Micro Services

1. Brown Field Project:-

Monolithic to micro services.

2. Green Field Project:-

Micro services in nature from scratch.

1. Monolithic Applications:-

Our Application contain several components

-presentation components

-web components

-Business components

-Persistence components

-Authorization components

-Integration tier components

->If develop all the components in the single projects then that project is called as monolithic Architecture based project.

->Once development is completed we will package our application as war file and will deploy into server.

->monolithic application packaged war is called as Fat war file.

->if load is increased on server, server might get crashed then application will be down

To avoid server crashes we need to scale our application

Scalling :

To handle multiple request to make on server that process is called as scalling .

One of the technic is load balancer

Scaling we can do in 2 ways

1.Vertical scalling :-

->vertical scalling nothing but increasing single server capacity.

Ex: ram,processor,…

2. Horizontal scalling

->Horizontal scalling means adding more no. of servers.

Load Balancer :-

1.Round ribbon algorithm

2.Sticky session

3.Ip hashing

1.Round ribbon algorithm

Once request’s come from users to Load balancer ,then it will tell, that request which server it will go handle by round ribbon algorithm

2.Sticky session algorthm

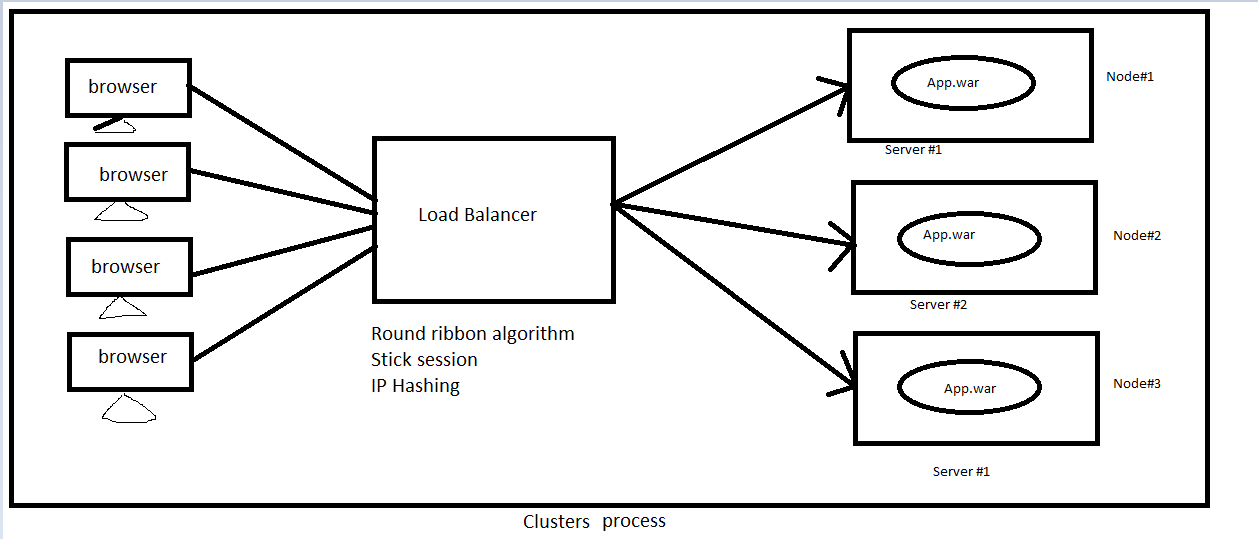
Once request’s come from users to Load balancer ,then it will tell, that request which

Node it will go handle by sticky session algorithm

Which node your session object is created ,your next request also go to that node .it will main the session id .

3.IP Session : it is having formula do mathematical calculations.

Cluster flow diagram :



Micro services :

Drawbacks:

1.Visibility: Micro services project can developed by individual developers, let assume 10 developers have developed 10 services .each one has developed one service only .So all developers doesn’t grip(fit) all services .

2. Configuration : We need to maintain each service configuration code individually.

3.Bounded Context: Let assume we have 10 services. Program has to start service to write code. First he has to identified which will take first .

4.Pack of cards : Let assume we have 10 services in flow .if any one micro services went down or missing or other problem .

Advantages :

1.Easy maintance : let assume instead of maintain 10 services ,we can main one service maintain individually easy

2.Faster release : we can easily deploy our code into server without brake other service.

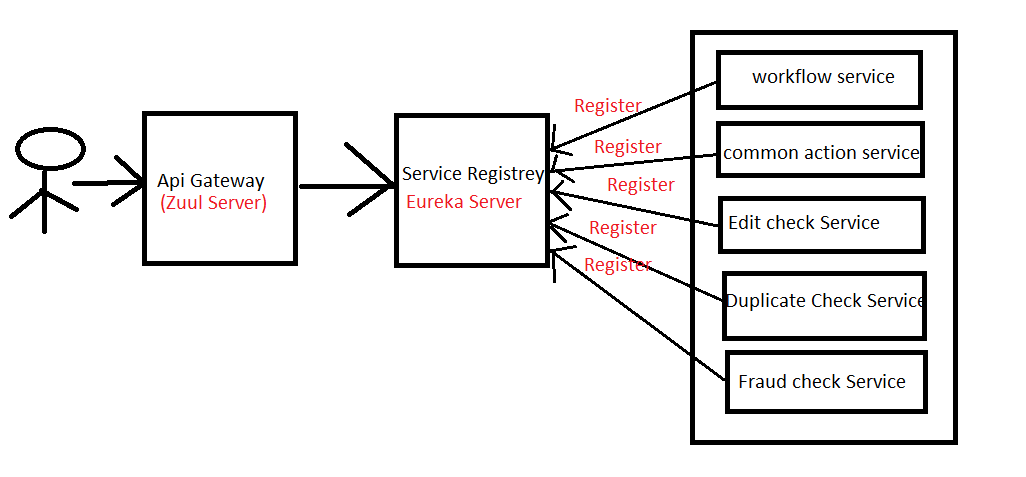
3.Technology independent : we can write our service in different technologies like ex:java,php,.net,phython……

Our company there is not much java developers. we have more python,.net peoples have it.

We can use them instead of taking new resources in java people (out side).

4.parallel development : we can develop paralley .java 1 sevice,.net one service,phython one service …possible.

Micro service Architecture:



Components :

1.Service(REST API):-we have multiple services .

2.service Registry: Every REST service(Eureka client) is needs to register in Service Registry(Eureka Server).

Eureka Server is also known as Discovery Server.

It will provide our Services details like : URL’s, Service Names, Our Service is UP/Down(running or not)--🡪Optional but we may use security purpose

3.API Gateway : It is routing our services where to go provided and filtering(making request is process or not) using ZUUL server. (EX: Front controller in MVC)---🡪 Optional but we may use security purpose .

Most of companies may not use these 2 component(Service registry and Api gateway) but directly they placed into servers using Restful.

Step to create service registrey project :

1) Create spring-boot application with below dependencies

A)spring-boot-starter-web

B)spring-cloud-Netflix-eureka-server

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web </artifactId>

</dependency>

<dependency>

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

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

</dependency>

2)specify @EnableEurekaServer at spring boot start class

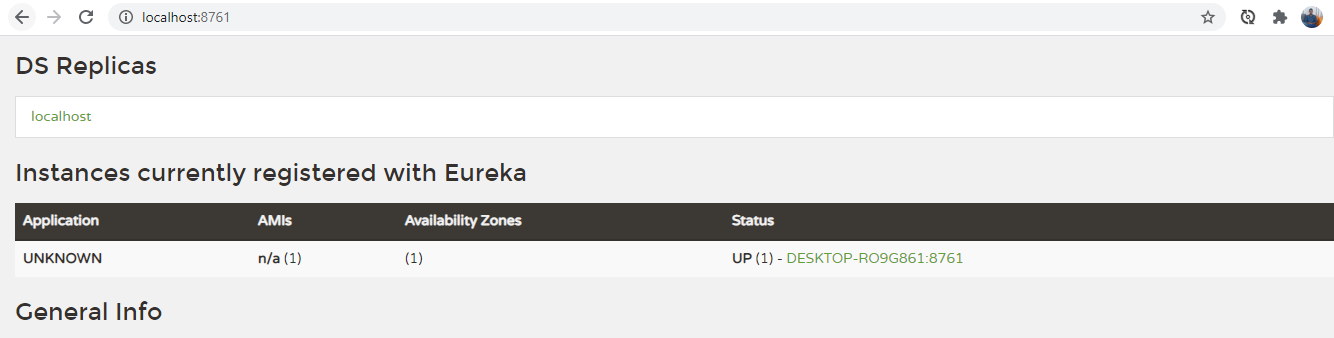
To represent this project act as a **Eureka Server(Discovery Server) project** use above annotation .

3) Configure embedded container port number as 8761

Note :If Embedded container port number is 8761,eureka client can auto register.

4) By default, the Eureka Server registers itself into the discovery. eureka.client.register-with-eureka=true(default)

Url : <http://localhost:8761/>

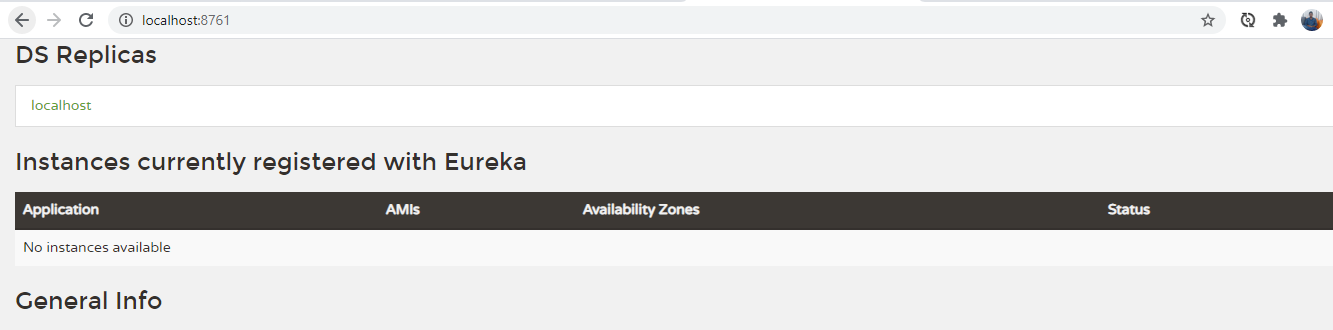


Note :

Service Registry project should not register with Eureka Server. We have to configure property in **application. Properties** file .

eureka.client.register-with-eureka=false

Url : <http://localhost:8761/>



5)Run application and access dashboard using below URl

Url : <http://localhost:8761/>

Coding Steps :-

Step 1 :-

Starter class :-

@SpringBootApplication

@EnableEurekaServer

public class MicroservicesServiceRegistryApplication {

public static void main(String[] args) {

SpringApplication.run(MicroservicesServiceRegistryApplication.class, args);

}

}

Step 2 :-

application.properties

server.port=8761

eureka.client.register-with-eureka=false

Eureka Client :-

Any service which is registered with Eureka Server is called Eureka client

If Eureka Server & Eureka client Running on same machine then Eureka Client will Search for Eureka Server with 8761 port number

If Eureka Server running on same machine with 8761 port number then client will auto-register.

If Eureka port number is not 8761 then we have to register client manually.

Steps to develop Eureka Client Application :-

1)Create Spring boot application with below dependencies

1)spring-boot-starter-web

2)spring-cloud-starter-netflix-eureka-client

3)Devtools

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web </artifactId>

<dependency>

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

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

</dependency>

2) Configure @EnableDisccoveryClient annotation at Spring boot starter class level

(To represent this act as a client project)

3)Create Rest controller with required methods

4)Configure below properties in application.yml file

a)server-port

b)application-name

c)register-with-eureka(optional🡪 server is running on 8761 same machine ,then no need register client ,because it can auto register) if different only need to register manually

5)Run an spring boot application and check Eureka dashboard(client application should be registered with Eureka).

Steps to write :-

Starter class :-

@SpringBootApplication

@EnableEurekaClient

public class MicroservicesServiceDiscoveryApplication {

public static void main(String[] args) {

SpringApplication.run(MicroservicesServiceDiscoveryApplication.class, args);

}

}

application.properties

eureka.client.serviceUrl.defaultZone = http://localhost:8761/eureka

eureka.client.instance.preferIpAddress = true

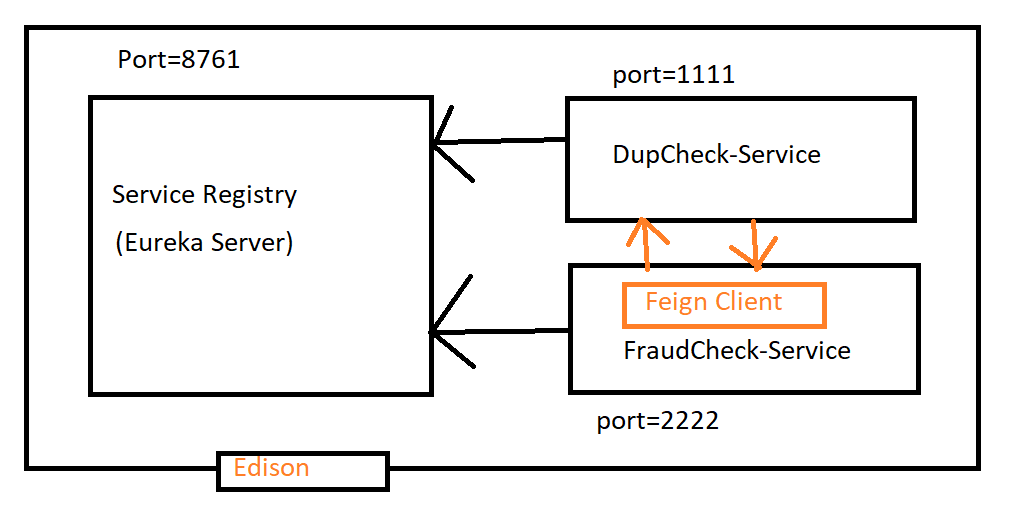
spring.application.name = Client App First

server.port=8085

How to register the Spring Boot Micro service application into the Eureka Server :

Feign client :-

Feign client is used to inter service communication for different rest services .



Mini project Example for Service Registry:

Eureka server code :-

Step 1 :add dependencies in pom.xml file:

1.web

2.Eureka server

Step 2 :configure annotation at starter class level

@SpringBootApplication

@EnableEurekaServer

**public** **class** SbmsEurekaServerApplication {

**public** **static** **void** main(String[] args) {

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

}

}

Step 3: application.properties

server.port=8761

eureka.client.register-with-eureka=false

Step 4 :

Run and check below Url

<http://localhost:8761/>

we can see Eureka server console Ui

Eureka client services code :-

Service 1: DupCheckService :-

Step 1 :add dependencies in pom.xml file:

1.web

2.Eureka discovery client

3.devtools

Step 2 :configure annotation at starter class level

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** SbmsEurekaClientDupCheckServiceApplication {

**public** **static** **void** main(String[] args) {

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

}

}

Step 3: application.properties

server.port=1111

spring.application.name=Dupcheck-Service

Step 3:create restcontroller class

@RestController

**public** **class** DupcheckController {

@GetMapping("/dupcheck")

**public** String getDupcheckService()

{

**return** "Welcome to Dupcheck-Sevice";

}

@GetMapping("/two")

**public** String getDupcheckServiceTwo()

{

**return** "Welcome to Dupcheck-Sevice Two";

}

}

Step 4 :

Run and check below Url

<http://localhost:8761/>

Here you can find service and then click url

[http://desktop-ro9g861:1111/actuator/info------>modify](http://desktop-ro9g861:1111/actuator/info------%3emodify) url below format like this

[http://desktop-ro9g861:1111/(pathname)---->output](http://desktop-ro9g861:1111/(pathname)----%3eoutput) came.

Service 2: FraudCheckService :-

Step 1 :add dependencies in pom.xml file:

1.web

2.Eureka server

Step 2 :configure annotation at starter class level

@SpringBootApplication

@EnableDiscoveryClient

@EnableFeignClients

**public** **class** SbmsEurekaClientFraudCheckServiceApplication {

**public** **static** **void** main(String[] args) {

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

}

}

Step 3: application.properties

server.port=2222

spring.application.name=Fraudcheck-Service

Step 4:create restcontroller class

@RestController

**public** **class** FraudcheckController {

@Autowired

**private** DupcheckFeign fraudcheckFeign;

@GetMapping("/fraudcheck/{name}")

**public** String getFraudcheckService(@PathVariable("name")String name)

{

String dupresponse=fraudcheckFeign.communicateAbstractMethodDupcheck();

String dupresponseTwo=fraudcheckFeign.communicateAbstractMethodDupcheckTwo();

String fraudresponse="Hi "+name;

String finalresponse=fraudresponse+" , "+dupresponse+" , "+dupresponseTwo;

**return** finalresponse;

}

}

Step 5 : create interface for inter communication for feign client

@FeignClient(name = "DUPCHECK-SERVICE")

**public** **interface** DupcheckFeign {

@GetMapping("/dupcheck")

**public** String communicateAbstractMethodDupcheck();

@GetMapping("/two")

**public** String communicateAbstractMethodDupcheckTwo();

}

Step 4 :

Run and check below Url

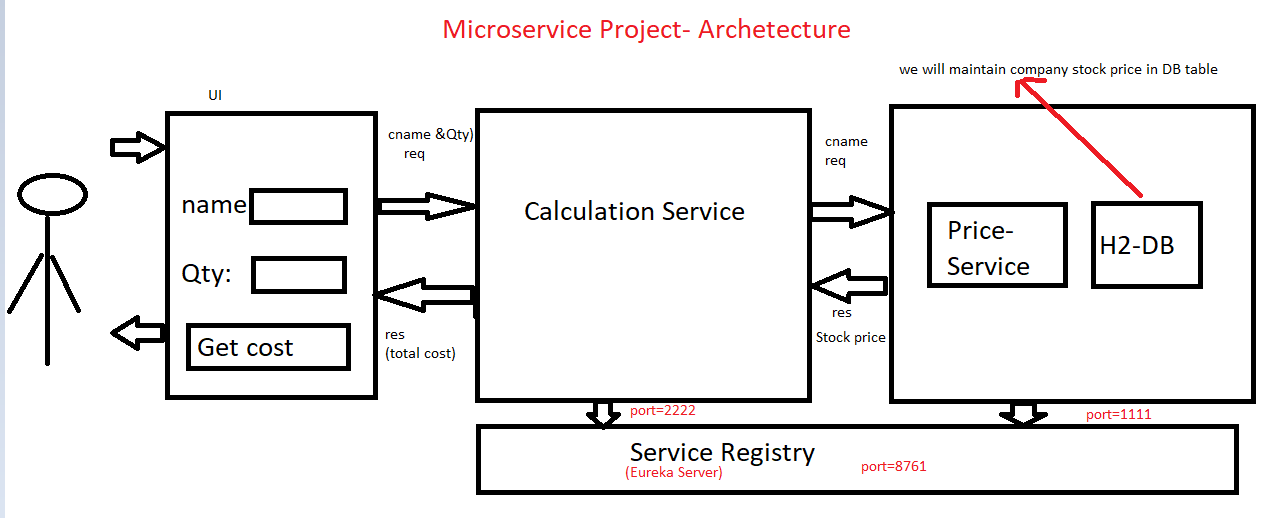
<http://localhost:8761/>

Here you can find service and then click url

[http://desktop-ro9g861:2222/actuator/info------>modify](http://desktop-ro9g861:2222/actuator/info------%3emodify) url below format like this

[http://desktop-ro9g861:2222/(pathname)---->output](http://desktop-ro9g861:2222/(pathname)----%3eoutput) came.

Mini project :



-> Stock-price-service it is one rest api which is maintaining company stock prices

-> It will take company name as Input and it will return stock price as output.

->Stock-calculation-service is responsible to calculate total cost for given quantity.For calculation it will get company stock price from stock-price-service using feignClient .

Eureka server code :-

Step 1 :add dependencies in pom.xml file:

1.web

2.Eureka server

Step 2 :configure annotation at starter class level

@SpringBootApplication

@EnableEurekaServer

**public** **class** SbmsEurekaServerApplication {

**public** **static** **void** main(String[] args) {

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

}

}

Step 3: application.properties

server.port=8761

eureka.client.register-with-eureka=false

Step 4 :

Run and check below Url

<http://localhost:8761/>

we can see Eureka server console Ui dashboard

Steps to develop STOCK-PRICE-SERVICE

Step 1:

Create application wit below dependencies

1.web 2.jpa 3.h2 4.lombok 5.devtools 6.eureka client 7.swagger

<!-- Swagger starts -->

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.0.2</version>

</dependency>

<!-- Swagger Ends -->

Step 2:

Configure @EnableDiscoveryclient annotation at starter class level.

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** SbmsServiceRegistryStockpriceServiceApplication {

**public** **static** **void** main(String[] args) {

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

}

}

Step 3:

Configure below application.properties file

server.port=1111

spring.application.name=STOCKPRICE-SERVICE

spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.password=null

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=sa

Step 4:

Create entity class and repository interface

@Data

@Entity

@Table(name="STOCK\_PRICE")

**public** **class** StockPrice {

@Id

@Column(name="STOCK\_ID")

**private** Integer stockId;

@Column(name="COMPANY\_NAME")

**private** String companyName;

@Column(name="COMPANY\_PRICE")

**private** Double companyPrice;

}

**public** **interface** StockpriceRepository **extends** JpaRepository<StockPrice, Serializable>{

**public** StockPrice findByCompanyName(String companyName);

}

Step 5:

Create data.sql file with insert queries to insert data in table why: actually we are using H2 Db ,so once server stop then data will deleted .

(src/main/resource)

Data.sql :-

INSERT INTO STOCK\_PRICE(STOCK\_ID,COMPANY\_NAME,COMPANY\_PRICE) VALUES(101,'HCL',500.00);

INSERT INTO STOCK\_PRICE(STOCK\_ID,COMPANY\_NAME,COMPANY\_PRICE) VALUES(102,'TCS',1000.00);

INSERT INTO STOCK\_PRICE(STOCK\_ID,COMPANY\_NAME,COMPANY\_PRICE) VALUES(103,'WIPRO',1500.00);

INSERT INTO STOCK\_PRICE(STOCK\_ID,COMPANY\_NAME,COMPANY\_PRICE) VALUES(104,'DELL',2000.00);

Step 6:

Create service interface and implementation

**public** **interface** StockPriceService {

**public** Double getStockPrice(String companyName);

}

@Service

**public** **class** StockPriceServiceImpl **implements** StockPriceService {

@Autowired

**private** StockpriceRepository stockpriceRepository;

@Override

**public** Double getStockPrice(String companyName) {

StockPrice entity = stockpriceRepository.findByCompanyName(companyName);

**if** (entity == **null**) {

**throw** **new** CompanyNotAvailableException("Company not available");

}

Double companyPrice = entity.getCompanyPrice();

**return** companyPrice;

}

}

Step 7:

Create restcontroller with required methods

@RestController

**public** **class** StockPriceServiceRestController {

@Autowired

**private** StockPriceService stockPriceService;

@GetMapping("/getPrice/{name}")

**public** ResponseEntity<Double> getPrice(@PathVariable("name")String companyName )

{

Double stockPrice=stockPriceService.getStockPrice(companyName);

**return** **new** ResponseEntity<>(stockPrice,HttpStatus.***OK***);

}

}

Step 8:

Create Exception handler for handling exception

**public** **class** CompanyNotAvailableException **extends** RuntimeException{

**public** CompanyNotAvailableException(String message) {

**super**(message);

}}

Actually if we write above code we have handle null pointer exception, but problem is status code got it as 500,to avoid for that we should create one extra class below code

@RestControllerAdvice

**public** **class** StockPriceExceptionHandler {

//we need to dipsplay message so return type is String

@ExceptionHandler(CompanyNotAvailableException.**class**)

**public** ResponseEntity<String> handleCompanyNotAvailableException(Exception e)

{

String msg=e.getMessage();

**return** **new** ResponseEntity<>(msg,HttpStatus.***BAD\_REQUEST***);

}

}

Step 9:

Configure swagger UI

@Configuration

@EnableSwagger2

**public** **class** SwaggerConfig {

@Bean

**public** Docket customerApi()

{

**return** **new** Docket(DocumentationType.***SWAGGER\_2***)

.select()

.apis(RequestHandlerSelectors.*basePackage*("com.app"))

.paths(PathSelectors.*any*())

.build();

}

}

Step 10 : run as spring boot client app and test it

<http://localhost:1111/swagger-ui.html>

<http://localhost:8761/>

Steps to develop STOCK-Calculation-SERVICE

Step 1:

Create application wit below dependencies

1. web 2.eureka client 3.feign client 4.swagger 5.devtools

<!-- Swagger starts -->

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.9.2</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.0.2</version>

</dependency>

<!-- Swagger Ends -->

Step 2:

Configure @EnableDiscoveryclient annotation at starter class level.

@SpringBootApplication

@EnableDiscoveryClient

@EnableFeignClients

**public** **class** SbmsServiceRegistryCalculationServiceApplication {

**public** **static** **void** main(String[] args) {

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

}

}

Step 3:

Configure below properties application. properties

server.port=2222

spring.application.name=CALCULATION-SERVICE

Step 4:

Create feign client to interservice communicate for stock-price-service

@FeignClient(name = "STOCKPRICE-SERVICE")

**public** **interface** StockPriceClient {

@GetMapping("/getPrice/{name}")

**public** ResponseEntity<Double> invoiceStockService(@PathVariable("name")String companyName);

}

Step 5:

Create service interface and impl class

**public** **interface** StockCalculationService {

**public** Double getCompanyPrice(String companyName);

}

@Service

**public** **class** StockCalculationServiceImpl **implements** StockCalculationService{

@Autowired

**private** StockPriceClient stockPriceClient;

@Override

**public** Double getCompanyPrice(String companyName) {

ResponseEntity<Double> resEntity=stockPriceClient.invoiceStockService(companyName);

**if**(resEntity!=**null** && resEntity.getStatusCodeValue()==200)

{

**return** resEntity.getBody();

}

**return** **null**;

}

}

Step 6:

Create Restcontroller with required methods

@RestController

**public** **class** StockCalculatorRestController {

@Autowired

**private** StockCalculationService calculatorService;

@GetMapping("/calculate/{name}/{qty}")

**public** ResponseEntity<String> getTotalCost(@PathVariable("name")String companyName,

@PathVariable("qty")Integer qty)

{

Double companyPrice=calculatorService.getCompanyPrice(companyName);

Double totalcost=companyPrice\*qty;

String resMsg="Total cost :: "+totalcost;

**return** **new** ResponseEntity<>(resMsg,HttpStatus.***OK***);

}

}

Step 7:

Configure swager

@Configuration

@EnableSwagger2

**public** **class** SwaggerConfig {

@Bean

**public** Docket customerApi()

{

**return** **new** Docket(DocumentationType.***SWAGGER\_2***)

.select()

.apis(RequestHandlerSelectors.*basePackage*("com.app"))

.paths(PathSelectors.*any*())

.build();

}

}

Step 8:

handle exceptions

Step 9:

Run and test application v

<http://localhost:2222/swagger-ui.html>’’p

2. Stock UI Applications:

Stock Ui application is responsible to access stock calculation service to get total cost

Sock UI application contains from to enter company name and qty

Step to create UI project :

Step 1:

Create spring boot application below dependencies

1)webflux

2)embedded jasper

3)devtools

Step 2:

Configure below annotations at starter class level

@SpringBootApplication

**public** **class** SbmsStockUiAppApplication {

**public** **static** **void** main(String[] args) {

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

}

}

Step 3:

Configure below properties in application. properties file

server.port=4444

spring.mvc.view.prefix=/WEB-INF/views/

spring.mvc.view.suffix=.jsp

Step 4:

Create controller to handle form display and form submission.

@Controller

**public** **class** StockUIController {

@Autowired

**private** StockUiService service;

@GetMapping("/")

**public** String loadForm()

{

**return** "index";

}

@GetMapping("/getTotalCost")

**public** String handleGetToalCost(HttpServletRequest req,Model model)

{

String companyName=req.getParameter("companyName");

String quantity=req.getParameter("qty");

String response=service.getTotalStockPrice(companyName, Integer.*parseInt*(quantity));

model.addAttribute("msg",response);

**return** "index";

}

}

Step 5:

Create form to Display fields

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<h3>Get Stock price here ::</h3>

${msg}

<form action=*"getTotalCost"*>

<table>

<tr>

<td>Company Name</td>

<td><input type=*"text"* name=*"companName"*></td>

</tr>

<tr>

<td>Quantity</td>

<td><input type=*"text"* name=*"quantity"*></td>

</tr>

<tr>

<td><input type=*"submit"* value=*"Get Cost"*></td>

</tr>

</table>

</form>

</body>

</html>

Step 6:

Create service class to access STOCK-CALCYLATION-SERVICE using web-client.

@Service

**public** **class** StockUiService {

**public** String getTotalStockPrice(String companyName,Integer qty)

{

String REST\_ENDPOINT="http://localhost:3333/stock/calc/calculate/{name}/{qty}";

String response=**null**;

WebClient webClient=WebClient.*create*();

**try** {

response=webClient.get()

.uri(REST\_ENDPOINT, companyName,qty)

.retrieve()

.bodyToMono(String.**class**)

.block();

} **catch** (Exception e) {

e.printStackTrace();

response="Company not found";

}

**return** response;

}

}

Step 7:

Run as spring-boot application

<http://localhost:3333>

3. API Gateway: It is routing our services where to go provided and filtering(making request is process or not) using ZUUL server. (EX: Front controller in MVC)---🡪 Optional but we may use security purpose.

Most of companies may not use these 2 Component (Service registry and Api gateway) but directly they placed into servers using Restful.

1) API Gateway is used to manage all APIs available in our Applications

2) API Gateway is management tool which sits between client request and backend services.

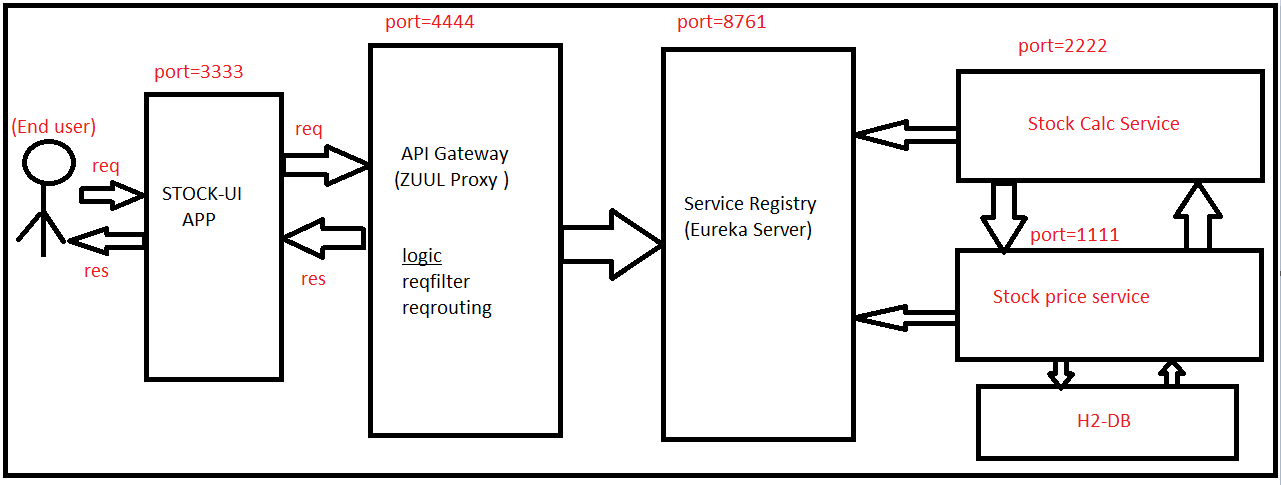
3) Request filtering logic we can do in API Gateway

4) API Gateway will act as an entry point for all our services.

5) API Gateway is also called as Edge Micro services.

Ex) ZUUL Proxy is an open Source API Gateway from Netflix

* Apigee is a commercial gateway from Google



Step to create API Gateway project :

Step 1:

Create spring boot application below dependencies

1)web

2)Eureka client

3)Zuul

Step 2:

Configure below annotations at starter class level

@SpringBootApplication

@EnableDiscoveryClient

@EnableZuulProxy

**public** **class** SbmsApiGatewayApplication {

**public** **static** **void** main(String[] args) {

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

}

}

Step 3:

Configure below properties in application. properties file

server:

port: 3333

spring:

application:

name: API-GATEWAY

zuul:

prefix: /stock

routes:

prices:

path: /price/\*\*

serviceId: STOCKPRICE-SERVICE

calc:

path: /calc/\*\*

serviceId: CALCULATION-SERVICE

# EX: below url for route CALCULATION-SERVICE Service in UI Application

# http://localhost:3333/stock/calc/calculate/{name}/{qty}

Step 4:

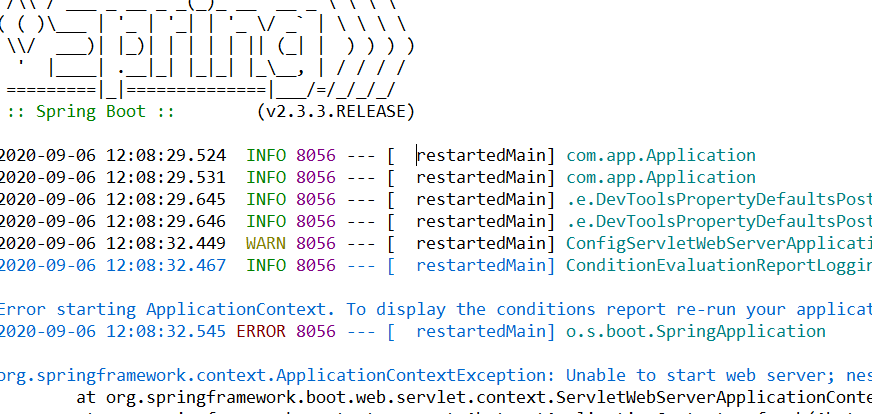
Step 5:

Ribbon :-

By Using Ribbon is used to achieve client side load balancing.

* First we need know below information then we can find Ribbon

If we run our spring boot application ,oracle port number 8080 already running so failed our application .



We are not able to run our application.

How to resolve the problem below :-

Process 1:

So we can configure property below in application. properties

server.port=1111

Process 2:

But I don’t want to change in application. properties file

Rightclick on project> Run as springboot app> run as> run configuration> (create,manage, and run configuration)> give Vm arguments

-Dserver.port=1111>Apply>run (Instance 1)

<http://localhost:1111/welcome> output will come

But here we can run many time in our spring boot application using above process

-Dserver.port=2222>Apply>run (Instance 2)

<http://localhost:2222/welcome> output will come

-Dserver.port=3333>Apply>run (Instance 3)……..

<http://localhost:1111/welcome> output will come

Above all are only one project with different runs.

Why we running multiple instances :

To reduce server burden (Distribute the load)

---🡪Here I want to see which port number is running, I want to know programmatically

Now see here

@RestController

**public** **class** WelcomeRestController {

@Autowired

**private** Environment env;

@GetMapping("/welcome")

**public** String welcomeMsg()

{

String msg="welcome to rest api";

String serverPort = env.getProperty("local.server.port");

System.***out***.println("Server Port ::"+serverPort);

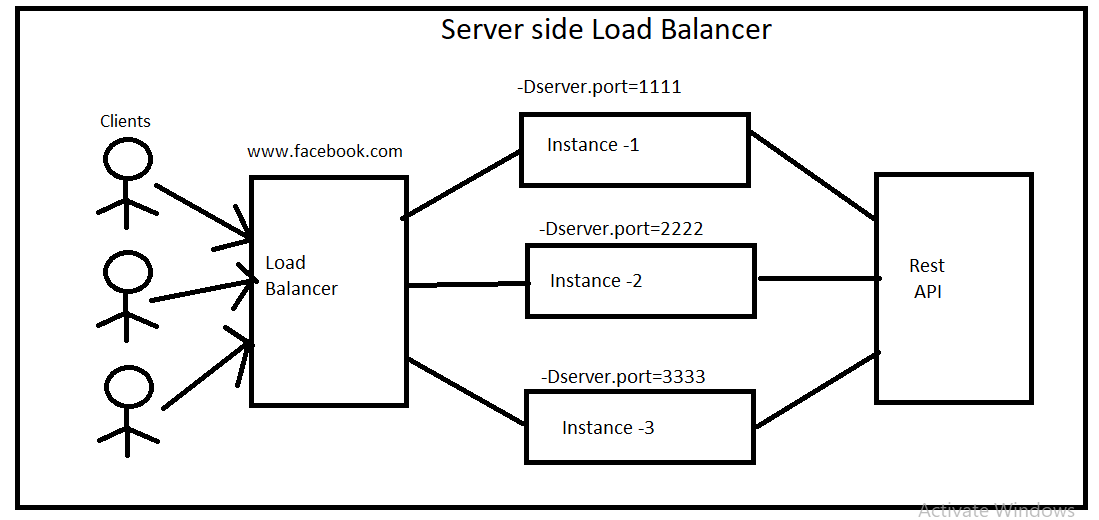
msg = msg.concat("I am from method port :: "+serverPort);

**return** msg;

}

}

What will do in the industry, we will configure one more server(Load balancer)



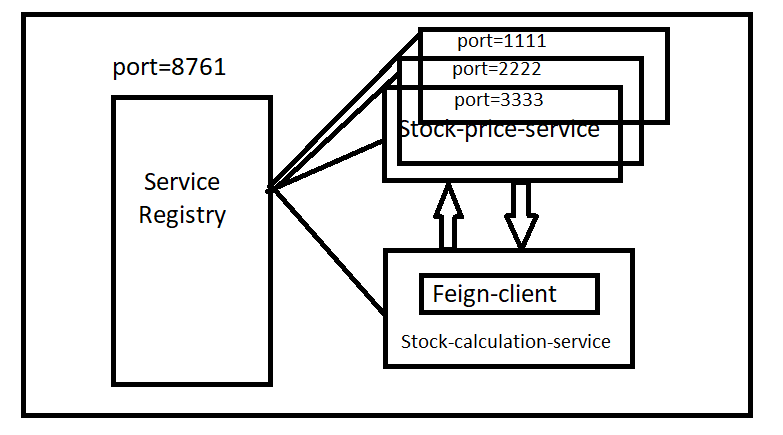
Coming to ribbon :

It will client side load balancing

It will decide which req-which Url has to send the request.

Here so many request are coming to price service from calculation service req are coming directly request are coming from the price service ,can we apply load balancer for stock price service ,yes

Stock-price-service running on 3 instances with different Url’s



Which Url feign client should access !!!!!!!!!!!!!!!

Earlier feign client will access the price service based on the service registry Url. For the feign client I am giving the name of the service which it has to access not an Url. If we give name of the servicename to the feign client then we can get the url from service registry.

But this time we have it 3 instances available with 3 different Urls, which Url feign client get access !!!.

Feign client can not decide confused .so there is a problem

Along with feign client+ Ribbon client

Here round ribbon algorithm

1st request🡪1 URL

2st request🡪2 URL

3st request🡪3 URL

Ribbon client is balancing the client side .

