**Spring Cloud**

* Rest Template
* Eureka Server
* @Loadbalanced
* Circuit Breaker
* API Gateway
* Zipkin Server
* Config Server

Rest Template: Its use for inter microservice communication.

**Step 1: Create delegate interface**

**public** **interface** StockSectorDeligate {

**public** List<Map> getAllSectors();

}

**Step 2:Create Service class to implement the interface.**

@Service

**public** **class** StockSectorDeligateImpl **implements** StockSectorDeligate {

@Autowired

@Lazy

RestTemplate restTemplate;

@Bean

@LoadBalanced

**public** RestTemplate getRestTemplate() {

**return** **new** RestTemplate();

}

@Override

@CircuitBreaker(name="SECTOR-CIRCUIT-BREAKER",fallbackMethod = "fallbackGetAllSectors")

**public** List<Map> getAllSectors() {

List sectors=**this**.restTemplate.getForObject("http://stock-sector/mymarketplace/sector", List.**class**);

**return** sectors;

}

**public** List<Map> fallbackGetAllSectors(Exception ex) {

System.***out***.println("StockSectorDeligateImpl.fallbackGetAllSectors()");

**return** **null**;

}

}

**Step 3: Call the delegate in actual service class.**

@Override

**public** List<Stocks> getAllStocks() {

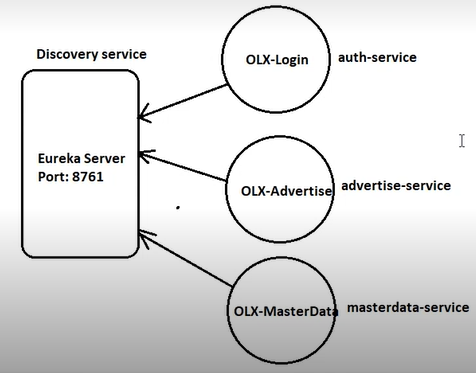
List<Stocks> stockList = stockRepo.findAll();

//for intercommunication microservice

sectorDeligate.getAllSectors();

**return** stockList; }

**Eureka Server:** It’s helps to remove the hardcoded IP Address and Port from the code by name. We have to create one Eureka server and all microservice as a client.



Steps to create Eureka server and client.

1) Create a spring boot project with Eureka Server dependency:

<dependency>

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

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

</dependency>

2) Add @EnableEurekaServer annotation to main class:

@SpringBootApplication

@EnableEurekaServer

public class TempEurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(TempEurekaServerApplication.class, args);

}

}

3) Add below properties into application.yml so that Eureka Server Application won't register itself into Eureka Server Registry.

server:

port: 8761

eureka:

client:

register-with-eureka: false

fetch-registry: false

4) Start Eureka Server using http://localhost:8761

5) In order to register microservices into Eureka Server, add below dependency in microservices spring boot app:

<dependency>

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

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

<version>3.0.3</version>

</dependency>

6) Add @EnableEurekaClient annotation to microservice main class.

@SpringBootApplication

@EnableEurekaClient

public class OlxLoginApplication {

}

7) Add below configuration in microservice application.yml:

spring.application.name=login-service

8) Now you can start microservice & make sure it is registered in Eureka Server on specified port using localhost:8761

9) If you wish Eureka to assign random port to microservice then add following properties in application.yml

server.port=0

eureka.instance.instance-id=${spring.application.name}:{random.uuid}

**@Loadbalanced:** If we assign name of server instead of IP & Port in Eureka server then @Loadbalanced helps to call the Remote call by Eureka server.

**Circuit Breaker:** It prevent unnecessary call if any service is down in remote calling. It has 3 states.

**CLOSED**: It is default state. It means everything is going fine.

**OPEN**: If service is down then it become in OPEN state. We can provide the time i.e after how many calling it will become in OPEN state.

**HALF OPEN:** If the service is in OPEN state, then after specified time HALF OPEN state will enable for specified time to check the service is now working fine or not. If service will work fine then it will move to CLOSE state otherwise it will again go to OPEN state.

**Implementation:**

**Step 1: Define the below dependencies in pom.xml file.**

<dependency>

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

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

<version>1.7.0</version>

</dependency>

<dependency>

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

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

</dependency>

**Step 2: Define the below properties in .yml file.**

management:

endpoints:

web:

exposure:

include: "\*"

health:

circuitbreakers:

enabled: true

endpoint:

health:

show-details: always

resilience4j:

circuitbreaker:

instances:

SECTOR-CIRCUIT-BREAKER: (It is the name of the circuit breaker that’s defined in our code)

registerHealthIndicator: true (It will register the circuit breaker in health indecator)

automaticTransitionFromOpenToHalfOpenEnabled: true

failureRateThreshold: 50

minimumNumberOfCalls: 5

permittedNumberOfCallsInHalfOpenState: 3

waitDurationInOpenState: 10s

**Step 3: Add circuit breaker annotation on remote call method and create a fallback method with same signature so if remote service is down then it will call the fallback method.**

@Override

@CircuitBreaker(name="SECTOR-CIRCUIT-BREAKER",fallbackMethod = "fallbackGetAllSectors")

**public** List<Map> getAllSectors() {

List sectors=**this**.restTemplate.getForObject("http://stock-sector/mymarketplace/sector", List.**class**);

**return** sectors;

}

**public** List<Map> fallbackGetAllSectors(Exception ex) {

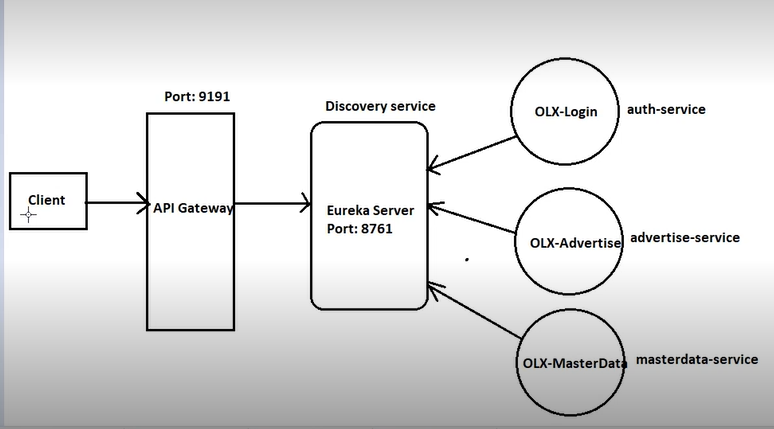
System.***out***.println("StockSectorDeligateImpl.fallbackGetAllSectors()");

**return** **null**;

}

NOTE: fallback method signature should be same as remote call method and need to add Exception as a last parameter in this method.

**API Gateway:** An API gateway is **programming that sits in front of an API and is the single-entry point for defined back-end APIs and microservices** (which can be both internal and external). For example: **If Android side need the base URL then we cann’t provide the URL with different IP and Port number. So we need to create a single entry point URL.**



If you not use the API gateway then for every call you need the (IP and Port for external client like postman or from application UI) or (Eureka client name for internal microservice communication).

Like: [**http://localhost:5000/mymarketplace/stock**](http://localhost:5000/mymarketplace/stock) **or** <http://localhost:4000/mymarketplace/stector> for external client request like postman.

Here, we have two port numbers for different port number to call the microservice. So we can replace it with single IP & Port.

<http://localhost:9191/mymarketplace/stock>

Here, 9191 is a port number of API gateway and it will work for both URL.

For internal microservice communication use like below URL.

[http://**API-GATEWAY**/mymarketplace/sector](http://API-GATEWAY/mymarketplace/sector)

Implementation:

**Step 1:** Create new Application like Eureka and Add “gateway” and “Eureka client” dependency.

**Step 2:** In application class add @EnableEurekaClient to register it with Eureka server as a client.

**Step 3:** Add below properties in .yml file.

server:

port: 9191

spring:

application:

name: API-GATEWAY

cloud:

gateway:

routes:

- id: STOCK-SECTOR

uri: lb://STOCK-SECTOR

predicates: Path=/mymarketplace/sector/\*\*

- id: STOCKS

uri: lb://STOCKS

predicates: Path=/mymarketplace/stock/\*\*

Here you need to add all routs and id should be match with register service name in Eureka server. “lb” indicates load balancer.

**Step 4**: Start Eureka server-> API Gateway-> All microservices and hit the URL with port number 9191.

**Zipkin Server:** We can find the execution time of each microservice using it.

Implementation:

**Step 1:** Download the Zipkin server from (<https://zipkin.io/pages/quickstart.html> -> latest release) and start it using below command of cmd.

**java -jar zipkin-server-2.23.16-exec.jar**

**Step 2:** Add the below two dependencies in pom.xml file.

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

Here, sleuth is API of Zipkin server to monitor the execution time.

**Step 3:** Add the Zipkin base Url in .yml file.

spring:

zipkin:

base-url: <http://localhost:9411/>

**Step 4:** Use <http://localhost:9411/> in google chrome to see the microservices taken time.

**Config Server:** It use to keep the properties that can be change in future like data base credentials, zepkin base URL of .yml file. So user no need to deploy the project again and again.

Implementation:

**Step 1**: First create a empty project on github.

EX: stock-config

**Step 2**: clone the project in system.

C:\Users\Test\Desktop\java spring files\SchoolProject>git clone <https://github.com/notoriouscode/stock-config>

Then go to under stock-config

C:\Users\Test\Desktop\java spring files\SchoolProject\stock-config>

**Step 3:** Create a .yml file in newly created github project in STS with same name that’s we used as a application name in StockUserLogin Application (login-service). So new file name is login-service.yml. we have to create follow same approach for every microservices.

**Step 4:** Put all yml file in stock-config directory and add these in local repository

C:\Users\Test\Desktop\java spring files\SchoolProject\stock-config>git add .

Then commit it

C:\Users\Test\Desktop\java spring files\SchoolProject\stock-config>git commit -m "commit message"

Then Push it

C:\Users\Test\Desktop\java spring files\SchoolProject\stock-config>git push

**Step 5:** Create a new project with spring web and config server dependencies. And add the properties in .yml file.

server:

port: 8888

spring:

application:

name: config-server

cloud:

config:

server:

git:

uri: <https://github.com/notoriouscode/stock-config> //config server url

default-label: main // main means main branch

username: notoriouscode

password: github password

clone-on-start: true //At the boot time of the config server to get the entire github repo in the memory.

**Step 6:** Add the below annotation in application main class and run the application.

@EnableConfigServer

**Step 7:** Hit the below URL to confirm that github yml file is loaded in config server application or not.

[http://localhost:8888/**login-service**/default](http://localhost:8888/login-service/default)

//login-service is yml file name on github and default refer to root yml file of the directory. Suppose I want to create different yml file for development/testing/production environment then we will create different folder for it and use the folder name instead for default.

Ex: [http://localhost:8888/**login-service**/dev](http://localhost:8888/login-service/dev)

**Step 8:** In login microservice add the **config client** and **actuator** dependencies.

**Step 9:** Add below properties in .yml file of login microservice and run the service.

management:

endpoints:

web:

exposure:

include: "\*"

Spring:

config:

import: configserver:http://localhost:8888

**Step 10:** To verify this create a controller class and run it.

@RestController

@RefreshScope //it will get latest value of datasource every time.

**public** **class** ConfigServerVerificationController {

@Value("${spring.datasource.url}") // get the value of this datasource.

**private** String dbUrl;

@GetMapping(value="/read-config")

**public** String readConfig() {

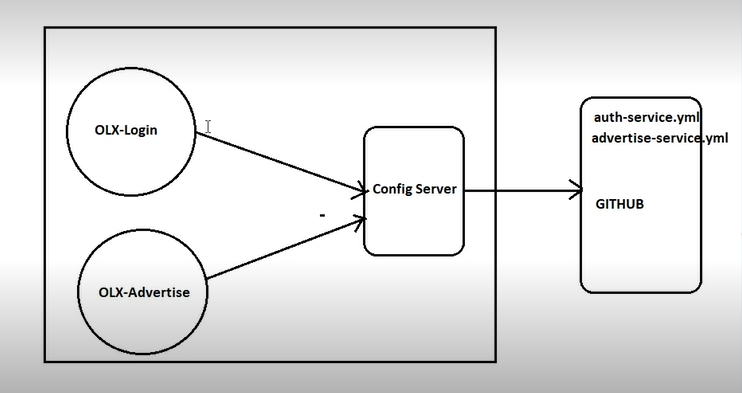
**return** "DB URL = "+dbUrl;

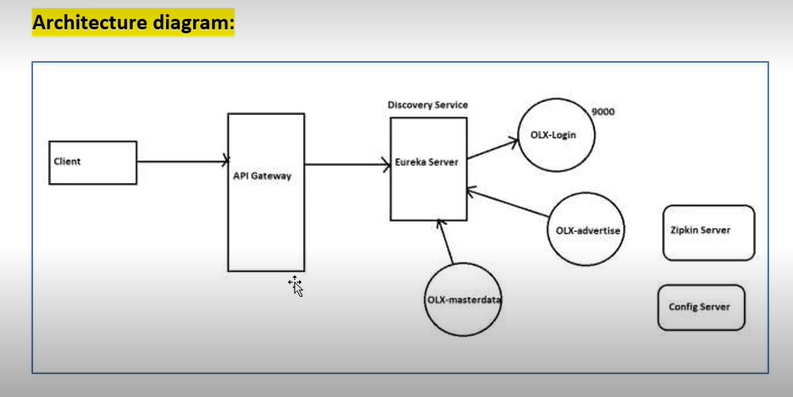
}

}

NOTE: Config server always keep the eyes on github repo and get the latest changes but microservice not get the latest changes automatically. So we will call the actuator/refresh to get the latest changes.

http://localhost:8001/actuator/refresh call this URL form post man with POST method. And for production we can also write the code that will call after 20 or 30 minutes to get the latest changes.



**s**