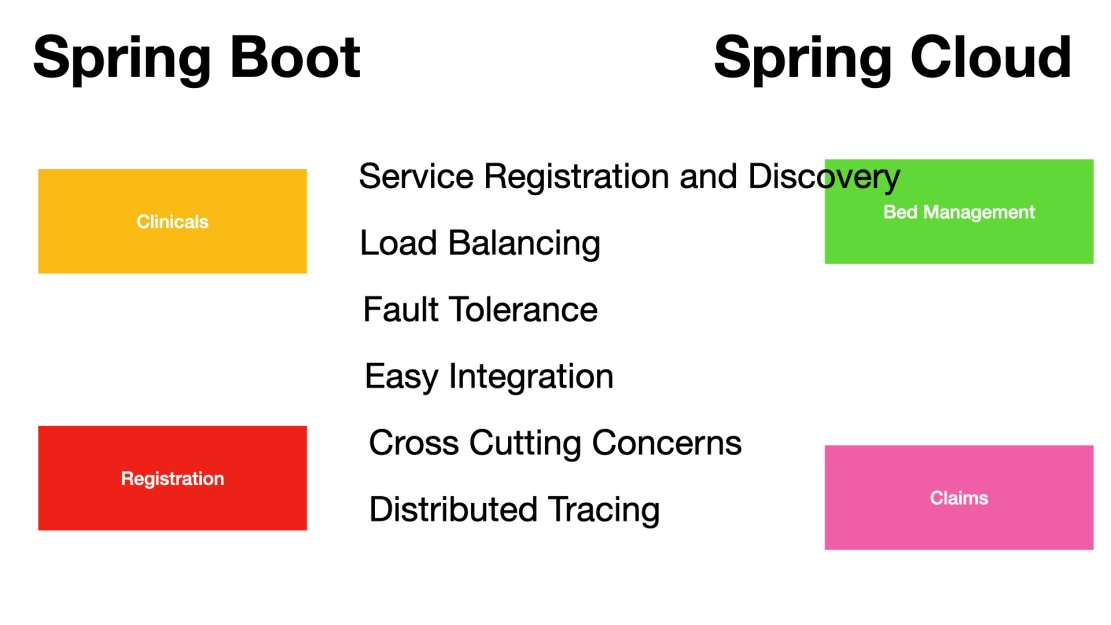
**Quick Review and Interview Questions**

**1) What is Spring Cloud?**



When we implement microservices using the modules provided by Spring Boot, there will be several non-functional requirements for these microservices as below -

**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.

**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.

**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.

**Easy integration** - They should be able to communicate with each other easily through restful client API.

**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.

**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 .

**All these are not available in Spring Boot.** That is where the **Spring Cloud** comes in. Spring cloud is a collection of open source components that help us implement all these non functional requirements and more.

**Service registration and discovery** is provided by **Eureka**

**Load Balancing** is provided by **Ribbon**

**Fault tolerance** is provided **Hystrix(deprecated) , reslient4j**

**Feign Client** which will help us create restful clients in a super easy way

The cross cutting concerns are configured using a **Zuul proxy gateway(deprecated), Spring cloud ApI Gateway** .Whatever is common across our microservices we can implement them in a single place in a single Zuul Server.

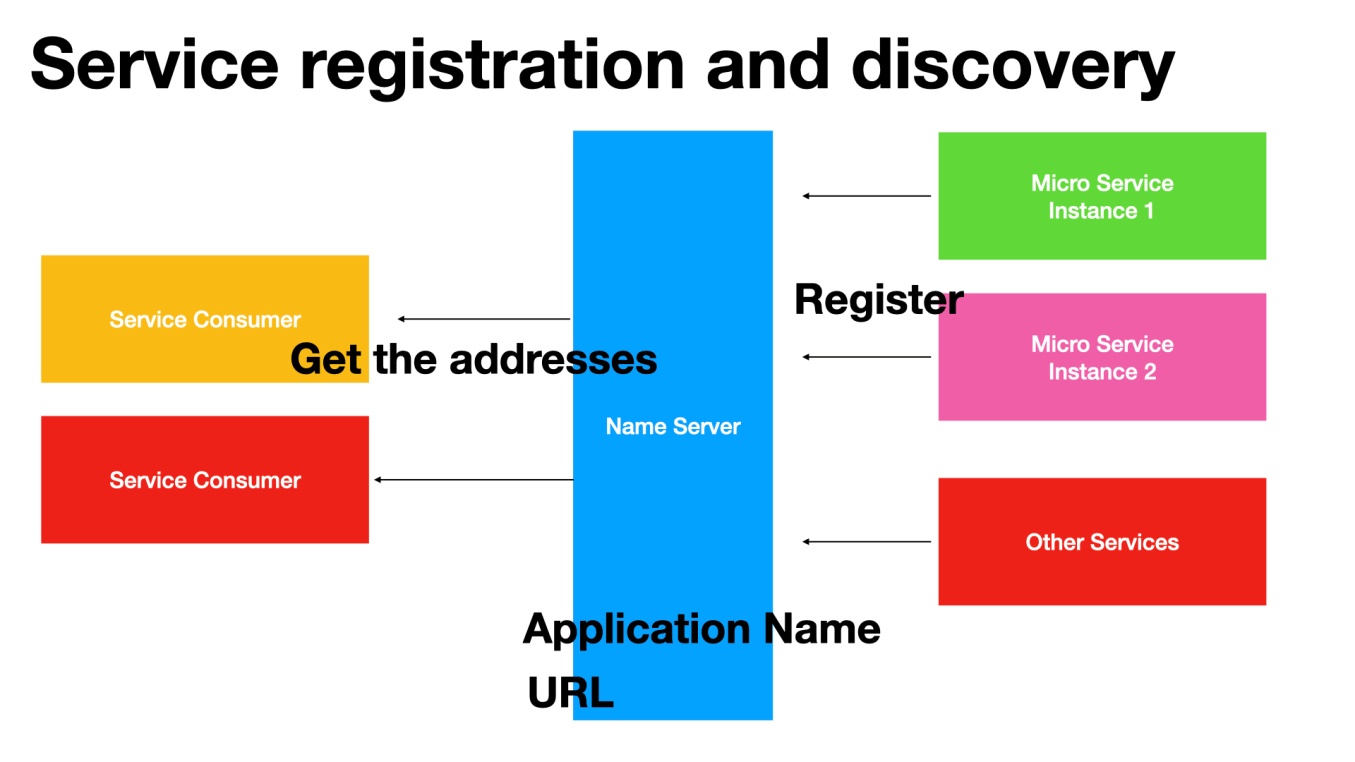
**Sleuth** allows us to do distributed tracing as the requests go from one micro service to another.

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.

**2) What is Eureka and how do we use it?**

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 in order 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.

We'll be registering our applications to start with the Eureka server .

**How to use Eureka?**

Create an Eureka server of our own .Create a project with the Eureka Server dependency from Spring cloud starters from Netflix

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

And on the Application class of this project add**@EnableEurekaServer**annotation that tells spring that this particular project is a Eureka Server

**And on the clients we use the following:**

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

**@EnableEurekaClient** annotation on application calss that tells spring that particular project is a Eureka Client.

spring.application.name=coupon-service

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

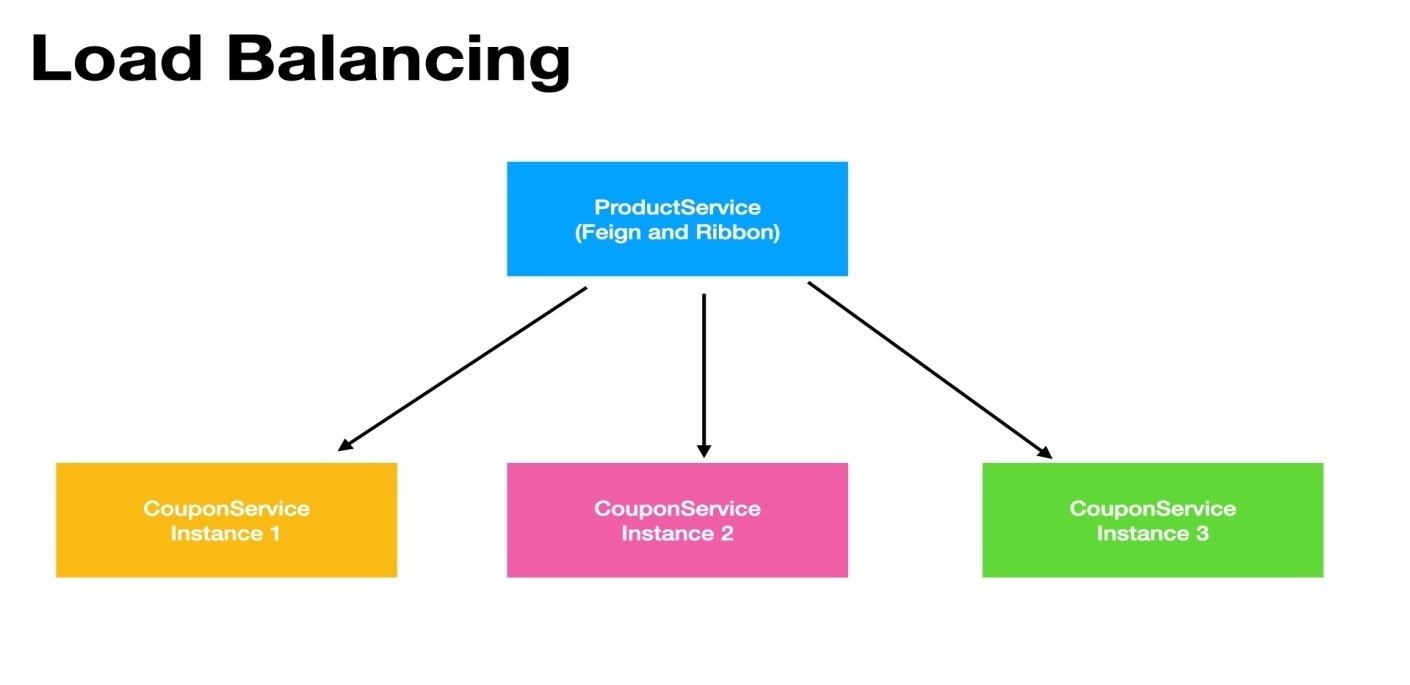
Once you do that and you start your application automatically these applications will register themselves with the Eureka server using the application name which we provide in the application or properties.

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.

**Note:** Default port for eureka server is 8761

**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 following dependency:**

<dependency>

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

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

</dependency>

Adding dependency on product microservice balances load on product microservice if multiple instances of product service are running just by adding ribbon depency to product service’s pom.xml.

**Client side load balancing on Feign Client:**

So, In our example we are balancing load from product service to multiple instances of coupon service running.

Add maven dependency to product service pom.xml:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>  
</dependency>

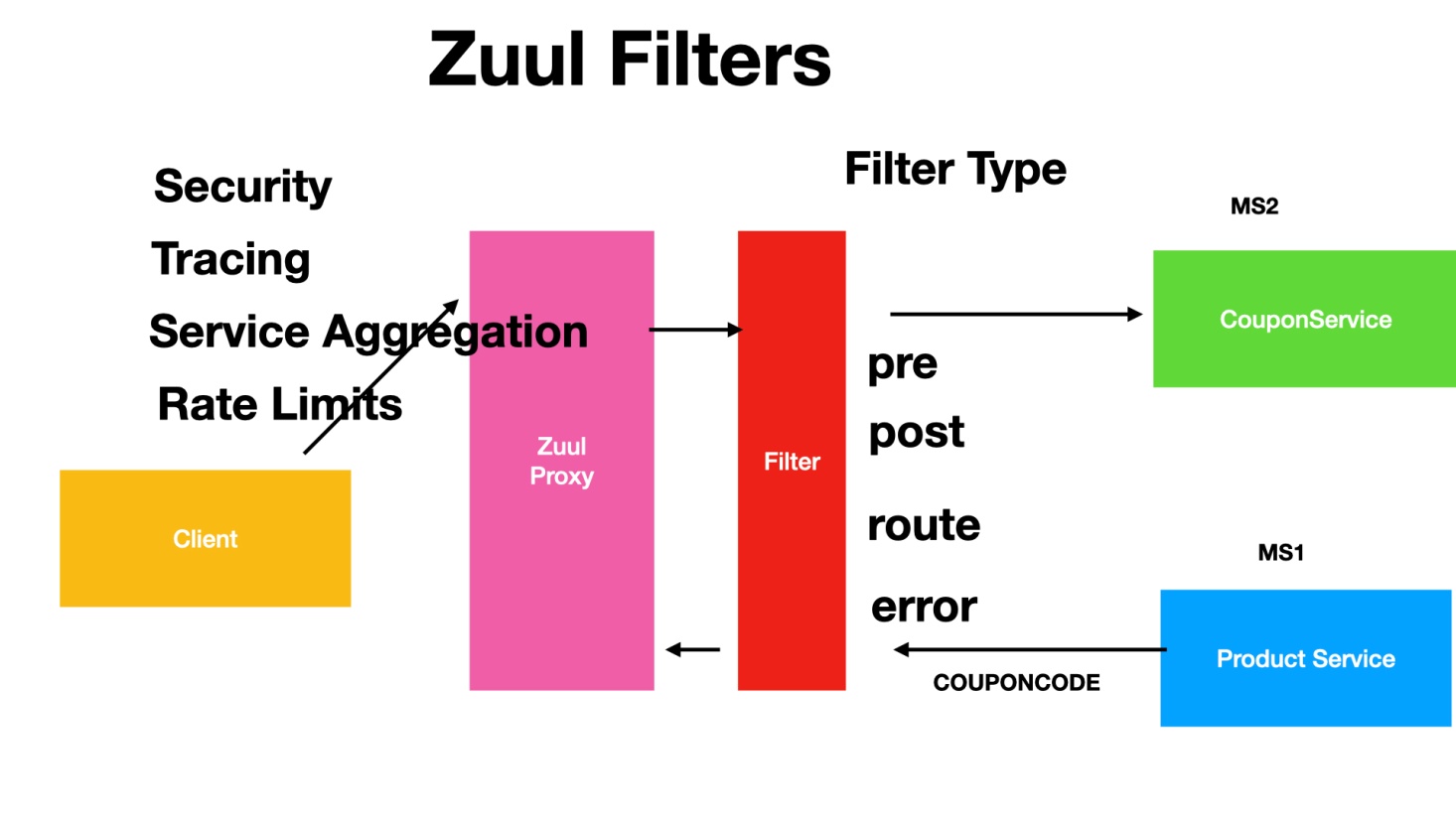
Enable and Configure Ribbon:

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

**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 .

1. We create a Zuul proxy server just like Eureka Server.
2. 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.

<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>

Add @EnableZuulProxy on top of application class

Configure ZuulProxy service application.properties

server.port=8081  
spring.application.name=coupon-service  
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

Now run the application

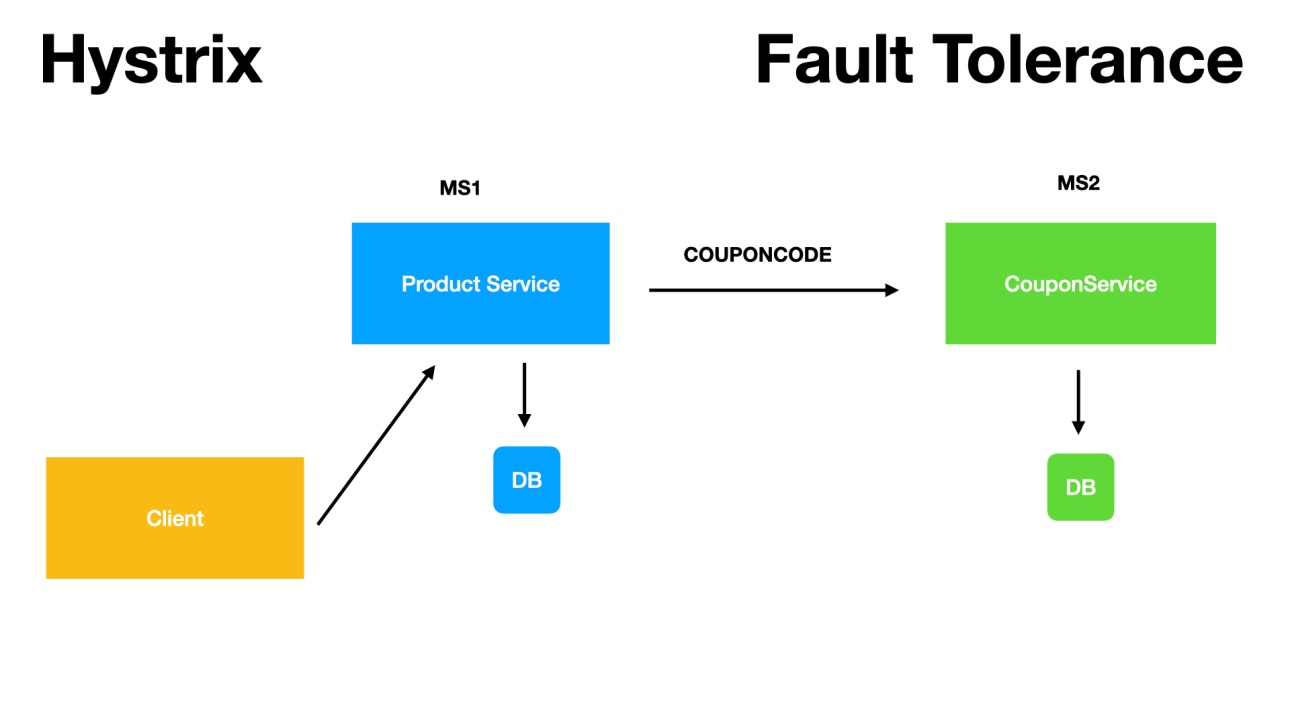
[http://localhost:8761**/product-service**/productapi/products](http://localhost:8761/product-service/productapi/products)

We should mention service name while hitting the micro service using zuul proxy

**Note:** default port in which zuul proxy runs is 8765

**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 is to enable hystrix on application class using **@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>

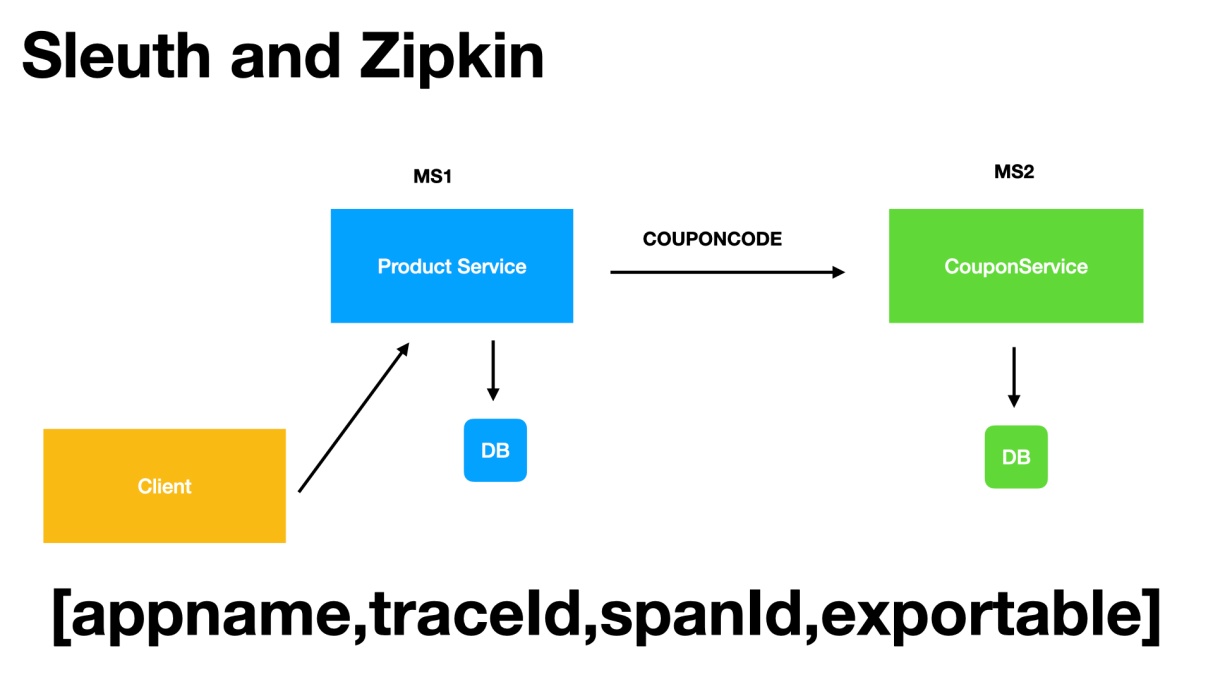
</dependency>

Add **@EnableHystrixDashboard** on the Application class

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

**5) What are Sleuth and Zipkin**

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.

**To use Sleuth just add the following to pom**

<dependency>

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

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

</dependency>

**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

**Note:** Default port for Zipkin is 9411

**6) How to use 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 micro service.

Steps:

1. Create Config Server
2. Create Config Repository
3. Create Config Client

**Spring Cloud Config** is Spring's client/server approach for storing and serving distributed configurations across multiple applications and environments.

1. **Create Config Server:**

We create a project with dependency

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-server</artifactId>  
</dependency>

We add @EnableConfigServer annotation on our application class to make our application a config server.

@SpringBootApplication  
@EnableConfigServer  
public class ConfigServerApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(ConfigServerApplication.class, args);  
 }  
  
}

Specify application name port in application.properties

spring.application.name=config-server  
server.port=8888

1. **Create Config Repository:**

To complete our server, we have to initialize a Git repository under the configured url, create some new properties files, and populate them with some values.

Here, we have added two config files to local repo for two environment dev and local

**product-service.properties**

management.security.enabled=false  
application.url=http://local

**product-service-dev.properties**

management.security.enabled=false  
application.url=http://dev

Now, update application.properties of config server micro service to point to local git repository

spring.cloud.config.server.git.uri=file:///C:/Users/praneetha/Documents/spring cloud workspace/localgitrepo

You can start application and can check configuration in web browser using urls below

<http://localhsot:8888/product-service/default/>

<http://localhost:8888/product-service/dev/>

1. **Create Config Client:**

We have product micro service and we can make it as config client by moving all its properties from application.properties to config server product-service.properties file.

**product-service.properties**

management.security.enabled=false  
application.url=http://local  
spring.datasource.url=jdbc:mysql://localhost:3306/mydb  
spring.datasource.username=root  
spring.datasource.password=Praneetha#1  
server.port=9090  
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/  
resilience4j.retry.instances.product-api.max-attempts=2  
resilience4j.retry.instances.product-api.wait-duration=3s

same way we can configure dev (Here we only changed port number between dev and local)

**product-service-dev.properties**

management.security.enabled=false  
application.url=http://local  
spring.datasource.url=jdbc:mysql://localhost:3306/mydb  
spring.datasource.username=root  
spring.datasource.password=Praneetha#1  
server.port=9093  
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/  
resilience4j.retry.instances.product-api.max-attempts=2  
resilience4j.retry.instances.product-api.wait-duration=3s

Now add, dependencies to product-service pom.xml

**pom.xml**

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bootstrap</artifactId>  
</dependency>

Now, update name of application.properties of product microservice to bootstrap.properties and make it point to listen to 8888 port to get properties.

**Bootstrap.properties**

spring.application.name=product-service  
spring.config.import=configserver:http://localhost:8888  
spring.profiles.active=dev

**Note:** By default config server runs on port 8888

**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.

**Refresh using Spring Boot Actuator:**

when we make the changes in the Git repository, we have to hit multiple instances of the micro service to refresh the configuration.

For example: Let’s add a customer property in **product-service** in local git reposiory

com.praneetha.springcloud.property=local1

To expose all endpoints in the actuator add the following line to local git repository’s product-service file

management.endpoints.web.exposure.include=\*

Commit git file

Add actuator dependency to product-service pom.xml:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

Let’s create a rest API to check custom property

**ProductRestController.java**

@Value("${com.praneetha.springcloud.property}")  
private String customProperty;  
  
@GetMapping("/customProperty")  
String getProperty()  
{  
 return this.customProperty;  
}

add **@ResfreshScope** annotation at the top on **ProductRestController** class **to reload bean configuration at run time** & start all applications

Now If we hit <http://localhost:9090/productapi/customProperty>, we get **local1** as result

Now let’s change the value of custom property in git repo and commit it.

com.praneetha.springcloud.property=local123

If we hit <http://localhost:9090/productapi/customProperty>, we still get **local1** as result

We should hit spring boot actuator’s refresh endpoint to refresh the microservice running on the given port

<http://localhost:9090/actuator/refresh>

Now if we hit <http://localhost:9090/productapi/customProperty>, we still get **local123**  as result which is updated value

Now let’s **change port number to 9095** **in product-service file** of local git repo, commit it and launch another instance of product microservice

Now if we hit <http://localhost:9095/productapi/customProperty>, we still get **local123**  as result

Now let’s change the value of custom property in git repo and commit it.

com.praneetha.springcloud.property=localabc

Let’s Perform Spring boot actuator refresh on port 9090 <http://localhost:9090/actuator/refresh>

Now if we hit <http://localhost:9090/productapi/customProperty>, we get **localabc** as result which is updated value

But hitting <http://localhost:9090/productapi/customProperty> , we still get **local123** as result

So, we need to hit spring boot actuator refresh on port 9095 also <http://localhost:9095/actuator/refresh>

The problem with actuator is it only refreshes one instance of micro service running on the specific port. To refresh all instances we should perform actuator refresh on each instance.

**Refresh using Spring Cloud Bus:**

There are many options available in the Spring Cloud Bus: **Apache** **Kafka, RabbitMQ,**etc. In this section, we will use **RabbitMQ**.

Install RabbitMQ in your system

Add following dependency to product-service pom.xml

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
</dependency>

Add RabbitMQ config to bootstrap.properties in product microservice

spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest

Restart the two instances of product microservices running on 9090 and 9095

Initially both instances will give same result **localabc** on hitting <http://localhost:9090/productapi/customProperty> ->localabc

<http://localhost:9090/productapi/customProperty> ->localabc

Now let’s change the custom property value in product-service file of local git and commit it

com.praneetha.springcloud.property=localbus

Now if we perfom bus refresh <http://localhost:9090/actuator/busrefresh>, it refreshes all instances

both instances will give same result **localbus** on hitting

<http://localhost:9090/productapi/customProperty> ->localbus

<http://localhost:9090/productapi/customProperty> ->localbus

**8) Explain about Feign Client?**

**Spring Cloud OpenFeign** provides OpenFeign integrations for Spring Boot apps through auto-configuration and binding to the Spring Environment. Without Feign, in Spring Boot application, we use **RestTemplate** to call the User service. To use the Feign, we need to add **spring-cloud-starter-openfeign** dependency in the pom.xml file.

**Dependencies:**

First, we’ll start by creating a Spring Boot web project and adding the spring-cloud-starter-openfeign dependency to our pom.xml file:

<dependency>

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

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

</dependency>

**Also, we'll need to add the spring-cloud-dependencies:**

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

**Next, we need to add @EnableFeignClients to our main class:** With this annotation, we enable component scanning for interfaces that declare they are Feign clients.

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

**Then we declare a Feign client using the @FeignClient annotation:**

@FeignClient(value = "jplaceholder", url = "https://jsonplaceholder.typicode.com/")  
public interface JSONPlaceHolderClient {  
  
 @RequestMapping(method = RequestMethod.GET, value = "/posts")  
 List<Post> getPosts();  
  
 @RequestMapping(method = RequestMethod.GET, value = "/posts/{postId}", produces = "application/json")  
 Post getPostById(@PathVariable("postId") Long postId);  
}

**9) Explain about Spring Cloud Gateway?**

In a distributed environment, services need to communicate with each other. However, this is interservice communication. We also have use-cases where a client outside our domain wants to hit our services for the API. So, either we can expose the address of all our microservices which can be called by clients OR we can create a Service Gateway which routes the request to various microservices and responds to the clients.

Creating a Gateway is much better approach here. There are two major advantages −

* The security for each individual services does not need to maintained.
* And, cross-cutting concerns, for example, addition of meta-information can be handled at a single place.

**Netflix Zuul** and **Spring Cloud Gateway** are two well-known Cloud Gateways which are used to handle such situations. In this tutorial, we will use Spring Cloud Gateway.

Spring Cloud Gateway provides a library for building an API Gateway on top of Spring WebFlux. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross-cutting concerns to them such as security, monitoring/metrics, and resiliency.

**Spring Cloud Gateway – Dependency Setting**

1)Let us use the case of Product and coupon which we have been using. Let us add a new service (gateway) in front of our two services, i.e., productservice and couponservice. First, let us update the pom.xml of the **gateway service** with the following dependency −

<dependencies>

<dependency>

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

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

**2) And then, annotate our Spring application class of gateway service with the annotation @EnableDiscoveryClient.**

**Dynamic Routing with Gateway**

The Spring Cloud Gateway has three important parts to it. Those are −

* **Route** − These are the building blocks of the gateway which contain URL to which request is to be forwarded to and the predicates and filters that are applied on the incoming requests.
* **Predicate** − These are the set of criteria which should match for the incoming requests to be forwarded to internal microservices. For example, a path predicate will forward the request only if the incoming URL contains that path.
* **Filters** − These act as the place where you can modify the incoming requests before sending the requests to the internal microservices or before responding back to the client

**Configuring Eureka Server URL and API Gateway Routes:**

**application.properties**

spring.application.name=gateway-service  
eureka.clientservice-url.defaultZone=http://localhost:8761/eureka  
  
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/\*\*

In **spring.cloud.gateway.routes[0].uri=lb://COUPON-SERVICE**, lb means load balancing

* Now, we can make Rest API call on the port which GATEWAY\_SERVICE Runs
* GATEWAY-SERVICE uses Eureka server to discover other micro services.
* It then routes to the microservice if the predicate matches.

**Custom Filters with Pre Processing and Post Processing logic:**

@Component  
public class MyFilter implements GlobalFilter {  
  
 @Override  
 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
 // *TODO Auto-generated method stub* System.*out*.println("Pre Processing Logic Goes here"+exchange.getRequest());  
 return chain.filter(exchange).then(Mono.*fromRunnable*(()->{System.*out*.println("Post Processing Logic Goes here"+exchange.getResponse());}  
 ));  
 }  
  
}

If we call coupon service using gateway service port, filter proceesing order would be:

* Gateway service pre proceeding logic
* Coupon service pre processing logic
* Coupon service post processing logic
* Gateway service post processing logic

**10) Explain about fault tolerance using resilience4j?**

[**Resilience4j**](https://www.baeldung.com/resilience4j) is a lightweight fault tolerance library that provides a variety of fault tolerance and stability patterns to a web application.

Dependency:

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

</dependency>

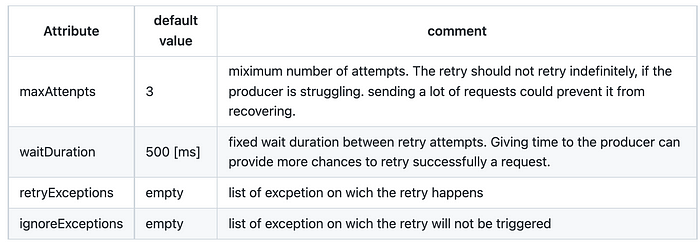
**Resilience4j implements multiple resiliency patterns :**

- Circuit Breaker  
- RateLimiter  
- TimeLimiter  
- Retry  
- Bulkhead  
- Cache

**Retry Pattern definition**

The retry pattern lets your consumer retry calls whenever they fail. This solution can solve cascading failure caused by transient errors, the basic deal is that if the error cause will resolve itself, we can be pretty sure one of the next retry calls will succeed, and this will prevent our consumer from cascading failure.

**Simple resilience4j retry configuration**



**Retry Consumer Client Code with FallBack Method**

@RequestMapping(value="/products",method=RequestMethod.*POST*)  
@Retry(name="product-api",fallbackMethod = "handleError")  
 public Product create(@RequestBody Product product) {  
 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;  
}

**application.properties**

resilience4j.retry.instances.product-api.max-attempts=2  
resilience4j.retry.instances.product-api.wait-duration=3s

**Retry** annotation has two properties, **name** that is valued with “product-api” the instance name in application.properties file and **fallbackMethod** wish take a method name that will be used as fall back in case all retry attempts fails, (the fall back method will be executed and it’s return value returned to the client).

### Fallback Method

Sometimes we may want to take a default action when all the retry attempts to the remote operation fail. This could be returning a default value or returning some data from a local cache.

The fallback method should be defined in the same class as the retrying class. It should have the same method signature as the retrying method with one additional parameter - the Exception that caused the retry to fail.