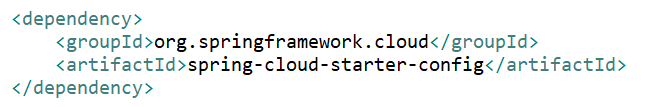
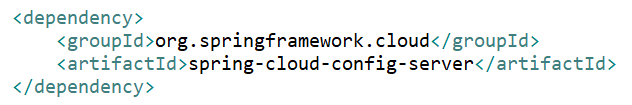
Microservices V1

1. Spring Cloud Config Client
   1. Add Dependency in pom.xml

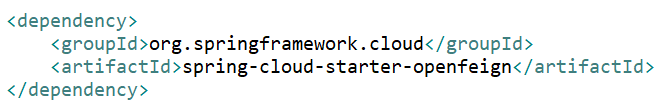


1. Spring Cloud Config Server
   1. Add Dependency in pom.xml

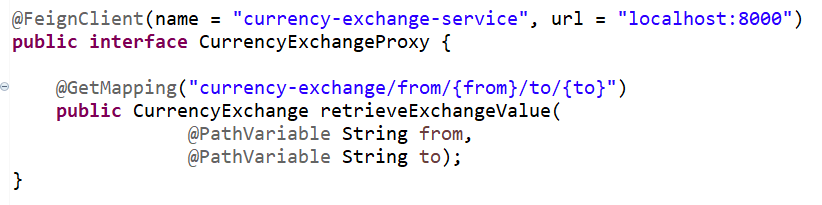


* 1. Add @EnableConfigServer at class level in application.java

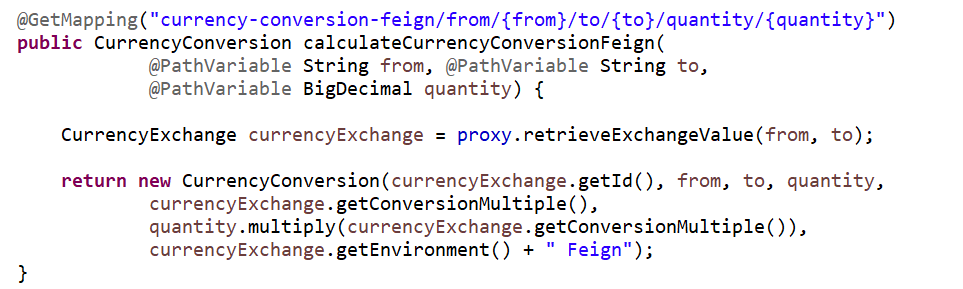
1. H2 In memory Database
2. JPA Repository
3. Rest Template
4. Spring Cloud Open Feign Client
   1. Add dependency in pom.xml



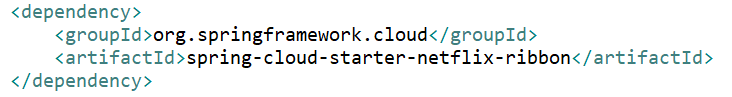
* 1. Create Proxy Interface



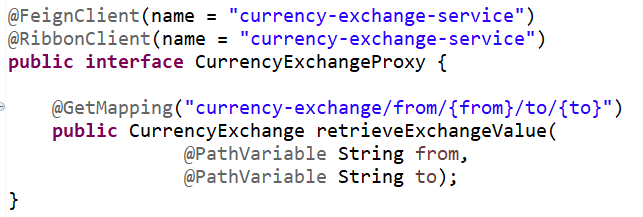
* 1. @Autowired Proxy Interface in controller call function instead of calling RestTemplate.getForEntity()



1. Client-side load balancing using Ribbon
   1. Add dependency in pom.xml



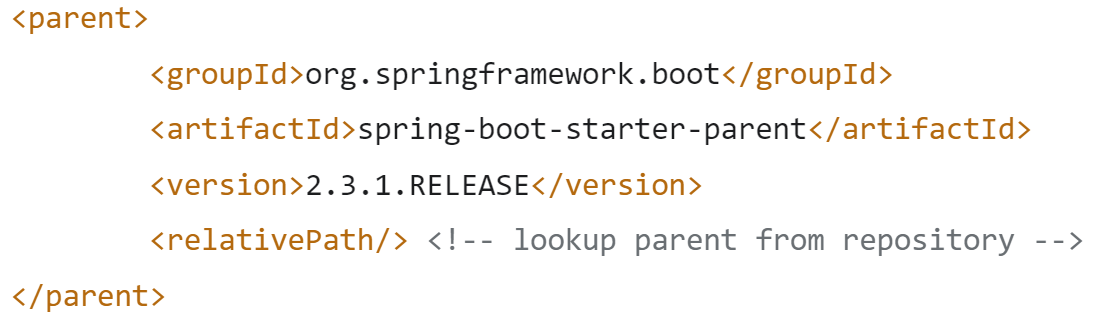
* 1. Add **@RibbonClient** in Proxy Interface at Interface level and remove url from @FeignClient

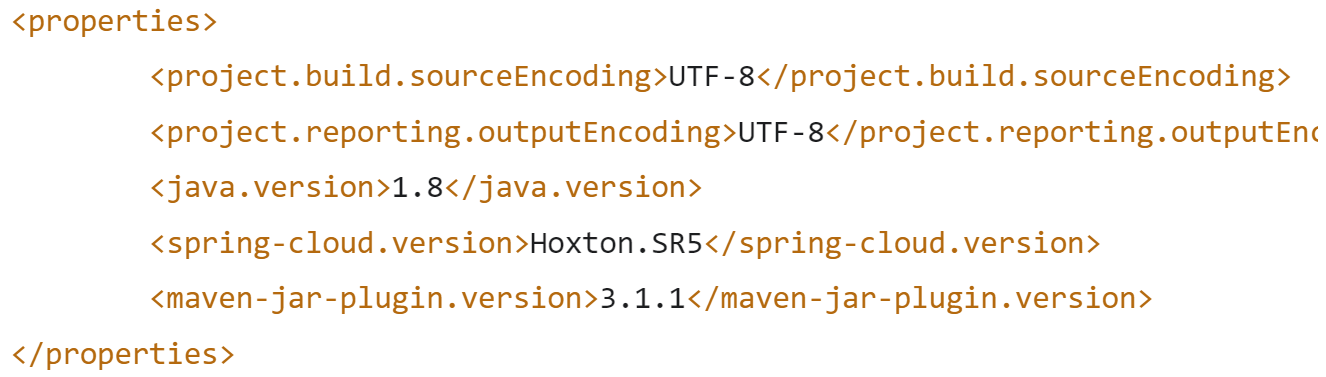


* 1. Add list of servers in application properties file

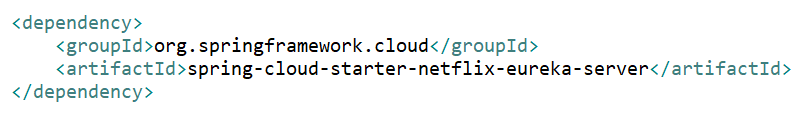


* 1. **Ribbon** DOES NOT work with **Spring Boot 2.4**
     1. For old spring boot concept we recommend using **2.3.1.RELEASE** with **Hoxton.SR5**.

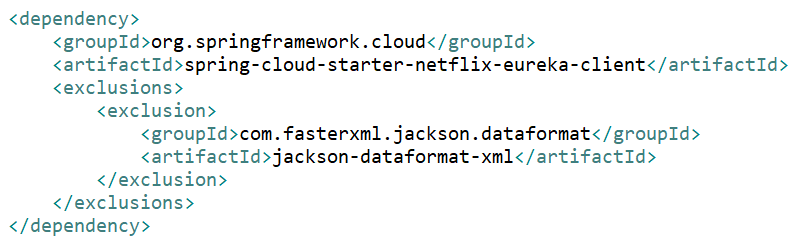




1. Spring Cloud Eureka Server (Service Discovery)
   1. We register all instance of microservices in eureka server
   2. Ribbon take instance url from eureka to call api instead of using hardcoded url in application properties file which is called Service Discovery.
   3. Add dependency in pom.xml



1. Register microservices to Eureka Server (Spring Boot Eureka Client)
   1. Add dependency in pom.xml

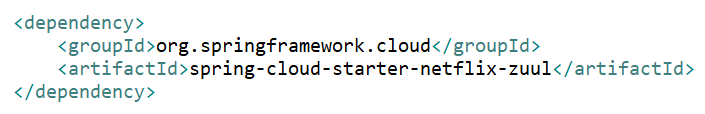


* 1. Add @EnableDiscoveryClient annotation at class level in Application.java file
  2. Add property in application properties file

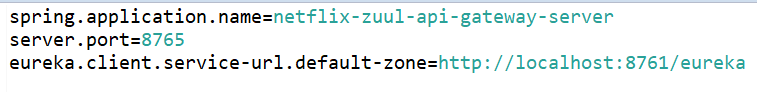


* 1. Distributing calls using Eureka and Ribbon
     1. Remove **currency-exchange.ribbon.listOfServers** from application properties file which enable to Ribbon to distribute call using eureka server.

1. Introduction to API Gateway (Spring Cloud Netflix Zuul)
   1. API gateway used for to implement:
      1. Authentication, authorization, and security
      2. Rate limits
      3. Fault toleration
      4. Service aggregation
   2. Add dependency in pom xml file



* 1. Add @EnableZuulProxy annotation at class level in Application.java file
  2. Change application properties file



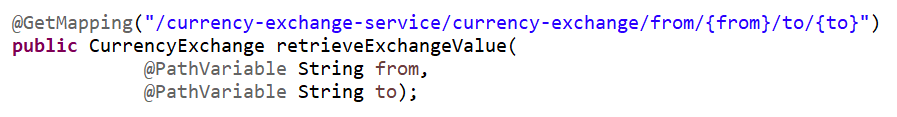
* 1. Implementing ZuulFilter for Logging
     1. Create Logging Class which extends ZuulFilter



* + 1. Set filterType as “pre”, “error” or “post”
  1. Use Zuul server application name in @FeignClient() at client webservices for using gateway api server for load balancing

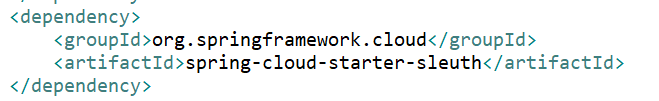


* 1. Add application name in @GetMapping() URL to call api through gateway api

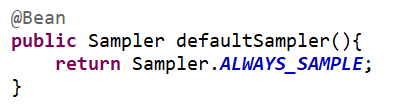


* 1. The Zuul uses Application Name in the url to talk to Eureka and find the url of the service

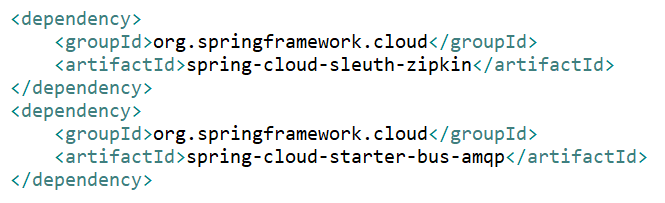
1. Introduction to Distributed Tracing
   1. We use Spring Cloud Sleuth with Zipkin for distributed tracing.
   2. Spring cloud sleuth would assign a unique ID to a request so that we can trace it across components zipkin is what we call a distributed tracing system.
   3. What we would do is all the log from all these services we would put it in MQ we would use RabbitMQ, and we would send it out to the zipkin server where it is consolidated, and we would be able to look through the different requests and find what happened with the specific request.
   4. Implementing Spring Cloud Sleuth
      1. Add dependency in API Gateway Service and other microservices



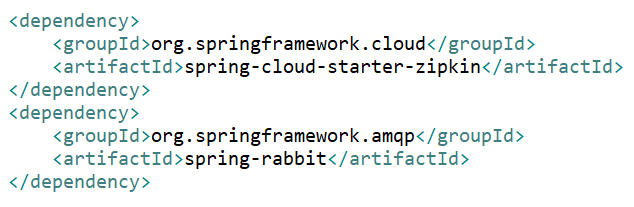
* + 1. Add below code inside Application java file API Gateway Service and other microservices



* 1. Setup Rabbit MQ
     1. Download and Install Erlang
     2. Download and Install Rabbit MQ (https://www.rabbitmq.com/install-windows.html)
     3. Run cmd as admin and run cd C:\Program Files\RabbitMQ Server\rabbitmq\_server-3.9.5\sbin
     4. To start Rabbit MQ service: rabbitmq-service start
     5. To stop Rabbit MQ service: rabbitmq-service stop
     6. To check all options: rabbitmq-service
  2. Setup Zipkin Server (https://zipkin.io/pages/quickstart)
     1. Download jar file (<https://search.maven.org/remote_content?g=io.zipkin&a=zipkin-server&v=LATEST&c=exec>)
     2. set RABBIT\_URI=amqp://localhost
     3. java -jar zipkin-server-2.7.0-exec.jar
     4. Access zipkin server using http://localhost:9411/zipkin/
  3. Connection microservice to zipkin server
     1. Add dependency in Zuul API Gateway and Microservices



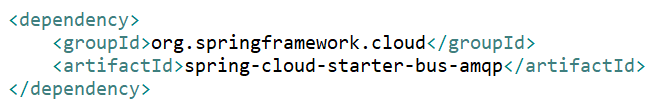
* + 1. For Spring Boot >= 2.1 use below dependency



1. Understanding the need for Spring Cloud Bus
   1. Call the API which will use config properties file from git.
   2. After change the config properties file then still, service will use the same old config data.
   3. Enable all actuator endpoints

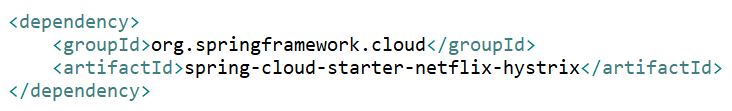


* 1. Call <http://localhost:8080/actuator/refresh> (Post) to refresh config file for each Instances
  2. Implementing Spring Cloud Bus
     1. Spring Cloud Bus links nodes of a distributed system with a lightweight message broker. This can then be used to broadcast state changes (e.g. configuration changes) or other management instructions.
     2. Add dependency in Config server and microservices

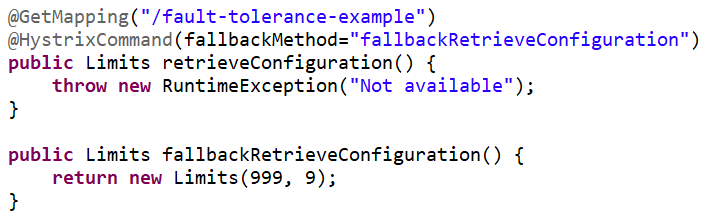


* + 1. If we call http://localhost:8080/actuator/bus-refresh (Post), it will refresh config for all instances.
    2. The way the Spring Cloud Bus works is at application start up, all the microservice instances register with the cloud bus. So, if Limits Service has three different instances, then instance1. instance2, and instance2, all of them register with the bus. When there is any change in configuration, when the refresh is called on any of these instances, the microservice instance would send an event over to the Spring Cloud Bus and the Spring Cloud Bus would propagate that event to all the microservice instances that are registered with it.

1. Fault Tolerance with Hystrix
   1. Add dependency



* 1. Add @EnableHystrix in Application java file
  2. Add fallback Method



* 1. Access <http://localhost:8080/fault-tolerance-example>