Web Services:

Types:

1. SOAP
2. REST

Designed for application to application interaction.

Allow communication over network.

Data Exchange between applications:

Request & Response (platform independent)

1. XML
2. JSON

Service Definition:

Request/Response format

Request Structure

Response Structure

EndPoint

SOAP web services:

Simple Object Access Protocol

We use XML as req/res

Service definition done using WSDL(web service definition language)

* Endpoint
* All operations
* Request structure
* Response structure

REST:

Representational state transfer.

Use of HTTP(methods POST,GET…) status codes (200,404…)

Java Brains:

Microservices Communication:

1. Synchronous Communication

* Client sends request and waits for a response from the service.
* Rest template, Webclient and Spring cloud open feign library.

1. Asynchronous Communication

* Client sends a request and does not wait for the response from the service.
* Apache kafka, RabbitMQ

RestTemplate:

Create Bean for it.

Use it in service like restTemplate.getForEntity(“url”,responseType);

WebClient:

Supports sync,async

Add Spring WebFlux dependency

<dependency>

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

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

<version>3.2.2</version>

</dependency>

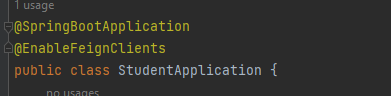
Configure Spring Bean for it.

Feign Client:

Add Spring cloud open feign maven dependency.

Enable feign client using @EnableFeignClients

Create Feign API Client.



A screen shot of a computer code

Description automatically generated



Take reference from Address microservice:

A screen shot of a computer code

Description automatically generated

Service Registry & Discovery:

To call a microservice we need its hostname as well as port number.

We may run multiple instance of a microservice and sometimes instances go down. So we need some automatic service registration and discovery mechanism. We can solve this by providing Spring Cloud Netflix Eureka.

Instead of hardcoding them we can register each microservice in service registry.

Development Steps:

1. Add @EnableEurekaServer annotation.
2. Disable eureka server as eureka client.
3. In client

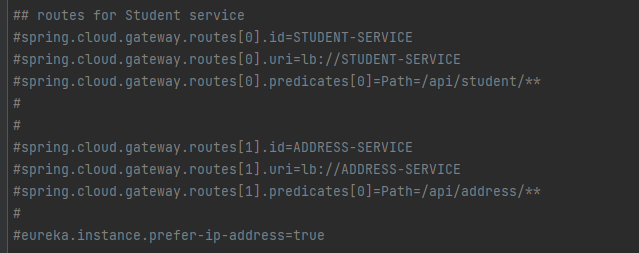
* Register it in eureka server as eureka client
* By default Load Balancing will happen
* To do this replace url with application name on FeignClient(name=”Application-Name”)

Api Gateway:

Route Request

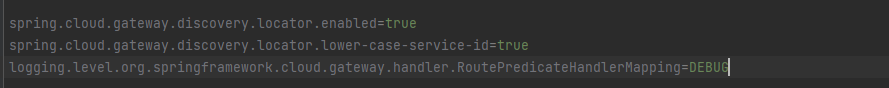
1. By using properties

We will create routes manually and no need to add application name in url



We can use spring cloud provided locator to create routes automatically but you need to mention application name in url

http://localhost:9191/student-service/api/student



1. Using java class

Load Balancing

spring.cloud.gateway.routes[1].uri=lb://ADDRESS-SERVICE

Security

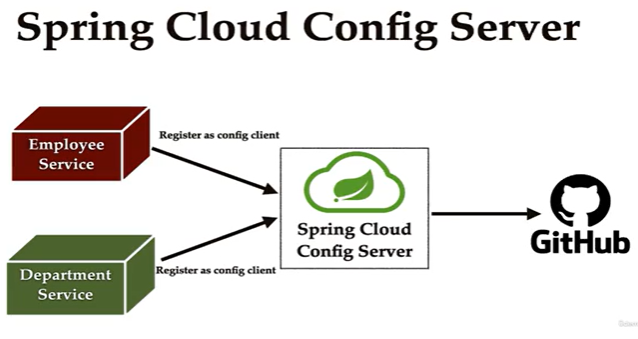
It provides a unified interface for a set of microservices so that clients no need to know about all the details of microservices internals

Centralize cross-cutting concerns

Development Steps:

1. Create springboot project as microservice
2. Register API-Gateway as Eureka Client to Eureka Server(service Registry)
3. Configuring API Gateway Routes and Test using Postman Client

Spring Cloud Config Server:



If you have config file when you made any changes then you need to restart the application along with the instances which is not good.

We can store all the microservices configurations in a centralized repository(eg: git)

Development Steps:

* Create Springboot project as microservice (config-server)
* Register Config-Server as Eureka Client
* Set up git location for Config Server.
* Refactor other services to use Config Server.

Whenever we change the configuration file then we don’t have to restart the microservice and it’s instances.

We need to call spring boot actuator/refresh API to reload the updated values from config server

Add @RefreshScope on top of controller (Actuator dependency need to be there)

In application.properties(in app not in git)

management.endpoints.web.exposure.include=\*

change anything in git and hit below url with POST

http://localhost:8080/actuator/refresh

Problem using Spring cloud config server:

In order to reload the config changes in config client applications ( Student service,Address Service) we need to trigger/refresh end point manually. This is not practical and viable if you have large number of applications.

Spring Cloud Bus module provides a solution.

Spring cloud bus module can be used to link multiple applications with a message broker and we can broadcast configuration changes

Distributed Tracing with Spring cloud sleuth and Zipkin:

1. We use Spring cloud sleuth for distributed tracing.
2. We use zipkin to visualize trace information through UI

Client 🡪 API Gateway 🡨🡪 Student Service 🡨🡪 Address Service

------Span id🡪 ---------Span id--🡪 -------Span id🡪

-------------------- Trace->trace id --------------------------------🡪

Trace id 🡪 common across all microservices

Span id 🡪 unique id for each microservice

Circuit Breaker problem:

In microservice architecture, one microservice will depend on the other microservices.

If one microservice is down or slow then the other microservices will also impact because of that microservice.

1. Fallback method:

Whenever one microservice is down, then default response will be returned from the microservices which are dependent on this.

1. Circuit breaker patter:

If A microservice is down then it won’t allow microservice B to hit request to A continuously.

1. Retry:

Sending request n times when one microservice is down

1. Ratelimit:

Limit the request calls to microservice which is down.

Circuit Breaker Pattern:

1. Closed

If microservices are up, then they will in closed state.

1. Open

If it reached a threshold limit, then microservice will go to Open state where it will be idle till time limit.

1. Half-Open

0nce the time limit is done then service will be in half-open state. From here based on n-limted-requests it’ll go to closed state if all requests are successful else it goes again to open state.

With these we can limit no. of requests to microservice which is down or slow.

Using Resilience4j:

Steps:

1. Add dependencies

Resilience4j, aop,actuator

1. Using @CircuitBreaker annotation to a method(which is calling to external service)
2. Fallback Method implementation
3. Add CircuitBreaker configuration in application.properties