1. Spring Cloud Eureka
2. Spring cloud config server using GitHub repository
3. Client side Load Balancer using Spring Cloud Ribbon | Spring Boot
4. Spring Cloud- Load Balancing using Netflix Ribbon + Eureka | Spring Boot
5. Spring Cloud Feign - Declarative REST Client
6. Distributed Tracing in Microservice with Spring Cloud Sleuth & Zipkin
7. How to configure SpringCloud Zuul – Routing and Filtering using SpringBoot
8. Microservice - Centralize configuration using spring cloud consul
9. What is MicroService | MicroService Architecture
10. Single Sign-On with Spring Boot & Okta | OAuth 2.0
11. Spring Cloud Function | @functional | Example
12. SpringBoot : Spring Cloud Task | Example |
13. Microservice - Spring Cloud Consul | Service Discovery |
14. Microservice -Circuit breaker | Hystrix | Pivotal Cloud Foundry
15. Microservice - Cloud Config Server in Pivotal Cloud Foundry
16. Microservice - Service Discovery with Eureka in Pivotal Cloud Foundry
17. Spring Cloud Stream With Apache Kafka Binder | Example |
18. Spring Cloud Data Flow | Microservice Stream Processing
19. Microservice | Spring Cloud Eureka + API Gateway + Spring Cloud Hystrix | PART-1
20. Microservice | Spring Cloud Eureka + Gateway + Hystrix | PART-2 |
21. Microservice | Spring Cloud Eureka + Gateway + Hystrix | PART-3 |
22. Microservice | Spring Cloud Eureka + Gateway + Hystrix | PART-4 |
23. Microservice | Spring Cloud Eureka + Gateway + Hystrix + Cloud Config Server | PART-5
24. Spring Cloud Security | OAuth2 Google Sign In using Spring Boot

**1- Spring Cloud Eureka:** Netflix component **Eureka** for service registry and discovery.

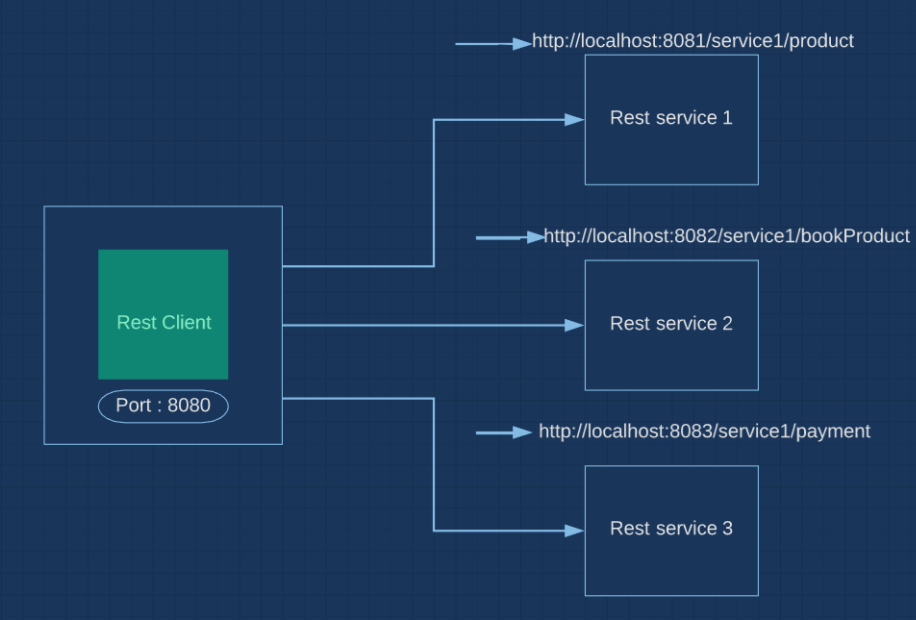
**What is Eureka?**

In simple word Eureka is a service Registry or we can say it is an embedded server provided by Netflix third party which integrates with spring framework.

**Main purpose to use Eureka:** Micro service Registration and Discovery with Spring Cloud and Netflix's Eureka

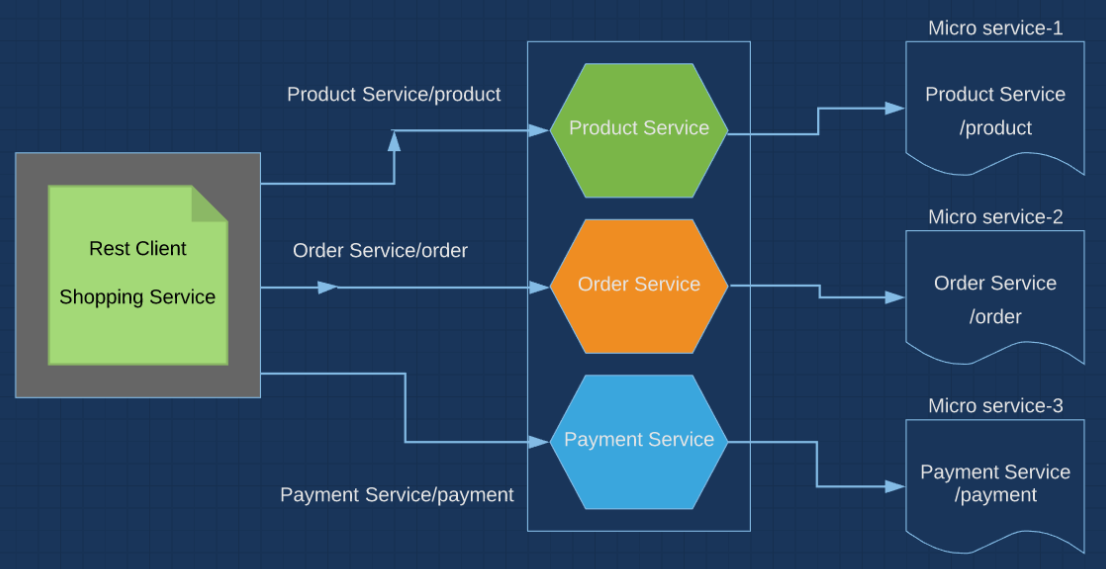
**What is the use of Eureka?**

Normally in Micro Service Architecture Design we are developing separate Services and exposing each API as service Endpoint and whenever we required to access other services in simple we are accessing it as a Rest client using third party API either Rest Template or Client Builder



As here we can see that we have three rest services 1,2,3 and each one may have different hostname and port number now if we want to call any of the rest service with the rest client then we will have to aware host name and port number of each services to be called, so this way we can see that there rest client is tightly coupled with each rest services and in future if any new service or instance gets deployed then it has to be communicated with its hostname and port number.

So to solve this kind of problem Eureka came into picture for spring cloud. Here each service gets registered with the Eureka server as given below.



As here we can see that each microservices is registered with Eureka server and now whenever we have to call to particular Microservice then we just need to provide Microservice\_Name/ resourceUrl and now when we hit the request then first it goes to Eureka server and search the service name in the Eureka Server Registry and further it searches for the resource URL and when match then it hit that particular microservices.

So my client is not communicating with the Microservice.

Step1- Mention the @**EnableEurekaServer** in the spring main launcher file.

|  |
| --- |
| @SpringBootApplication  @EnableEurekaServer  **public** **class** EurekaServerApplication |

Step2: Mention the below properties in the application.properties or yaml file.

|  |
| --- |
| spring.application.name=naming-server  server.port=8761  eureka.client.register-with-eureka=false  eureka.client.fetch-registry=false |

Now let’s start the server at: <https://localhost:8761> and we will get below screen.

**Note**: Here while creating the project as Eureka-server to register the services we will have add dependency [Eureka Server] from the dependency [spring-cloud-starter-netflix-eureka-**server**]

While if we have to register any service in the Eureka-server then we will have to add dependency [Eureka-Discovery client]

To register the service to Eureka server:

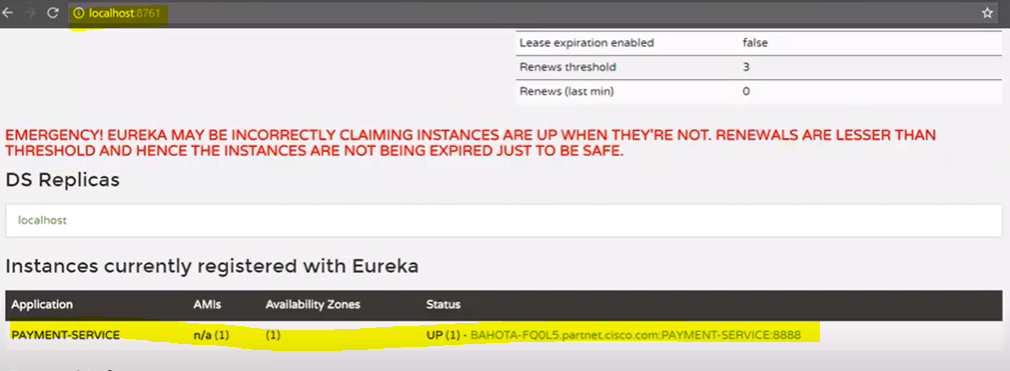
1. Add one dependency – [spring-cloud-starter-netflix-eureka-**client**]
2. Add annotation at spring boot launcher class.

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  **public** **class** PaymentServiceApplication |

3- Now in the application.properties file mention the below properties:

|  |
| --- |
| **eureka:**  **client:**  **registerWithEureka:** true  **fetchRegistry:** true  **serviceUrl:**  **defaultZone:** https://localhost:8761/eurkea  **instance:**  **hostname:** localhost  # OR  #eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka  #eureka.instance.hostname=localhost |

Example: Project: Eureka-Server & Payment-Service in eclipse:



When we start a project, we usually have all the configurations in the properties file.  
As more and more services are developed and deployed, adding and modifying these properties become more complex. Some services might go down, while some the location might change. This manual changing of properties may create issues.  
Eureka Service Registration and Discovery helps in such scenarios. As all services are registered to the Eureka server and lookup done by calling the Eureka Server, any change in service locations need not be handled and is taken care of

<https://www.javainuse.com/spring/spring_eurekaregister>

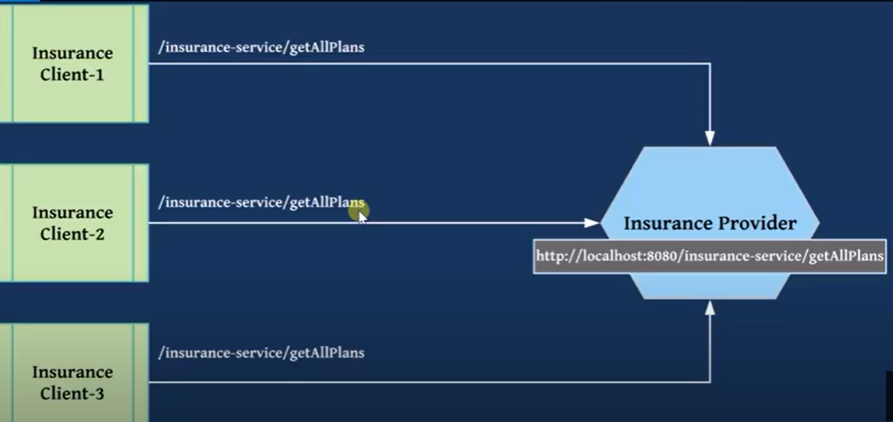
**2- Spring cloud config server using GitHub repository**

Here we will see how this spring cloud config server works and how the configuration related changes can be reflected without restart and redeploy our applications.

Basically this provides the server and client side configurations in our distributed system so that we can have a central place to manage our external properties of our application in different kind of environments.

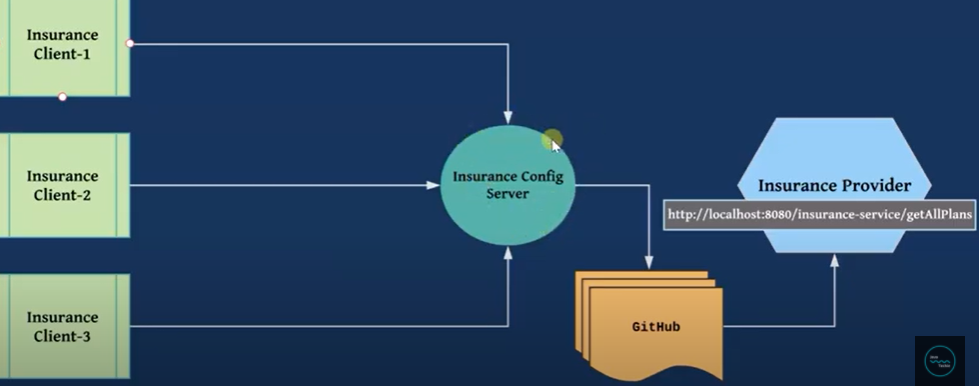
Here we can declare all the properties globally so that different kind of the microservices can access it in loosely manner.

**Q: What would be the draw without this cloud config server?**



As here we can see that all three clients are trying to Insurance provide with same URL [<https://localhost:8080/insurance-service/getAllPlans>], now if Insurance Provider changes its url [<https://localhost:8080/insurance-service/getAllUpdatedPlans> ] then all the client will have to change the URL to access the resources, so we can see that this configuration is tightly coupled.

To solve this problem cloud config server came into picture: Solution:



So here we can see that all the clients does not talk to Insurance provide directly, instead they talk to Config server. Now if Insurance Provider makes any changes in the URL then it will commit the same changes to the git-hub and now the latest update changes will start reflecting into Config server automatically because config server also connected to centralized repository git-hub.

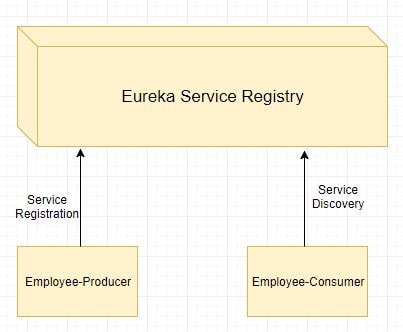
In this way cleint don’t need to bother about the changes they just need to talk to Config server. So here the Config server behaves like a gate way or entry point.

Note: When we make any changes and commit the same in git hub & Config server then just the reflect this updated changes we need to add one annotation @RefreshScope at the controller class level so that automatically it can have latest changes done in the git-hub and config server.

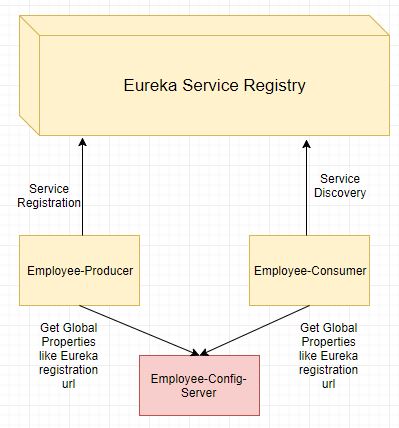
In shot:

Example to be done:

**What is Spring Cloud Config ?Need for it?**

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.  
  
[Mostly in all our previous tutorials](https://www.javainuse.com/spring/spring_eurekaregister) we were creating the modules as follows-  
  
Modules can have common global properties which are repeated in all the modules. For example we have have properties related to Database, Messaging Queues etc. For example in our employee-consumer and employee-producer we are having the following property for registering to Eureka Server.

eureka.client.serviceUrl.defaultZone=http://localhost:8090/eureka

We can **externalize this property using Spring Cloud Config.**  


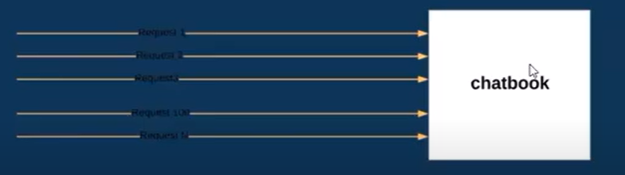
How Spring Cloud Config Works?

Spring Cloud Config Server can be either configured in following 2 ways-

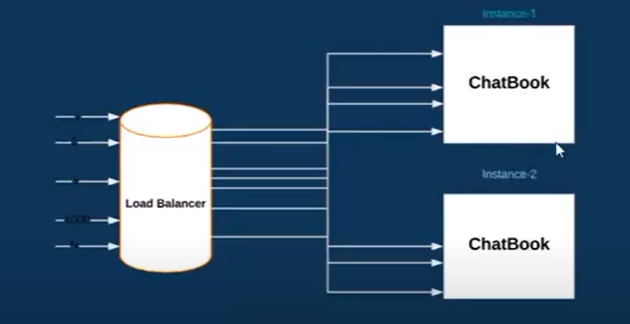
|  |  |
| --- | --- |
| **Using Local File System -**Properties to be externalized are stored in the local file system of the Spring Cloud Config Server. Externalize Properties Using Spring Cloud Example | **Using GIT Repo -**Properties to be externalized are stored in the GIT Repo.  Externalize Properties Using Spring Cloud and GIT Repo |

**Client side Load Balancer using Spring Cloud Ribbon | Spring Boot**

Load balancing means distribute the incoming request to the multiple servers.



Without load balancer: All the request is going to one instance of the application.



With Load Balancer: Placed and connected with the different servers to balance the load. Here we have placed different instances of the same application into several servers now the load balancer based on some algorithm sends the request to these servers.

1. **Round Robin Algorithm**: Requests will go one-by-one to each server in sequential order. Since here we are using Cloud Ribbon for load balancing so its default algorithm is Round Robin
2. **Least Connection:** If we want 100 requests to one server and another 100 request to another server then this algo is nothing but least connection.
3. **IP-Has:** Here based on the IP address of the server, we want to re-direct the request

**So these three are the algorithm used in load balancing.**

Example:

**Spring Cloud- Load Balancing using Netflix Ribbon + Eureka | Spring Boot**

Example needs to be done: <https://www.javainuse.com/spring/spring_ribbon>

**Spring Cloud Feign - Declarative REST Client**

What is Feign Client? : It is an http Rest client developed by Netflix. Best use case of Feign client is when we are performing load balancing in spring cloud and we are integrating Spring cloud Eureka service registry to invoke external service then we should not use Rest Template, and in this case we have to use the feign cleint where it internally balance the load.

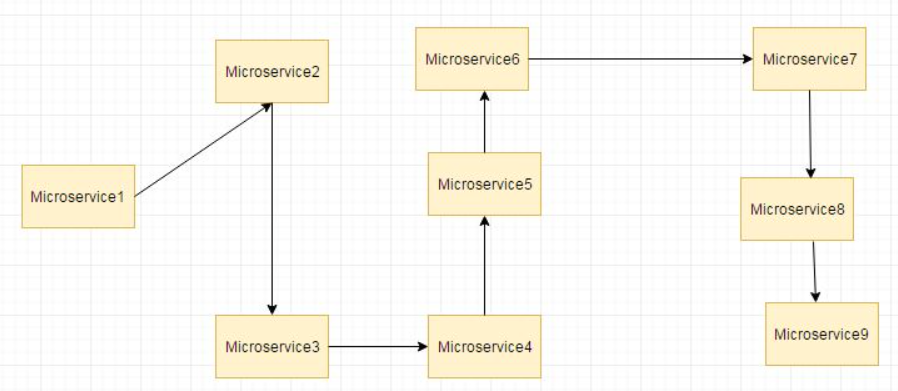
The advantage of Feign client is we no need to write the unit test case for Rest Client because there is no code. The developer just needs to declare and annotate an interface while the actual implementation will be provided at run time.

<https://www.javainuse.com/spring/spring-cloud-netflix-feign-tutorial>

Example:

**Spring Cloud Tutorial - Distributed Log Tracing using Sleuth and Zipkin**

Microservices architecture involves multiple services which interact with each other. So functionality may involve call to multiple microservices. Usually for systems developed using Microservices architecture, there are many Microservices involved. These microservices collaborate with each other.  
Consider the following microservices-



If suppose during such calls there are some issues like exception has occurred Or maybe there are latency issues due to a particular service taking more than expected time. How do we identify where the issue is occurring. In regular project we would have used logging to analyze the logs to know more about occurred exceptions and also performance timing. But since a microservice involves multiple services so we cannot use regular logging. Each Service will be having its own separate logs. So we will need to go through the logs of each service. Also how do we correlate the logs to a request call chain i.e which logs of microservices are related to Request1, which are related to Request2? To resolve these issues we make use of Spring Cloud Sleuth and Zipkin

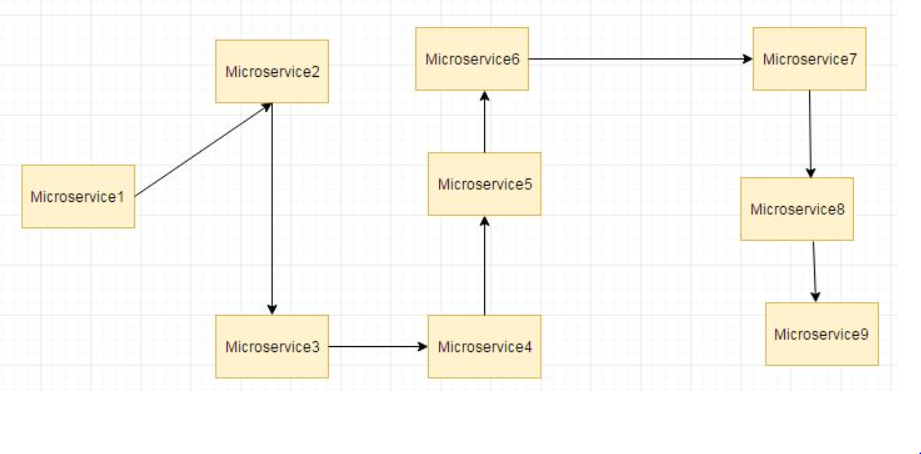
* **Spring Cloud Sleuth** is used to generate and attach the trace id, span id to the logs so that these can then be used by tools like Zipkin and ELK for storage and analysis
* **Zipkin** is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

Example : <https://www.javainuse.com/spring/cloud-sleuth>

**Spring Cloud- Netflix Eureka + Ribbon + Hystrix**

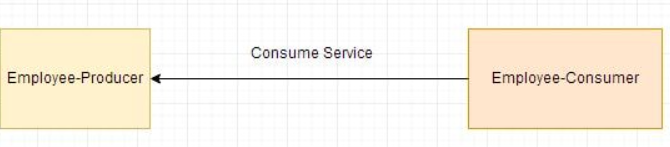
**What is Netflix Hystrix? Need for it?**

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.  
Usually for systems developed using Microservices architecture, there are many microservices involved. These microservices collaborate with each other.  
Consider the following microservices-



Suppose if the microservice 9 in the above diagram failed, then using the traditional approach we will propagate an exception. But this will still cause the whole system to crash anyways.  
This problem gets more complex as the number of microservices increase. The number of microservices can be as high as 1000. This is where hystrix comes into picture-  
We will be using two features of Hystrix-

* Fallback method
* Circuit Breaker



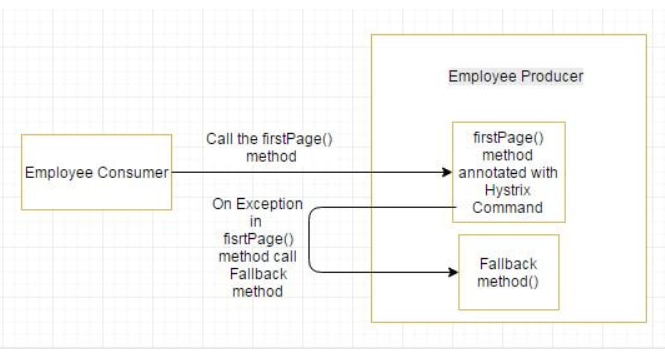
Now suppose due to some reason the employee-producer exposed service throws an exception. In this case using Hystrix we define a fallback method. This fallback method should have the same return type as the exposed service. In case of exception in the exposed service the fallback method will return some value.

<https://www.javainuse.com/spring/spring_hystrix>

**What is the Netflix Hystrix Circuit Breaker Feature? Need for it?**

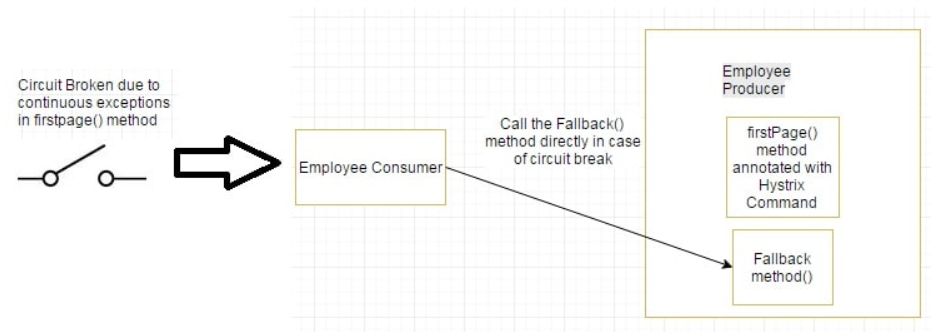
In previous posts we had two services- employee-consumer consuming the service exposed by the employee-producer.

Due to some reason the employee-producer exposed service throws an exception. In this case using Hystrix we defined a fallback method. In case of exception in the exposed service the fallback method returned some default value.



If the exceptions keep on occuring in the firstPage method () then the Hystrix circuit will break and the employee consumer will skip the firtsPage method all together and directly call the fallback method.

The purpose of circuit breaker is to give time to the first page method or other methods that the firstpage method might be calling and is causing the exception to recover. It might happen that on fewer loads the issue causing the exceptions has better chance of recovering.

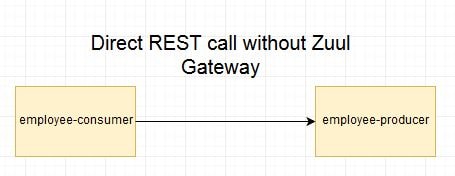
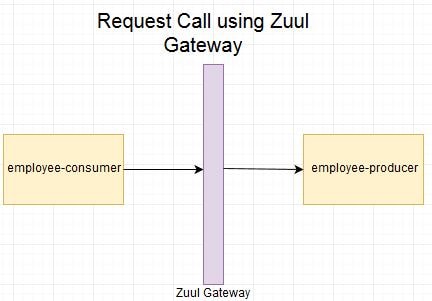


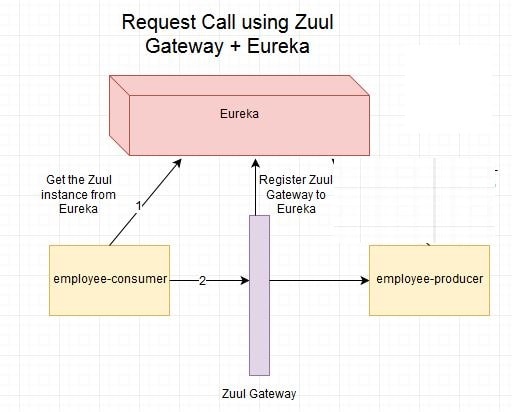
<https://www.javainuse.com/spring/spring_hystrix_circuitbreaker>

**Spring Cloud- Netflix Zuul + Eureka**

Zuul is the front door for all requests from devices and web sites to the backend of the Netflix streaming application. As an edge service application, Zuul is built to enable dynamic routing, monitoring, resiliency and security.

Zuul is a JVM based router and server side load balancer by Netflix.  
It provides a single entry to our system, which allows a browser, mobile app, or other user interface to consume services from multiple hosts without managing cross-origin resource sharing (CORS) and authentication for each one. We can integrate Zuul with other Netflix projects like Hystrix for fault tolerance and Eureka for service discovery, or use it to manage routing rules, filters, and load balancing across your system.

* **Microservice call without Netflix Zuul**  
  
* **Microservice call with Netflix Zuul**  
  
* **Microservice call with Netflix Zuul + Netflix Eureka**



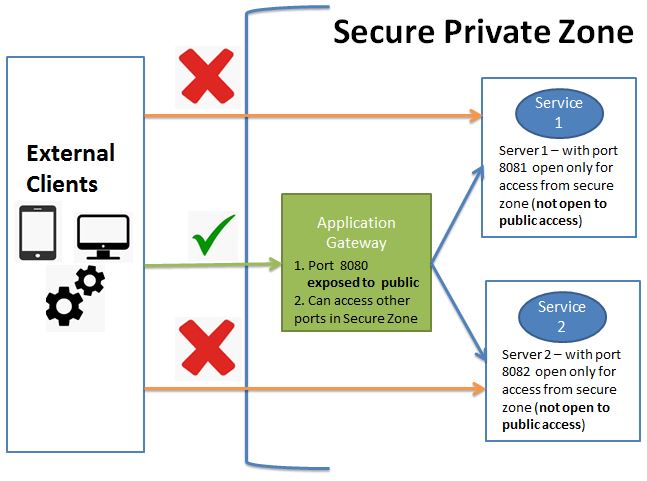
**Spring Cloud Tutorial - Spring Cloud Gateway Hello World**

In a [previous tutorial we had implemented API Gateway using Netflix Zuul Component.](https://www.javainuse.com/spring/spring-cloud-netflix-zuul-tutorial) However Zuul is a blocking API. A blocking gateway api makes use of as many threads as the number of incoming requests. So this approach is more resource intensive. If no threads are available to process incoming request then the request has to wait in queue.  
In this tutorial we will be implementing API Gateway using Spring Cloud Gateway. Spring Cloud Gateway is a non blocking API. When using non blocking API, a thread is always available to process the incoming request. These request are then processed asynchronously in the background and once completed the response is returned. So no incoming request never gets blocked when using Spring Cloud Gateway.  
We will first look at what is API gateway and why are they needed. Then we will be exploring the Spring Cloud Gateway Architecture and implement an API Gateway using it.

## What is an API Gateway? Why do we need it?

An API Gateway acts as a single entry point for a collection of microservices. Any external client cannot access the microservices directly but can access them only through the application gateway  
In a real world scenario an external client can be any one of the three-

* Mobile Application
* Desktop Application
* External Services or third party Apps



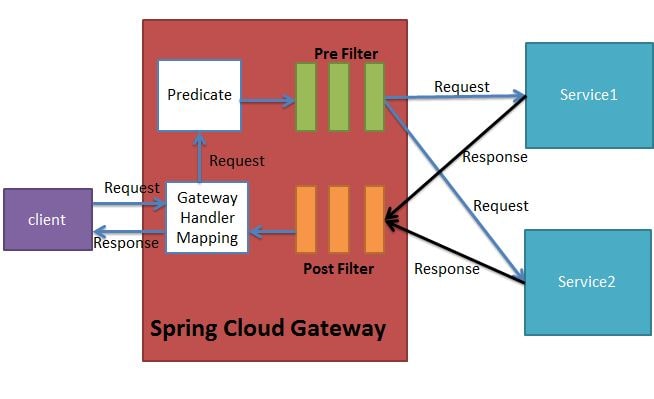
The advantages of this approach are as follows-

* This improves the security of the microservices as we limit the access of external calls to all our services.
* The cross cutting concerns like authentication, monitoring/metrics, and resiliency will be needed to be implemented only in the API Gateway as all our calls will be routed through it.
* The client does not know about the internal architecture of our microservices system. Client will not be able to determine the location of the microservices instances.
* Simplifies client interaction as he will need to access only a single service for all the requirements.

## Spring Cloud Gateway Architecture

Spring Cloud Gateway is API Gateway implementation by Spring Cloud team on top of Spring reactive ecosystem. It consists of the following building blocks-

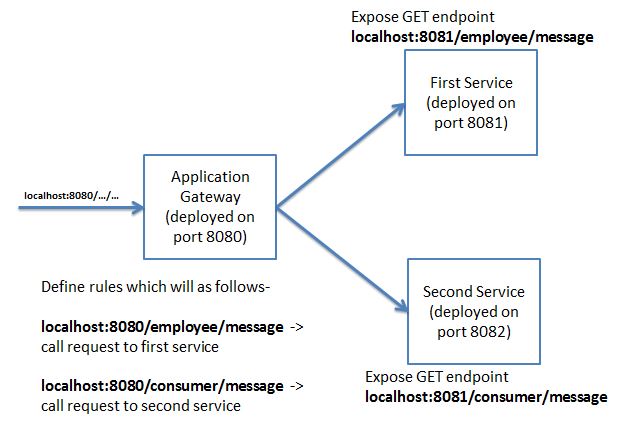
* Route: Route the basic building block of the gateway. It consists of
  + ID
  + destination URI
  + Collection of predicates and a collection of filters
* A route is matched if aggregate predicate is true.
* Predicate: This is similar to Java 8 Function Predicate. Using this functionality we can match HTTP request, such as headers , url, cookies or parameters.
* Filter: These are instances Spring Framework GatewayFilter. Using this we can modify the request or response as per the requirement. We will be looking at filters in detail in the next tutorial - [Spring Cloud Tutorial - Spring Cloud Gateway Filters Example](https://www.javainuse.com/spring/cloud-filter)

  
When the client makes a request to the Spring Cloud Gateway, the Gateway Handler Mapping first checks if the request matches a route. This matching is done using the predicates. If it matches the predicate then the request is sent to the filters.

## Implementing Spring Cloud Gateway

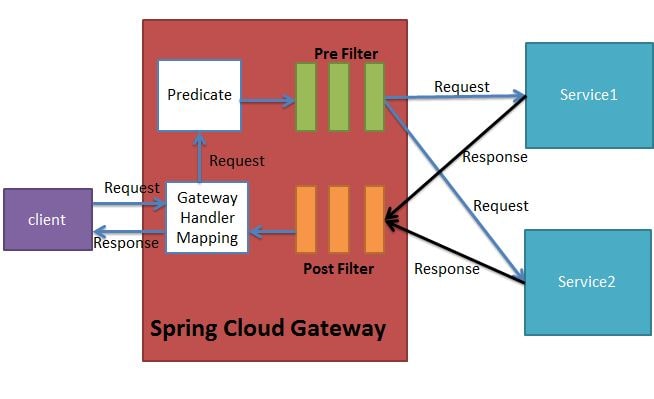
Using Spring Cloud Gateway we can create routes in either of the two ways -

* Use java based configuration to programmatically create routes
* Use property based configuration(i.e application.properties or application.yml) to create routes.

In this tutorial we will be implementing Spring Cloud Gateway using both configurations.  
We will be implementing Spring Cloud Gateway application which routes request to two other microservices depending on the url pattern.  


<https://www.javainuse.com/spring/cloud-gateway>

**Spring Cloud Tutorial - Spring Cloud Gateway Filters Example**

In a [previous tutorial we had implemented Spring Cloud Gateway Hello World Example](https://www.javainuse.com/spring/cloud-gateway). In this tutorial we will be making use of Spring Cloud provided filters and also create custom filters for our spring cloud gateway. In the next tutorial we will be [integrating Spring Cloud Gateway with Eureka Service Discovery.](https://www.javainuse.com/spring/cloud-gateway-eureka)  
Using Predicates Spring Cloud Gateway determines which route should get called. Once decided the request is the routed to the intended microservice. Before routing this request we can apply some filters to the request. These filters are known as pre filters. After applying the filters the intended micoservice call is made and the response is returned back to the Spring Cloud Gateway which returns this response back to the caller. Before returning the response we can again apply some filters to this response. Such filters are called post filters.  
  
As specified in the [Spring Cloud Gateway Documentation](https://cloud.spring.io/spring-cloud-gateway/reference/html/), Spring Cloud provides a number of built in filters. Also we can create our own custom filter to suit our business requirement. In this tutorial we will be looking at the various Filters that can be used with Spring Cloud Gateway.

## Implementing Spring Cloud Gateway Filters

Spring Cloud Gateway filters can be classified as

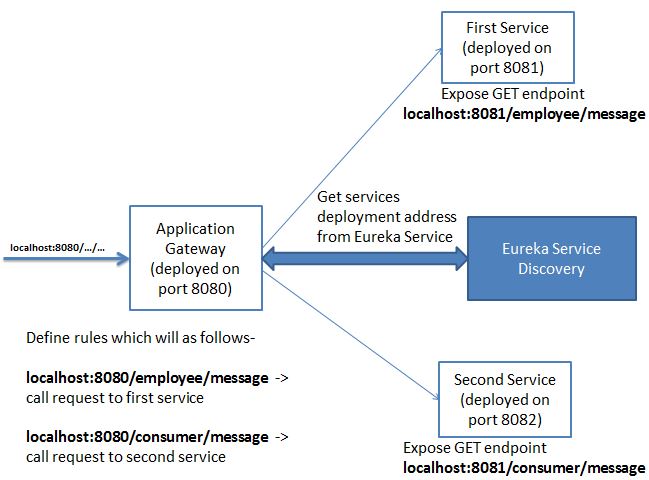
|  |  |
| --- | --- |
| * Spring Cloud Gateway Pre Filters * Spring Cloud Gateway Post Filters   spring cloud gateway filter classification | Spring Cloud Filters can be implemented in following two ways-   * Spring Cloud Gateway Filters using Java Configuration * Spring Cloud Gateway Filters using Property Configuration   spring cloud gateway filter implementation |

<https://www.javainuse.com/spring/cloud-filter>

**Spring Cloud Tutorial - Spring Cloud Gateway + Netflix Eureka Example**

In a [previous tutorial we had implemented Spring Cloud Gateway Hello World Example.](https://www.javainuse.com/spring/cloud-gateway) In this tutorial we will be implementing Spring Cloud Gateway and get the microservices deployment url using Netflix Eureka Discovery Service. In a [previous tutorial we had implemented Netflix Eureka Discovery Service.](https://www.javainuse.com/spring/spring_eurekaregister) When we start a project, we usually have all the configurations in the properties file. As more and more services are developed and deployed, adding and modifying these properties become more complex. Some services might go down, while some the location might change. This manual changing of properties may create issues. Eureka Service Registration and Discovery helps in such scenarios. As all services are registered to the Eureka server and lookup done by calling the Eureka Server, any change in service locations need not be handled and is taken care of

## Spring Cloud Gateway + Netflix Eureka Discovery Service

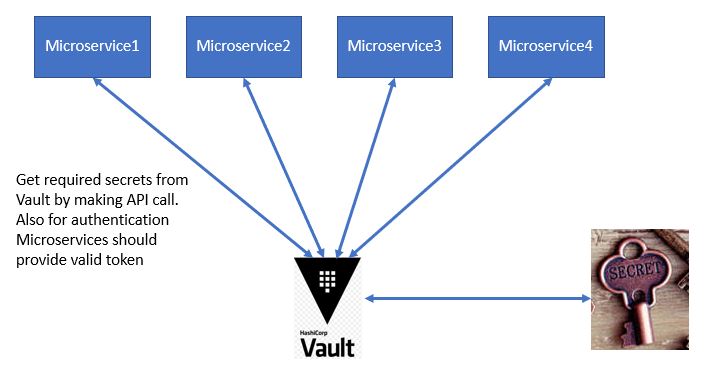
We will be modifying the code we had implemented in the previous Spring Cloud Tutorial - Spring Cloud Gateway Hello World Example.  


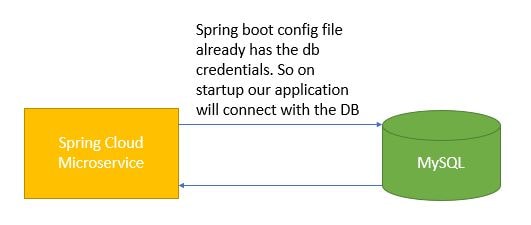
<https://www.javainuse.com/spring/cloud-gateway-eureka>

**Spring Cloud Tutorial - Secure Secrets using Spring Cloud Config + Vault**

Microservices architecture have multiple services which interact with each other and external resources like databases. They also need access to usernames and passwords to access these resources. Usually these credentials are stored in config properties. So each microservice will have its own copy of credentials. If any credentials change we will need to update the configurations in all microservices. We have previously discussed one solution to this problem is using [Spring Cloud Config Native Server](https://www.javainuse.com/spring/spring_cloud_config_server) or [Spring Cloud Config Git Server](https://www.javainuse.com/spring/spring_cloud_config_server_using_git) where common global properties which are repeated in all the microservices are usually stored.  But still storing the secrets in configuration file is a security concern. Above approach as 2 drawbacks-

* No single point of Truth
* Security risk of exposing the credentials

In this tutorial will be using Spring Cloud Config and Hashicorp Vault to manage secrets and protect sensitive data.  
  
Hashicorp Vault is a platform to secure, store, and tightly control access to tokens, passwords, certificates, encryption keys for protecting sensitive data and other secrets in a dynamic infrastructure.  
Using vault we will be retrieving the credentials from the vault key/value store.

We will be implementing a simple Spring Boot Microservice which returns employee details from MySql Database. We will be creating the Spring Boot + MySQL Application using Spring Boot JDBC. We have already seen [Spring Boot MYSQL JDBC basics in a previous tutorial.](https://www.javainuse.com/spring/bootjdbc) In We will be initially storing the MySql credentials in the configuration file.  
  
**Later we will be modifying this application to fetch the MySQL credentials from HashiCorp Vault.**  
