**Guia Microservicios**

**Spring Cloud, Spring Boot Eureka y Zuul**

1. **Creación de proyecto**
   1. Configurar nombre en application.properties
      1. spring.application.name=servicio-productos
   2. Establecer puerto (cada microservicio debe tener un puerto distinto)
      1. server.port=8001
   3. Configurar base de datos h2 (si se necesita)
      1. spring.h2.console.enabled=true
      2. spring.datasource.url=jdbc:h2:mem:testdb
      3. spring.datasource.username=root
      4. spring.datasource.password=root
   4. Conectar microservicios
      1. Crear otro microservicio
      2. Crear RestTeplate (accede a los recursos de los demás microservicios ) en mediante clase AppConfig.
      3. Se crea el restTemplate en la clase que peticiona al otro microservicio

Ruta endpoint y el tipo que se quiere obtener

List<Producto> productos =

Arrays.asList( clienteRest.getForObject("http://localhost:8001/listar", Producto[].class));

return productos stream().map( p -> new Item(p, 1)).collect(Collectors.toList());

1. **Micro servicios con FEIGN ( librería de Netflix)**
   1. Agregar dependencias
      1. Botón derecho sobre el proyecto -> Spring -> Edit Starters -> seleccionar OpenFeign. Se añaden OpenFeign y Spring Cloud al pom.xml
      2. Añadir @EnableFeignClients en la Clase principal de SpringBoot.
      3. Crear interfaz ProductoClienteRest para implementar Feign,
      4. Añadir anotación para habilitar el cliente Feign. Debe indicar el nombre del servicio, tal y como se ha declarado en el archivo application.properties y el nombre del servidor.

@FeignClient(name = “servicio-productos”, url=“localhost:8001”)

* + 1. Añadir métodos y anotaciones de los métodos. Deben ser iguales a los microservicios a los que nos queremos conectar.

@GetMapping(“/listar”)

public List<Producto> listar();

@GetMapping("/ver/{id}")

public List<Producto> detalle(@PathVariable Long id);

* + 1. Crear implementación del Servicio de Feign que implemente la interfaz creada.

@GetMapping(“/listar”)

public List<Producto> listar();

@GetMapping("/ver/{id}")

public List<Producto> detalle(@PathVariable Long id);

* + 1. Inyectar ProductoClienteRest en la implementación.

@Autowired

private ProductoClienteRest clienteFeign;

* + 1. Si se tiene más de una implementación se debe indicar cuál es la implementación principal con la anotación @Primary o usar @Qualifier.
  1. Implementar múltiples instancias y balanceo de carga con Ribbon.
     1. Añadir Ribbon al proyecto (Ejemplo 2.1 Agregar dependencias)
     2. Quitar url de la clase ProductoClienteRest.
        1. @FeignClient(name=“servicio-productos”)
     3. Añadir la anotación @RibbonClient en la clase principal de Spring Boot y se establece como nombre el microservicio o microservicios (@RibbonClients) al que nos queremos conectar.
        1. @RibbonClient(name=“servicio-productos”)

∫

* + 1. Registrar en application.properties las url de los servicios
       1. servicio-productos.ribbon.listOfServers=localhost:8001,localhost:9001
    2. Crear otra instancia del microservicio productos
       1. Botón derecho en el proyecto -> Run As -> Run Configurations
       2. Seleccionar el servicio productos
       3. Ir a Argumentos
       4. En la pestaña **Argumentos**, espacio de texto **VM Arguments** añadir -Dserver.port=9001
       5. Visualizar puertos (Opcional). En la clase Producto, añadir atributo para visualizar el puerto. Añadir anotación @Transient para excluirlo de la base de datos.
          1. @Transient
          2. private Integer port;
       6. En el controlador inyectamos la clase Environment de org.springframework.core.env.Environment para obtener el puerto
          1. @Autowired
          2. private Environment environment;
       7. Obtener puerto (mediante environment)
          1. producto.setPort(Integer.parseInt(environment.getProperty(“local.server.port”)));
       8. Obtener puerto mediante anotación @Value
          1. @Value("${server.port}")
          2. private Integer port;
       9. Añadir anotación @LoadBalanced de Ribbon para el balanceador de carga en AppConfig.
       10. Eliminar las referencias a localhost y sustituirlas por el nombre del servicio en la implementación del servicio Rest (ItemServiceImpl).
       11. Cambiar el @Qualifier(“serviceFeign”) por @Qualifier(“serviceRestTemplate”) en el ItemController

1. **Creación de Eureka Server**
   1. Añadir dependencias DevTools y Eureka Server.
   2. Configurar nombre y puerto en application-properties. Por defecto Eureka se configura a sí mismo como servidor y microservicio.
      1. spring.application.name=servicio-eureka-server
      2. server.port=8761
   3. Cambiar configuración de Eureka para que no se registre como cliente
      1. eureka.client.register-with-eureka=false
      2. eureka.client.fetch-registry=false

* 1. Añadir dependencia JAXV si se está usando jdk11 o superior (en jdk8 viene por defecto y en jdk9 se elimina, se debe instalar aparte)
     1. <dependency>
     2. <groupId>org.glassfish.jaxb</groupId>
     3. <artifactId>jaxb-runtime</artifactId>
     4. </dependency>
  2. Habilitar Eureka server mediante la anotación @EnableEurekaServer en la clase principal del servidor Eureka.
  3. Agregar la dependencia de **Eureka Discovery Client** a los demás proyectos para que puedan registrarse en el servidor de Eureka.
  4. Añadir anotación @EnableEurekaClient en la clase principal de Spring Boot. (No es necesario ya que se realiza mediante el pom.xml, pero si es recomendado).
  5. Añadir la configuración de la ruta del servidor de Eureka en los microservicios. (Si los microservicios están alojados en la misma máquina no es necesario, se encuentran automáticamente)
     1. eureka.client.service-url.defaultZone=<http://localhost:8761/eureka>
     2. Eliminar la dependencia de ribbon del pom.xml de ítems (en eureka viene por defecto)
     3. Eliminar la lista de instancias de la aplicación en el archivo application.properties
        1. servicio-productos.ribbon.listOfServers=localhost:8001,localhost:9001
     4. Eliminar de la clase principal la anotación @RibbonClient.
     5. Al arrancar el servidor de eureka y las aplicaciones, estas se registran automáticamente en el servidor de nombres.

1. **Configuración dinámica de puertos (Solo en los microservicios escalables donde se desee tener más de una instancia).**
   1. En el archivo application.properties del servidor eureka sustituir:
      1. server.por=número por server.port=${PORT:0}
   2. Configurar la instancia en eureka
      1. eureka.instance.instance-id=${spring.application.name}:${spring.application.instance\_id:${random.value}}
2. **Instalación de Hystrix (Herramienta para el manejo de errores, gestiona las conexiones, añade la lógica a la tolerancia a fallos, latencia, etc.) Se carga como un hilo separado de la comunicación de los microservicios.**
   1. Añadir dependencias en servicio-item
      1. Añadir Starters -> Hystrix
   2. Habilitar Hystrix en el proyecto
      1. Añadir la anotación @EnableCircuitBreaker en la clase principal de Spring Boot.
      2. Añadir anotación @HystrixCommand(fallbackMethod=“nombre del método”) para abrir un circuito alternativo cuando se genera un error. Se debe crear un método con el mismo nombre y parámetros al elegido para la anotación y que contenga la lógica en caso de error.
   3. Configurar tiempos de espera con Hystrix (Se deben configurar Hystrix y Ribbon ya que tienen tiempos de espera de solo 1 segundo. Hystrix envuelve al cliente Ribbon, por lo tanto, debe tener un tiempo de espera mayor)
      1. Añadir en el archivo application.properties del proyecto que se conecta con los demás microservicios (servicio-item) la configuración de Hystrix y Ribbon:
         1. hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 60000
         2. ribbon.ConnectTimeout: 3000
         3. ribbon.ReadTimeout: 60000
3. **Instalación de Zuul (microservicio que se encarga del acceso a los demás microservicios. Puerta de enlace o gateway. Tiene enrutamiento dinámico y enruta a cada uno de los microservicios registrados en eureka)**
   1. Crear proyecto con las dependencias **Dev Tools**, **Eureka Discovery Client, Spring Web** y **Zuul**
   2. Añadir las anotaciones @EnableEurekaClient y @EnableZuulProxy en la clase principal de Spring Boot del proyecto.
   3. Configurar el nombre del proyecto y puerto y la ruta hacia eureka en el archivo application.properties
   4. Enrutar todos los microservicios (en nombre del servicio puede ser cualquiera, en esta caso productos)
      1. zuul.routes.productos.service-id=servicio-productos
   5. Mapear las rutas hacia el microservicio (url=/api/productos, endpoint=/\*\*):
      1. zuul.routes.productos.path=/api/productos/\*\*
   6. Filtros con Zuul
      1. **PRE**: Se ejecuta antes de que el request sea enrutado hacia 1 microservicio. Se usa para asignar datos del request para que sean utilizados en los otros filtros.
      2. **POST**: Se ejecuta después de que el request haya sido enrutado. Se usa para modificar la respuesta
      3. **ROUTE**: Se ejecuta durante el enrutado del request, aquí se resuelve la ruta. Se usa para la comunicación con el microservicio.

Por debajo los microservicios ya están implementados:

* + - 1. **RibbonRoutingFilter**: Usa Ribbon, Hystrix y un cliente http (por defecto Apache Http Cliente) para la comunicación.
    1. Implementar Filtros:
       1. Crear clase que herede de ZuulFilter y configurar con @Component.
       2. Ejemplo en el proyecto springboot-servicio-zuul -> filter
  1. Configurar Timeout en Zuul
     1. Copiar configuración del Timeout de Hystrix

1. **Crear Servidor de configuración Spring Cloud** 
   1. Crear proyecto con las dependencias **DevTools**, **Config** **Server**
   2. Añadir la dependencia @EnableConfigServer en la clase principal de Spring Boot
   3. Configurar nombre de proyecto y puerto en el archivo **application.properties**.
   4. Añadir configuración para indicar dónde se ubicará el repositorio GIT.
      1. spring.cloud.config.server.git.uri=file:///url ( Windows )
      2. spring.cloud.config.server.git.uri=file:/url ( Linux o Mac )
   5. Instalar GIT si no está instalado
   6. Crear repositorio de configuración local.
      1. Añadir configuración local (si no se ha realizado)
         1. git config --global user.email “email”
         2. git config --global user.name “usuario”
         3. git config --global user.password “password”
   7. Crear repositorio de inicio
   8. git init
   9. Inicializar puerto en el microservicio
      1. echo server.port=puerto > “nombre microservicio.properties”
         1. echo server.port=8005 > servicio-items.properties
      2. Añadir configuraciones personalizadas
         1. configuracion.texto=Configurando entorno de Desarrollo
      3. Añadir archivo al repositorio
         1. git add .
         2. git commit -am “texto”
   10. Conectar servidor de configuración con microservicio
       1. Comprobar lanzando la siguiente petición: http://localhost:8888/servicio-items/default
   11. Modificar dependencias en servicio-item
       1. Botón derecho en el proyecto -> Spring -> Edit Starters.
          1. Añadir dependencia **Config Client**
   12. Configurar la conexión con el servidor de configuración
       1. Copiar el archivo **application.properties** y renombrarlo a **bootstrap.properties** (en este archivo se realiza la configuración, se carga en el lanzamiento de la aplicación y no se puede sobrescribir mediante la configuración del archivo application.properties)
       2. Siguiendo el siguiente orden debe ir:
          1. Nombre microservicio
          2. Ruta del servidor de configuración
             1. spring.cloud.config.uri=<http://localhost:8888>
          3. Configurar el perfil (DES, PRE o PRO)
   13. Obtener la configuración de los archivos properties mediante clase Handler.
       1. Inyectar un atributo mediante la anotación @Value(“${nombre del atributo en el archivo application.properties}”) en el controlador del microservicio cliente (springboot-servicio-item)

@Value("${configuracion.texto}")

private String texto;

* + 1. Probar configuración con Postman mediante la url <http://localhost:8005/obtener-config>
    2. Probar en zuul con la siguiente url <http://localhost:8090/obtener-config>
    3. Probar configuración a través del GateWay <http://localhost:8090/api/items/obtener-config>
  1. Configurar entornos de desarrollo y producción
     1. Crear archivos properties en la carpeta de configuración del servidor cloud
        1. Ejemplo de configuración de desarrollo:
           1. configuracion.texto=Configurando entorno de desarrollo
           2. configuracion.autor.nombre=Juan
           3. configuracion.autor.email=juancorvelle@gmail.com
     2. Añadir el perfil en el archivo bootstrap.properties. Esta configuración se obtiene con el atributo Environment de Spring y se inyecta con la anotación @Autowired.
        1. spring.profiles.active=dev
        2. Ejemplo de obtención del parámetro configuracion.autor.nombre:
           1. env.getProperty(“configuracion.autor.nombre”)
        3. Ver archivos de configuración mediante Postman:
           1. <http://localhost:8888/servicio-items/dev>
           2. <http://localhost:8888/servicio-items/pro>
     3. Actualizar cambios en tiempo real
        1. Añadir anotación @RefreshScope en el controlador
        2. Añadir Spring Actuator mediante Spring Starter
        3. En bootstrap.properties modificar el entorno a dev
        4. Habilitar el endpoint de actuator en el archivo bootstrap.properties
           1. management.endpoints.web.exposure.include=\* (el \* habilita todos los endpoints de spring actuator)
        5. Actualizar en tiempo real. Ejecutar actuator (No configuraciones de bbdd, solo propias)
           1. Desde Postman realizar petición POST <http://localhost:8005/actuator/refresh>
     4. Configuración desde repositorio git
        1. Subir repositorio de configuración a git
        2. Comentar url de repositorio local en el archivo application.properties del microservicio springboot-servicio-cloud-config-server
        3. Añadir url de repositorio git
        4. spring.cloud.config.server.git.uri=<https://github.com/juanlod/cloud-config.git>
        5. Si da error Timeout al probar el balanceador de carga (Zuul) se soluciona activando los timeouts en el microservicio ítems y en zuul.
  2. Crear Crud de productos si no se había realizado ya
  3. Crear Crud de ítems si no se había realizado ya

1. Crear proyecto que contenga las clases comunes y librerías ( springboot-servicio-commons ) con las librerías ( No necesario ):
   * 1. Spring Data JPA
     2. H2 Database
   1. Eliminar el método main de la clase principal del proyecto ya que es un proyecto de librerías.
   2. Eliminar el plugin de maven del archivo pom.xml.
   3. Crear package que almacene las entidades.
   4. Copiar la clase entity Producto al package nuevo.
   5. Quitar dependencia h2 del proyecto commons
   6. Modificar clase principal de Spring Boot y deshabilitar la configuración del DataSource mediante la anotación:

@EnableAutoConfiguration(exclude = {DataSourceAutoConfiguration.class})

* 1. Generar jar del proyecto.
  2. Inyectar el jar como dependencia en el pom.xml
  3. Copiar groupId, artifactId y versión del pom del proyecto commons
     + 1. <dependency>

<groupId>com.jaunos.app.commons</groupId>

<artifactId>springboot-servicio-commons</artifactId>

<version>1.0.0.0</version>

</dependency>

* 1. Modificar imports para que utilice la entidad del proyecto commons
  2. Si no aparece el jar en maven dependecies (aparece como proyecto), cambiar la versión en el proyecto y actualizar el proyecto con maven.
  3. Habilitar el escaneado para que encuentre la clase Productos (al estar en otro proyecto, no lo encuentra) en la clase principal del proyecto mediante la anotación @EntityScan(“{com.jaunos.app.commons.entity}") e indicándole que busque las clases entity en el/los paquetes especificados separados por coma.

1. **Añadiendo Seguridad con Jason Web Token (JWT, <https://jwt.io/>), HATEOAS**
   1. Crear proyecto de usuarios para gestionar los permisos con las siguientes librerías:
      1. Spring Boot DevTools
      2. Spring Data JPA
      3. H2 Database
      4. Eureka Discovery Client
      5. Spring Web
      6. Rest Repositories
   2. Declarar nombre del microservicio en el archivo application.properties.
      1. Opción copiar configuración del proyecto Productos y cambiar nombre del microservicio
   3. Opcional. Activar SQL Nativo:
      1. logging.level.org.hibernate.SQL=debug
   4. Crear entidades en package entity
   5. Crear clases Usuario.java y Role.java
   6. Crear relaciones en Usuario.java

@ManyToMany(fetch = FetchType.**LAZY**)

// Personalización

@JoinTable(name="usuarios\_roles",

joinColumns = @JoinColumn(name="usuario\_id"),

inverseJoinColumns = @JoinColumn(name="role\_id"),

// 1 usuario no puede tener rol repetido

uniqueConstraints = {@UniqueConstraint(columnNames = {"usuario\_id", "role\_id"})})

private List<Role> roles;

* 1. Crear relación inversa en Role.java

@ManyToMany(fetch = FetchType.**LAZY,** mappedBy = "roles")

private List<Usuario> usuarios;

* 1. Crear repositorio UsuarioRepository
  2. Añadir anotación @RepositoryRestResource y el path de la api rest a la cual se va a exportar el repositorio

// Path donde se va a exportar el repositorio

@RepositoryRestResource(path="usuarios")

* 1. Configurar el enrutado dinámico en Zuul en el archivo.properties

zuul.routes.usuarios.service-id=servicio-usuarios

zuul.routes.usuarios.path=/api/usuarios/\*\*

* 1. El crud se utiliza a través de peticiones http sin necesidad de controlador mediante el endpoint del repositorio y la anotación @RepositoryRestResource(path=“usuarios”) y a través del gateway de zuul
     1. <http://localhost:8090/api/usuarios/usuarios>
  2. Creación de usuario mediante POST <http://localhost:8090/api/usuarios/usuarios:>

{

"username": "pepe",

"enabled": true,

"password": "12345",

"nombre": "Jose",

"apellido": "Doe",

"email": "jose.doe@bolsadeideas.com",

"roles": [

{

"id":1,

"nombre": "ROLE\_USER"

},

{

"id":2,

"nombre": "ROLE\_ADMIN"

}

]

}

* 1. Utilizar los métodos del repositorio mediante search:

GET <http://localhost:8090/api/usuarios/usuarios/search/findByUsername?username=admin>

* 1. Personalizar el path de los métodos con la anotación @RestResource y Renombrar parámetro co la anotación @Param.

@RestResource(path=“buscar-username”)

public Usuario findByUsername(@Param(“nombre”) String username);

* 1. Petición:

GET <http://localhost:8090/api/usuarios/usuarios/search/buscar-username?nombre=admin>

* 1. Configurar repositorio para exponer y devolver la id
     1. Creamos clase RepositoryConfig que implementa la interfaz RepositoryRestConfigurer y anotamos con la anotación @Configuration.
     2. Sobreescribimos el método configureRepositoryRestConfiguration mediante Source -> Override/Implement Methods.
     3. Borramos lo autogenerado dentro del método y añadimos la siguiente linea:

config.exposeIdsFor(Usuario.class, Role.class);

1. **Crear microservicio de control de accesos mediante la api OAuth2 con las dependencias:**
   1. Dependencias:
      1. Eureka Discovery Client
      2. Cloud OAuth2
      3. Spring Boot DevTools
      4. Spring Web Starter
      5. OpenFeign
   2. Habilitamos Eureka Client en la clase principal del proyecto con la anotación @EnableEurekaClient.
   3. Configurar nombre del microservicio y el puerto y activar el servicio en eureka:

spring.application.name=servicio-oauth

server.port=9100

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

* 1. Habilitar cliente Feign mediante la anotación @EnableFeignClients en la clase principal del microservicio oauth para que se pueda comunicar con el microservicio usuarios.
  2. Crear interfaz UsuarioFeignClient.java en el package clients y decorarlo con la anotación @FeignClient y el nombre del microservicio con el que nos queremos conectar.

@FeignClient(name="servicio-usuarios")

* 1. Crear método para buscar el usuario por su username.

@GetMapping("/usuarios/search/buscar-username")

public Usuario findByUsername(@RequestParam String username);

* 1. Crear package services para implementar los servicios de Spring Security.
     1. UsuarioService.java
  2. Crear package security y las clases de configuración de Spring Security
     1. SpringSecurityConfig extends WebSecurityConfigurerAdapter e implementamos el método configure(AuthenticationManagerBuilder) y el método authenticationManager();
     2. Creamos la clase AuthorizationServerConfig con las anotaciones @Configuration y @EnableAuthorizationServer para habilitar la clase como Servidor de Autorización y que extienda de AuthorizationServerConfigurerAdapter.
  3. Se inyectan los @Beans de la clase SpringSecurityConfig
  4. Implementamos los métodos de AuthorizatonServerConfigurerAdapter
  5. En el método configure de los endpoints se configura el authentication manager, el token storage (JWT) y el Access Token Converter que se encarga de guardar los datos en el token.
  6. Implementar la interfaz CommandLineRunner el método run para encriptar las contraseñas con BCryptPasswordEncoder. (Solo sirve para obtener contraseñas de ejemplo encriptadas)
  7. Implementar los métodos de oauth en Zuul
     1. Añadir el endpoint del microservicio oauth en el archivo properties.

zuul.routes.security.service-id=servicio-oauth

zuul.routes.security.path=/api/security/\*\*

* + 1. Añadir servicio de cookies necesarias para el servicio oauth para eliminarlas de la cabecera

zuul.routes.security.sensitive-headers=Cookie, Set-Cookie

* + 1. Probar desde Postman
       1. En Authorization seleccionar Basic Auth
          1. Username: frontendapp o nombre de usuario de prueba elegido
          2. Password: 12345 o password de prueba elegido
       2. POST localhost:8090/api/security/oauth/token
       3. En Body seleccionar x-www-form-urlencoded
          1. username admin
          2. password 12345
          3. grant\_type password
  1. Añadir información al token (Claims)
  2. Crear clase en el package security del microservicio OAuth con el nombre InfoAdicionalToken que implemente la interfaz TokenEnhancer y decorarla con la anotación @Component
  3. Crear interfaz IUsuarioService que implemente el método findByUsername() para poder devolver el Usuario de la clase UsuarioService
  4. Implementamos IUsuarioService en UsuarioService
  5. Inyectamos IUsuarioService en la clase InfoAdicionalToken
  6. Modificar el método configure(AuthorizationServerEndpointsConfigurer) de la clase AuthorizationServerConfig
  7. Configurar Zuul como servidor de recurso (Protege los accesos a los distintos microservicios y valida que el token sea correcto)
     1. Añadir la dependencia Cloud OAuth2 de Spring Cloud Security mediante Edit Starters
     2. Crear el package oauth
     3. Crear la clase de configuración para el servidor de recursos llamada ResourceServerConfig y la decoramos con la anotación @Configuration.
     4. Habilitar la configuración del servidor de recursos mediante la anotación @EnableResourceServer y extender de la clase ResourceServerConfigurerAdapter.
     5. Implementar los métodos para proteger las rutas y configurar el token de la misma manera que los del microservicio oauth
     6. Implementar los métodos
     7. Probar desde Postman
        1. GET localhost:8090/api/usuarios/usuarios
        2. GET localhost:8090/api/usuarios/1
           1. Después de obtener el token mediante la petición POST localhost:8090/api/security/oauth/token se añade el token en la pestaña Authorization y se cambia el tipo a Bear Token
  8. Crear configuración oauth en el servidor de configuración
     1. Crear archivo de configuración application.properties en el proyecto de configuración git-config
     2. Configurar clienteId

[config.security.oauth.client.id](http://config.security.oauth.client.id)=frontendapp

* + 1. Configurar el secret ID

config.security.oauth.client.secret=12345

* + 1. Configurar la Key para firmar el token

config.security.oauth.jwt.key=algun\_codigo\_secreto\_aeiou

* + 1. Hacer commit y push
    2. Agregar la dependencia Cloud Client de Spring Cloud Config mediante la opción Edit Starter en los microservicios oauth, zuul
    3. Copiar bootstrap.properties del microservicio ítems y lo pegamos en zuul y en oauth, eliminamos el perfil y modificamos los nombres de los microservicios
    4. Inyectar Environment en la clase de configuración del microservicio oauth AuthorizationServerConfig para sustituir la configuración anterior por la configuración desde el servidor de configuración
    5. Inyectar Environment en la clase de configuración del microservicio zuul ResourceServerConfig para sustituir la configuración anterior por la configuración desde el servidor de configuración
    6. Opcional: Añadir anotación en las clases si se desea actualizar sin reiniciar
    7. Renovar token antes de que expire a partir de refresh token y mediante la petición de generar un token copiar el refresh token y en configurar el body con los siguientes parámetros:
       - 1. grant\_type refresh\_token
         2. refresh\_token “el refresh token”
    8. Configurar CORS
       1. Añadir en el método configure(HttpSecurity http) en la clase ResouceServerConfig del microservicio oauth:

.and().cors().configurationSource(corsConfigurationSource())

* + - 1. Añadir los siguientes métodos

// Configuración de CORs

@Bean

public CorsConfigurationSource corsConfigurationSource() {

CorsConfiguration corsConfig = new CorsConfiguration();

// Configurar el origen

corsConfig.addAllowedOrigin("\*");

// Configurar varios orígenes

corsConfig.setAllowedOrigins(Arrays.asList("\*"));

// Configurar método de acceso. Options lo utiliza oauth2 por debajo

corsConfig.setAllowedMethods(Arrays.asList("POST", "GET", "PUT", "DELETE", "OPTIONS"));

// Configurar credenciales

corsConfig.setAllowCredentials(true);

corsConfig.setAllowedHeaders(Arrays.asList("Authorization", "Content- Type"));

// Enviar configuración de cors a la url

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

// Registrar la configuración

source.registerCorsConfiguration("/\*\*", corsConfig);

return source;

}

// Filtro de CORS a nivel de aplicación

@Bean

public FilterRegistrationBean<CorsFilter> corsFilter() {

FilterRegistrationBean<CorsFilter> bean =

new FilterRegistrationBean<CorsFilter>(new CorsFilter(corsConfigurationSource()));

// Configurar como prioridad alta

bean.setOrder(Ordered.***HIGHEST\_PRECEDENCE***);

return bean;

}

* + 1. Creación de Eventos Success y Error
       1. Se crea un package de nombre events en el microservicio oauth y dentro de él, una nueva clases llamada, por ejemplo, AuthenticationSuccesErrorHandler, anotar como @Component e implementar la interfaz AuthenticationEventPublisher.
       2. Implementar los métodos
       3. Inyectar la clase en SpringSecurityConfig.

@Autowired

private AuthenticationEventPublisher eventPublisher;

* + - 1. Registrar el eventPublisher en el método configure(AuthenticationManagerBuilder auth) añadiendo .and().authenticationEventPublisher(eventPublisher)
      2. Probar desde Postman un login correcto.
      3. Añadir bloqueo por fallos.
      4. Añadir campo intentos de tipo Integer en la clase usuario
      5. Añadir método para actualizar en UsuarioClienteFeign del microservicio oauth

@PutMapping("/usuarios/{id}")

public Usuario update(@RequestBody Usuario usuario, @PathVariable Long id);

* + - 1. Copiar el método e implementarlo en la interfaz IUsuarioService (sin anotaciones) y el nuevo método de update en la clase UsuarioService.
      2. Inyectar la clase IUsuarioService en la clase AuthenticationSuccessErrorHandler para actualizar en usuario cada vez que falla o reseteandola si consigue loguearse correctamente antes de los 3 intentos.
      3. Personalizar el error sobre Feign mediante try catch copiando el error de log y el throw error al catch y cambiando el tipo de error que se captura por FeignException en la clase UsuarioService.
    1. Configurar conexión a base de datos MySQL
       1. Añadir dependencia para mysql desde Spring Starters y añadir MySQL Driver en el microservicio productos
       2. Añadir en application.properties

spring.datasource.url= jdbc:mysql://localhost:3306/db\_springboot\_cloud?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC&allowPublicKeyRetrieval=true&useSSL=false

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=create

logging.level.org.hibernate.SQL=debug

* + - 1. Migrar configuración de MySQL al archivo de configuración.
         1. Crear un archivo.properties en el servidor de configuración llamado servicio-productos-dev.properties
         2. Configurar el microservicio-productos para que sea un cliente del servicio de configuración

Copiar bootstrap.properties de servicio-ítems

spring.application.name=servicio-productos

spring.profiles.active=dev

spring.cloud.config.uri=http://localhost:8888

Añadir la dependencia Config Client de Spring Cloud Config mediante Edit Starters de Spring

* + 1. Trazabilidad distribuida con Sleuth (dependencia que provee una solución de trazado distribuido (se realizan trazas de todas las peticiones realizadas a los microservicios) para Spring Cloud. Permite identificar la petición completa de un microservicio) y Zipkin (guarda y permite mostrar toda la trazabilidad de Sleuth)
       1. Añadir la dependencia Sleuth de Spring Cloud Tracing a los microservicios que se desee realizar el seguimiento de las trazas, por ejemplo, item, oauth, usuarios, zuul-server etc
       2. Descargar la versión jar de Zipkin de <https://zipkin.io/> version de java latest release
       3. Desplegar el jar mediante el comando java -jar nombre\_del \_archivo
       4. Abrir una pestaña en el navegador con la dirección localhost:9411
       5. Añadir la dependencia Zipkin Client de Spring Cloud Tracing a los microservicios a los cuales se había inyectado la dependencia Sleuth
       6. Configurar los microservicios para que funcionen con Zipkin
       7. Configurar Zipkin en el application properties de cada microservicio
          1. Cambiar la probabilidad al 100% (1) en la que envía las trazas en la que por defecto 10% (0,1)

spring.sleuth.sampler.probability=1.0

* + - * 1. Configurar el endpoint del servidor zipkin (opcional), la siguiente ruta es la ruta por defecto,

spring.zipkin.base-url=<http://localhost:9411/>

* + - * 1. Añadir información a las trazas

Inyectar el componente Tracer mediante @Autowired en la clase UsuarioService del microservicio oauth

tracer.currentSpan().tag("error.mensaje", error.concat(" : “).concat(e.getMessage()));

* + - 1. Configurar los microservicios para que envíen las trazas mediante RabbitMQ
         1. Descargar RabbitMQ de <https://www.rabbitmq.com/download.html>
         2. Instalar RabbitMQ
         3. Añadir ruta al path

nano ~/.bash\_profile

export PATH=$PATH:/usr/local/opt/rabbitmq/sbin

* + - * 1. Activar consola web desde consola y como administrador

sudo rabbitmq-plugins enable rabbitmq\_management

* + - * 1. Abrir consola web en localhost:15672
        2. Abrir rabbitmq en segundo plano en mac

brew services start rabbitmq

* + - * 1. Iniciar rabbitmq en primer plano

rabbitmq-server

* + - * 1. Usuario y password por primera vez = guest
      1. Conectar los microservicios de RabbitMQ con Zipkin
         1. Añadir la dependencia Spring for RabbitMQ de Messagin a cada microservicio con Zipkin
         2. Configurar el servidor Zipkin para que pueda consumir los mensajes y guardar las trazas en el storage

Crear archivo zipkin.sh en el directorio del proyecto con la siguiente configuración

#!/bin/sh

#This program returns the zipkin variable configurations

#contents of my Home folder

echo off

export RABBIT\_ADDRESSES=localhost:5672

export STORAGE\_TYPE=mysql

export MYSQL\_USER=zipkin

export MYSQL\_PASS=zipkin

java -jar ./zipkin-server-2.21.1-exec.jar —server.port=9512

Orden de apertura. Los productores (microservicios) se conectan cuando envían la traza

./zipkin.sh

config-server

eureka-server

oauth

usuarios

items

productos

zuul

* + - * 1. Configurar MySQL con Zipkin

export STORAGE\_TYPE=mysql

export MYSQL\_USER=zipkin

export MYSQL\_PASS=zipkin

* + - * 1. Crear base de datos en MySQL

CREATE SCHEMA `zipkin` DEFAULT CHARACTER SET utf8 COLLATE utf8\_bin ;

* + - * 1. Crear usuario

Username zipkin

Password zipkin

Tipo de autenticación standard

Host localhost

* + - * 1. Dar privilegios

Ir a la pestaña Schema privileges -> Add privileges -> seleccionar base de datos zipkin

* + - * 1. Copiamos el esquema de la página <https://github.com/openzipkin/zipkin/blob/master/zipkin-storage/mysql-v1/src/main/resources/mysql.sql>
    1. Desplegar aplicaciones con Docker
       1. Descargar de <https://www.docker.com/products/docker-desktop>
       2. Instalar Docker
       3. Crear jar de los microservicios con maven
       4. Crear archivo de configuración de Docker en la raíz del proyecto
          1. Boton derecho -> new File -> nombre Dockerfile
          2. Añadir jdk existente en el repositorio de docker

FROM openjdk:12

* + - * 1. Añadir volumen a montar (opcional)

VOLUME /tmp

* + - * 1. Configurar puerto (indica en que puerto se va a exponer el contenedor).

EXPOSE 8888

* + - * 1. Indicar el directorio de la aplicación que se va a desplegar y el nombre con el que se va a desplegar

ADD ./target/springboot-servicio-cloud-config-server-1.0.0.0.war cloud-config-server.jar

* + - * 1. Indicar que se ejecute la aplicación cuando arranque el contenedor

ENTRYPOINT ["java", "-jar", "/cloud-config-server.jar"]

* + - * 1. Generar imagen con Docker en la ruta raid del proyecto. Debe tener el docker file dentro del directorio

docker build -t cloud-config-server:v1 .

* + - * 1. Verificar que se ha generado bien

docker images

* + - * 1. Crear red para ejecutar los microservicios y las bases de datos

docker network create springcloud

* + - * 1. Ejecutar imagen para crear el contenedor

(puerto externo: puerto interno del microservicio)

docker run -p 8888:8888 --name config-server --network springcloud cloud-config-server:v1

* + - * 1. Listar imagen

docker ps -a

* + - * 1. Borrar imagen

docker rm nombre\_imagen

* + - * 1. Borrar imágenes <none>

docker rmi $(docker images -f "dangling=true" -q)

* + - * 1. Eliminar todos los contenedores ejecutándose

docker rm -f $(docker ps -aq)

* + - * 1. Ver configuración en el navegador

localhost:8888/servicio-items/default

localhost:8888/servicio-productos/default

localhost:8888/servicio-productos/dev

localhost:8888/servicio-usuarios/dev

* + - * 1. Configurar Dockfile en **eureka server**

FROM openjdk:12

VOLUME /tmp

EXPOSE 8761

ADD ./target/springboot-servicio-eureka-server-1.0.0.0.war eureka-server.jar

ENTRYPOINT ["java", "-jar", “/eureka-server.jar"]

* + - * 1. Construir imagen

docker build -t servicio-eureka-server:v1 .

* + - * 1. Ejecutar imagen de **eureka server**

docker run -p 8761:8761 --name servicio-eureka-server --network springcloud servicio-eureka-server:v1

* + - * 1. Instalar imagen de **mysql** <https://hub.docker.com/_/mysql>

docker pull mysql:version

docker pull mysql:8

* + - * 1. Crear contenedor de **mysql**

docker run -p 3306:3306 --name servicio-mysql8 --network springcloud -e MYSQL\_ROOT\_PASSWORD=root -e MYSQL\_DATABASE=db\_springboot\_cloud -d mysql:8

* + - * 1. Ver logs para ver estado del microservicio

docker logs -f servicio-mysql8

* + - * 1. Conectarse al contenedor mediante MySQL Workbench mediante una nueva conexión
        2. Conectarse con Postgres (Opcional)

docker pull postgres:12-alpine

* + - * 1. Crear contenedor de **postgres**

docker run -p 5432:5432 --name servicio-postgres12 --network springcloud -e POSTGRES\_PASSWORD=root -e POSTGRES\_DB=db\_springboot\_cloud -d postgres:12-alpine

* + - * 1. Configuración de base de datos en el microservicio cloud-config

Cambiar localhost por en nombre de la imagen de docker

* + - * 1. Modificar los microservicios para que apunten a la imagen de eureka

Cambiar localhost por en nombre de la imagen de docker

Triplicar el tiempo en los timeout

* + - * 1. Modificar las url del servidor de configuración para que apunten a la imagen de docker

Sustituir localhost por en nombre de la imagen de docker

* + - * 1. Crear configuración de despliegue para **productos**

Copiar el Dockerfile de otro proyecto y modificarlo:

FROM openjdk:12

VOLUME /tmp

ADD ./target/springboot-servicio-productos-1.0.0.0.war servicio-productos.jar

ENTRYPOINT ["java", "-jar", “/servicio-productos.jar"]

Generar el jar

mvn clean install -DskipTests

Construir imagen

docker build -t servicio-productos:v1 .

Crear instancia del contenedor con puerto aleatorio (-P). Name es opcional ya que nadie va a hacer referencia a este microservicio

docker run -P --network springcloud servicio-productos:v1

* + - * 1. Crear configuración de despliegue para **zuul**

Copiar el Dockerfile de eureka y modificarlo:

FROM openjdk:12

VOLUME /tmp

EXPOSE 8090

ADD ./target/springboot-servicio-zuul-1.0.0.0.war zuul-server.jar

ENTRYPOINT ["java", "-jar", "/zuul-server.jar"]

Generar el jar

mvn clean install -DskipTests

Construir imagen

docker build -t servicio-zuul-server:v1 .

Crear instancia del contenedor con puerto aleatorio (-P). Name es opcional ya que nadie va a hacer referencia a este microservicio

docker run -p 8090:8090 --name servicio-zuul-server --network springcloud servicio-zuul-server:v1

Peticionar en postman

GET <http://localhost:8090/api/productos/listar>

* + - * 1. Crear configuración de despliegue para **usuarios**

Copiar el Dockerfile de productos y modificarlo:

FROM openjdk:12

VOLUME /tmp

ADD ./target/springboot-servicio-usuarios-1.0.0.0.war servicio-usuarios.jar

ENTRYPOINT ["java", "-jar", "/servicio-usuarios.jar"]

Generar el jar

mvn clean install -DskipTests

Construir imagen

docker build -t servicio-usuarios:v1 .

Crear instancia del contenedor con puerto aleatorio (-P). Name es opcional ya que nadie va a hacer referencia a este microservicio

docker run -P --network springcloud servicio-usuarios:v1

Peticionar desde Postman

GET <http://localhost:8090/api/usuarios/usuarios>

* + - * 1. Crear configuración de despliegue para **oauth**

Copiar el Dockerfile de zuul y modificarlo:

FROM openjdk:12

VOLUME /tmp

EXPOSE 9100

ADD ./target/springboot-servicio-oauth-1.0.0.0.war servicio-oauth.jar

ENTRYPOINT ["java", "-jar", "/servicio-oauth.jar"]

Generar el jar

mvn clean install -DskipTests

Construir imagen

docker build -t servicio-oauth:v1 .

Crear instancia del contenedor con puerto aleatorio (-P). Name es opcional ya que nadie va a hacer referencia a este microservicio

docker run -p 8090:8090 --name servicio-oauth --network springcloud servicio-oauth:v1

Peticionar en postman

POST http://localhost:8090/api/security/oauth/token

* + - * 1. Crear configuración de despliegue para **ítems**

Copiar el Dockerfile de zuul y modificarlo:

FROM openjdk:12

VOLUME /tmp

EXPOSE 8002

ADD ./target/springboot-servicio-items-1.0.0.0.war servicio-items.jar

ENTRYPOINT ["java", "-jar", "/servicio-items.jar"]

Generar el jar

mvn clean install -DskipTests

Construir imagen

docker build -t servicio-items:v1 .

Crear instancia del contenedor con puerto aleatorio (-P). Name es opcional ya que nadie va a hacer referencia a este microservicio

docker run -p 8090:8090 --name servicio-ítems --network springcloud servicio-items:v1

Peticionar en postman

GET http://localhost:8090/api/items/listar

* + - * 1. Instalar **rabitMQ**

Descargar de <https://hub.docker.com/_/rabbitmq>

docker pull rabbitmq:version

docker pull rabbitmq:3.8-management-alpine

Crear contenedor de rabbitmq

docker run -p 15672:15672 -p 5672:5672 --name servicio-rabbitmq38 --network springcloud -d rabbitmq:3.8-management-alpine

Preparar base de datos para zipkin desde workbench (igual de la forma anterior pero para docker)

Crear base de datos para zipkin

Crear usuario

Crear las tablas a partir del ddl de zipkin como en la forma anterior

Añadir el esquema al usuario

Crear carpeta de zipkin en el proyecto de los microservicios llamado zipkin-server y copiar el jar de zipkin dentro

Copiar dockerfile de config-server en la carpeta para realizar la configuración de zipkin:

FROM openjdk:12

VOLUME /tmp

EXPOSE 9411

ADD ./zipkin-server-2.21.1-exec.jar zipkin-server.jar

ENTRYPOINT ["java", "-jar", “/zipkin-server.jar"]

Construir la imagen en el directorio donde está ubicado el ejecutable de zipkin

docker build -t zipkin-server:v1 .

Levantar imagen

docker run -p 9411:9422 --name zipkin-server --network springcloud -e RABBIT\_ADDRESSES= servicio-rabbitmq38:5672

-e STORAGE\_TYPE=mysql -e MYSQL\_USER=zipkin -e MYSQL\_PASS=zipkin -e MYSQL\_HOST= servicio-mysql8 zipkin-server:v1

Modo 1. Modificar el host de los microservicios que se conectan a zipkin.

Sustituir localhost por el nombre de la imagen

Modo 2. Enviar configuración al servidor de configuración

# local

#spring.zipkin.base-url=http://localhost:9411/

#docker

spring.zipkin.base-url=http://zipkin-server:9411/

spring.rabbitmq.host=servicio-rabbitmq38

Copiar en el archivo de configuración application.properties, commit y push

Volver a levantar las instancias

docker restart idcontainer

* + - * 1. Desplegar los contenedores mediante Docker Compose

Detener y eliminar los contenedores creados mediante docker run

Crear carpeta llamada docker-compose dentro del directorio de los proyectos

Crear archivo docler-compose.yml

version: "3.7"

services:

config-server:

image: cloud-config-server:v1

ports:

- "8888:8888"

restart: always

networks:

- springcloud

servicio-eureka-server:

image: servicio-eureka-server:v1

ports:

- "8761:8671"

restart: always

networks:

- springcloud

networks:

springcloud:

Levantar docker compose en el directorio donde se encuentra el archivo

docker-compose up

* + - * 1. Eliminar contenedores

docker-compose down

* + - * 1. Añadir contenedor mysql y productos

servicio-mysql8:

image: mysql:8

ports:

- "3306:3306"

restart: always

networks:

- springcloud

environment:

MYSQL\_DATABASE: db\_springboot\_cloud

MYSQL\_ROOT\_PASSWORD: root

servicio-productos:

image: servicio-productos:v1

restart: always

networks:

- springcloud

depends\_on:

- config-server

- servicio-eureka-server

- servicio-mysql8

* + - * 1. Levantar docker compose poco a poco (el nombre del servicio es el dado en el archivo)

docker-compose up -d config-server

docker-compose up -d servicio-eureka-server

docker-compose up -d servicio-mysql8

* + - * 1. Añadir contenedor ítems y zuul

servicio-items:

image: servicio-items:v1

ports:

- "8002:8002"

- "8005:8005"

- "8007:8007"

restart: always

networks:

- springcloud

depends\_on:

- config-server

- servicio-eureka-server

- servicio-productos

servicio-zuul-server:

image: servicio-zuul-server:v1

ports:

- "8090:8090"

restart: always

networks:

- springcloud

depends\_on:

- config-server

- servicio-eureka-server

- servicio-productos

- servicio-items

* + - * 1. Ver logs

docker-compose logs -f

∫