

# Tutoriel pour apprendre à mettre en place des patterns de microservices avec Spring Cloud

Par Oleksandr Slynko

Date de publication : 4 mai 2020

Une première partie présentera la gestion de la configuration, avec Spring Config, et la découverte de services, avec Eureka. Elle sera suivie d'une seconde partie, qui abordera les passerelles de services avec Zuul et les notions de circuit breakers, de fallback processing et de bulkhead avec Hystrix.

Pour réagir à ce tutoriel, un espace de dialogue vous est proposé sur le forum **Commentez**

I - Introduction.....	3
II - Gestion de configuration – Spring Cloud Config.....	3
II-A - Introduction du problème.....	3
II-B - Diagramme du pattern.....	3
II-C - Configuration de Spring Cloud Config.....	4
II-D - Exemple de Postman.....	5
II-E - Exemple d'interaction.....	6
II-F - Conclusion.....	7
III - Découverte de services – Spring Cloud/Netflix Eureka.....	8
III-A - Introduction du problème.....	8
III-B - Description du pattern.....	8
III-C - Configuration de Eureka client.....	9
III-D - Exemple d'interaction.....	10
IV - Passerelle de services – Spring Cloud/Netflix Zuul.....	11
IV-A - Introduction du problème.....	11
IV-B - Diagramme du pattern.....	11
IV-C - Description du pattern.....	12
IV-D - Configuration de Zuul.....	12
IV-E - Résumé.....	14
V - Circuit breaker, Fallback, Bulkhead – Spring Cloud/Netflix Hystrix.....	14
V-A - Introduction du problème.....	14
V-B - Description des patterns.....	14
V-B-1 - Circuit breaker.....	14
V-B-2 - Fallback processing.....	14
V-B-3 - Bulkhead.....	14
V-C - Configuration de Hystrix.....	14
VI - Résumé.....	17
VII - Remerciements.....	17

## I - Introduction

Il y a une dizaine d'années, la plupart des applications Web étaient construites selon un style architectural monolithique. Mais la plupart du temps, plusieurs équipes de développement travaillent sur l'application. Chaque équipe de développement a ses propres composants de l'application dont elle est responsable. Avec le temps, l'application grossit et il faut découper le monolithe pour pouvoir développer l'application en parallèle et pouvoir la déployer indépendamment. Le concept de microservices est une réponse directe au défi de la mise à l'échelle d'applications monolithiques.

Parallèlement, **Spring**, avec en particulier **Spring Boot**, est devenu le choix par défaut pour la création d'applications développées en Java. Les microservices étant devenus l'un des modèles d'architecture les plus courants pour la création d'applications basées sur le cloud, la communauté de développement Spring nous a fourni **Spring Cloud** parce que Spring Boot n'implémente pas les patterns spécifiques aux architectures microservices qui vont être présentées. Dans cette série d'articles, nous passerons en revue plusieurs des patterns proposés par Spring Cloud :

- Gestion de la configuration ;
- Découverte de services ;
- Passerelle de service ;
- Circuit breaker, Fallback processing, Bulkhead.

Un premier article présentera la gestion de la configuration, avec Spring Config, et la découverte de services, avec Eureka. Il sera suivi d'un second article, qui abordera les passerelles de services avec Zuul et les notions de circuit breakers, de fallback processing et de bulkhead avec Hystrix.

## II - Gestion de configuration – Spring Cloud Config

### II-A - Introduction du problème

Les données de configuration d'application écrites directement dans le code sont souvent problématiques, car chaque fois qu'une modification de la configuration doit être effectuée, l'application doit être recompilée et/ou redéployée. Pour éviter cela, il est recommandé aux développeurs de séparer complètement les informations de configuration du code de l'application (le troisième facteur de la méthodologie **Twelve-Factor App**).

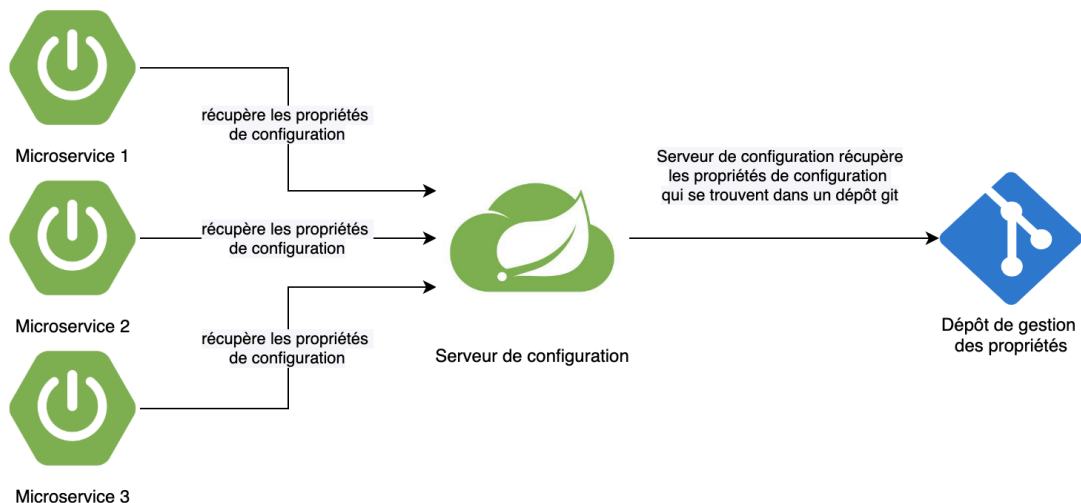
Il est possible d'utiliser un fichier de propriétés (YAML, JSON, XML, etc.) pour stocker leurs informations de configuration.

Cette approche peut s'appliquer à un petit nombre d'applications, mais elle devient rapidement trop lourde lorsqu'il s'agit d'applications distribuées pouvant contenir des centaines de microservices.

Pour résoudre ce problème, nous allons utiliser Spring Cloud Config qui implémente le pattern Gestion de configuration.

### II-B - Diagramme du pattern

Le diagramme ci-dessous présente la récupération de la configuration par les instances de microservices lors de leur démarrage.



## Description du pattern

Spring Cloud Config gère les données de configuration des applications via un service centralisé, de sorte que les données de configuration de vos applications (en particulier celles de votre environnement) soient clairement séparées de votre microservice déployé. Cela garantit que, peu importe le nombre d'instances de microservices que vous avez lancées, elles auront toujours la même configuration. Spring Cloud Config possède son propre dépôt de gestion de propriétés, mais s'intègre également à des projets open source tels que **Git**, **Consul** et **Eureka**. Nous allons utiliser Git dans cet article.

Le développement de microservices dans le cloud nécessite :

- la séparation de la configuration d'une application du code ;
- la construction du serveur et de l'application : les livrables ne seront plus recompilés, mais promus d'un environnement à un autre ;
- l'injection d'informations de configuration d'application au démarrage du serveur par le biais de variables d'environnement ou d'un dépôt centralisé

## II-C - Configuration de Spring Cloud Config

### pom.xml

```
1. <!-- Les dépendances de Spring Cloud Config -->
2. <dependency>
3.   <groupId>org.springframework.cloud</groupId>
4.   <artifactId>spring-cloud-config-server</artifactId>
5. </dependency>
```

### application.yml

```
1. # Spring Cloud Config application.yml
2. server:
3.   port: 8888 # le port de Spring Cloud Config
4. spring:
5.   cloud:
6.     config:
7.       server:
8.         git:
9.           uri: file:///Users/oslynko/IdeaProjects/springmicroservices/config-repo/ # uri du
   dépôt Git qui contient la configuration
```

### ConfigServerApplication.java

```
1. @SpringBootApplication
2. @EnableConfigServer // l'annotation qui permet la gestion de la configuration
3. public class ConfigServerApplication {
```

**ConfigServerApplication.java**

```
4.     public static void main(String[] args) {  
5.         SpringApplication.run(ConfigServerApplication.class, args);  
6.     }  
7. }
```

## II-D - Exemple de Postman

Nous allons utiliser **les profils Spring** pour les tests Postman. Il y a deux fichiers dans le dépôt :

- *message-service.yml* ;
- *message-service-dev.yml*.

La convention de nommage des fichiers est {application}-{profile}.yml.

Nous allons utiliser Postman pour récupérer les propriétés de configuration avec le profil par défaut, puis avec le profil dev.

The top screenshot shows a GET request to `http://localhost:8888/message-service/default`. The response body is a JSON object:

```

1 {
2   "name": "message-service",
3   "profiles": [
4     "default"
5   ],
6   "label": null,
7   "version": "7e1a20b54e5fc8f3031337ea943189b9b20e799a",
8   "state": null,
9   "propertySources": [
10    {
11      "name": "file:///Users/oslynko/IdeaProjects/springmicroservices/config-repo//message-service.yml",
12      "source": {
13        "custom-message": "Hello from default properties"
14      }
15    }
16  ]
17 }

```

The bottom screenshot shows a GET request to `http://localhost:8888/message-service/dev`. The response body is a JSON object:

```

1 {
2   "name": "message-service",
3   "profiles": [
4     "dev"
5   ],
6   "label": null,
7   "version": "7e1a20b54e5fc8f3031337ea943189b9b20e799a",
8   "state": null,
9   "propertySources": [
10    {
11      "name": "file:///Users/oslynko/IdeaProjects/springmicroservices/config-repo//message-service-dev.yml",
12      "source": {
13        "custom-message": "Hello from dev properties"
14      }
15    },
16    {
17      "name": "file:///Users/oslynko/IdeaProjects/springmicroservices/config-repo//message-service.yml",
18      "source": {
19        "custom-message": "Hello from default properties"
20      }
21    }
22  ]
23 }

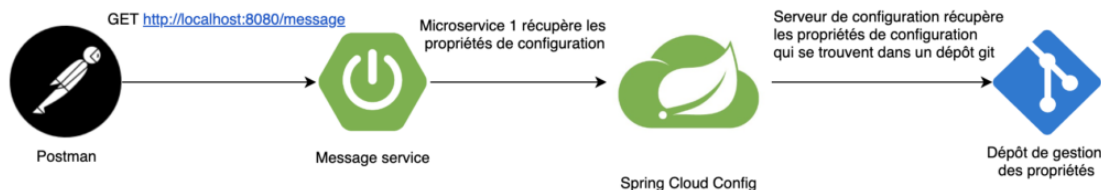
```

En cas d'appel à <http://localhost:8888/message-service/default>, Spring Cloud Config prend les propriétés du fichier `message-service.yml` qui se trouve dans le dépôt.

En cas d'appel à <http://localhost:8888/message-service/dev>, Spring Cloud Config prend les propriétés du fichier `message-service-dev.yml` qui se trouve dans le dépôt.

## II-E - Exemple d'interaction

Pour tester l'intégration, nous allons créer un microservice d'exemple (le service « Message ») qui va récupérer ses propriétés depuis Config Server.



#### application.yml

```

1. # Message service application.yml
2. server:
3.   port: 8080 # le port du microservice d'exemple (Message service)
4. spring:
5.   application:
6.     name: message-service # le nom du service, le nom est utilisé pour identification
7.   cloud:
8.     config:
9.       uri: http://localhost:8888 # Spring Cloud Config uri, toutes les propriétés viennent de ce service
  
```

#### bootstrap.properties

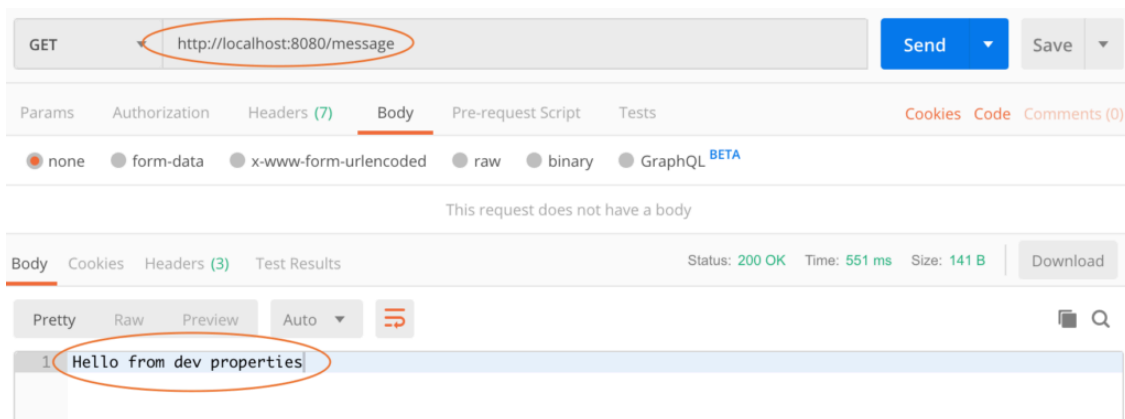
```

# Message service bootstrap.properties
spring.profiles.active=dev
  
```

#### MessageController.java

```

1. @RestController
2. public class MessageController {
3.     @Value("${message}") // la valeur 'message' vient de Spring Cloud Config service
4.     private String message;
5.     @GetMapping("/message")
6.     public String getMessage() {
7.         return message;
8.     }
9. }
  
```



En cas d'appel à <http://localhost:8080/message>, Message service prend les propriétés de Spring Cloud Config qui prend les propriétés du fichier *message-service-dev.yml*.

## II-F - Conclusion

- Spring Cloud Config vous permet de configurer les propriétés de l'application avec des valeurs spécifiques à l'environnement.
- Spring utilise des profils Spring pour lancer un service afin de déterminer les propriétés d'environnement à extraire du service Spring Cloud Config.

- Spring Cloud Config peut utiliser un dépôt de configuration d'application basé sur un fichier ou sur Git pour stocker les propriétés de l'application.

### III - Découverte de services – Spring Cloud/Netflix Eureka

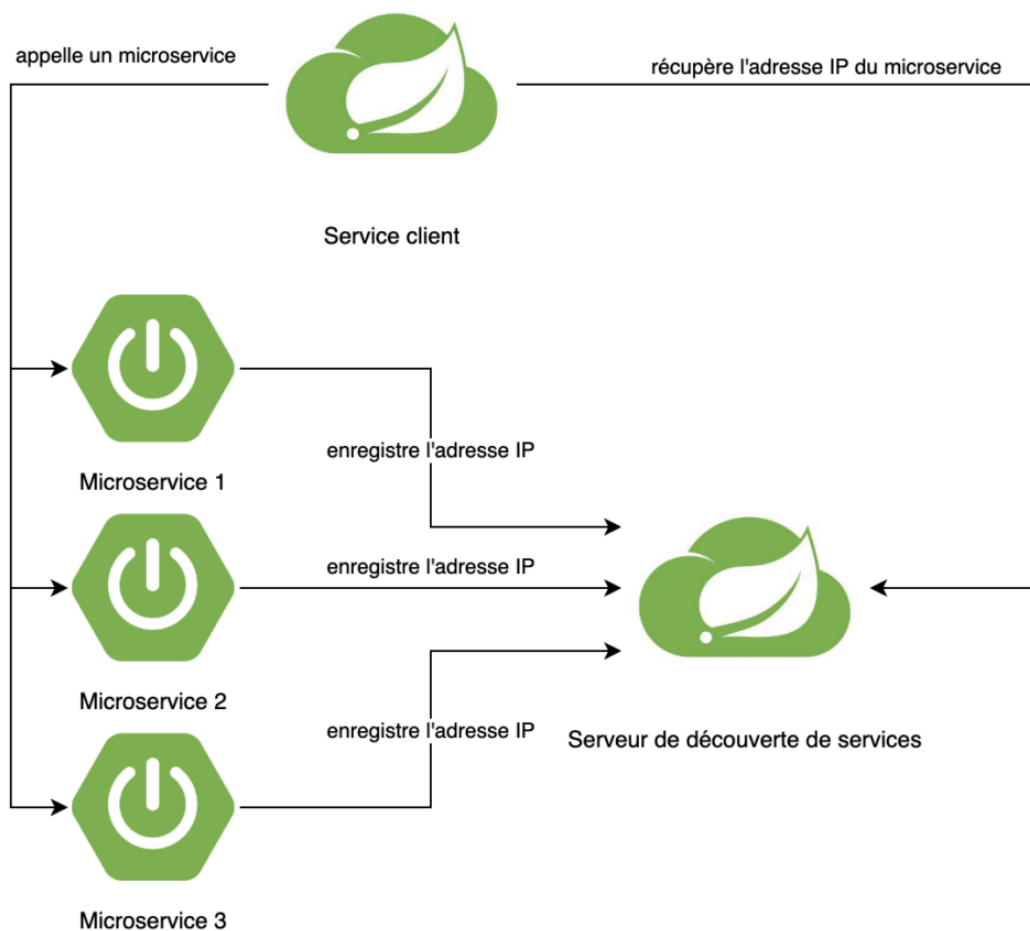
#### III-A - Introduction du problème

Si une équipe gère des serveurs physiques, un fichier de configuration répond essentiellement au besoin de découverte de services, ce fichier permet de savoir quelle adresse utiliser pour appeler un service.

Mais vos services peuvent avoir un emplacement réseau dynamique en raison d'un redémarrage, d'une défaillance et d'une mise à l'échelle. Maintenir manuellement un fichier de configuration n'est tout simplement pas faisable.

Pour résoudre ce problème, nous allons utiliser Eureka qui implémente le pattern Découverte de services.

#### Diagramme du pattern



#### III-B - Description du pattern

- La découverte de services est essentielle pour les microservices pour deux raisons principales. Elle offre à l'application la possibilité d'augmenter rapidement le nombre d'instances de service exécutées dans un environnement. Les consommateurs de services sont abstraits de l'emplacement physique du service via la découverte de services. Étant donné que les consommateurs de services ne connaissent pas



l'emplacement physique des instances de service réelles, de nouvelles instances de service peuvent être ajoutées ou supprimées du pool de services disponibles.

- Cette capacité à adapter rapidement les services sans perturber les utilisateurs de services permet à une équipe de développement habituée à la création d'applications monolithiques d'adopter l'approche la plus puissante pour la mise à l'échelle en ajoutant davantage de serveurs.
- Le deuxième avantage de la découverte de services est qu'elle contribue à augmenter la résilience des applications. Lorsqu'une instance de microservice devient instable ou indisponible, la plupart des moteurs de découverte de services la supprime de la liste interne des services disponibles. Les dommages causés par un service indisponible seront minimisés, car le moteur de découverte de services acheminera les appels aux services disponibles.

## Configuration de Eureka

### pom.xml

```
1. <!-- Les dépendances de Eureka client (message service) -->
2. <dependency>
3.   <groupId>org.springframework.cloud</groupId>
4.   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
5. </dependency>
6. <dependency>
7.   <groupId>org.springframework.cloud</groupId>
8.   <artifactId>spring-cloud-starter-openfeign</artifactId>
9. </dependency>
```

### application.yml

```
1. # Eureka application.yml
2. server:
3.   port: 8761
4. eureka:
5.   client:
6.     registerWithEureka: false # Ne pas s'enregistrer auprès du service Eureka.
7.     fetchRegistry: false # Ne cache pas les informations de registre localement.
8.   server:
9.     waitTimeInMsWhenSyncEmpty: 5 # Temps initial d'attente avant que le serveur accepte les
    demandes
```

### EurekaApplication.java

```
1. @SpringBootApplication
2. @EnableEurekaServer // l'annotation qui permet la découverte de services
3. public class EurekaApplication {
4.   public static void main(String[] args) {
5.     SpringApplication.run(EurekaApplication.class, args);
6.   }
7. }
```

## III-C - Configuration de Eureka client

### pom.xml

```
1. <!-- Les dépendances de Eureka client (message service) -->
2. <dependency>
3.   <groupId>org.springframework.cloud</groupId>
4.   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
5. </dependency>
6. <dependency>
7.   <groupId>org.springframework.cloud</groupId>
8.   <artifactId>spring-cloud-starter-openfeign</artifactId>
9. </dependency>
```

### application.yml

```
1. # Eureka client (message service) application.yml
2. server:
3.   port: 8080
4. spring:
5.   application:
```

## application.yml

```
6.   name: message-service # le nom du service, le nom est utilisé pour identification du
    service
7.   eureka:
8.     instance:
9.       preferIpAddress: true
10.    client:
11.      registerWithEureka: true
12.      fetchRegistry: true
13.      serviceUrl:
14.        defaultZone: http://localhost:8761/eureka/ # url de Eureka Server
```

## III-D - Exemple d'interaction

Nous avons configuré le serveur Eureka et le service Message. Nous allons maintenant les démarrer. Dans les logs du service Message, nous voyons les détails de l'enregistrement du service.

```
2019-07-22 21:17:47.771 INFO 58312 [ Thread-47] o.s.eurekaserverinitializerconfiguration : Started Eureka Server
2019-07-22 21:17:47.797 INFO 58312 [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8761 (http) with context path ''
2019-07-22 21:17:47.798 INFO 58312 [ main] o.s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8761
2019-07-22 21:17:47.799 INFO 58312 [ main] fr.xebia.eureka.EurekaApplication : Started EurekaApplication in 17.941 seconds (JVM running for 23.466)
2019-07-22 21:17:48.254 INFO 58312 [3]-192.168.0.11] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2019-07-22 21:17:48.254 INFO 58312 [3]-192.168.0.11] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2019-07-22 21:17:48.261 INFO 58312 [3]-192.168.0.11] o.s.web.servlet.DispatcherServlet : Completed initialization in 7 ms
2019-07-22 21:18:06.474 INFO 58312 [nio-8761-exec-2] c.n.e.registry.AbstractInstanceRegistry : Registered instance MESSAGE-SERVICE/192.168.0.11:message-service with status UP (replication=false)
2019-07-22 21:18:07.061 INFO 58312 [nio-8761-exec-3] c.n.e.registry.AbstractInstanceRegistry : Registered instance MESSAGE-SERVICE/192.168.0.11:message-service with status UP (replication=true)
```

Dans les logs on peut voir que Message service enregistre l'adresse IP sur le serveur Eureka lors du démarrage.

The screenshot shows the Spring Eureka web interface at localhost:8761. The 'System Status' section displays environment 'test', data center 'default', current time '2019-07-22T21:19:40+0200', uptime '00:02', lease expiration enabled 'false', renew threshold '3', and renew interval '2'. The 'DS Replicas' section shows 'localhost'. The 'Instances currently registered with Eureka' table has columns: Application, AMIs, Availability Zones, and Status. A row shows 'MESSAGE-SERVICE' with 'n/a (1)' AMIs, '(1)' Availability Zones, and 'UP (1) - 192.168.0.11:message-service' status. A green arrow points from the log entry 'Registered instance MESSAGE-SERVICE/192.168.0.11:message-service' to the 'UP (1) - 192.168.0.11:message-service' status in the table.

Application	AMIs	Availability Zones	Status
MESSAGE-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.11:message-service

En cas d'appel à <http://localhost:8761>, on voit que Message service a été enregistré auprès d'Eureka.

Pour tester l'intégration avec Eureka, nous utiliserons **OpenFeign Declarative REST Client** : OpenFeign crée une implémentation dynamique d'une interface décorée avec des annotations JAX-RS ou Spring MVC.

## MessageServiceClient.java

```
1. @FeignClient("message-service") // le nom du service
2. public interface MessageServiceClient {
3.     @RequestMapping("/message") // openfeign va implémenter cette interface pour appeler le
    point de terminaison /message
4.     String getMessage();
5. }
```

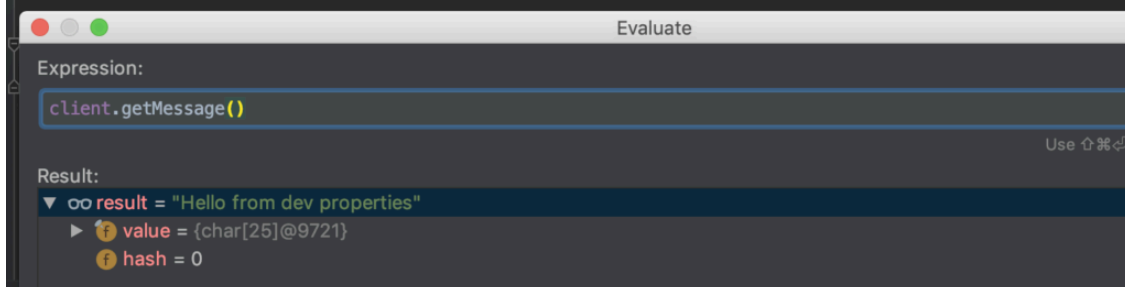
```
@RestController
public class MessageController {

    @Value("${custom-message}")
    private String message; message: "Hello from dev properties"

    private final MessageServiceClient client; client: "HardCodedTarget(type=MessageServiceClient, name=me

    @Autowired
    public MessageController(MessageServiceClient client) {
        this.client = client;
    }

    @GetMapping("/client/message")
    public String getClientMessage() {
        return client.getMessage(); client: "HardCodedTarget(type=MessageServiceClient, name=message-servi
    }
}
```



Sur la capture d'écran, nous voyons qu'OpenFeign a récupéré l'IP de Message service et a appelé le point de terminaison/message.

## Conclusion

- Eureka est utilisé pour abstraire la localisation physique des services.
- Un moteur de découverte de services tel qu'Eureka peut ajouter et supprimer de manière transparente des instances de service d'un environnement sans que les clients du service ne soient affectés.
- Eureka est un projet Netflix qui, lorsqu'il est utilisé avec Spring Cloud, est facile à configurer.

Pour aller plus loin, le code source du projet utilisé comme exemple est disponible sur Github : <https://github.com/slynko/springmicroservices>.

## IV - Passerelle de services – Spring Cloud/Netflix Zuul

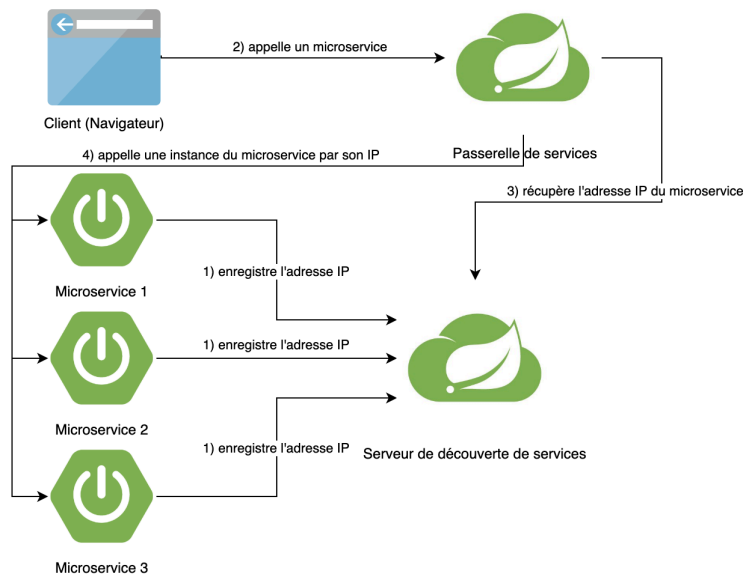
### IV-A - Introduction du problème

Dans une architecture distribuée comme celle des microservices, vous devrez vous assurer que des problématiques clés telles que la sécurité, la journalisation et le suivi des utilisateurs surviennent lors de plusieurs appels de service. Pour implémenter cette fonctionnalité, vous souhaitez que ces attributs soient systématiquement appliqués à tous vos services sans que chaque équipe de développement ait besoin de créer ses propres solutions.

Une passerelle de services sert d'intermédiaire entre un service et le client qui l'appelle. Le client ne parle qu'à une seule URL gérée par la passerelle de services. La passerelle de services distingue le chemin provenant de l'appel du client et détermine le service que le client tente d'invoquer.

### IV-B - Diagramme du pattern

Sur le diagramme, vous pouvez voir comment la passerelle de services dirige l'utilisateur vers une instance du microservice cible.



## IV-C - Description du pattern

Dans la mesure où une passerelle de services se situe entre tous les appels du client aux services individuels, elle agit également comme un PEP (Policy Enforcement Point) central pour les appels de service. L'utilisation d'un PEP centralisé signifie que les problèmes de services transversaux peuvent être mis en œuvre à un seul endroit sans que les équipes de développement aient à mettre en œuvre individuellement des solutions. Les exemples de problèmes transversaux pouvant être implémentés dans une passerelle de services incluent :

- **Routing statique** : une passerelle de services place tous les appels de service derrière une URL et une route d'API uniques. Cela simplifie le développement, car les développeurs n'ont besoin de connaître qu'un seul endpoint final de service pour chacun de leurs services ;
- **Routing dynamique** : une passerelle de services peut inspecter les appels entrants et, en fonction des données de la demande entrante, effectuer un routage intelligent en fonction de l'identité de l'appelant. Par exemple, les clients participant à un programme bêta peuvent avoir tous les appels d'un service acheminés vers un cluster de services spécifique qui exécute une version de code différente de celle utilisée par le reste des utilisateurs ;
- **Authentification et autorisation** : étant donné que tous les appels de service passent par une passerelle de services, cette dernière est un endroit naturel pour vérifier si le client s'est authentifié et est autorisé à appeler le service ;
- **Collecte et journalisation des métriques** : une passerelle de services peut être utilisée pour collecter des métriques et enregistrer des informations lorsqu'un appel de service passe par la passerelle de services. Vous pouvez également utiliser la passerelle de services pour vous assurer que les informations essentielles sont en place dans la demande de l'utilisateur afin de garantir une journalisation uniforme. Cela ne veut pas dire que vous ne devriez pas continuer à collecter des métriques au sein de vos services individuels, mais uniquement qu'une passerelle de services vous permet de centraliser la collecte de vos métriques de base, telles que le nombre de fois où le service est appelé et le temps de réponse du service.

## IV-D - Configuration de Zuul

pom.xml

```

1. <!-- Les dépendances de Zuul -->
2. <dependency>
3.   <groupId>org.springframework.cloud</groupId>
4.   <artifactId>spring-cloud-starter-zuul</artifactId>
5. </dependency>
  
```

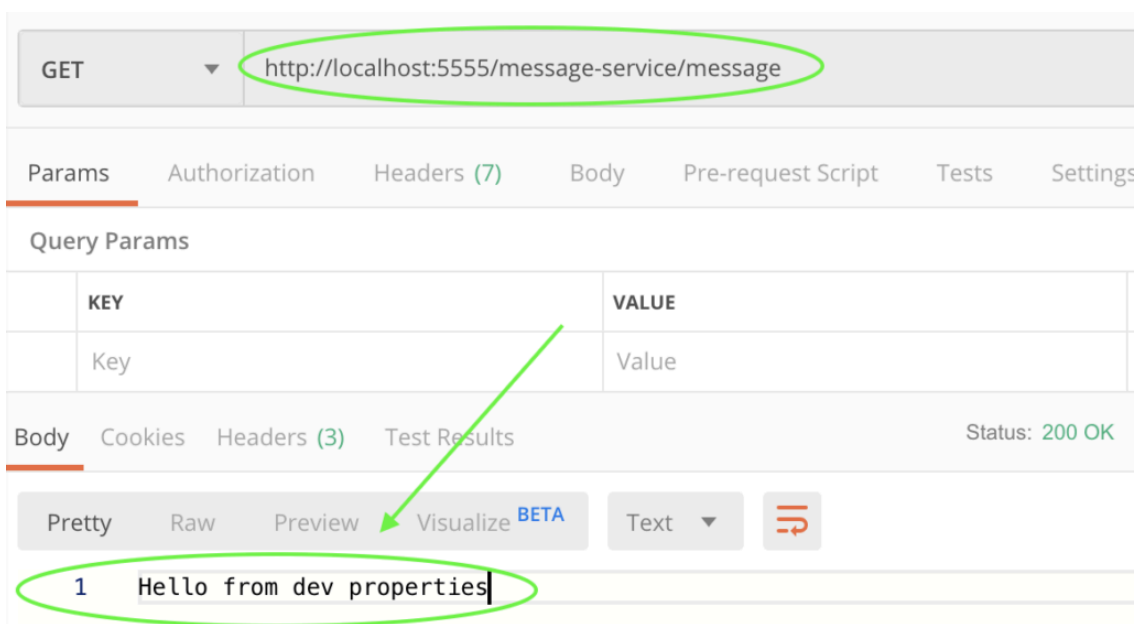
#### application.yml

```
1. server:
2.   port: 5555
3. spring:
4.   application:
5.     name: zuul-gateway
6. eureka:
7.   instance:
8.     preferIpAddress: true
9.   client:
10.    registerWithEureka: true
11.    fetchRegistry: true
12.    serviceUrl:
13.      defaultZone: http://localhost:8761/eureka/
```

#### ZuulApplication.java

```
1. @SpringBootApplication
2. @EnableZuulProxy
3. public class ZuulApplication {
4.   public static void main(String[] args) {
5.     SpringApplication.run(ZuulApplication.class, args);
6.   }
7. }
```

Nous allons utiliser Postman pour appeler l'endpoint/message du service Message-service (dont le port est 8080) via Zuul (dont le port est 5555).



En cas d'appel à <http://localhost:5555/message-service/message>, Zuul récupère son adresse IP auprès d'Eureka, redirige l'appel à Message-service (<http://localhost:8080/message>) et récupère le message « Hello from dev properties » que vous pouvez voir sur la capture d'écran.

Le « message-service » dans l'URL est l'ID du service, l'ID se configure dans *application.yml* sous le nom *spring.application.name*.

Le « /message » correspond à l'endpoint de Message-service.

Par conséquent, <http://localhost:5555/message-service/message> est équivalent à <http://localhost:8080/message> (8080 est le port de Message-service et 5555 est le port de Zuul).

## IV-E - Résumé

- Spring Cloud facilite la création d'une passerelle de services.
- La passerelle de services Zuul s'intègre au serveur Eureka de Netflix et peut automatiquement mapper les services enregistrés auprès d'Eureka sur une route Zuul.
- Zuul peut préfixer tous les itinéraires gérés, afin que vous puissiez facilement préfixer vos itinéraires avec quelque chose comme /api.
- En utilisant le serveur Spring Cloud Config, vous pouvez recharger dynamiquement les mappings de route sans avoir à redémarrer le serveur Zuul.

## V - Circuit breaker, Fallback, Bulkhead – Spring Cloud/Netflix Hystrix

### V-A - Introduction du problème

Les patterns de résilience des clients sont axés sur la protection d'un client de ladite ressource (un autre appel de microservice ou une base de données) en cas de défaillance de la ressource distante. Le but de ces modèles est de permettre au client d'échouer rapidement (**Fail-fast**), de ne pas consommer de ressources, telles que les connexions de base de données et les pools de threads, et d'empêcher le problème du service distant de se propager « en amont » aux consommateurs du client.

### V-B - Description des patterns

#### V-B-1 - Circuit breaker

Lorsqu'un service à distance est appelé, le **circuit breaker** surveillera l'appel. Si les appels prennent trop de temps, le circuit breaker interviendra et met fin à l'appel. De plus, le circuit breaker surveillera tous les appels vers une ressource distante et si trop d'appels échouent, l'implémentation de la coupure de circuit apparaîtra, échouant rapidement et empêchant les futurs appels vers la ressource distante défaillante.

#### V-B-2 - Fallback processing

Avec le fallback pattern, lorsqu'un appel de service distant échoue, plutôt que de générer une exception, le consommateur de service exécute un code alternatif et tente d'exécuter une action par un autre moyen. Cela implique généralement la recherche de données provenant d'une autre source de données ou la mise en file d'attente de la demande de l'utilisateur en vue d'un traitement ultérieur. L'appel de l'utilisateur ne fera pas l'objet d'une exception signalant un problème, mais il se peut qu'il soit averti que sa demande devra être satisfaite à une date ultérieure.

#### V-B-3 - Bulkhead

En utilisant le bulkhead pattern, vous pouvez grouper les appels de ressources distantes dans leurs propres pools de threads et réduire le risque qu'un problème lié à une lenteur sur un appel de ressource distante entraîne la destruction de toute l'application. Les pools de threads agissent comme des cloisons pour votre service. Chaque ressource distante est séparée et assignée au pool de threads. Si un service répond lentement, le pool d'unités d'exécution pour ce type d'appel de service devient saturé et arrête le traitement des demandes. Les appels à d'autres services ne deviennent pas saturés, car ils sont attribués à d'autres pools de threads.

### V-C - Configuration de Hystrix

**Hystrix** n'est plus en développement actif et est actuellement en mode maintenance. **Resilience4j** constitue une alternative, mais **Spring Cloud** ne l'utilise pas encore. Voir [ici](#) pour les détails de la configuration.

### pom.xml

```
1. <!-- Les dépendances de Hystrix -->
2. <dependency>
3.     <groupId>org.springframework.cloud</groupId>
4.     <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
5. </dependency>
```

### MessageServiceApplication.java

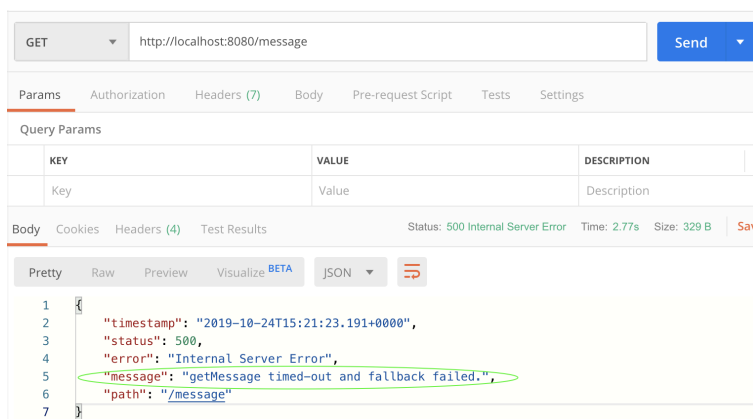
```
1. @SpringBootApplication
2. @EnableCircuitBreaker
3. public class MessageServiceApplication {
4.     public static void main(String[] args) {
5.         SpringApplication.run(MessageServiceApplication.class, args);
6.     }
7. }
```

### MessageController.java

```
1. @RestController
2. public class MessageController {
3.     @Value("${custom-message}")
4.     private String message;
5.     @GetMapping("/message")
6.     @HystrixCommand
7.     public String getMessage() {
8.         return message;
9.     }
10. }
```

L'annotation `@HystrixCommand` active le circuit breaker.

Maintenant, avec l'annotation `@HystrixCommand` en place, le service interrompra un appel vers sa base de données si la requête prend trop de temps. Si les appels de base de données prennent plus de 1000 millisecondes pour exécuter le wrapping de code Hystrix, votre appel de service lève une exception `com.netflix.hystrix.exception.HystrixRuntimeException` que nous pouvons voir sur la capture d'écran.



Maintenant nous allons changer la configuration du MessageController pour configurer la Fallback processing et un pool de threads (Bulkhead).

### MessageController.java

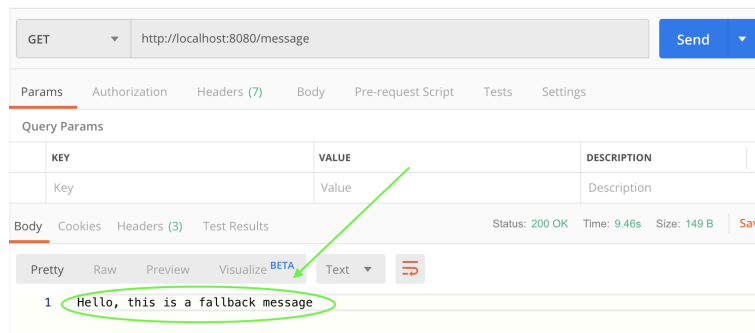
```
1. @RestController
2. public class MessageController {
3.     @Value("${custom-message}")
4.     private String message;
5.     @GetMapping("/message")
6.     @HystrixCommand(fallbackMethod = "buildFallbackMessage",
7.         threadPoolKey = "messageThreadPool")
8.     public String getMessage() {
9.         return message;
10.     }
```

## MessageController.java

```
9.    }
10.   private String buildFallbackMessage() {
11.       return "Hello, this is a fallback message";
12.   }
13. }
```

L'annotation `@HystrixCommand(fallbackMethod = "buildFallbackMessage"` va utiliser la méthode `buildFallbackMessage` si la méthode `getMessage` prend trop de temps pour s'exécuter.

Sur la capture d'écran, nous pouvons voir le message « Hello, this is a fallback message » qui est construit dans la méthode `buildFallbackMessage`.



Dans une application basée sur une architecture en microservices, il est souvent nécessaire d'appeler plusieurs microservices pour effectuer une tâche particulière. Sans utiliser de bulkhead pattern, le comportement par défaut de ces appels est que ceux-ci sont exécutés à l'aide des mêmes threads que ceux réservés pour la gestion des demandes pour l'ensemble du conteneur Java. En cas de volumes élevés, des problèmes de performances avec un service sur plusieurs peuvent avoir pour résultat que tous les threads du conteneur Java sont saturés et en attente de traitement, tandis que les nouvelles demandes de travail sont sauvegardées. Le conteneur Java finira par planter. Le bulkhead sépare les appels de ressources distantes dans leurs propres pools de threads, de sorte qu'un seul service défectueux puisse être contenu et ne provoque pas le blocage du conteneur.

Hystrix utilise un pool de threads pour déléguer toutes les demandes de services distants. Par défaut, toutes les commandes Hystrix partageront le même pool de threads pour traiter les demandes. Ce pool de threads comportera 10 threads pour traiter les appels de service distant. Voir [ici](#) pour plus de détails sur les propriétés de `@HystrixCommand`. Il est également possible de faire des appels à des services REST, à une base de données, etc.





Lorsque la cloison est activée, le pool de threads est utilisé, par exemple `messageThreadPool` comme nous pouvons voir sur la capture d'écran.


## VI - Résumé

- Lors de la conception d'applications hautement distribuées telles qu'une application basée sur une architecture en microservices, la résilience du client doit être prise en compte.
- Un seul service peu performant peut déclencher un effet en cascade d'épuisement des ressources, car les threads du client appelant sont bloqués dans l'attente de la fin du service.
- Les principaux modèles de résilience des clients sont le circuit breaker, le fallback et le bulkhead.
- Le circuit breaker cherche à supprimer les appels système lents et dégradés afin que les appels échouent rapidement et évitent l'épuisement des ressources.
- Le fallback vous permet, en tant que développeur, de définir d'autres chemins de code en cas d'échec d'un appel de service distant.
- Le bulkhead sépare les appels de ressources distantes les uns des autres, isolant les appels d'un service distant dans leur propre pool de threads. Si un ensemble d'appels de service échoue, ses échecs ne doivent pas être autorisés à consommer toutes les ressources du conteneur d'applications.
- Spring Cloud et les bibliothèques Netflix Hystrix fournissent des implémentations pour les patterns circuit breaker, fallback et bulkhead.

Pour aller plus loin, le code source du projet utilisé comme exemple est disponible sur Github : [https://github.com/slynko/springmicroservices/tree/zuul\\_hystrix](https://github.com/slynko/springmicroservices/tree/zuul_hystrix).

## VII - Remerciements

Cet article a été publié avec l'aimable autorisation de la société **Publicis Sapient Engineering (anciennement Xebia)** qui est la communauté Tech de Publicis Sapient, la branche de transformation numérique du groupe Publicis.

Nous tenons à remercier **Claude Leloup** pour sa correction orthographique et  **Winjerome** pour la mise au gabarit.