Api

Api

Weater-Api

fund-Api

currency-Api

GIT

Spring-cloud-config-server

Currency-Api

DEV

DEV1

QA

QA1

QA2

STAGE

STAGE 1

PROD

PROD 1

PROD 2

PROD 3

DEV

DEV1

QA

QA1

QA2

STAGE

STAGE 1

Fund-Api

PROD

PROD 1

Communiquer avec les 3 instances avec l’implantation de Ribbon comme load balancer

EurekaNamingServer

Weater-Api

Currency-Api

Fund-Api

**EurekaNamingServer** joue 2 roles

\*Service registration : les apis démarrées sont enregistrées dans l’annuaire d eureka

\*Service discovery : eureka fourni l’annuaire de la liste des api activés au clients

\* fonction utile aussi , elle permet au load balancer(Ribbon) de trouver les instance sans

paramètré les url avec les port en dur dans l’application.properties

Ribbon

Fund-Api

NamingServer

Fund-api communique avez 3 apis Currency-Api , on passant par le load balancer **Ribbon**

Currency-Api-1

Currency-Api-2

Currency-Api-3

Eureka server

**Tracing server**

Pour implémenter la fonction de tracing server ou un log centralise des applications, on a besoin d’installer rabbitMQ , Zipkin server qui écoute au changement fait dans la queue de rabitMQ et Sleuth-zipkin plus un bus-amqp pour l’envoi des log au rabbitMQ

Spring-cloud-sleuth permet de tagger chaque requête par un identifiant unique(nom-application + idRequestParent + idRequestchild)

Pour l’installation de rabbitMQ il faut :

Installer rabbitMQ : <https://www.rabbitmq.com/install-windows.html>

Installer Erland : <https://www.rabbitmq.com/which-erlang.html>

<http://www.erlang.org/downloads>

Activé la queue de rabbitMQ : rabbitmq-server start / stop

Pour intaller Zipkin , il faut récurer le jar de zipkin à partir de https :zipkin.io/pages/quickstart et cliquer sur last release pour récupérer le jar

Pour démarrer Zipkin , dans l’invite exécuté

SET RABBIT\_URI=amqp://localhost ajouter cette variable dans les variables d’environnement ou exécuter la dans l’invite de commande ça permet de dire à zipkin de savoir où se trouve rabbitmq

java –jar zipkin.jar lancer zipkin pour écouter qui va écouter les messages de la queue zipkin

dans rabbitMQ

Tester le démarrage de zipking :

Utiliser l’url suivante : http://localhost/9411/zipkin

Il faut savoir qu’il faut lancer le rabbitMQ par ce que zipkin écoute toujours les mouvements

dans le rabbitMQ

Pour dire à zipkin qu’il y a un rabbit-MQ on ajouter un code a l’url de démarrage de Zipking

Dans les versions 2+ de spring boot zipkin est lancer dans un invite de commande

Pour afficher ce qui ce passe dans la queue de rabbitMQ il faut activer le managger avec la commande rabbitmq-plugins enable rabbitmq\_management

l’url pour la configuration est <http://localhost:15672/>

Session : guest Passwor : guest c’est par défaut après on peut le changer mais c’est user admin

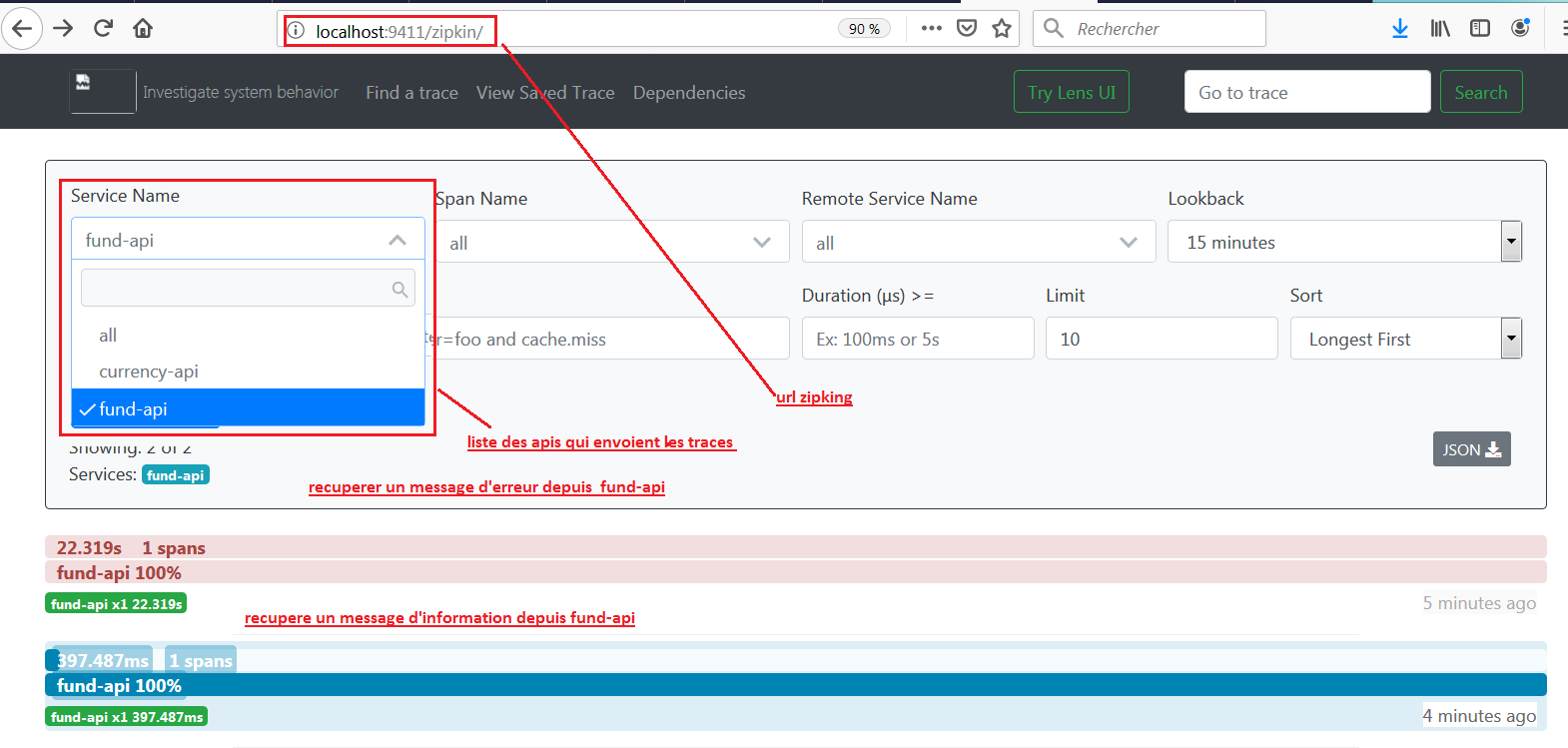
**Move messages**

Il faut activer le plugin shovel pour deplacer les messages

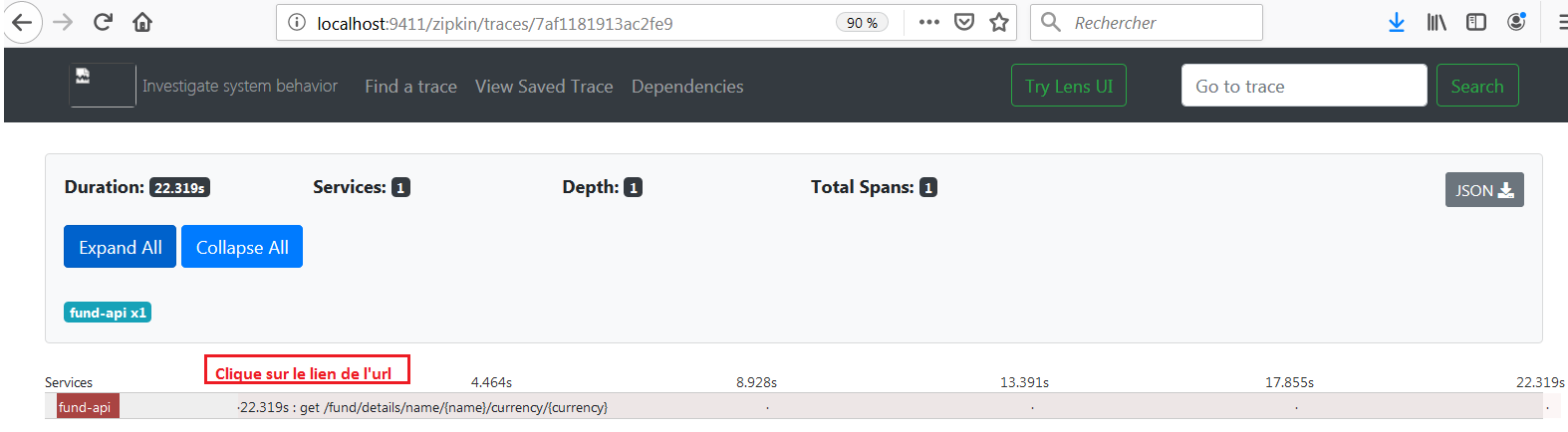
rabbitmq-plugins enable rabbitmq\_shovel rabbitmq\_shovel\_management

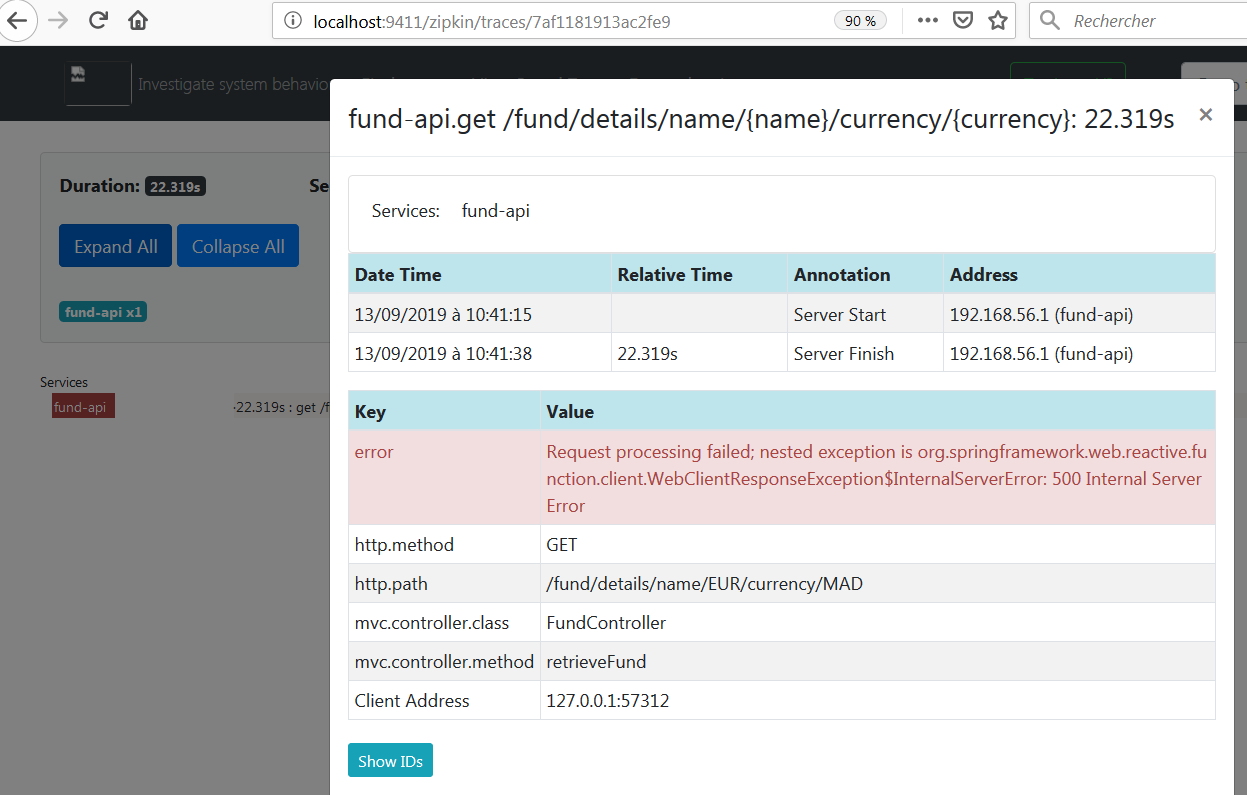
RabbitMQ est lancé comme un service windows en arrière-plan

Plus de détails : https://www.rabbitmq.com/management.html

Visuel des traces envoyées par les apis

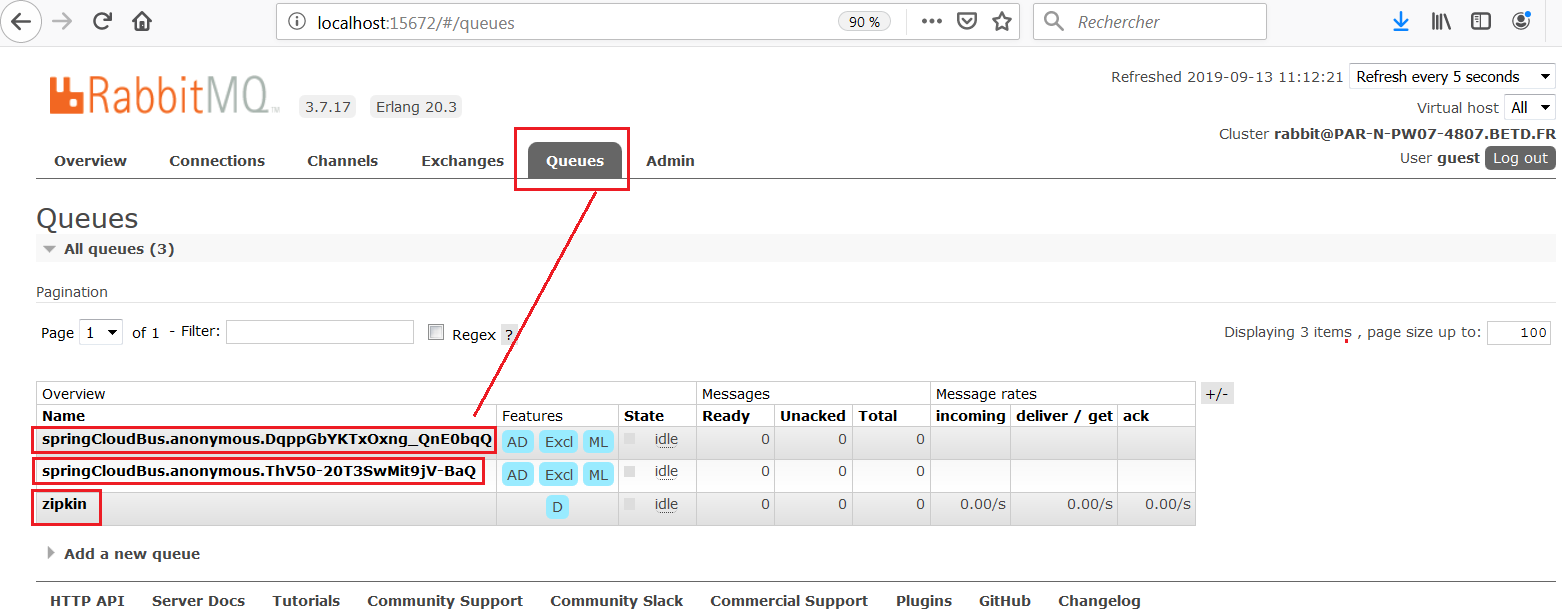
Détails de message d’erreur :





On peut voir les différentes queues dans le manger de rabbitMQ

Les queues sont créé par spring avec le module **spring-cloud-starter-bus-amqp**

**spring-cloud-starter-bus-amqp** relie les nœuds d’un système distribué à un courtier de messages léger. Ce courtier peut ensuite être utilisé pour diffuser les changements d'état (tels que les modifications de configuration) ou d'autres instructions de gestion. Une idée clé est que le bus est comme un actionneur distribué pour une application Spring Boot qui est mise à l'échelle. Cependant, il peut également être utilisé comme canal de communication entre les applications. Ce projet fournit des démarreurs pour un courtier AMQP ou pour Kafka en tant que transport.

Les dépendances Maven

<!-- spring zipkin server des traces applicatifs -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-sleuth</artifactId>

</dependency>

Cette dépendance ajoute un ID unique aux requêtes

<!-- spring zipkin server des traces applicatifs -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-sleuth-zipkin</artifactId>

</dependency>

Cette dépendance permette de formater le message de log sous le format attendu par zipkin et qui sera envoyé pour être stocké dans une queue de rabbitMQ

<!-- spring outil pour l'envoi de message de rabitMQ à zipkin server -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

Cette dépendance permette d’envoyer un message à rabbitMQ , on utilisant le protocole amqp on a déjà spécifié que zipkin server va utiliser le protocol amqp SET RABBIT\_URI=amqp://localhost

Weater-Api

currency-Api

fund-Api

Rabbit-MQ

ZipkinDistributedTracingServer

DB

Envoi Log applicatif

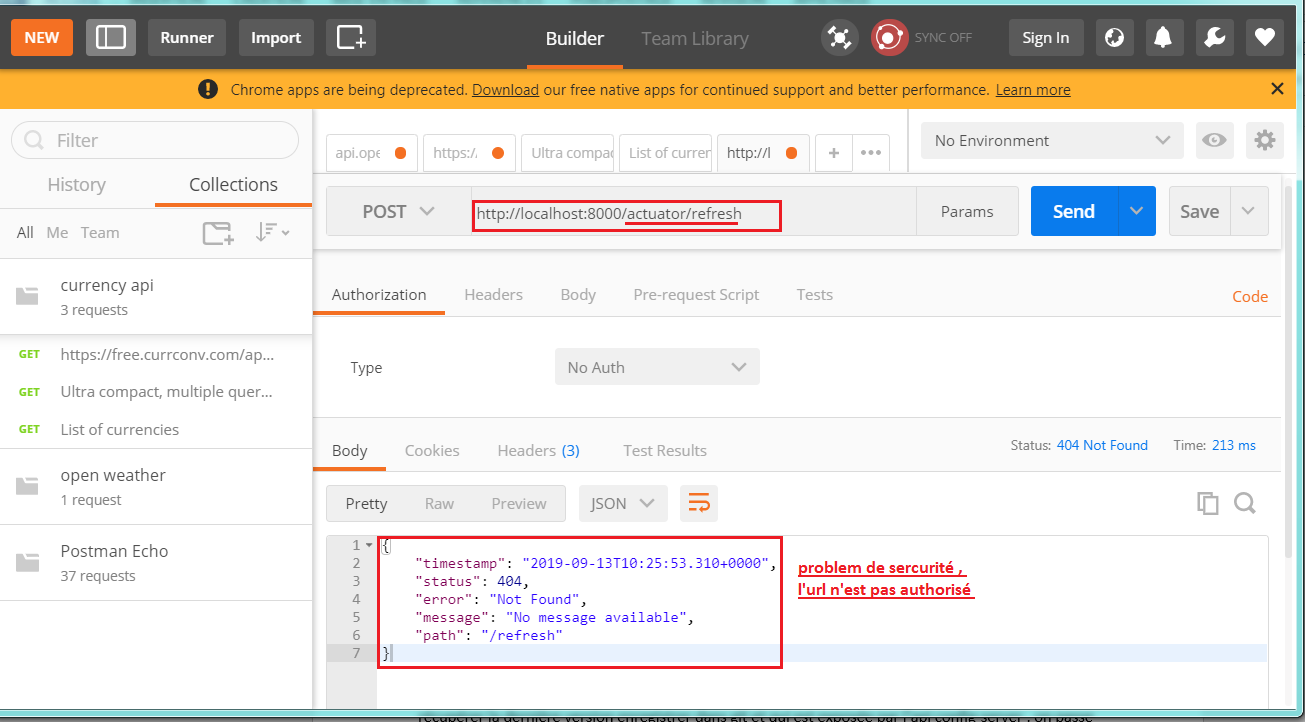
Envoi Log applicatif

Envoi Log applicatif

# **Spring Cloud Bus**

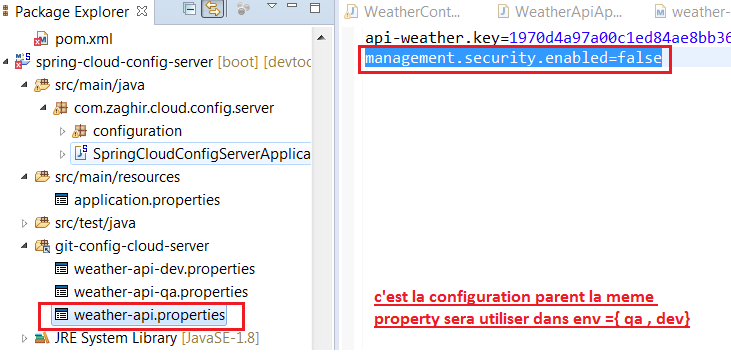
On va partir de l’exemple intéressant pour expliquer le rôle de spring cloud bus

On va faire un passe sur le service spring-config-server qui expose les config des applications sous format api

Pour mettre à jour la configuration de l’api weather-api , il faut passer une requête de type post pour récupérer la dernière version enregistrer dans git et qui est exposée par l’api config-server , on passe la requête avec postman : **http://localhost:8000/actuator/refresh**

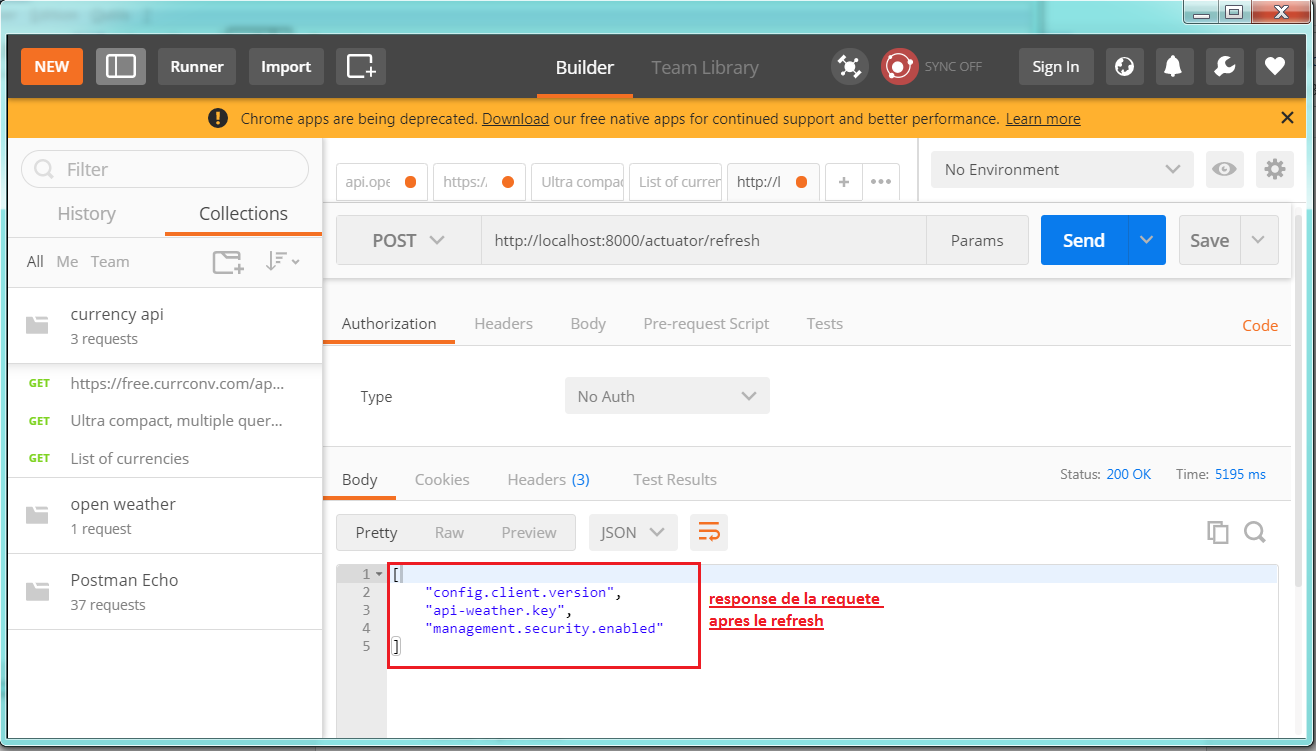
Pour résoudre ce problème de sécurité, on va désactiver la sécurité sur l’actuator

Dans le fichier versionner par git , on ajoute la propriété **management.security.enabled=false**



Apres la modification des fichier on pousse les modification sur git avec git add , commit

Et faire une requête de refresh **http://localhost:8000/actuator/refresh** sur l’instance de l’api , ici c’est sur le port 8000



Il faut savoir que si on a plusieurs instances de l’api weather(8000,8001,8002) et si on change dans les propriétés il faut faire une requête refresh sur toutes les instances et c’est compliqué pour gérer ça dans une architecture cloud

La solution c’est d’utiliser un bus avec un queue dans notre car en va utiliser rabbitMQ avec le protocole amqp , donc ce qu’il faut faire :

Ajouter la dépendance spring-cloud-starter-bus-amqp dans l’api sprring-cloud-config-server et Weather-api

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

On crée une nouvelle instance pour faire le test , instance sur le port 8001

Click droit sur la classe main -> run as -> faire une copie sur l’instance qui existe

Maintenant on va mettre à jour toutes les instances de Weather-api on utilisant le bus spring cloud

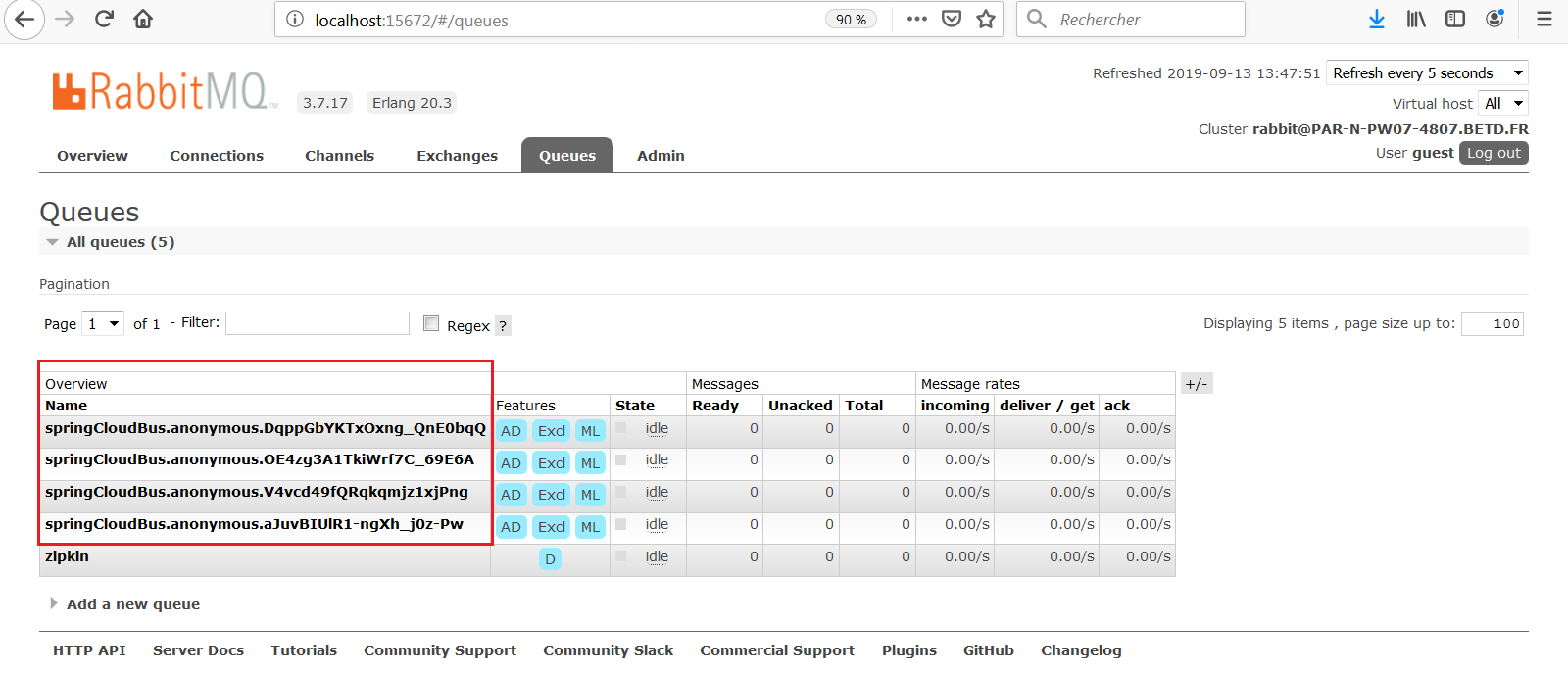
L’url pour le refresh est [**http://localhost:8000/actuator/bus-refresh**](http://localhost:8000/actuator/bus-refresh)et non plus [**http://localhost:8000/actuator/refresh**](http://localhost:8000/actuator/refresh) **<- c’est seulement pour une instance**

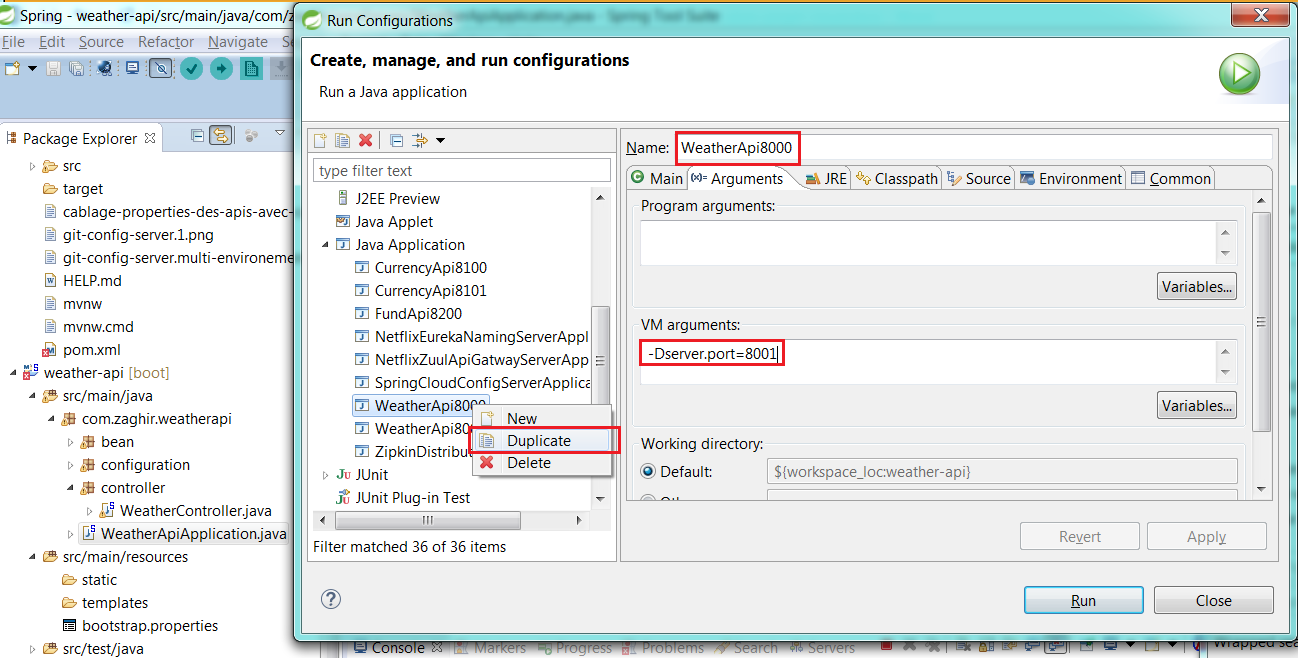
Cette requête va mettre à jour toutes les instances démarrée et c’est cool

La magique de ce fonctionnement c’est que chaque instance est enregistrée avec un bus utilisant le protocole amqp et enregistrer dans rabbitMQ

S’il y a un changement dans une instance de micro-service et s’il y a un appel à refresh , le micro-service envoie un évènement est envoyé à travers spring-cloud-bus , spring-cloud-bus propage cet évènement à tous les micrco-service enregistrée eux aussi avec un bus de spring-cloud-bus

Si en regarde rabbitMQ , on trouve les instances des apis avec leur bus de communications





# **Circuit Breacker avec Spring Cloud hystrix fault tolerance (tolérance de panne)**

L’architecture micro service est composée de plusieurs composants. Les services peuvent communiquer entre eux, et c’est possible qu’ils soient bloqués par d’autres services qui ne répondent plus ou ils sont désactivés, c’est dans ce cas ou la tolérance de panne des micros services intervient

Dans cette approche pour la tolérance des pannes, il faut spécifier un comportement en mode dégradé qui répond aux besoins des autres services au cas problèmes

Hystrix Framework permet de créer des services qui tolèrent les pannes

currency-Api

Weater-Api

fund-Api

Spring-cloud-config-server

Fund-api is down

FallBack

currency-Api

Weater-Api

fund-Api

Fund-api is down

FallBack

Spring-cloud-config-server

Api A

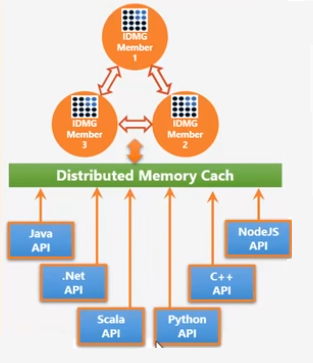
Api B

Utilisation de l’api B même si elle est chère car l’api A ne répond plus

**Utilisation d’un cache pour minimiser le nombre d’appels au apis externes**

Solution utilisé : **Hazelcast**

Url : https://hazelcast.org

Petit définition :

* Hazelcast IMDG(In Distributed Memory Grid) est un middleware Open source en java , qui permet de créer un cache en mémoire distribué.
* Dans une grille Hazelcast, les données sont reparties uniformément entre les nœuds d’un groupe d’ordinateurs, ce qui permet
  + Un stockage distribué Scalable(Distributed Memory Cache)
  + Un traitement distribué Scalable (Distributed Computing).
  + Réplication des données sur plusieurs nœud pour tolérance au panes
* Ces technique réduisent la charge de requête sur une base de données et améliorent les performances des systèmes distribués.

Hazelcast est une solution quand vous avez besoin de :

* Application analytiques nécessitant un traitement de données volumineuses en partitionnant les données (Big Data)
* Pour conserver les données fréquemment consultées dans la grille mémoires
* Un cache mémoire distribué hautement disponible pour les applications
* Un magasin de données principal pour les applications avec des exigences de performances d’évolutivité et de latence maximales
* In Memory NoSQL data base de type clé-valeurs
* Solution de messagerie(publier / souscrire) très rapide avec un scalabilté entre les applications
* Solution faire distribuer les traitements(Distribued Compution)
* Une alternative aux autres solutions comme Coherence and Terracotta