Contents

[MICRO SERVICE WITH SPRING CLOUD 2](#_Toc81846665)

[NON-FUNCTIONAL REQUIREMENTS OF SPRING MICROSERVICES 2](#_Toc81846666)

[USE CASE 2](#_Toc81846667)

[CREATING SPRING BOOT MICROSERVICE PROJECTS 3](#_Toc81846668)

[SERVICE REGISTRATION AND DISCOVERY 3](#_Toc81846669)

[SERVICE REGISTRATION AND DISCOVERY USING EUREKA SERVER 4](#_Toc81846670)

[SETTING-UP EUREKA SERVER 4](#_Toc81846671)

[COMMUNICATION BETWEEN MICROSERVICES 5](#_Toc81846672)

[SETTING UP FEIGN REST CLIENT 5](#_Toc81846673)

[CLIENT-SIDE LOAD BALANCING 5](#_Toc81846674)

[ADDING LOAD BALANCING SUPPORT 5](#_Toc81846675)

[GATEWAY 5](#_Toc81846676)

[SETING UP GATEWAY 6](#_Toc81846677)

[API GATEWAY FILTERS 6](#_Toc81846678)

[FAULT TOLERANCE (Resilence4J) 7](#_Toc81846679)

[ADDING RESILENCE4J DEPENDENCIES 7](#_Toc81846680)

[SETTING UP RESILENCE4J 7](#_Toc81846681)

[CUSTOMIZING RETRY- RESILENCE4J 8](#_Toc81846682)

[TRACING THE MICRO SERVICE REQUESTS 8](#_Toc81846683)

[ENABLING DISTRIBUTED TRACING USING SLEUTH 8](#_Toc81846684)

# MICRO SERVICE WITH SPRING CLOUD

## MICROSERVICES

Micro services are the services

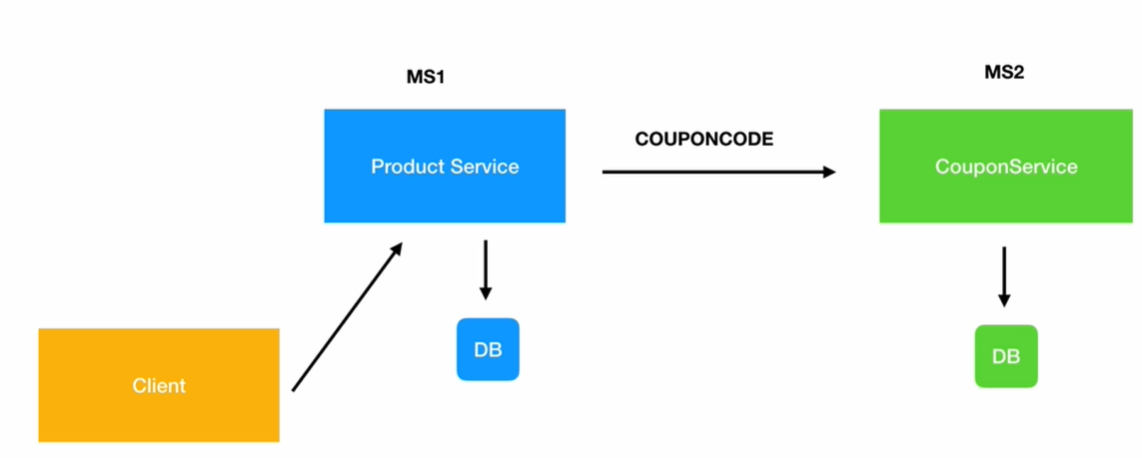
* Which are exposed by REST
* Small well-chosen Deployable unit
* Cloud Enabled : When a micro services are cloud enabled – we can able to scale up or down the micro service by adding new instance of the service- when the load increases and decreases.
* ***Spring cloud is a collection of open-source components that help us implement all these nonfunctional requirements***

## **NON-FUNCTIONAL REQUIREMENTS OF SPRING MICROSERVICES**

|  |  |
| --- | --- |
| **SERVICE REGISTRATION**  **AND DISCOVERY** | * Each microservice will have to register itself with a centralized server so that the other microservices will be able to discover that particular micro service and communicate with it dynamically. Without that each microservice will be tightly coupled to another microservice it is using. * This is provided by **Eureka** |
| **LOAD BALANCING** | * As the load to our microservices increases, there should be multiple instances of the same microservice running on different servers and the load should be balanced/distributed. * **Load Balancing** is provided by **Ribbon** |
| **FAULT TOLERANT** | * If something goes wrong in one of these microservices then entire systems should not come down. They should be fault tolerant and handle the faults gracefully. * **Fault tolerance** is provided **Hystrix** |
| **EASY INTEGRATION** | * They should be able to communicate with each other easily through restful client API. * **Feign Client** which will help us in creating restful clients |
| **CROSS CUTTING CONCERNS** | * Are common requirements across microservices like security, authentication, authorization, logging etc. . Instead of repeating those cross-cutting concerns across these microservices we should have one place where we can address them. * The cross cutting concerns are configured using a **Zuul proxy gateway**. Whatever is common across our microservices we can implement them in a single place in a single Zuul Server. |
| **DISTRIBUTED TRACING** | * As the requests go from one microservice to another, we should be able to trace how the requests are going and how the responses are coming back. When something goes wrong we can exactly pinpoint where it went wrong using distributed tracing. * **Sleuth** allows us to do distributed tracing as the requests go from one micro service to another. * We use**Sleuth and Zipkin** together. Zipkin will give us a dashboard where we can see and track the requests from one micro service to another. |

* **All these are not available in Spring Boot.** That is where the **Spring Cloud** comes in.

## **USE CASE**



* The use-case we will be building is creating 2 microservice “product” and coupon microservices.
* These microservices will expose the Restful APIs.
* The client can be able to create a product in DB.
* The product micro service will take help of Coupon Micro Services to create a product. It will use a coupon code – that will be applied to the created product for discount.
* The product service will pass the coupon code 🡪 get the discount corresponding to coupon code from coupon service micro service 🡪 Apply the discount on the product price 🡪 Save the product with the discounted price in DB.

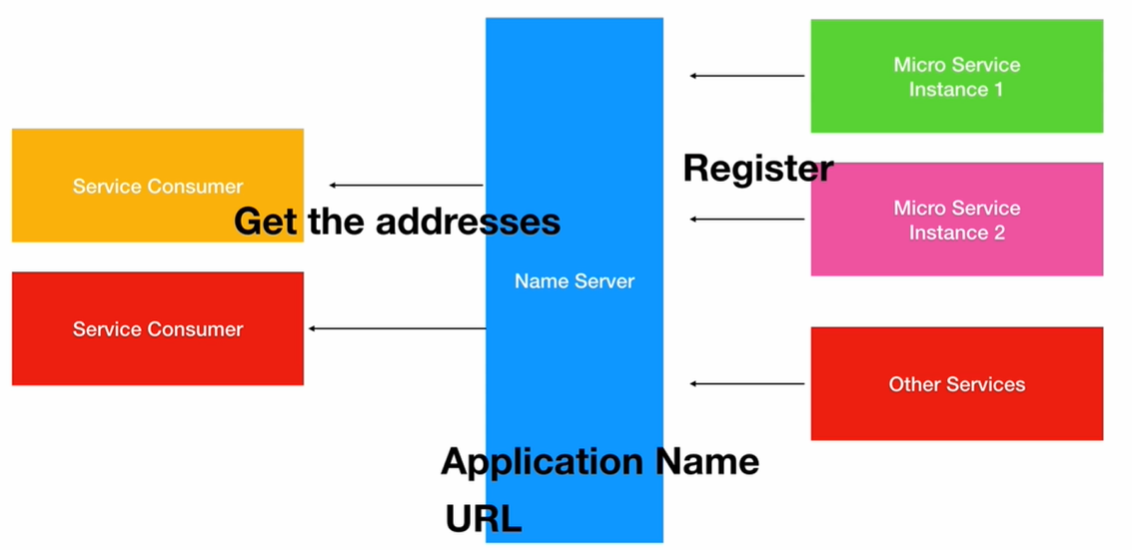
Note: Since we are running multiple microservices we will be running it different Server port

|  |  |
| --- | --- |
| **SERVICES** | **PORT** |
| **productservice** | 9090 |
| **couponservice** | 8080 |
| **EurekaServer** | 8761 |
| **Gateway Service** | 9095 |
| **Zipkin** | 9411 |

## **CREATING SPRING BOOT MICROSERVICE PROJECTS**

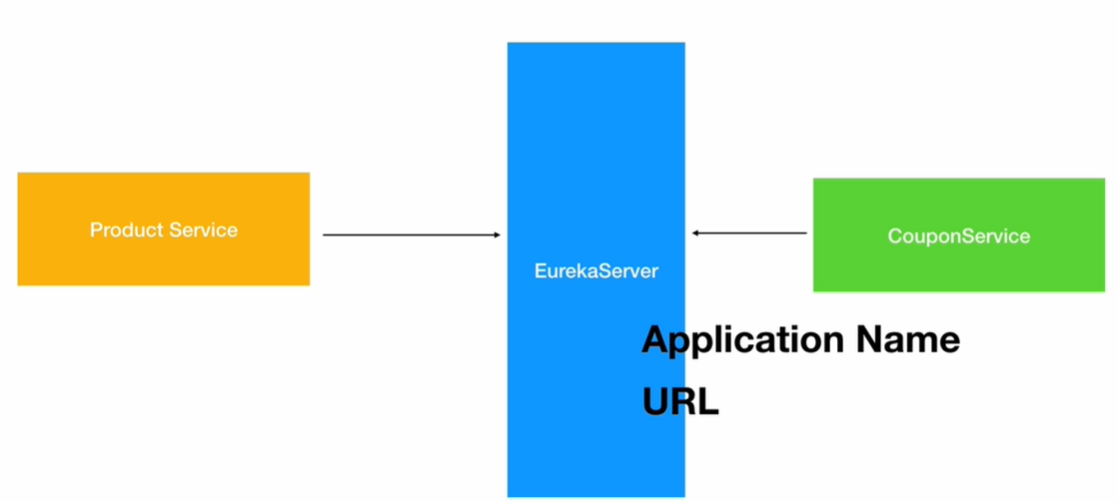
* **DEPENDENCIES**: Spring web, JPA and MySQL, Dev Tool- (Dev tool dependency helps in hot reloading of the application)
* **CREATE MODEL AND RESPOSITORY**
* **CREATE A REST CONTROLLER**
* **CONFIGURE DATA SOURCE**

## **SERVICE REGISTRATION AND DISCOVERY**



* There will be multiple Microservices that require or communicate with each other through Restful calls.
* To do that the consumers will have to know the URL, the port number etc. to communicate with the appropriate microservice.
* This will be very hard to maintain as there could be multiple instances of the same microservices running. That is where spring cloud offers naming services, or a naming server called Eureka. Micro services will register themselves as soon as they start or come up . They will register using an application name or application ID which is a unique Id for each application and also the URL that is required to communicate with them.
* Service consumers when they come up can communicate with the naming server to fetch those details based on just the application name. As long as the consumers know the unique application ID of the provider microservice they can fetch the URL and Port number.
* All that communication details will be maintained by the Eureka server. Eureka server will decouple these microservices and they can communicate with any other microservice through registration and discovery without maintaining any URL,Port Etc that are required otherwise.

### **SERVICE REGISTRATION AND DISCOVERY USING EUREKA SERVER**



### **SETTING-UP EUREKA SERVER**

1. Create a project with the Eureka Server dependency from Spring cloud starters from Netflix (Select “Eureka Server” dependency in STS)

|  |
| --- |
| **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>**  **</dependency>** |

1. And on the Application class of this project add**@EnableEurekaServer**annotation that tells spring that this particular project is a Eureka Server
2. On the client microapp – which will register themself has to Eureka Client Dependency

|  |
| --- |
| **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>**  **</dependency>** |

1. And on the Application class of client projects add**@**EnableEurekaClient annotation.
2. Give a name to each micro-service in application.properties

|  |
| --- |
| **spring.application.name=coupon-service**  **eureka.client.service-url.defaultZone=http://localhost:8761/eureka/** |

1. Start your application automatically these applications will register themselves with the Eureka server using the application name which we provide in the application or properties.
2. And then other micro services can use this name and communicate with that particular micro service. We provide the Eureka server URL as a property in the client projects as well.

|  |  |
| --- | --- |
|  | 1. After Registry configuration : Start the Eureka Server Application 🡪 Start all the client MicroApp 2. Open the Eureka Server Console from browser at : <http://localhost:8761/> |

## **COMMUNICATION BETWEEN MICROSERVICES**

* In the application – “productservice” micro service is acting as client for “couponservice” micro service.
* The productservice becomes “rest client” for couponservice.
* **Feign** can be used to create Rest Client. It can be enabled in using @FeignClient annotation in productservice microservice

### **SETTING UP FEIGN REST CLIENT**

1. **ADD FEIGN CLIENT DEPENDENCY (In productservice)**

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-**openfeign**</artifactId>  </dependency> |

1. **CREATING REST CLIENT USING FEIGN CLIENT**

* **Create an interface and configure the application name to Feign Client**

|  |
| --- |
| @FeignClient("COUPON-SERVICE")  **public** **interface** CouponClient {  @GetMapping("/couponapi/coupons/{code}")  Coupon getCoupon(@PathVariable("code") String code);  } |

1. **ENABLE FEIGN CLIENT**

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @EnableFeignClients  **public** **class** ProductserviceApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ProductserviceApplication.**class**, args);  }  } |

## **CLIENT-SIDE LOAD BALANCING**

* After the load increases – we need to deploy it to different servers. So when request comes – it is sent to different instances rather than sending it to one single instance.
* The load balancing is doen by spring cloud component called load balancer which works closely with Feign client .
* The Load balancing happenes on client side (***productservices***)

### ADDING LOAD BALANCING SUPPORT

**ADD THE DEPENDECY ON CLIENT APPLICATION FOR CLIENT SIDE LOAD BALANCING**

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>**spring-cloud-starter-loadbalancer**</artifactId>  </dependency> |

## **GATEWAY**

* The API Gateway do the server-side load balancing.
* The cross cutting concerns can be configured using API Gateway filters

|  |  |
| --- | --- |
|  | It handles cross cutting functionality like   1. **SECURITY:**     1. Authentication and Authorization    2. Encryption and Decryption 2. **TRACING**   Tracking the coming request when it flows through various microservices   1. **SERVICE AGGREGATION**   When a client has to make a call to various other microservices   1. **RATE LIMITS:** To limit – To limit much how much cloud resource a microservice can use |

* These cross-cutting concerns are common for all micro services we use spring cloud API Gateway component.
* All the microservices call are redirected via Gateway . For example if the productservice want to call couponservice , it has to call gateways service which the redirected to couponservice.
* This helps the gate ways service to apply the cross-cutting functionality.

### SETING UP GATEWAY

#### CREATING GATEWAY SPRING BOOT PROJECT

1. To set up the gateways create a spring boot project.
2. Select Dependencies
   1. Gateway
   2. Eureka Discovery Client.

#### REGISTERING GATEWAY PROJECT TO EUREKA

* The gateway service also needs to register with Eureka Server. Add the following to application.properties.
* Add the server port to run the gateway service on a desired port.

|  |
| --- |
| spring.application.name=gateway-service  eureka.client.service-url.defaultZone=http://localhost:8761/eureka/  server.port=9095 |
| **ENABLE EUREKA CLINET**  @SpringBootApplication  **@EnableEurekaClient**  **public** **class** ApigatewayserviceApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ApigatewayserviceApplication.**class**, args);  }  } |

#### CONFIGURING ROUTES OF MICROSERVICES IN GATEWAYS

* Add the routes in application.properties

|  |
| --- |
| spring.cloud.gateway.routes[0].id=couponModule  spring.cloud.gateway.routes[0].uri=lb://COUPON-SERVICE  spring.cloud.gateway.routes[0].predicates[0]=Path=/couponapi/\*\*  spring.cloud.gateway.routes[1].id=productModule  spring.cloud.gateway.routes[1].uri=lb://PRODUCT-SERVICE  spring.cloud.gateway.routes[1].predicates[0]=Path=/productapi/\*\* |

### API GATEWAY FILTERS

|  |  |
| --- | --- |
|  | * To add any pre or post processing logic, before the request/ response is handed over a microservice using API Gateway filter. * All the cross-cutting functionality call be applied using API Gateway filters |

#### SETTING UP API GATEWAY FILTERS

## **FAULT TOLERANCE (**Resilence4J**)**

* In a system – which has multiple micro services talking to each other, if any one of the microservice goes down – We never expect the other dependent microservice go down. In our example , If the couponservice goes down – we don’t want productservice to go down as well and sending exception message to consumer.
* **We expect productservice to “retry” or “handle the error gracefully using fallback methods” – This is where Resilence4J component comes into play and allow us to implement Fault Tolerance.**

### ADDING RESILENCE4J DEPENDENCIES

* Resilience 4J expects actuator dependency added in pom.xml
* <https://resilience4j.readme.io/docs/getting-started-3>

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>io.github.resilience4j</groupId>  <artifactId>resilience4j-spring-boot2</artifactId>  </dependency> |

### SETTING UP RESILENCE4J

* We user @Retry Annotation on the controller method which we want to retry. Usually, we annotate the method where we call other microservice.
* To handle the error gracefully after the retries we can configure fallback methods

Note:

* The syntax of fallback method signature must be same as the method where the exception occurred.
* The fallback method can have some back up logic after the exception occurred.

|  |
| --- |
| @RestController  @RequestMapping("/productapi")  **public** **class** ProductRestController {  @Autowired  ProductRepo productRepo;    @Autowired  CouponClient couponClient;  @RequestMapping(value = "/products", method = RequestMethod.***POST***)  **@Retry(name = "product-api" ,fallbackMethod = "handleError")**  **public** Product create(@RequestBody Product product) {  System.***out***.println("Retry");  Coupon coupon = couponClient.getCoupon(product.getCouponCode());  product.setPrice(product.getPrice().subtract(coupon.getDiscount()));  **return** productRepo.save(product);  }  **public** Product handleError(Product product, Exception exception) {  System.***out***.println("Inside Handle Error");  **return** product;  }  } |
| * handleError is the fallback method which will be called after the configured retries. * In the above example – We are just returning the product received in request with any “id” and without saving the data in DB |

### CUSTOMIZING RETRY- RESILENCE4J

* By default - Resilience 4j retries 3 times before it throws an exception
* We can configure the retries in application.properties file

|  |
| --- |
| **#The above configuration will set the number for retries =2 with retry interval= 3s**  resilience4j.retry.instances.product-api.max-attempts=2  resilience4j.retry.instances.product-api.wait-duration=3s |

## **TRACING THE MICRO SERVICE REQUESTS**

* It is very important that we should be able to trace the microservices requests as they flow from one service to another and if they fail where exactly they failed. **Sleuth and Zipkin provide these distributor tracing capabilities**.
* As our requests flow from micro service to another Sleuth is responsible for adding a trace id for these requests. That trace id will show up in the logs .That way we can track our request across micro services and we can easily figure out where things are going wrong if they do.
* ***Zipkin is responsible for providing a dashboard where all traces by sleuth are exported and it will display them beautifully.***

### ENABLING DISTRIBUTED TRACING USING SLEUTH

* To Enable Sleuth add the dependencies to the Micro-services
* **Adding this dependencies - Will help in generating the a unique trace id. The trace id can be seen in the logs**

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-sleuth</artifactId>  </dependency> |

#### TRACING FORMAT

This looks like a normal log, except for the part in the beginning between the brackets. This is the core information that Spring Sleuth has added. This data follows the format of:

**[application name, traceId, spanId, export]**

|  |  |
| --- | --- |
| **APPLICATION NAME** | This is the name we set in the properties file and can be used to aggregate logs from multiple instances of the same application. |
| **TRACEID** | This is an id that is assigned to a single request, job, or action. Something like each unique user initiated web request will have its own traceId. |
| **SPANID** | Tracks a unit of work. Think of a request that consists of multiple steps. Each step could have its own spanId and be tracked individually. By default, any application flow will start with same TraceId and SpanId. |
| **EXPORT** | This property is a boolean that indicates whether or not this log was exported to an aggregator like Zipkin. Zipkin is beyond the scope of this article but plays an important role in analyzing logs created by Sleuth. |

**And for Zipkin add**

<dependency>

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

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

</dependency>

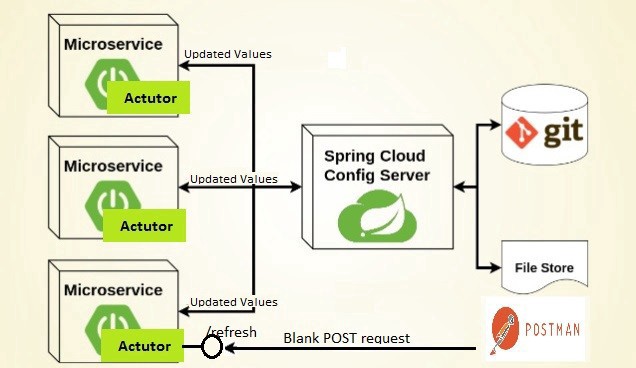
**Download and run Zipkin server**

curl -sSL https://zipkin.io/quickstart.sh | bash -s

java -jar zipkin.jar

## **CENTRALIZED CONFIGURATION**

|  |  |
| --- | --- |
|  |  |



* Every Microservice application will have associated configuration. This could be the database connection information , the message broker information if it is a messaging application and application specific configuration as well. And this information will be different for different environments.
* On a local developer's machine a database that he wants to connect is different from the one dev environment shared by all the developers and the testing team will have their own database servers their own message brokers and also application specific configuration. Same with staging before our application is pushed to production finally the production will have its own set of databases, message brokers etc. Each application can have multiple configurations as well, not just one. So it is our responsibility to make sure the right configuration information is being pushed across these environments.
* That is where centralized configuration comes in to make our job as a developer easy. Spring cloud provides a **Configuration Server.**All the micro service applications will ask this configuration server for the configuration information for a particular environment. On dev environments the configuration server is responsible for giving them the development specific configuration, which will have the dev specific databases, dev specific JMS brokers etc. And if they're deployed to staging or production the appropriate configuration should be given by this configuration server.

**How will the configuration server know the correct configuration for an environment?**

* We will push all the configuration to the GIT repository. The configuration server knows how to connect to the GIT or SVN repository and it will pull the configuration information and share it across the microservices. We'll be simply creating the configuration as developers. We can do it on the local machine and it is our responsibility to put that configuration into this GIT repository.
* All the microservices will have the configuration available. The configuration server is responsible for maintaining and handing over the correct configuration for a correct micros service.

### **SETTING UP CONFIG SERVER**

|  |  |
| --- | --- |
| * **CREATE A CONFIG SERVER** * **CREATE A CONFIG REPOSITORY** * **CREATE A CONFIG CLIENT**   **Note:**   * Limit Microservice , MS X and MS Y are client microservice – * They will fetch the configuration from a centralized config server , * The config server which in turn pull the configuration from local or remote Git Repo |  |

#### SETTING UP CLIENT MICROSERVICE

* We are creating Client microservices that will later connect to Spring cloud config server.
* Dependency Added : ***Spring Web, Dev Tools, Spring Boot Actuator and Config Client***.

##### USE CASE: HOW TO READ VALUES FROM application.properties

|  |  |
| --- | --- |
| application.properties  limits-service.minimum=2  limits-service.maximum=999 | **CREATE A BEAN CLASS**  **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  @Component  @ConfigurationProperties("limits-service")  **public** **class** Configuration {  **private** **int** minimum;  **private** **int** maximum;  **public** **int** getMinimum() {  **return** minimum;  }  **public** **void** setMinimum(**int** minimum) {  **this**.minimum = minimum;  }  **public** **int** getMaximum() {  **return** maximum;  }  **public** **void** setMaximum(**int** maximum) {  **this**.maximum = maximum;  }  } |
| **NOTE :**  The properties name of the class should match with the keys of application.properties(minimum and maximum) |
| **USING THE CONFIGURATION**  @RestController  **public** **class** LimitController {    @Autowired  Configuration configuration;    @GetMapping("/limits")  **public** Limits retriveLimits() {  **return** **new** Limits(configuration.getMinimum(), configuration.getMaximum());  }  } | |

#### SETTING UP CONFIG SERVER

* Create a spring boot application with dependencies: ***Spring Web and Config Server***

**7) What is Spring Cloud Bus**

When configurations are updated on the fly those changes will not be loaded by the microservice applications running until we restart the application or use a spring boot actuator refresh url for each micro service.Once you enable the spring cloud bus , with a single refresh at the cloud bus level all the micro services will get the latest configuration values that are updated.Spring cloud bus uses messaging brokers to communicate these changes to the applications.

**3) What is Ribbon or What and how to use client side load balancing?**

As the number of requests to our micro services increase we will be scaling our application across different servers.There will be multiple instances of the same microservice running on different servers so that the client can make calls to one of these instances as required.

Client side Load balancing is done using Ribbon from Spring cloud .Feign client also a Spring Cloud component which allows easy Rest Client creation works with Ribbon and the load will be balanced from the client side itself.The client will decide to which instance the request should go to.

Configuring Ribbon is easy in the pom.xml.Just Add

<dependency>

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

<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>

</dependency>

**4) What is Zuul Proxy**

There will be some non-functional requirements that are common across micro services.

For example security where the client needs to authenticate and authorize with a particular microservice ,tracing where we need to trace the request as it goes from one micro service to another, service aggregation, i.e if we have calls that need to go from the client to multiple micro services . Instead of the client making those separate calls we can have a service aggregator component in one place that can make all those calls .For example Rate Limits.If you want to charge your client based on the request like Amazon or Microsoft azure cloud you can apply all those rate limits.If we have a centralized place so all these non-functional requirements that are common across microservices can go into each micro service which will repeat the same code the same components across microservices are we can put these cross cutting concerns in one place and that is where the Zuul proxy component of spring cloud comes in.

Once we create a Zuul proxy server all the requests will route through the Zuul proxy gateway the client requests will go through the Zuul proxy .We can implement Security, Tracing , Service aggregation and all that inside the Zuul proxy .

We create a Zuul proxy server just like Eureka Server . Add Netflix Zuul dependency and then it will register itself with Eureka . Zuul will be just like any other micro service; it will register itself with Eureka when it starts and other microservices can start using it.

**5) What is Hystrix**

When we implement micro services architecture there will be a lot of microservices running.And if one of these micro services goes down for some reason the entire system should not collapse.The errors should be handled gracefully and the minimum functionality should be delivered to the end user.That is where this spring cloud hystrix component comes in and it provides fault tolerance capabilities.

**Steps to use Hystrix:**

Add Hystrix dependency to the project

<dependency>

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

<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>

</dependency>

Second Step enable hystrix by using the following on application class

**@EnableHystrix**

Create a Fault tolerant method in the REST controllers

public Product **sendErrorResponse**(Product product) {

…….

And use it on a method where the fault might occur

**@HystrixCommand(fallbackMethod = "sendErrorResponse")**

@RequestMapping(value = "/products", method = RequestMethod.POST)

public Product create(@RequestBody Product product)

When something goes wrong inside of this method this is the fault tolerant method that will be invoked by Hystrix and whatever response this method returns that will be sent back to the client .

Hystrix also offers a super cool **dashboard**. To us it we add the following to pom.xml

<dependency>

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

<artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>

</dependency>

Add **@EnableHystrixDashboard** on the Application class

Hystrix metrics can then be seen on the hystrix dashboard from a browser.