**Spring cloud and microservices**

Microservice is an architecture design for dividing big application into small units where each unit(microservice) can be individually developed and deployed. These units(microservices) coordinate with each other to provide the functionality of the complete application.

Spring cloud is built on top of spring boot or in other words it is dependent on spring-boot and spring-boot is the minimum dependency/library required to use spring cloud support

**Advantages of Microservice architecture**

1. Easier to test small microservice than one big monolithic application
2. Each microservice can be developed with different language and library,

foreg. One microservice can be running on java, other microservice can be developed on python and running on its platform

Similarly one microservice can be running on java 6 while other microservice might be running on java 8

And All these microservices are still coordinating with each other in the application

1. Microservices scale(increase size of operations) independently, more memory, other hardware etc can be increased for each microservices individually as each microservice must be running on different machines in production.

Foreg. A lot of hardware like RAM and other hardware can be given to

microservice1 if traffic is high

And moderate amount of RAM and other hardware can be given to microservice2 if the traffic is not that high

**Spring Cloud Components**

1) Service Discovery

2) Service Registry

3) Load Balancer

4) Api Gateway

5) Configuration Server

6) Circuit Breaker

7) Service Monitoring

**Service Discovery**

Service Discovery is the minimum concept for developing microservice

Requires DiscoveryServer and Discovery Client

1. **Discovery Server** : this is where all services registers and this is therefore which helps in discovering services when a client service requests

Creating Discovery server in 2 steps

A)

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

1. Mentioning **hostname and port number** are the minimum requirements for running discovery server

application.properties

*# Configure this Discovery Server***eureka.instance.hostname**= **localhost***# HTTP (Tomcat) port***server.port**=**8585**

1. **Discovery Client** :

A)

@SpringBootApplication  
@EnableDiscoveryClient  
**public class** EmployeeApplication {

SpringApplication.*run*(EmployeeApplication.**class**, args);

}

B) application name, defaultzone and port number are the minimum things which discovery client should mention

*# Service registers under this name***spring.application.name**= **employee-microservice***# Discovery Server Access assuming discovery server running on same machine***eureka.client.serviceUrl.defaultZone**= **http://localhost:8585/eureka/***# HTTP Server (Tomcat) Port***server.port**= **8586**

**Clientside Load balanacer**

In traditional model load balancing is done at server side where requests are distributed on different server machines,

In Client side load balancingclient decideswhich server machine it will send request to for eg. Request1 sent to server 1, request2 sent to server2 and both these servers acting as single micro service(ie.Registered with same name). All this client load side balancing is done automatically by

1)Important steps provide beans of resttemplate and mark the method with @LoadBalanced

@LoadBalanced

@Bean  
RestTemplate restTemplate() {  
 **return new** RestTemplate();  
}

1. And where request is to be sent to others services get restTemplate

@Autowired

//correct way  
**private** RestTemplate **restTemplate**;

//below is wrong,you can discover service this way

**RestTemplate restTemplate=new RestTemplate()**

**Circuit Breaker(provides failover capability)**

Circuit Breakers are similar in analogy to try/catch block where original method is similar to try block and catch block is similar to fallback method.

Circuit breakers are however more intelligent, if original method is failing too often then original method will be bypassed and fallbackmethod will be called.

Failures can be due to reasons like Remote service is not available etc

**Primary job of circuit breaker is due to provide failover capability**

**Externalizing Configuration**

Spring cloud provides server and client side support for externalized configuration in distributed environment.

**Steps to use**

1. Create configuration server
2. Run configuration server with application class like below

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

B) bootstrap.properties files should have below information

**#port on which config server runs**

**server.port** = **9494***#Git repo location where properties file will be kept, can be local machine or maintained in a server in network*

*#or git repository can be hosted on github and url provided here***spring.cloud.config.server.git.uri**=path to git repository

1. **Create Configuration client which will use properties from git repository**
2. Add springboot actuator dependency to the client application in pom

B)

@RefreshScope  
@RestController  
**public class** MessagesController {  
 @Value(**"${msg1}"**)  
 **private** String **message1**;  
 @GetMapping(**"/message1"**)  
 **public** String getMessage1() {  
 **return this**.**message1**;  
 }  
}

**C)**

**Mention in properties file**

**spring.application.name**=**client***#spring.profiles.active=development*

*#configuration server running on same machine, if running on other machine provide ip of that machine***spring.cloud.config.uri**=**http://localhost:9494  
management.security.enabled**=**false  
management.endpoints.web.exposure.include**=**refresh**

**Refresh The Changes when there is change in properties**

1. **Manual**

Trigger the client refresh to fetch new values by sending an empty post request at actuator refresh endpoint, in our case below is the url for post request since client is running on local machine at port 8585

<http://localhost:8585/actuator/refresh>

Confirm it is working by visting the <http://localhost:8585/message>

1. **Automatic**

Whenever there is commit in properties, it will trigger changes on all the clients of configuration server, This requires **Spring cloud bus , message brokers like kafka/rabbitmq**