

Microservicios con Spring Boot 3 y Spring Cloud

Dockerizar una aplicación	2
Buildpack	2
Subiendo las imágenes a docker hub	3
Docker Compose	3
Crear contenedores con docker compose	3
Ejecutar el docker compose	4
Configuración centralizada en microservicios	4
Generando microservicio config server	5
Actualizar Microservicio Hotels para que tome las configuraciones	6
Configurar application.properties de los microservicios	7
Crear clase de configuración en los microservicios	8
Actualizando imágenes Docker	9
Crear imagen para el microservicio ConfigServer	9
Crear nuevamente las imágenes docker de hotels	10
Subir las imágenes a Docker Hub	10
Actualizando docker-compose.yml	10
Levantar contenedores de imágenes	13
Eureka - Registro y descubrimiento de microservicios	13
Crear proyecto Eureka Server	14
Registrar microservicios en EurekaServer	15
Actualizando imagenes docker	16
Subir las imágenes a docker hub	16
Levantando contenedores con docker compose	16
RestTemplate y Feign para comunicación entre microservicios	17
Consumir microservicios con Feign	19
Spring Cloud Gateway	20
Generando proyecto gateway	21

Dockerizar una aplicación

1) Crear docker file

```
#inicia con la imagen base que contiene Java runtime
FROM openjdk:17-jdk-slim as build

# se agregará el jar del microservicio al contenedor
COPY target/hotels-0.0.1-SNAPSHOT.jar hotels-0.0.1-SNAPSHOT.jar

#se ejecuta el microservicio
ENTRYPOINT ["java", "-jar", "/hotels-0.0.1-SNAPSHOT.jar"]
```

2) docker build . -t jzapana/hotels

3) docker images

```
docker inspect <tres caracteres del id>
```

4) corremos la imagen para crear el contenedor

```
docker run -p 8080:8080 jzapana/hotels
```

```
docker ps (para ver los contenedores que están corriendo)
```

```
docker run -p 8090:8080 jzapana/hotels (inicia dentro de otro contenedor)
```

Buildpack

Es otra opción para crear una imagen de nuestro proyecto, mediante un comando de Maven utilizado en proyectos de Spring Boot que te permite construir una imagen de contenedor Docker a partir de una aplicación de Spring Boot. Este comando es parte del ecosistema de Spring Boot y se utiliza en combinación con Spring Boot's Native Image Support para crear una imagen de contenedor que contiene tu aplicación de Spring Boot.

Maven se encargará de realizar varias tareas, incluyendo:

- Compilar tu aplicación de Spring Boot.
- Empaquetarla en un formato adecuado para ser incluido en una imagen de contenedor Docker.
- Generar un archivo Dockerfile si no lo has proporcionado explícitamente en tu proyecto.
- Invocar Docker para construir la imagen de contenedor.

El resultado final será una imagen de contenedor Docker que contiene tu aplicación de Spring Boot, lista para ser ejecutada en un entorno de contenedor.

Para utilizar buildpack se debe agregar lo siguiente en el pom del proyecto correspondiente

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
```

```

        <configuration>
            <excludes>
                <exclude>
                    <groupId>org.projectlombok</groupId>
                    <artifactId>lombok</artifactId>
                </exclude>
            </excludes>
            <image>
                <name>jzapana/${project.artifactid}</name>
            </image>
        </configuration>
    </plugin>
</plugins>
</build>

```

para generar la imagen se debe correr el siguiente comando, tener en cuenta que docker debe estar funcionando (la segunda opción es para evitar los test):

```

mvn spring-boot:build-image
mvn spring-boot:build-image -DskipTests

```

Luego las imágenes se pueden ver con

```
docker images
```

Iniciamos el microservicio en el puerto que corresponda:

```
docker run -p 8081:8081 jzapana/rooms
```

Subiendo las imágenes a docker hub

```

login -u "jzapana" -p xxxxx
docker push docker.io/jzapana/hotels

```

Docker Compose

Es otra herramienta para gestionar contenedores y crear imágenes. Es útil cuando hay muchos microservicios en la aplicación. Previamente hay que verificar que el docker desktop tenga configurado el docker compose con el comando

```
docker-compose --version
```

También es recomendable verificar si están instaladas las últimas actualizaciones de docker

Crear contenedores con docker compose

Se realiza mediante un archivo yaml, aquí se indican los servicios que son candidatos para crear los contenedores, es decir los que se quieren dockerizar. Para cada servicio se indica la imagen que se va a descargar de docker hub (por ejemplo: jzapana/hotels:latest)

```
services:
  hotels:
    image: jzapana/hotels:latest
    mem_limit: 800m
    ports:
      - "8080:8080"
    networks:
      - jzapana-network

  rooms:
    image: jzapana/rooms:latest
    mem_limit: 800m
    ports:
      - "8081:8081"
    networks:
      - jzapana-network

  reservations:
    image: jzapana/reservations:latest
    mem_limit: 800m
    ports:
      - "8082:8082"
    networks:
      - jzapana-network

networks:
  jzapana-network:
```

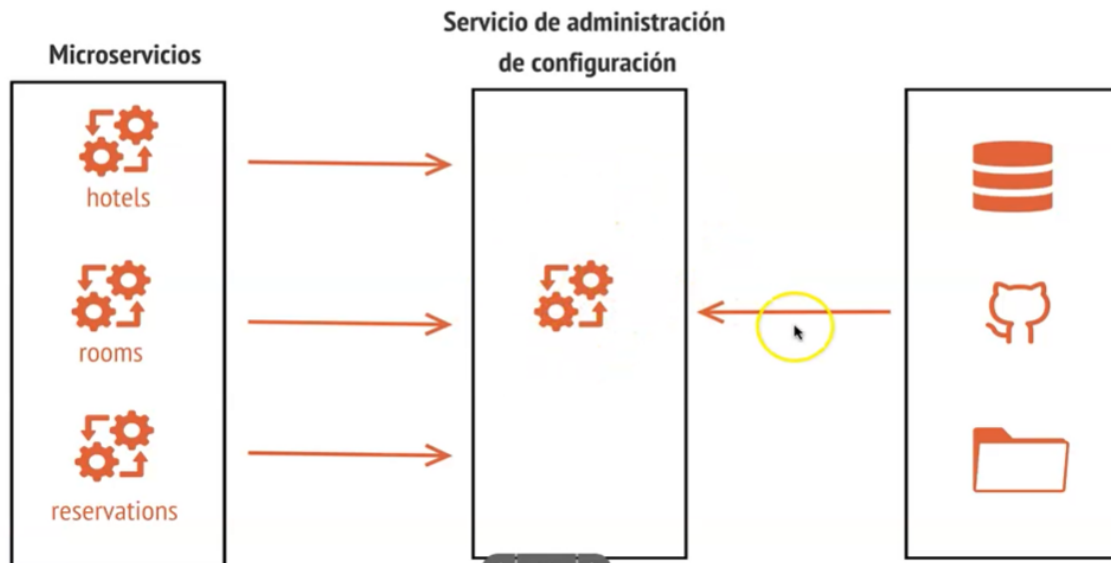
Ejecutar el docker compose

`docker compose up`

Configuración centralizada en microservicios

- Permite desacoplar la configuración de los archivos properties
- Inyectan las configuraciones que el microservicio necesita cuando se levanta en los diferentes ambientes.
- Mantener las configuraciones en un lugar centralizado con distintas versiones

Administración de Configuraciones



Spring Cloud Config

<https://spring.io/projects/spring-cloud>

- Da soporte para externalizar configuraciones en sistemas distribuidos (microservicios)
- Con un servidor de configuraciones tienes un lugar centralizado para administrar configuraciones de aplicaciones, para todos los ambientes.

Generando microservicio config server

- Crear un proyecto denominado configserver
- Agregar las siguientes dependencias
 - Config Server
 - Spring Boot Actuator

The screenshot shows the configuration for a Spring Boot project. The 'Project' section includes 'Language' (Java), 'Spring Boot' (3.0.4), and 'Project Metadata' (Group: com.aleal.onfigserver, Artifact: Config-Server, Name: Config-Server, Description: microservices with config server for configurations, Package name: com.aleal.onfigserver). The 'Dependencies' section shows 'Config Server' (with 'SPRING CLOUD CONFIG' dependency) and 'Spring Boot Actuator' (with 'OPS' dependency). A yellow circle highlights the 'Config Server' dependency.

El archivo Pom debe incluir el tag image tal como se muestra a continuación (al igual que el ms rooms)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <image>
          <name>jzapana/${project.artifactid}</name>
        </image>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Configurar el proyecto

- **Agregar la anotación @EnableConfigserver** en la clase principal del proyecto. Esta anotación permite que el proyecto trabaje como un servidor de configuraciones y permita leer toda la información de configuración desde un repositorio git.
- **Crear otro proyecto en gitlab denominado configserver-ms** que contendrá los archivos de configuración para todo el proyecto, este proyecto debe ser de tipo público (no privado).
- **Configurar el archivo de propiedades** para que permita leer la información desde gitlab (ver documentación en proyecto spring cloud)
spring.application.name=configserver

```
spring.profiles.active=git
#configurar la ubicación del proyecto gitlab
spring.cloud.config.server.git.uri=https://gitlab.com/cu-rso-microservicios2435144/configserver-ms.git
```

```
spring.cloud.config.server.git.clone-on-start=true
spring.cloud.config.server.git.default-label=main
```

```
server.port=8085
```

- Probar proyecto: iniciar el proyecto configserver y probar en el browser <http://localhost:8085/hotels/prod>
 - Puede agregar el plugin JSON Viewer para mejorar chrome

Actualizar Microservicio Hotels para que tome las configuraciones

El objetivo es agregar la configuración de spring cloud a los microservicios que van a consumir los valores que están guardados en configserver-ms.

En los POM de los microservicios que van a consumir la configuración se debe agregar la misma versión de spring cloud que figura en el microservicio configserver-ms, esto se agrega debajo de la versión de java de cada microservicio, es decir:

```
<properties>
  <java.version>17</java.version>
  <spring-cloud.version>2022.0.4</spring-cloud.version>
</properties>
```

También se debe copiar toda la sección de las dependency-management, se debe pegar debajo de dependencies:

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

Los microservicios también tienen que incluir la dependencia de spring cloud:

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

Configurar application.properties de los microservicios

Finalmente resta que los microservicios lean las configuraciones, para ello agregamos esta configuración en el application.properties de los microservicios.

Optional: significa que si está caído el ms de configuración que NO impida que los demás microservicios funcionen.

```
spring.application.name=hotels
```

```
#ambiente, en este caso dev
spring.profiles.active=dev
```

```
#nombre obtenido de la clave spring.application.name del configserver-ms
spring.config.import=optional:configserver:http://localhost:8085
```

Crear clase de configuración en los microservicios

```
package com.reservahotel.hotels.config;
import lombok.Data;
/**
 * Configuración para consumir las properties que empiecen con "hotels"
 * @author jzapana
 *
 */
@Configuration
@ConfigurationProperties(prefix="hotels")
@Data
public class HotelsServiceConfiguration {

    private String msg;
    private String buildVersion;
    private Map<String,String>mailDetails;

}
```

Se define un modelo para leer las propiedades

```
package com.reservahotel.hotels.model;
import java.util.Map;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;

/**
 *
 * @author jzapana
 */
@Getter
@Setter
@AllArgsConstructor
public class PropertiesHotels {
    private String msg;
    private String buildVersion;
    private Map<String, String> mailDetail;
}
```

y finalmente se agrega un endpoint en el controlador de hoteles para realizar pruebas, para esto se inyecta una instancia de la configuración.

```
@Autowired
private HotelsServiceConfiguration configHotels;
```



```

/**
 * EndPoint de prueba para leer las properties del microservicio
 * configserver-ms
 *
 * @return
 * @throws JsonProcessingException
 */
@GetMapping("/hotels/read/properties")
public String getPropertiesHotels()throws JsonProcessingException{
    ObjectWriter owj = new ObjectMapper().writer().withDefaultPrettyPrinter();
    PropertiesHotels propHotels = new PropertiesHotels(configHotels.getMsg(),
configHotels.getBuildVersion(),configHotels.getMailDetails());

    String jsonString = owj.writeValueAsString(propHotels);
    return jsonString;
}

```

Resultado del endpoint

```

localhost:8080/hotels/read/properties
1 // 20231106202334
2 // http://localhost:8080/hotels/read/properties
3
4 {
5   "msg": "Welcome to the Reservations Hotels applications",
6   "buildVersion": "1",
7   "mailDetail": {
8     "hostName": "dev-alfredo@gmail.com",
9     "port": "5400",
10    "from": "dev-alfredo@gmail.com",
11    "subject": "Your Reservations is ready"
12  }
13 }

```

Actualizando imágenes Docker

Crear imagen para el microservicio ConfigServer

previamente hay que corregir el nombre del artifact-id de este microservicio ya que no debe tener guiones ni mayúsculas, es decir que debe quedar del siguiente modo:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://maven.apache.org/POM/4.0.0" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.5</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.reservahotel.configserver</groupId>
  <artifactId>configserver</artifactId>
  <packaging>jar</packaging>
  <name>configserver</name>
  <description>configserver</description>
  <url>http://maven.apache.org</url>
  <licenses>
    <license>
      <name>Apache License, Version 2.0</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0</url>
    </license>
  </licenses>
  <developers>
    <developer>
      <name>Jzapana</name>
    </developer>
  </developers>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

para dockerizar ejecutamos el comando:

```
mvn spring-boot:build-image
```

Crear nuevamente las imágenes docker de hotels

Crear el jar y construir las imágenes con build pack

```
mvn spring-boot:build-image
```

```
mvn clean install
docker build . -t jzapana/hotels
```

Subir las imágenes a Docker Hub

```
docker push docker.io/jzapana/hotels
docker push docker.io/jzapana/rooms
docker push docker.io/jzapana/reservations
docker push docker.io/jzapana/configserver
```

Esto es necesario porque docker-compose.yml baja la última versión de estas imágenes y empieza a construir los contenedores.

Actualizando docker-compose.yml

El docker compose fue creado en el microservicio de reservations para generar las imágenes para luego levantar el contenedor y poder utilizarlas.

```
services:
  configserver:
    image: jzapana/configserver:latest
```

```
mem_limit: 800m
ports:
  - "8085:8085"
networks:
  - jzapana-network
```

eureka-server:

```
image: jzapana/eureka-server:latest
mem_limit: 800m
ports:
  - "8065:8065"
networks:
  - jzapana-network
depends_on:
  - config-server
deploy:
  restart_policy:
    condition: on-failure
    delay: 5s
    max_attempts: 3
    window: 120s
environment:
  SPRING_PROFILES_ACTIVE: default
  SPRING_CONFIG_IMPORT: config-server:http://config-server:8085/
```

hotels:

```
image: jzapana/hotels:latest
mem_limit: 800m
ports:
  - "8080:8080"
networks:
  - jzapana-network
depends_on:
  - config-server
  - eureka-server
deploy:
  restart_policy:
    condition: on-failure
    delay: 5s
    max_attempts: 3
    window: 120s
environment:
  SPRING_PROFILES_ACTIVE: default
  SPRING_CONFIG_IMPORT: config-server:http://config-server:8085/
  EUREKA_CLIENT_SERVICEURL_DEFAULTZONE:
http://eureka-server:8065/eureka/
```

```
rooms:
  image: jzapana/rooms:latest
  mem_limit: 800m
  ports:
    - "8081:8081"
  networks:
    - jzapana-network
  depends_on:
    - configserver
    - eurekaserver
  deploy:
    restart_policy:
      condition: on-failure
      delay: 5s
      max_attempts: 3
      window: 120s
  environment:
    SPRING_PROFILES_ACTIVE: default
    SPRING_CONFIG_IMPORT: configserver:http://configserver:8085/
    EUREKA_CLIENT_SERVICEURL_DEFAULTZONE:
http://eurekaserver:8065/eureka/
```

```
reservations:
  image: jzapana/reservations:latest
  mem_limit: 800m
  ports:
    - "8082:8082"
  networks:
    - jzapana-network
  depends_on:
    - configserver
    - eurekaserver
  deploy:
    restart_policy:
      condition: on-failure
      delay: 5s
      max_attempts: 3
      window: 120s
  environment:
    SPRING_PROFILES_ACTIVE: default
    SPRING_CONFIG_IMPORT: configserver:http://configserver:8085/
    EUREKA_CLIENT_SERVICEURL_DEFAULTZONE:
http://eurekaserver:8065/eureka/
```

```
networks:
```

jzapana-network:

Levantar contenedores de imágenes

```
docker compose up
```

con docker ps puedo ver los contenedores creados.

Eureka - Registro y descubrimiento de microservicios

Respecto de spring cloud:

- El servicio de eureka permite actuar como agente de descubrimiento y registro de microservicios
- La librería de spring cloud balancer para la implementación del load balancing
- Feign para la comunicación con otros microservicios. Otra opción es RestTemplate

Netflix ha sido un pionero en la implementación de microservicios ya que spring cloud tomó todos estos conceptos de los desarrolladores de netflix.

Eureka es un componente de Spring Cloud utilizado para la implementación de un servicio de registro y descubrimiento en arquitecturas de microservicios. Se basa en el patrón de registro de servicios y permite a los microservicios registrarse a sí mismos y descubrir otros servicios en la red.

Básicamente, Eureka proporciona una manera de que los servicios se registren a sí mismos con un servidor central (llamado servidor Eureka) durante su inicio. Esto permite que otros servicios en la red se encuentren y se comuniquen con estos servicios registrados. De esta manera, facilita la comunicación entre microservicios sin la necesidad de conocer sus ubicaciones físicas.

Las principales características de Eureka incluyen:

Registro de Servicios: Los servicios pueden registrarse en el servidor Eureka durante su inicio. Esto incluye información sobre su ubicación (host, puerto, etc.) y otros metadatos.

Descubrimiento de Servicios: Los servicios que necesitan interactuar con otros servicios pueden consultar el servidor Eureka para descubrir la ubicación (dirección, puerto, etc.) de esos servicios. Esto se hace utilizando el nombre lógico del servicio registrado en el servidor Eureka.

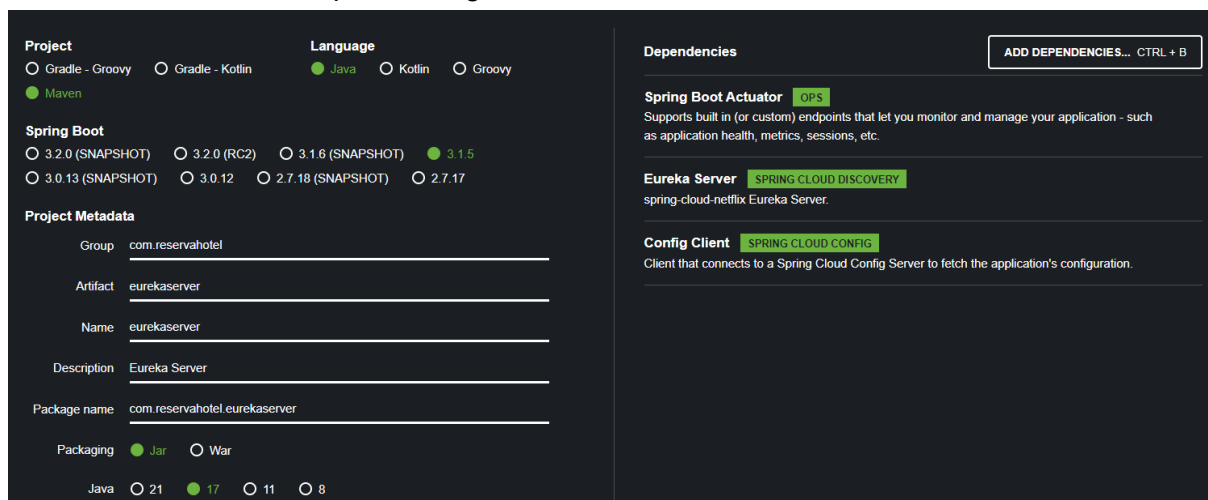
Balaneo de Carga y Resiliencia: Eureka puede manejar el balanceo de carga y la resiliencia al permitir que los servicios registren múltiples instancias de sí mismos. Además,

Eureka supervisa constantemente el estado de los servicios registrados y elimina automáticamente las instancias que no responden, mejorando así la resiliencia del sistema.

Eureka se integra bien con otros componentes de Spring Cloud, como Ribbon (para el balanceo de carga del lado del cliente) y Zuul (para enrutamiento y proxy). En resumen, Eureka simplifica el desarrollo y la gestión de aplicaciones basadas en microservicios al proporcionar un mecanismo para el registro y descubrimiento de servicios en un entorno distribuido.

Crear proyecto Eureka Server

El objetivo de este microservicio es que funcione como un servidor de Eureka para que los demás microservicios se puedan registrar allí



agregar en el pom lo relacionado al build pack y generar la imagen del proyecto:

```
<configuration>
  <image>
    <name>jzapana/${project.artifactId}</name>
  </image>
</configuration>
```

También se debe agregar la anotación `@EnableEurekaServer` en la clase principal del proyecto:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaserverApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaserverApplication.class, args);
    }

}
```

Configurar las siguientes propiedades:

```
spring.application.name=eureka-server  
spring.config.import=optional:configserver:http://localhost:8085
```

Finalmente en el repositorio de configuraciones agregar un archivo de propiedades denominado eureka-server.properties que contenga lo siguiente

```
server.port=8065  
eureka.instance.hostname=localhost  
eureka.client.registerWithEureka=false  
eureka.client.fetchRegistry=false  
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:  
${server.port}/eureka/
```

Para probar el servicio debemos iniciar los proyectos config-server y luego eureka-server en la url <http://localhost:8065/>

Registrar microservicios en EurekaServer

En el pom.xml del microservicio **hoteles** agregar la dependencia que permita a los microservicios clientes, registrarse en el servidor de eureka

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

También es necesario agregar las siguientes propiedades en este microservicio, las mismas tienen que ver con la dependencia de actuator y permiten conocer información del servicio en eureka.

```
eureka.instance.preferIpAddress=true  
eureka.client.registerWithEureka=true  
eureka.client.fetchRegistry=true  
eureka.client.serviceUrl.defaultZone=http://localhost:8065/eureka/
```

```
management.endpoints.web.exposure.include=*
```

```
#información para consultar por actuator  
info.app.name=Hotels microservice  
info.app.description=Application to reserve an rooms  
info.app.version=1.0.0
```

```
management.info.env.enabled=true
```

```
endpoint.shutdown.enabled=true
management.endpoint.shutdown.enabled=true
```

Actualizando imagenes docker

Previamente eliminar todas las imágenes que existen y luego crear las nuevas imágenes

Hotels:

```
mvn clean install "-Dmaven.test.skip=true"
docker build . -t jzapana/hotels
```

Reservations: creado con buildpack

```
mvn spring-boot:build-image "-Dmaven.test.skip=true"
```

Rooms: creado con buildpack

```
mvn spring-boot:build-image "-Dmaven.test.skip=true"
```

Eureka Server

```
mvn spring-boot:build-image "-Dmaven.test.skip=true"
```

Importante: Si fallan estos comando se debe verificar que los archivos se encuentre utilizando el encoding UTF-8 y de ser necesario agregar este plugin al pom (sección build plugins)

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-resources-plugin</artifactId>
  <version>3.1.0</version>
</plugin>
```

Subir las imágenes a docker hub

Ejecutar estos comandos

```
docker push docker.io/jzapana/hotels
docker push docker.io/jzapana/reservations
docker push docker.io/jzapana/rooms
docker push docker.io/jzapana/eureka-server
```

Levantando contenedores con docker compose

ver archivos docker compose en las carpetas default, developer y production del proyecto reservation.

Se debe ejecutar el que se encuentra en la carpeta default para levantar los contenedores

```
docker compose up
```


si alguno de los contenedores no levanta se puede iniciarlo en forma manual desde docker desktop

RestTemplate y Feign para comunicación entre microservicios

Ejemplo: devolver todas las habitaciones de un hotel.

Microservicio rooms: implementar el endpoint que permita buscar las habitaciones de un hotel por su id:

```
/**
 * Devuelve las habitaciones de un hotel determinado
 * @param id
 * @return
 */
@GetMapping("rooms/{id}")
public List<Room> searchByHotelId(@PathVariable long id){

    return (List<Room>) this.service.searchRoomByHotelId(id);
}
```

Microservicio hotels: consumir el endpoint anterior, para lo cuál utilizamos RestTemplate. Primero creamos una clase de configuración

```
package com.reservahotel.hotels.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {

    /**
     * Bean para devolver una instancia de RestTemplate
     * @return
     */
    @Bean("clientRest")
    public RestTemplate registreRestTemplate(){
        return new RestTemplate();
    }
}
```

Luego creamos el método y todo lo necesario para consumir el servicio del microservicio de rooms.

Primero se crea el modelo room

```
package com.reservahotel.hotels.model;
import lombok.Data;

@Data
public class Room {
    private long roomId;
    private long hotelId;
    private String roomName;
    private String roomAvailable;
}
```

generamos la respuesta del modelo

```
package com.reservahotel.hotels.model;
import lombok.Data;
import java.util.List;

/**
 * Dominio para devolver la lista de habitaciones
 * obtenidas del microservicio rooms mediante RestTemplate
 */
@Data
public class HotelRooms {
    private long hotelId;
    private String hotelName;
    private String hotelAddress;
    private List<Room> rooms;
}
```

Implementamos el método HotelRooms searchHotelById en el servicio de hotels, primero inyectamos el Bean clientRest

```
@Autowired
private RestTemplate clientRest;

@Override
public HotelRooms searchHotelById(long hotelId) {
    HotelRooms response = new HotelRooms();
    Optional<Hotel> hotel = hotelDao.findById(hotelId);
    Map<String, Long> pathVariable = new HashMap<String, Long>();
    pathVariable.put("id", hotelId);
    List<Room> rooms =
Arrays.asList(clientRest.getForObject("http://localhost:8081/rooms/{id}",
Room[].class, pathVariable));
    response.setHotelId(hotel.get().getHotelId());
}
```

```

        response.setHotelName(hotel.get().getHotelName());
        response.setHotelAddress(hotel.get().getHotelAddress());
        response.setRooms(rooms);
        return response;
    }

```

Finalmente creamos el endpoint en el controlador de hoteles

```

/**
 * Consumiendo las habitaciones de un hotel determinado extraído del
 * microservicio de rooms
 * @param id
 * @return
 */
@GetMapping("hotels/{id}")
public HotelRooms searchHotelById(@PathVariable long id){
    return this.service.searchHotelById(id);
}

```

Consumir microservicios con Feign

Feign es una librería de netflix y se encuentra en el proyecto spring cloud. El inconveniente de la implementación con RestTemplate es que se produce un fuerte acoplamiento entre los microservicios involucrados. Feign es una alternativa que resuelve este inconveniente. Para implementar Feign es necesario agregar la siguiente dependencia:

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>

```

En la clase principal del microservicio de hoteles habilitar FeignClients

```

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

```

La implementación requiere crear lo siguiente

```

package com.reservahotel.hotels.services.client;

import com.reservahotel.hotels.model.Room;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

```

```

import java.util.List;

/**
 * Configuración para indicar que se va a consumir el servicio
 * utilizando Feign.
 * Se indica con que nombre se registra el
 * microservicio de rooms en Eureka "rooms"
 * @author jzapana
 */
@FeignClient("rooms")
public interface RoomsFeignClient {

    @RequestMapping(method = RequestMethod.GET, value="rooms/{id}", consumes
    ="application/json")
    public List<Room> searchHotelById(@PathVariable long id);
}

```

Finalmente modificamos el método searchHotelById de HotelServiceImpl

```

@Autowired
RoomsFeignClient roomsFeignClient;

@Override
public HotelRooms searchHotelById(long hotelId) {
    HotelRooms response = new HotelRooms();
    Optional<Hotel> hotel = hotelDao.findById(hotelId);

    List<Room> rooms = roomsFeignClient.searchHotelById(hotelId);

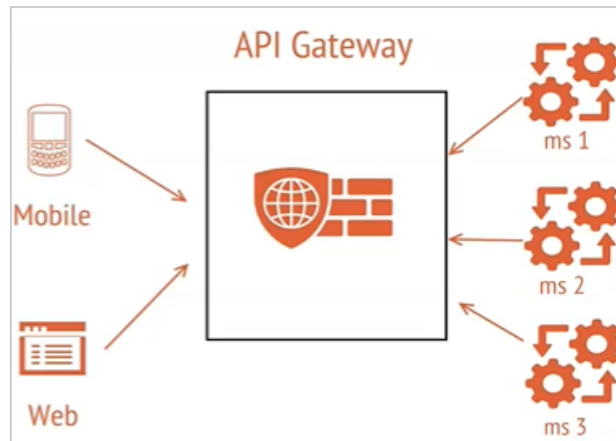
    response.setHotelId(hotel.get().getHotelId());
    response.setHotelName(hotel.get().getHotelName());
    response.setHotelAddress(hotel.get().getHotelAddress());
    response.setRooms(rooms);
    return response;
}

```

Spring Cloud Gateway

API Gateway

Es el gestor de tráfico que interactúa con los datos o el servicio backend real y aplica políticas, autenticación y control de acceso general para las llamadas de una API para proteger los datos. Es el responsable de enrutar la solicitud al servicio adecuado y enviar una respuesta al solicitante



Spring Cloud Gateway

Este proyecto proporciona una biblioteca para crear una API Gateway sobre Spring WebFlux

Tiene como objetivo proporcionar una forma simple y efectiva de enrutar a las APIs y brindarles seguridad, monitoreo / métricas, etc.

Generando proyecto gateway

Ver documentación en: <https://spring.io/projects/spring-cloud-gateway>

Crear un nuevo proyecto **gatewayserver** que contenga las librerías

- Gateway
- Spring Boot Actuator
- Eureka Discovery Client
- Config Client

agregar las siguientes propiedades en application.properties:

```
spring.application.name=gatewayserver
spring.config.import=optional:configserver:http://localhost:8085

management.endpoints.web.exposure.include=*

info.app.name=Gateway Server microservice
info.app.description=microservices for gateway in microservices
info.app.version=1.0.0
management.info.env.enabled=true
management.endpoint.gateway.enabled=true

spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true
```

Configurar para que lea del proyecto gitlab las siguientes properties en *gatewayserver.properties*

```
server.port=8066
eureka.instance.preferIpAddress=true
eureka.client.registerWithEureka=true
```

```
eureka.client.fetchRegistry=true
```

```
eureka.client.serviceUrl.defaultZone=http://localhost:8065/eureka/
```

Configurar el enrutamiento de los demás microservicios creando un Bean