Découvrez les Edge Microservices avec Spring Cloud

Philémon Globléhi
Développeur Back-end Java

Introduction

Les Edge Microservices orchestrent les microservices principaux qui gèrent la logique métier. Netflix a popularisé ces services, aujourd'hui largement disponibles via Spring Cloud, où ils fonctionnent naturellement ensemble.

- Spring Cloud Config
- Eureka
- Ribbon
- Zuul
- Zipkin

1- La configuration

Avec de nombreux microservices en production, modifier un simple paramètre de configuration peut devenir un casse-tête. Changer application.properties signifie arrêter et redéployer toutes les instances, ce qui bloque l'application.

Spring Cloud Config résout ce problème en centralisant toutes les configurations dans un dépôt Git. Ce serveur distribue les fichiers aux microservices, et lorsque vous modifiez une configuration, les microservices la récupèrent automatiquement sans interruption de service.

Spring Cloud Config en pratique ? C'est simple :

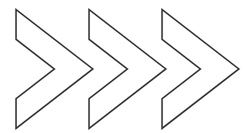
Côté serveur :

1. Ajoutez la dépendance config-server
2. Pointez vers votre dépôt Git avec vos configurations
3. Activez le serveur avec @EnableConfigServer

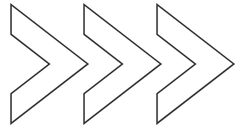
Côté microservice client :

1. Ajoutez la dépendance client config
2. Configurez l'URL du serveur de config
3. Utilisez @RefreshScope sur vos composants
4. Accédez aux propriétés avec @Value("${ma.propriete}")
5. Activez l'endpoint /actuator/refresh pour mettre à jour votre config sans redémarrage

Un simple POST sur /actuator/refresh et votre microservice adopte instantanément les nouvelles configurations



```
// 1. Configuration du serveur Spring Cloud Config
// Dans pom.xml du serveur de configuration
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>

// Dans application.properties du serveur
server.port=8888
spring.cloud.config.server.git.uri=https://github.com/votre-compte/config-repo
spring.application.name=config-server

// Dans la classe principale du serveur
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}

```



```
// 2. Configuration du client (microservice)
// Dans pom.xml du client
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>

// Dans bootstrap.properties du client
spring.application.name=mon-microservice
spring.cloud.config.uri=http://localhost:8888

// Pour permettre le rechargement dynamique des configurations, ajoutez
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

// Dans application.properties du client
management.endpoints.web.exposure.include=refresh

// Dans un contrôleur pour utiliser la configuration
@RestController
@RefreshScope
public class ConfigController {
    @Value("${ma.propriete}")
    private String maPropriete;

    @GetMapping("/config")
    public String getConfig() {
        return "Valeur configurée: " + maPropriete;
    }
}
```

2- La découvrabilité

Quand votre application monte en charge, vous avez besoin de plusieurs instances du même microservice. Comment suivre toutes ces instances et savoir lesquelles sont actives ?

C'est là qu'intervient **Eureka**, le serveur d'annuaire intelligent. Une simple annotation suffit pour qu'une nouvelle instance s'enregistre automatiquement auprès d'Eureka.

Vos clients interrogent simplement Eureka pour trouver les instances disponibles d'un service. Bonus : Eureka vérifie régulièrement quelles instances sont toujours actives et nettoie automatiquement son registre des instances mortes.

Eureka en pratique : comment gérer vos instances de services en 5 minutes

Côté serveur Eureka :

- Une dépendance dans le pom.xml
- Configuration minimale dans application.properties
- Une simple annotation @EnableEurekaServer et c'est prêt !

Côté microservice :

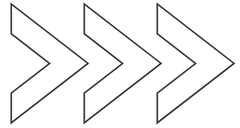
- Ajoutez la dépendance client Eureka
- Configurez l'URL du serveur (defaultZone)
- Activez avec @EnableDiscoveryClient
- Lancez plusieurs instances sur différents ports

Communication entre services :

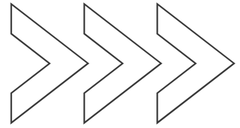
- Utilisez @LoadBalanced sur votre RestTemplate
- Appelez directement par nom de service : "http://mon-service/api"
- L'équilibrage de charge est automatique !

```
// 1. Configuration du serveur Eureka
// Dans pom.xml du serveur Eureka
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>

// Dans application.properties du serveur Eureka
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
spring.application.name=eureka-server

// Dans la classe principale du serveur
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}

```

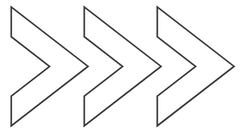
```
22
23 // 2. Configuration d'un microservice client
24 // Dans pom.xml du microservice
25 <dependency>
26     <groupId>org.springframework.cloud</groupId>
27     <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
28 </dependency>
29
30 // Dans application.properties du microservice
31 spring.application.name=mon-microservice
32 server.port=8080
33 eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
34
35 // Dans la classe principale du microservice
36 @SpringBootApplication
37 @EnableDiscoveryClient
38 public class MonMicroserviceApplication {
39     public static void main(String[] args) {
40         SpringApplication.run(MonMicroserviceApplication.class, args);
41     }
42 }
43
```



```java
// 3. Utilisation du client Eureka pour appeler un autre service
// Dans un service client
@Service
public class ClientService {
    @Autowired
    private DiscoveryClient discoveryClient;

    @Autowired
    private RestTemplate restTemplate;

    public String appellerAutreService() {
        // Obtenir les instances d'un service depuis Eureka
        List<ServiceInstance> instances = discoveryClient.getInstances("autre-service");

        if (instances != null && !instances.isEmpty()) {
            URI uri = instances.get(0).getUri();
            return restTemplate.getForObject(uri + "/api/ressource", String.class);
        }
        return null;
    }
}

```



```java
// 4. Configuration du RestTemplate avec équilibrage de charge
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}

// 5. Utilisation simplifiée avec @LoadBalanced
@Service
public class ClientSimpleService {
    @Autowired
    private RestTemplate restTemplate;

    public String appellerAutreService() {
        // Appel direct en utilisant le nom de service au lieu de l'URL
        return restTemplate.getForObject("http://autre-service/api/ressource", String.class);
    }
}

```



3- L'équilibrage de la charge (Load Balancing)

Nos microservices doivent rester découplés et autonomes. Avec plusieurs instances d'un même microservice, comment répartir la charge sans créer un point de défaillance unique ?

Un équilibreur de charge traditionnel centralisé compromettrait la résilience de notre application.

C'est là qu'intervient **Ribbon** : un équilibreur de charge côté client. Chaque microservice décide lui-même quelle instance distante appeler, sans dépendre d'un service central. Cette approche distribue intelligemment le trafic tout en préservant l'autonomie de chaque composant.

Ribbon en pratique :

Configuration minimale :

- Ajoutez la dépendance ribbon dans votre pom.xml
- Combinez avec Eureka pour la découverte de services
- Un seul @LoadBalanced sur votre RestTemplate et c'est prêt !

Utilisation intuitive :

- Appelez vos services par leur nom : "http://mon-service/api"
- Oubliez les adresses IP et les ports spécifiques
- Ribbon sélectionne automatiquement l'instance optimale

Personnalisation puissante :

- Choisissez votre algorithme d'équilibrage (round-robin, aléatoire, réponse rapide...)
- Configurez des règles par service
- Fonctionne même sans Eureka si nécessaire

Résultat : chaque microservice fait ses propres choix d'équilibrage, élimine les points uniques de défaillance et augmente la résilience globale de votre architecture !

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

// 2. Configuration de base dans application.properties
spring.application.name=service-client
server.port=8080
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

// 3. Configuration du RestTemplate avec équilibrage de charge
@SpringBootApplication
@EnableDiscoveryClient
public class ClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(ClientApplication.class, args);
    }

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

// 4. Utilisation dans un service client
@Service
public class ClientService {
    @Autowired
    private RestTemplate restTemplate;

    public String appelServiceDistant() {
        // Utilisation du nom du service au lieu de l'URL concrète
        // Ribbon choisira automatiquement l'instance à appeler
        return restTemplate.getForObject("http://service-produit/produits", String.class);
    }
}
```

```
// 5. Configuration personnalisée de Ribbon (optionnel)
// Dans application.properties pour configurer une stratégie spécifique
service-produit.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule

// 6. Configuration alternative de Ribbon sans Eureka (si besoin)
// Dans application.properties
ribbon.eureka.enabled=false
service-produit.ribbon.listOfServers=http://serveur1:8090,http://serveur2:8090

```

4- API Gateway

Dans une application e-commerce réelle, afficher une simple fiche produit implique d'appeler de nombreux microservices : images, prix, recommandations, caractéristiques, comparaison, livraison, etc.

Sans solution adaptée, votre client devrait :

- Identifier chaque service
- Localiser leurs instances
- S'authentifier séparément
- Transformer les données selon l'appareil

Cette complexité devient rapidement ingérable. C'est pourquoi il faut une API Gateway. **ZUUL** joue ce rôle parfaitement. Ce point d'entrée unique simplifie drastiquement l'architecture en centralisant :

- L'accès aux microservices
- L'authentification (une seule fois !)
- Les règles de sécurité
- La transformation des données

Résultat : vos clients communiquent avec un seul point, pendant que ZUUL orchestre intelligemment tous les microservices en arrière-plan.

Zuul en pratique : simplifiez l'accès à vos microservices en quelques lignes

Comment transformer une architecture complexe en solution élégante avec Zuul :

Installation express :

- Ajoutez la dépendance Zuul dans votre pom.xml
- Utilisez @EnableZuulProxy dans votre classe principale
- Connectez-le à Eureka pour découvrir automatiquement vos services

Routage intelligent :

- Avec Eureka : accédez à n'importe quel service via /nom-du-service/**
- Sans Eureka : définissez des routes avec zuul.routes.mon-service.path=/chemin/**
- Le client n'a plus besoin de connaître la localisation réelle des services

Sécurité centralisée avec les filtres :

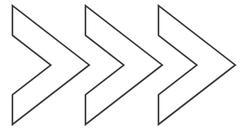
- Filtres pré-requête : authentification, autorisation, validation
- Filtres post-requête : transformation des réponses selon l'appareil
- Tout cela dans un seul endroit, plus besoin de dupliquer dans chaque service !

Résilience automatique :

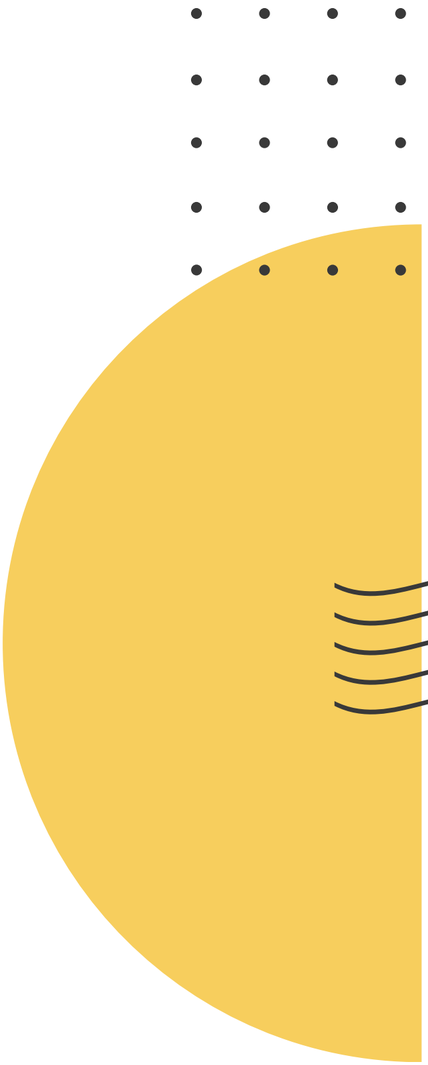
- Configurez des fallbacks pour les services indisponibles
- Définissez des timeouts adaptés à vos besoins

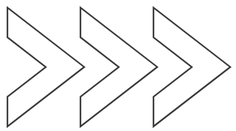
Résultat : votre client appelle simplement api-gateway.com/produits/123 et Zuul s'occupe de tout le reste. Architecture simplifiée et sécurité renforcée en un seul composant !

```
// 1. Configuration du serveur Zuul
// Dans pom.xml du serveur Zuul
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

// Dans application.properties du serveur Zuul
server.port=8080
spring.application.name=api-gateway
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

// Routes statiques (si vous n'utilisez pas Eureka)
zuul.routes.service-produit.path=/produits/**
zuul.routes.service-produit.url=http://localhost:8081/

zuul.routes.service-prix.path=/prix/**
zuul.routes.service-prix.url=http://localhost:8082/

// Configuration de timeouts
zuul.host.connect-timeout-millis=5000
zuul.host.socket-timeout-millis=10000

// Dans la classe principale du serveur
@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ApiGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
}

```

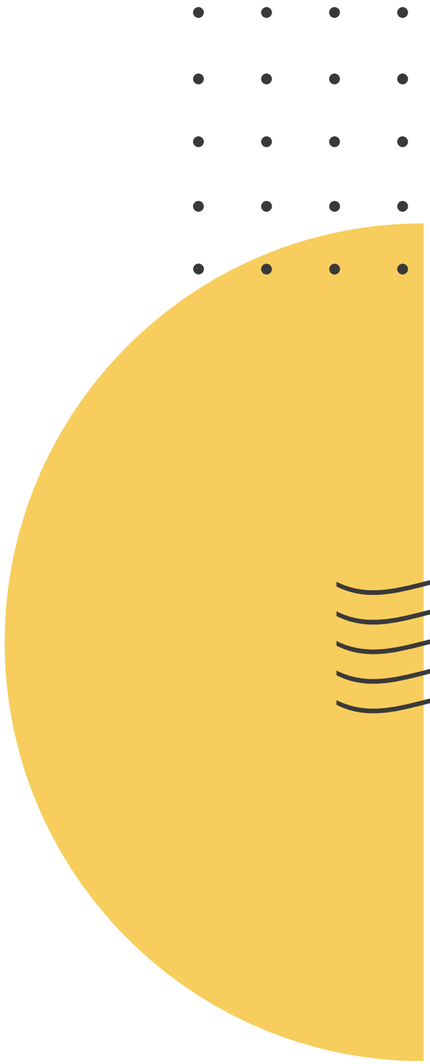


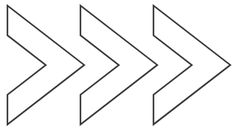
```java
// 2. Création d'un filtre pré-requête (pour l'authentification par exemple)
@Component
public class AuthenticationFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return "pre"; // Filtre appliqué avant la requête
    }

    @Override
    public int filterOrder() {
        return 1; // Ordre d'exécution des filtres
    }

    @Override
    public boolean shouldFilter() {
        return true; // Toujours appliqué
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();

        // Vérification du token d'authentification
        String authHeader = request.getHeader("Authorization");

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            ctx.setSendZuulResponse(false);
            ctx.setResponseStatusCode(401);
            ctx.setResponseBody("Unauthorized");
            return null;
        }

        // Ajout d'un header pour le microservice
        ctx.addZuulRequestHeader("User-Id", "12345");
        return null;
    }
}
```



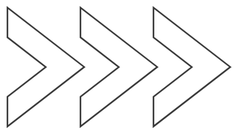


```java
// 3. Création d'un filtre post-requête (pour la transformation de réponse)
@Component
public class ResponseTransformationFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return "post"; // Filtre appliqué après la requête
    }

    @Override
    public int filterOrder() {
        return 10;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletResponse response = ctx.getResponse();

        // Exemple: ajout d'un header à la réponse
        response.addHeader("X-Processed-By", "API-Gateway");
        return null;
    }
}

```





```java
// 4. Fallback pour un service indisponible
@Component
public class ProductServiceFallback implements FallbackProvider {
    @Override
    public String getRoute() {
        return "service-produit"; // Nom du service
    }

    @Override
    public ClientHttpResponse fallbackResponse(String route, Throwable cause) {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() {
                return 200;
            }

            @Override
            public String getStatusText() {
                return "OK";
            }

            @Override
            public void close() {}

            @Override
            public InputStream getBody() {
                return new ByteArrayInputStream("{\"message\":\"Produit temporairement indisponible\"}
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```



5- Le traçage des requêtes

Lorsque votre fiche produit nécessite des appels en cascade entre microservices (par exemple, une demande de paiement qui passe par le service produits puis le service commandes), localiser l'origine d'une erreur devient un vrai casse-tête.
Et quand votre requête traverse des dizaines de microservices ? Impossible de savoir où se situe le problème !

C'est là qu'intervient **Zipkin**, votre détective des microservices. Il trace le parcours complet de chaque requête à travers tous les services, enregistre les temps de réponse et les statuts, puis vous présente une visualisation claire de tout le trajet.

Résultat : vous identifiez immédiatement le microservice défaillant, même dans les architectures les plus complexes.

Zipkin en pratique : tracer vos requêtes en 3 étapes simples
Pour enfin savoir où se cachent vos bugs dans un labyrinthe de microservices :

Configuration ultra-rapide :

- Ajoutez les dépendances zipkin et sleuth dans chaque microservice
- Configurez l'URL du serveur Zipkin et activez l'échantillonnage
- Aucune modification de code nécessaire !

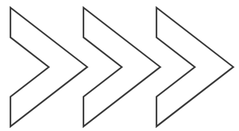
Traçage automatique :

- Sleuth ajoute automatiquement des identifiants uniques à chaque requête
- Ces IDs suivent votre requête à travers tous les microservices
- Les logs incluent désormais trace-id et span-id pour faciliter le débogage

Visualisation intuitive :

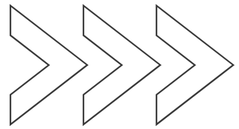
- Lancez le serveur Zipkin (une simple commande Docker)
- Consultez l'interface web sur le port 9411
- Visualisez le parcours complet de chaque requête avec les temps d'exécution

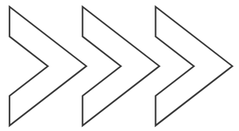
```
// 1. Configuration dans pom.xml pour tous vos microservices
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>

// 2. Configuration dans application.properties de chaque microservice
spring.application.name=service-produit
spring.zipkin.base-url=http://localhost:9411/
spring.sleuth.sampler.probability=1.0

// 3. Lancement du serveur Zipkin
// Option 1: via Docker
// docker run -d -p 9411:9411 openzipkin/zipkin

// Option 2: via Java
// java -jar zipkin-server-2.23.2-exec.jar

```

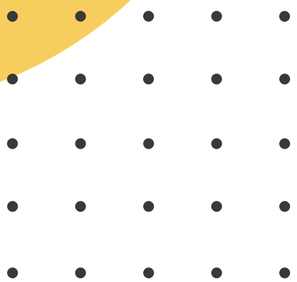


```java
// 4. Exemple d'utilisation dans un service
@RestController
@RequestMapping("/produits")
public class ProduitController {
    private static final Logger logger = LoggerFactory.getLogger(ProduitController.class);

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/{id}")
    public Produit getProduit(@PathVariable Long id) {
        logger.info("Récupération du produit avec l'ID: {}", id);

        // Sleuth ajoutera automatiquement les IDs de trace et span à cette requête
        CommandeStatus status = restTemplate.getForObject(
            "http://service-commande/commandes/statut/{id}",
            CommandeStatus.class,
            id
        );

        logger.info("Statut de la commande récupéré: {}", status);

        // Autre logique...
        return new Produit(id, "Produit Example", 99.99);
    }
}

// 5. Configuration du RestTemplate pour propager les traces
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}

```

```java
// 5. Configuration du RestTemplate pour propager les traces
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}

// 6. Personnalisation avancée du traçage
@Configuration
public class SleuthConfig {
    @Bean
    public SamplerProperties defaultSampler() {
        SamplerProperties samplerProperties = new SamplerProperties();
        samplerProperties.setProbability(1.0); // Trace 100% des requêtes
        return samplerProperties;
    }

    @Bean
    public SpanCustomizer customizer() {
        return new SpanCustomizer() {
            @Override
            public SpanCustomizer name(String name) {
                // Personnalisation du nom du span
                return this;
            }

            @Override
            public SpanCustomizer tag(String key, String value) {
                // Ajout de tags personnalisés
                return this;
            }
        };
    }
}
```







Merci

Philémon Globléhi, Back-end Developer & DevOps Lover
Linkedin: https://www.linkedin.com/in/philemon-globlehi/