Micro Service : It is nothing but we need to break a monolithic Application in to small services which can interact with each other. Breaking down in logical manner, as per the thumb rule, a Micro Service can server a final work which cannot be break.

Like : Sending mail : In real work we can not go with this.

Developing MicroService by springBoot :

1. Web : providing all the dependencies which is needed to develop a web application.

2. Devtools : if we need that server need not be started every time, It would be started automatically once it got any changes.

3. Actuator : For monitoring prospective

Working functionality of Spring Boot :

**Key components:**

**Spring boot starters:** Combine all related jar in to a single jar.

**Spring boot auto configurator:** The main responsibility of Spring Boot AutoConfigurator is to reduce the Spring Configuration. If we develop Spring applications in Spring Boot,then We dont need to define single XML configuration and almost no or minimal Annotation configuration. Spring Boot AutoConfigurator component will take care of providing those information.

**Spring boot CLI**: Spring Boot CLI(Command Line Interface) is a Spring Boot software to run and test Spring Boot applications from command prompt. When we run Spring Boot applications using CLI, then it internally uses Spring Boot Starter and Spring Boot AutoConfigurate components to resolve all dependencies and execute the application.

**Spring boot actuator:**

1. Providing Management EndPoints to Spring Boot Applications.

2. Provide Spring boot application metrics

Once Application develop by Spring boot:

**Problem1** : There are thousands of MicroServices are running and there are thousands of properties are there, how we can centralized that.

**Solution**: We can to configure : spring-cloud server which contains all the information of each application.properties :

**Implementation**:

**Steps**:

1. You have a running springboot project limitservice having **spring.application.name=limitservice** in application.properties.
2. Add spring-cloud-starter-config dependencies inside this.
3. Create another Spring boot project (**spring-cloud-server**), which has dependency of spring-cloud-server.
4. Create a folder anywhere let say inside **d:\myfolder** and go inside that and run **git init** command.
5. Create limitservice.properties file inside d:\myfolder and add it into git bycommand git add –A, and then commit by command git commit –m “”. (**for profiling create name like : limitservice-qa.propertie**s
6. Put all name value pairs (which you want to centralized) inside this file i.e. in limitservice.properties.
7. Go to project **spring-cloud-server**, right click go to build path, click link source and add path till git folder.
8. In application.properties of Project **spring-cloud-server** add below line
   1. **spring.cloud.config.server.git.uri= git file path like file:///path into point 1 project application.properties**
9. Now go to project limitservice.
10. Change application.properties to bootstrap.properties and add below key value

spring.cloud.config.uri=<http://localhost:8888> (URL of **spring-cloud-server server**)

**Problem 2: Calling Rest Service :**

We can call another Rest Service by providing the **hardcoded** URL by RestTemplate and it will return you the RestEntity and we can get object by body part of that. But it will need a lot of code and again and again we need to write same code. To overcome this we will use feign client.

Steps:

1. We have currency-exchange-service(ces) and currency-conversion-service (ccs). Currency conversion-service use or call currency-exchange-service.
2. Either we can use RestTemplate inside ccs to ces. Or we can introduce feign client.
3. Add spring-cloud-starter-openfeign dependency in ccs.
4. Add @EnableFeignClient in main class.
5. Create a interface with annotation @FeignClient (name=”name of application” i.e. currecy-exchange-service” , url = localhost:port) on which ces is running.
6. Create a same signature method which is in controller. Like below.

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversionBean retrieveCurrencyValue(@PathVariable String from, @PathVariable String to) ;

1. Declare variable of this Interface in Main controller with autowire.
   1. Call method by proxy only.

*Add dependency like : spring-cloud-starter-config and spring-cloud-starter-openfeign*

1. *Create an interface and add annotation like : @FeignClient (name =“a”, url =“b”)*

*a: application name which you need to call or from where you get the response.*

*b. Url : where your rest service is running : like* [*http://localhost*](http://localhost)

*3. Code inside Interface would be like this :*

*@GetMapping("/currency-exchange/from/{from}/to/{to}")*

*public CurrencyConversionBean retrieveCurrencyValue(@PathVariable String from, @PathVariable String to) ;*

*4. Go to spring initializer class and put annotation like : @EnableFeignClients("com.ramji.apptad.\*")*

*4. Now goto Controller from where you write down the RestTemplate Code. Call like*

*proxy.methodName()*

*It will reduce lots of the code.*

**Problem 3**: How to manage cluster which contains single application running on various port.

like an application running on 8000 and 8001 and you want to access this by Feign client. Or you can say application need load balancing. Load balancing would be done by Ribbon.

**Solution** :

1. To manage cluster we need to Introduce **Ribbon,** go to pom.xml and Introduce a new dependency like :

Spring-cloud-starter-netflix-ribbon

2. Now go to Proxy class and add annotation @RibbonClient(name=“applicationname”)

3. Remove URL from @FeignClient so now annotation would be like : @FeignCLient(applicationname)

4. Go to application.properties and add your server URL (which you want to call and run on two ports or cluster)

currency-exchange-service.ribbon.listOfServers=http://localhost:8000, <http://localhost:8001>

Run application, now it will contacted with 8001 and 8000, it will work like load balancer.

**Problem 4:** Need to register your service and discover it.

Solution :

1. Create a Spring Boot Application and add dependencies like :

spring-starter-cloud-config and spring-cloud-starter-netflix-eureka-server

2. Go to spring initializer class and add annotation @EnableEurekaServer

3. Go to application.properties and add below

eureka.client.registry-with-eureka=false

eureka.client.fetch-registry=true

4. Now go to application which need to be registered with Eureka

5. Add dependencies

spring-cloud-starter-netflix-eureka-client

6. Go to spring initializer and add below annotation

@EnableDiscoverCLient

7. Go to application.properties and add Eureka server URL:

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

Remove any other URL like : ribbon list of servers : check Eureka URL it will ask Eureka for service which has the application name.

Now :

Problem5 :

API Gateways:

1. Authentication & Authorization with centralized
2. Rate limit (Request for a MS rate limit)
3. Fault tolerance
4. Service aggregation (if one need to call 25 MS then it is better that we can give a URL on which customer can call a single service.

To get this we need ZUUL server:

Requirement :

1. Need a component for zuul .
2. What we need when zuul will intercept a MS request.
3. All request need to passed through the zuul server.

Steps :

1. Create a springboot project with zuul dependency.
2. Add @ZuulEnableProxy
3. Add a class and Implement Class ZULLFilter
   1. Run : actual business logic
   2. shouldFilter : this filter need to run or not
   3. FilterType : pre, post, error
   4. FilterOrder : order of filter

Now If the request was : <http://localhost:port/app_name/>

Now request would be [http://localhost:<zull\_port>/app\_name/](http://localhost:%3czull_port%3e/app_name/)

**Problem :**

If there are so many MS and they are calling with each other, how I can trace my error or exception for one request.

**Solution** : Distributed Tracing by Spring Cloud sleuth : It will create a unique requesteId for each request.

Steps : add below dependency

<dependency>

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

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

</dependency>

Now add bean for

@Bean

**public** Sampler defaultSampler(){

**return** Sampler.***ALWAYS\_SAMPLE***;

}

Now when you made any request, a unique id would be assigned and it spawn to all other microservice request.

Now problem is we need to look different console or log file for each appl’n log so Zipkin is server which is responsible for central matrix.