version>1.5.4.RELEASE</version>
<relativePath/>
<!-- lookup parent from repository -->
</parent>
<properties>
<project.build.sourceEncoding>UTF-8</project.bu
<project.reporting.outputEncoding>UTF-8</projec
<java.version>1.8</java.version>
<spring-cloud.version>Dalston.SR1</spring-cloud
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artif
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-config</artifa
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-eureka-server<
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactI
<scope>test</scope>
</dependency>
```

```
</dependencies>
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifact
<version>${spring-cloud.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactI
</plugin>
</plugins>
</build>
</project>
```

Now create the bootstrap.properties:

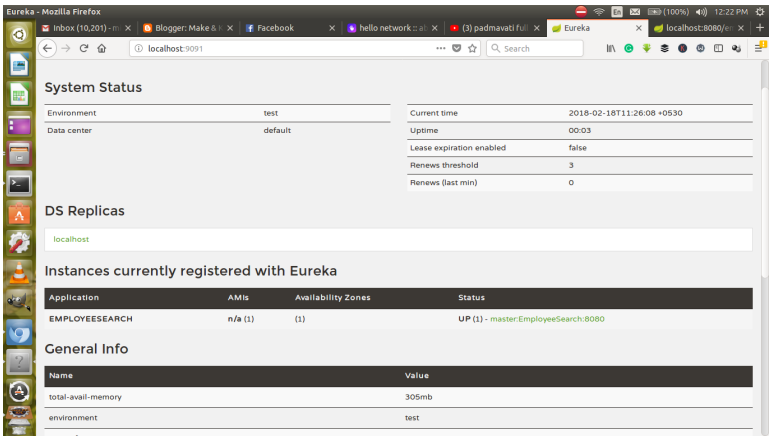
```
spring.application.name=EmployeeEurekaServer
eureka.client.serviceUrl.defaultZone:http://loc
server.port=9091
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Here, I give a logical name **EmployeeEurekaServer** to this application, and the location of the Eureka server is http://localhost:9091; the embedded server will start on port 9091. Please note that the Eureka server itself can be a Eureka client; because there may be multiple instances of Eureka servers, it needs to be in sync with others. With this **eureka.client.register-with-eureka=false,** I explicitly instruct Spring Boot to not treat Eureka server as a client, because I created only one Eureka server, so it does not require to register itself as a client.

Now I will create the Java file:

```
package com.example.EmployeeEurekaServer;

import org.springframework.boot.SpringApplicati
import org.springframework.boot.autoconfigure.S
import org.springframework.cloud.netflix.eureka

@EnableEurekaServer
@SpringBootApplication
public class EmployeeEurekaServerApplication {

 public static void main(String[] args) {
  SpringApplication.run(EmployeeEurekaServerApp

 }
}
```

With the @EnableEurekaServer annotation, Spring Boot spawns this service as Eureka server. We are all set now; if I run the service and hit the http://localhost:9091/ in the browser, we will see the following screen:



In Part 2, we will create the Employee Search and other services, then learn about deploying your microservices with Docker. Stay tuned!

Link to second part: https://dzone.com/articles/building-microservice-using-spring-boot-and-docker?fromrel=true

---

Microservices for the Enterprise eBook: Get Your

Topics: MICROSERVICIOS, TUTORIAL, ARQUITECTURA DE SOFTWARE, ARRANQUE DE PRIMAVERA, ACOPLADOR

Recursos para socios de **microservicios**