**Microservices**

**What is a microservice?**

Microservice is a small and focused that is also autonomous, that is it can be built and deployed on its own without impacting other services. Each microservices communicates with each other with API network calls.

**What are advantages using microservices ?**

1. **Heterogenous** : Each of our microservice can be written in a different programming language and they can run on different platform or operating system and they communicate with APIs they expose.
2. **Robustness** : When one microservice is down its wont effect the whole application.
3. **Scalability** : If huge load is coming for one or two microservice we need to just deploy an another for that two services only, but in case of monolith application whole application must be deployed.
4. **Reusability and Replaceable** : One microservice can be used by another microservices and if we want to replace an microservice with third party vendor it will be easy.

**Centralized configuration:**

Create a new project spring cloud config server by adding ***spring-cloud-config-server*** dependency and add **@EnableConfigServer** annotation on the main class. Also add below properties in application.properties file.

*spring.application.name=spring-cloud-config-server*

*server.port=8888*

spring.cloud.config.server.git.uri=https://github.com/vishalkumar392392/configserver-microservices2

spring.cloud.config.server.git.username=vishalkumar392392

spring.cloud.config.server.git.password=accesstoken

spring.cloud.config.server.git.clone-on-start=true

spring.cloud.config.server.git.default-label=main

*spring.cloud.config.server.git.uri=file:///C:/Users/vpalla/Desktop/vishal/Microservices/git-localconfig-repo*

file. If another service say ***limits-service*** wants to consume properties(limit-service.minimum=10,limit-service.maximum=1000) from github using ***spring-cloud-config-server***, we need to add dependency ***spring-cloud-starter-config.*** . Also add below properties in application.properties file.

*spring.application.name=limits-service*

*spring.config.import=optional:configserver:http://localhost:8888*

In our github repo we have application.properties file which will serve as default config file to all microservices which are registered with config-server. But if we want a specific microservice should only able to config some specific properties then create a bootstrap.properties file and add below two lines.

spring.cloud.config.uri=http://localhost:8012

spring.cloud.config.name=user.properties //-> user -> application.name of a service

**Spring cloud bus :**

Spring cloud bus helps us to push our configuration changes to our microservices while they are running. And microservices to use updated configuration properties they don’t need to be restarted. So using spring cloud bus we can change the configuration properties in remote git repository and push updates to our microservices without restarting the microservices. The protocol used is AMQP, advanced message queuing protocol.

Add dependency spring cloud starter bus amqp, spring boot starter actuator in config server application. In other applications add only spring cloud starter bus amqp.

To install the RabbitMQ which acts as a message broker execute below command in terminal.

docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.9-management

In browser use localhost:15672 to see RabbitMQ dashboard.

In application.properties file of config-sever add below lines.

management.endpoints.web.exposure.include=busrefresh

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

In application.properties file of other microservices add below lines.

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

**Spring-cloud-starter-openfeign:**

We use spring cloud feign in our services to consume other microservices rest endpoints. We need to add spring-cloud-starter-openfeign as dependency and add @EnableFeignClients annotation in main class and create a proxy interface and see below snapshots from Proxy and Controller.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Eureka Naming Sever or Service Registry**

To make any project or service as naming server we must add Eureka server dependency which is ***spring-cloud-starter-netflix-eureka-server***. Also add @EnableEurekaServer annotation in the main class. Give a good name for server spring.application.name=naming-server. We should add some config in application.properties so eureka server won’t register itself in registry.

*eureka.client.register-with-eureka=false*

*eureka.client.fetch-registry=false*

*server.port = 8761*

Now we go to browser and hit ***localhost:8761*** then we can see eureka server dashboard. To register other services in eureka open other microservices and dependency ***spring-cloud-starter-netflix-eureka-client***. And add below line in application.properties.

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

**Note** : ***Feign asks eureka server for instances of currency-exchange service and load balance between them. When we open dependency hierarchy, we can see spring-cloud-starter-loadbalancer in eureka naming server dependency. And feign uses this loadbalancer framework to distribute the load among the multiple instances which are returned by eureka. In the earlier versions of Spring Cloud, the load balancer which was used was Ribbon and in the recent versions, Spring Cloud shifted to using Spring Cloud Load Balancer as the load balancer. The great thing is, if you're using Eureka and Feign, then load balancing comes for free. This is client-side load balancing, and this comes for free for you.***

**API Gateway**

In previous versions we used Zuel API as gateway but now in latest we are using spring-cloud-API-gateway. We need to add dependency ***spring-cloud-starter-gateway*** and eureka client dependency in a new project to make it as a gateway.

#spring.application.name=api-gateway

#server.port=8765

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

And in application.properties file add below line so that it will talk eureka server to know the registered microservices. So, we can call currency exchange service with api gateway server with below URL and currency-exchange is the name of currency-exchange service in eureka so we can use it **http://localhost:8765/currency-exchange**/appendcorrectURL.

**http://localhost:8765/currency-exchange**/currency-exchange/from/{from}/to/{to}

#spring.cloud.gateway.discovery.locator.enabled=true

#spring.cloud.gateway.discovery.locator.lower-case-service-id=true

But URLS looks overwritten so we need to eliminate extra currency-exchange in the path. The way we can do that is configuring routes. Whenever we get a request from “/currency-exchange/\*\*” we can ask eureka to find that service and that service handle the request and load balance it. “lb://currency-exchange”

Graphical user interface, text, application

Description automatically generated

**Spring cloud Gateway**

* Simple, yet effective way to route to APIs.
* Provide cross cutting concerns:

1. Security
2. Monitoring/metrices

* Build on top of Spring web flux (Reactive Approach)
* Features :

1. Match routes on any request attributes.
2. Define Predicates and filters.
3. Integrates with Spring cloud Discovery client (Load Balancing).

**Circuit Breaker** :

Graphical user interface, text, application

Description automatically generated

To start we need to add spring-boot-starter-aop, actuator and resilience4j-spring-boot2 in the currency-exchange service.

**@Retry** : We use this annotation on a method if the method throws an error, we are saying to method please retry 3 more times and after give the response. We can also set fallBackMethod response in case after 3 times also if method throws error. We can also configure the no. of retries.

**@CircuitBreaker** : If a microservice or an API is down or failing for a period time, then circuit breaker says hey the API is failing for a certain period time now I wont call the API anymore, I will return default fall back method response.

**@RateLimiter** : We can configure like this, in 10 seconds(limit-refresh-period) allow only 2(limit-for-period) requests. We can do that in application.properties.

**@Bulkhead** : We can configure how may concurrent calls can be allowed trough application.properties.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Running users service application in multiple ports** :

Graphical user interface, text, application

Description automatically generated

When we start the application through STS IDE, the application will run in different random port numbers. But only one instance will register on eureka, to avoid that we can configure properties file like above. If you want to configure the port number and eureka instance id we can use below command.

***mvn spring-boot:run -Dspring-boot.run.arguments=”--spring.application.instance\_id=vishal --server.port=8999”***

**Sleuth and Zipkin**

Sleuth and Zipkin is used to trace communication between microservices. Sleuth adds traceId and spanID to every request happening between microservices. If two microservices are communicating with each other then TraceId will be same for the two microservices in a request but spanid will differ. If we need we can send this logs to zipkin, so we can see the path of communication between microservices in a good UI interface.

Download the docker image of zipkin using below docker command

**docker run -d -p 9411:9411 openzipkin/zipkin**

Add spring-cloud-starter-sleuth and spring-cloud-starter-zipkin dependencies in pom.xml and also below lines to application.properties.

spring.zipkin.base-url=http://localhost:9411

spring.zipkin.sender.type=web

spring.sleuth.sampler.probability=0.1

**ELK (Elastic search, Logstash, Kibana)**

To collect the log files in one place we use logstash. ***Logstash*** is a tool used to collect logs from microservices which can filter, transform and it will pass to the next step in which it needs to be indexed so that they can be searchable and as a search engine we use ElasticSearch. ElasticSearch is a tool which will index and store the log files data centrally so that we can search this log file records and analyzie them. As visual interface we use Kibana. Run rabbitmq sever as well.

We have to download logstash, elsatic search, kibana from elastic.co . For logstash to to know about the logfile of our microservices write logging.file.name=users.log or anything. In logstash folder create a file with name simple-config.conf and paste below code.

input {

file {

type=>"users-log"

path=>"C:/Users/vpalla/Desktop/vishal/Microservice-course2/user-service/users.log"

}

file {

type=>"albums-log"

path=>"C:/Users/vpalla/Desktop/vishal/Microservice-course2/album-service/albums.log"

}

}

output {

if [type] == "users-log" {

elasticsearch {

hosts => ["localhost:9200"]

index => "users-%{+YYYY.MM.dd}"

}

} else if [type] == "albums-log" {

elasticsearch {

hosts => ["localhost:9200"]

index => "albums-%{+YYYY.MM.dd}"

}

}

stdout { codec => rubydebug}

}