**Author: Subhjeet Singh**

🡨 Spring Micro Services →

**Help: https://github.com/LearnCodeWithDurgesh/Microservices-Tutorial-Series**

Microservices is an architectural style of developing an application as a suite of small services, each capable of independently deployable. Here we will learn:

* Split a monolithic application into microservices.
* Store configuration details on cloud using CloudConfig
* Load balance requests between microservices using Ribbon
* Discover services in cloud using Eureka
* Increase resilience through Hystrix
* Use asynchronous communication to improve performance
* Create an API gateway using Zuul
* Simplify REST calls through Feign
* Secure microservices using OAuth through Spring Cloud Security
* Monitor your microservices through Turbine, Sleuth and Zipkin

**Similarity b/w SOA(Presentation, Db, Service, implementation) and Micro-services: 🡪**

Both of them are intended to address issues of monolithic architecture

Both focus on splitting the application into smaller individual components

Both use service end points for communication

Though they are similar, SOA failed whereas Microservices are trending. This is due to two reasons:

SOA has been there for a decade and a lot of lessons were learnt on the mistakes arising from it

The tools needed for SOA to succeed are available only now. Hence one can say that Microservice is SOA done right.

**Differences**:

* **Communication**: Microservices use simpler communication protocols like REST, AMQP (Advanced Message Queuing Protocol). SOA used SOAP based comm. Protocol.
* **Security**: SOAP has very complex security standards. Also, XML can contain even executable data thus increasing security risks. Microservices now use advanced resilience concepts like circuit breaker, fallback, OAuth2 integration etc.
* **Complexity**: SOA had a more complex service orchestration as the services have strong contractual agreements. SOA used ESB (Enterprise Service Bus) as the central command mechanism. Microservices have a simpler contract agreement.
* **Implementation:** SOA implementation differed from each other and many were very vendor dependent
* **DevOps:** In SOA, one had control over the development aspects. One was still dependent on other teams for operations. Microservice is best suited for DevOps as each team not only develops and maintains, but also deploys. Thus one team is responsible for entire life cycle of the microservice.

**Monolithic has its own advantages ( as long as it is reasonable in size )\*, such as:**

• Easier to get the bigger picture of the application, as the entire code base is in one place

• You have to deploy only one jar/war file

• Relatively simple and straightforward to develop

• The dependency on network latency and security is greatly reduced

**Problems in Mono. App →**

The monolithic version of our application, though working flawlessly, has issues. For example,

* If we have to scale up the add friend functionality what should we do?
* We will have to scale the entire application though only add friend functionality is facing a peak request rate. This leads to loss of resources
* If the get plan functionality can be better implemented in python, what should we do?
* This is difficult as we are stuck with one code base written in Java. If we need to update the logic of call details, what should we do?
* We end up stopping the entire application when we deploy the newer version. Thus even unrelated functionalities get affected
* If we make a modification to Customer functionality, what is the impact? We have to do a regression testing of the whole application
* Microservices are by nature distributed applications. One immediate impact of microservices is whether the database should be shared or distributed?
* Having a single shared DB is dangerous. If the DB fails the entire application fails. Hence the best practice is to not use a shared DB

**The right approach in dealing with databases in a microservice architecture is:**

* Each microservice should have its own set of tables in a separate database or schema
* One microservice should not try to directly access tables owned by another microservice. If there is a data dependency then a microservice should call another service through REST api to fetch data.
* Foreign key constraints should not be there. We can still have columns to establish link between entities.
* We should not use joins between tables owned by two different microservices. If you need data from both tables make a database call to fetch data owned by one service and make REST call to fetch data from the other. This pattern is called API Composition.
* Remove foreign key constraints from all the tables
* All microservices should have only entity classes related to their own functionality alone. Therefore remove other entity classes from the customer microservice.
* Since there is no foreign key constraints and no other entity classes in the microservices, add end points in the microservices which will provide relevant data.
* View Profile end point in customer microservice currently fetches data from Friend and Family and Plan table directly. Now it needs to get these data from other microservices.
* In Friend and Family microservice, create a new end point for fetching friend and family details for a customer
* In Plan Microservice, create a new end point for fetching plan details for a customer
* Modify View Profile end point in Customer Microservice to invoke these two new end points for fetch relevant details

**NOTE 🡪**

We can see that even though we have avoided joins in the table level, we have ended up increasing the REST end points. Because of this the com-plexity in the application level has increased. Also, the number of network calls needed also has increased. Initially the profile request involved one call to the customer REST endpoint and one call to the DB. Now it has become 3 calls to DB and 3 calls to endpoints in different microservices. Also, joining at API level is slower than table level join.

But despite these issues, we use this approach as the benefit of loose coupling and independence of microservices far outweigh these cons.

**Topics Which we Cover Here🡪**

* **Cloud**
* **Load Balancing (Ribbon, Rest Template)**
* **Service Discovery (Eureka[Discovery Client])**
* **Resilience (Circuit Breaker, Fallback, Hystrix)**
* **Api Gateway(Zuul)**
* **Declarative Client(Feign)**
* **Tracing(Slueth, Zipkin)**
* **Security**

**------------ Cloud (Config-Server, bootstrap.properties):--------------**

Cloud-enabled applications are not cloud-native applications. Cloud native applications are specifically written in such a way that it takes advantage of the cloud. For example,

* Will your application take advantage of multiple instances of related services and load balance across them?
* In the cloud, a lot of things can go wrong. If a service is suddenly unavailable, is your application resilient enough?
* In the cloud, instances can be increased, decreased, moved around and so on. Will your application automatically adapt to such changes?

If yes, then the application is cloud-native as it was designed to work on cloud advantages.

We will be using Spring Cloud in this course. Spring cloud uses many open source components from Netfilx OSS. Netflix OSS is the open- source project for a variety of solutions including build, deployment, data analytics, etc.

**Challenges of properties in application.properties file:**

All these are placed inside the application itself. The problem with this approach is that if we have to modify any configuration then we have to change these configurations in multiple services and redeploy all these services. If a service has multiple instances then all the instances have to be redeployed.

**Ways of placing our configuration details?**

* Placing the configuration as environment variables - but there is a limit on creation of environment variables
* Placing them in external files - but access to such file system is difficult in cloud

An ideal solution would be to use an external version control system like GIT, as it not only avoids the above-mentioned problems but also gives us traceability of changes.

How to use cloud:--

Steps

1)Create git repository and place one application.properties file there. Place all the common configurations there. Like

spring.datasource.driverClassName=com.mysql.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=update

ii) Create separate properties file for each microservice inside git with name matching with spring.application.name = CustomerMS of the service (CustomerMS.properties). The properties which are not common we can place in respective prop. files in git.

iii) Create a spring starter project for config server with relevant dependencies. This is the mediator between Different micro services and repository properties file.

**CustomerMicroservice--call--->ConfigServerMicroservice----fetch-->Properties from Git**

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency>

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

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

iv) Add information about the Git server in the configserver(micro service) properties file.

server.port=1111

spring.application.name=ConfigServer

spring.cloud.config.server.git.uri = http://infygit/Deepak\_M05/Infytel-config or http://github.com/PlayGroundConfiguration/SpringMicroservices.git

v) Add @EnableConfigServer annotation in the application file of config server(Bootstrap or Main class).

vi) Add relevant dependencies to all the microservices

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency>

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

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

</dependency>

vii) BootstrapContext get loaded before ApplicationContext. Create a bootstrap.properties file in each microservice (other than config-server) with a property for the config server

spring.application.name=CustomerMS

spring.cloud.config.uri=http://localhost:1111

server.port=8001

viii) Remove all properties except spring.application.name from the individual services.

Note-Now the different microservices read common properties from application.properties file in GIT and individual properties from relative spring.application.name.properties file in GIT. Each microservice have two application.properties file one in git and another locally.precedence will be given to git application.properties file. This happen because of bootstrap context.Because it got picked-up first. This context is responsible for loading configuration details from an external source.

When we use cloud config, this property source takes a higher priority. That means any duplicate properties in other property sources are ignored.

The properties file used in the GIT must have the same name as that of the client's spring.application.name

We can have multiple profiles(properties file) as well and if a profile is not mentioned it will load the default profile, which is the same name as that of the spring.application.name.

Profile here Means a MicroService.

If we try to access a microservice property for a given profile, the order of files used will be:

* Yaml file for that profile
* Properties file for that profile
* application.yml file
* application.properties file

For example, if we have a property called x=10 in application.yml and x=20 in application.properties, the final value used will be 10.

Note-The config server is contacted during start of the project. After this any change in config server not reflected.

→If a port is not specified for the config server, it runs in its default port 8888. Also, if the cloud-config server is down, then the client will throw an error not during startup, but while trying to access a property at runtime. To avoid this we can have the failFast property set to true. By this the client will fail at startup time rather than at run time.

**Spring.cloud.config.failfast = true**

Also, in order to avoid config-server to be a single point of failure, we usually deploy multiple instances of it to ensure high availability. If the cloud config server is unavailable, it will use the properties files in the individual applications as a fallback.

**-----:Dynamically config. Changes in GIT:------**

When we make any changes to the properties file in GIT, the config-clients automatically do not update themselves with the modified values. This is because the configurations are taken only once at the time of startup. To over- come this we need to:

* Add @RefreshScope annotation on the bean which is using the property. Suppose in controller layer property is required, then add @RefreshScope there.
* Add spring-boot-actuator end point dependency

<dependency>

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

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

</dependency>

* Disable security to the refresh end point by adding the below property in the relevant microservices:

management.security.enabled=false

We should not disable the management security in deployment. We will look at security of cloud config later in the course.

* Send a POST request to the /refresh end point of the service. This refreshes the microservice without restarting/redepolying it.

In this case we would add @RefreshScope on the CustomerController, add actuator dependency on the CustomerMS and send a POST request to http://localhost:8001/refresh which is where the CustomerMS is running.

====================================================================

***CHATGPT HELP:***

1. Create a Spring Cloud Config Server:

1.1. Add Dependencies:

In your Spring Boot project, add the following dependencies to your pom.xml:

<dependency>

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

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

1.2. Enable Config Server:

Add @EnableConfigServer annotation to your main application class.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication

@EnableConfigServer

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

1.3. Configure application.properties for Config Server:

In src/main/resources/application.properties, specify the Git repository where configuration files will be stored.

spring.cloud.config.server.git.uri=https://github.com/your-username/config-repository.git

2. Create a Spring Boot Microservice:

2.1. Add Dependencies:

In your microservice project, add the following dependency:

<dependency>

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

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

</dependency>

2.2. Configure bootstrap.properties:

Create a bootstrap.properties file in src/main/resources of your microservice.

spring.application.name=my-microservice

spring.cloud.config.uri=http://localhost:8888

The spring.application.name property specifies the application name, and

spring.cloud.config.uri points to the Config Server.

2.3. Access Configured Properties:

In your microservice code, you can now use @Value annotation to inject properties from the Config Server.

import org.springframework.beans.factory.annotation.Value;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/api")

public class MyController {

@Value("${example.property}")

private String exampleProperty;

@GetMapping("/property")

public String getProperty() {

return "Value from Config Server: " + exampleProperty;

}

}

3. Set Up a Git Repository for Config Files:

3.1. Create a Git Repository:

Create a Git repository to store your configuration files. Ensure it contains a file named my-microservice.properties with the required configurations.

example.property=value-from-config-server

3.2. Configure Config Server to Use Git:

Ensure that the spring.cloud.config.server.git.uri in your Config Server application.properties points to this Git repository.

4. Run Your Applications:

4.1. Start Config Server:

Run the Config Server application.

4.2. Start Microservice:

Run your microservice application. It will fetch configurations from the Config Server.

4.3. Access Microservice Endpoint:

Access the microservice endpoint (e.g., http://localhost:8080/api/property) to see the configured property value.

This tutorial outlines the basic steps for setting up Spring Cloud Config for a microservice using Git as a configuration source. Customize the configuration files and Git repository structure according to your project needs. Additionally, refer to the official documentation for more advanced configurations and options: Spring Cloud Config Documentation.

**Here's how you can use @RefreshScope:**

Add Dependencies:

Ensure you have the necessary dependencies for Spring Cloud Config in your project. Add the following to your pom.xml:

<dependency>

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

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

</dependency>

Enable Refresh in application.properties or application.yml:

In your bootstrap.properties or bootstrap.yml (or application.properties/application.yml for Spring Boot 2.4+), enable the refresh feature:

spring.cloud.config.refresh=true

Add @RefreshScope to Beans:

In your Spring components (typically @Service, @Component, or @Configuration classes), add the @RefreshScope annotation to indicate that the bean's state should be refreshed:

import org.springframework.cloud.context.config.annotation.RefreshScope;

import org.springframework.stereotype.Service;

@Service

@RefreshScope

public class MyService {

// Your bean logic here

}

Trigger Refresh:

To trigger a refresh of the beans annotated with @RefreshScope, you need to expose an endpoint that can be called to initiate the refresh process.

import org.springframework.cloud.context.refresh.ContextRefresher;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class RefreshController {

private final ContextRefresher contextRefresher;

public RefreshController(ContextRefresher contextRefresher) {

this.contextRefresher = contextRefresher;

}

@PostMapping("/refresh")

public void refresh() {

contextRefresher.refresh();

}

}

By default, the refresh endpoint is /actuator/refresh. You can customize this path in your application properties if needed.

Invoke Refresh Endpoint:

To refresh the beans annotated with @RefreshScope, you can make a POST request to the refresh endpoint. For example, using curl:

curl -X POST http://localhost:8080/actuator/refresh

Alternatively, you can use tools like Postman or write a script to trigger the refresh.

When the refresh endpoint is invoked, beans annotated with @RefreshScope will be reinitialized, and their state will reflect the updated configuration values.

Keep in mind that not all beans need to be annotated with @RefreshScope; only those that should be reinitialized with refreshed configuration. Additionally, be cautious about the impact of refreshing beans on the state of your application, especially in a production environment.

===========================================================================

**PRACTICE:**

Steps:

Related to Config-Server:

1. Dependencies:

<dependency>

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

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

1. Bootstrap Class:

package com.subh.config.server;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication

@EnableDiscoveryClient

@EnableConfigServer

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

1. Properties:

spring:

cloud:

config:

server:

git:

uri: https://github.com/subhjeetsingh/microservices-configurations

clone-on-start: true

Related to Config-Client:

1. Dependency:

<dependency>

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

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

</dependency>

1. Property:

spring:

config:

import: configserver:http://localhost:8196

CONFIG-SERVER LOCAL URL to see PROPERTIES:

**<http://192.168.43.157:8196/application/dev> :** This will display default and dev related properties.

**<http://192.168.43.157:8196/application/prod> :** This will display default and prod related properties.

**----------: LOAD BALANCING (Ribbon (from Netfilx), RestTemplate) :-----------**

Server-side load balancer can be hardware or software based. Load balancing between different instances of same Microservices. Client side Load balancing is performed using Ribbon (provided by Spring-Cloud Netfilx).

Suppose we have four microservices Customer, Friend, Plan and CallDetails. And from UI many requests coming to Customer microservice and these request going further to Friend microservice which have multiple instances. Then how we do Load balancing using Ribbon:-

Steps:

1. Add depencency in customer microservice.

<dependency>

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

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

</dependency>

ii)we need to create a configuration class(separate class with any name like CustomerConfig or xyz) with a bean in CustomerMS

@Configuration

public class CustomerConfig{

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate(); //This will return a load balanced rest template.

} //if we do xml way,then we have to con-figure it in xml file

}

iii) Now in controller of customer service Autowire restTemplate like

@controller

@RibbonClient

Public class CustomerController {

@Autowired

RestTemplate template;

}

iv) Now we got load balanced rest templates, but this load balancer should work on which instances,how does it know? Use RibbonClient

@Controller

@EnableAutoConfiguration

@RibbonClient(name="custribbon")

Public class CustomerController {

@Autowired

RestTemplate template;

List<Long> friends = template.getForObject("http://custribbon/customers/"+phoneNo+"/friends",

List.class);

//no confusing why using customers when try to hit friends, here customers is RequestMapping not Micro service name.

}

Now in any properties file inside Customer microservice. Put property

custribbon.ribbon.eureka.enabled=false //eureka is disabled

custribbon.ribbon.listOfServers=http://localhost:8001,http://localhost:8082; //instances of friends

microservice

Note-It follows by default round robin strategy for load balancing, it also changed by adding configuration file. Suppose one of the instances is stopped, at that case when request hit the running instance its ok, but when try to hit the stopped instance then throws error. We can configure the Ping strategy so that we stop sending requests to services which are down.

**-----------------: Service Discovery (Netflix Eureka):------------------**

services may run in one port and after sometime, the cloud provider may shift it to another. How can one service find out the **current port and host** where other service is running?

**→**In micro service the solution is service discovery pattern. A service registers itself with the central server called service registry. Now two thing happens:--

a) Its name, port, host etc are stored in service registry.

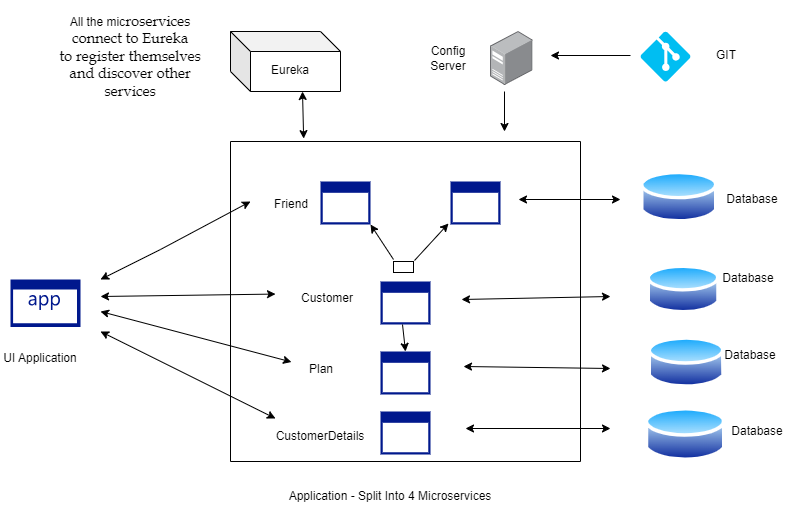
b) Others registered services also available to it.

Suppose the host, port is changed of the service and redeployed, then it update its info into service registry. The other services would discover about its updated details through the service registry. Service registry solutions like Netflix Eureka, etcd, Zookeeper, consul, etc.

c) Microservices will be registered with Eureka service discovery server.

Details of Eureka is also stored in GIT repo.

Details of eureka in GIT are accessed using config server.



Steps:→ @EnableDiscoveryClient annotation makes an application an Eureka instance as well as an Eureka Client

i) Create a Spring Starter project with name infytel-eureka.

11) Add the below dependencies:

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependency>

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

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

</dependency>

1) Add the below properties in the application.properties file of infytel-eureka spring.application.name=Eureka1

server.port=5555

eureka.client.fetch-registry=false // to run only a single eureka server instance

eureka.client.register-with-eureka=false // to run only a single eureka server instance

eureka.client.service-url.defaultZone=http://localhost:5555/eureka

iv) Add @EnableDiscoveryServer annotation in the application file(bootstrap class) of infytel-eureka

v) Add the below dependencies in all other microservices:

<dependency>

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

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

</dependency>

vi)Add @EnableDiscoveryClient in all microservices application file

vii)Add the below property in the application.properties file in git

eureka.client.service-url.default Zone=http://localhost:5555/eureka

viii)Autowire Discovery client in CustomerController class as @Autowired DiscoveryClient client;

ix)No need of Autowired RestTemplate and ribbon.Remove the String friendUri; from CustomerController and update the code in accessing the friend-family-service as:

List<ServiceInstance> instances = client.getInstances("FriendFamilyMS");

ServiceInstance instance=instances.get(0);

URI friendUri = instance.getUri();

List<Long> friends = new RestTemplate().getForObject(friendUri"/customers/" +phoneNo+"/friends");

Note: The service discovery happens through the spring.application.name value of the services. Hence they should not change.

**Note**: register-with-eureka property when set to true (which is by default), will register an application with the Eureka Sever. Such an application is also called a Eureka Instance. The Eureka Instance will start sending heartbeats to the Eureka Server. If the Eureka server does not receive heartbeats from a Instance within a configurable time limit (every 30 secs by default), it considers the Instance to be down and deregisters it from the registry.

fetch-registry property will fetch the registry from the Eureka Sever once at startup time and will cache it. It will check the Eureka Server at regular intervals (by default at every 30 secs) to see if there are any changes. If there are changes, it fetches only the updates and the unchanged parts will be continued to be accessed from cache.

A Eureka server is also a client. Because it can register itself with other Eureka servers and form a cluster.

Every client has to register itself with Eureka and it will also try to fetch the registry details. If we want to run only a single Eureka Server instance, then these two properties( fetch registry, register-with-eureka) should be false. Otherwise it is trying to register itself with itself

Actuator Endpoints To see all the MS instances registered: →

**http://(eureka-host):(eureka-port)/(eureka)/apps/**

or to see instances of any specific MS.

**<http://{eureka-host):(eureka-port)/{eureka)/apps/(spring-application-name)>**

**ChatGpt Code**

Eureka Server:

1.1. Create a Spring Boot Application (Eureka Server):

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class EurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaServerApplication.class, args);

}

}

1.2. Configure Eureka Server in application.properties or application.yml:

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

2. Microservice 1:

2.1. Create a Spring Boot Application (Microservice 1):

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

public class Microservice1Application {

public static void main(String[] args) {

SpringApplication.run(Microservice1Application.class, args);

}

}

2.2. Configure Microservice 1 in application.properties or application.yml:

server.port=8081

spring.application.name=microservice1

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

2.3. Create a Controller in Microservice 1:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.client.RestTemplate;

@RestController

@RequestMapping("/api")

public class Microservice1Controller {

@Autowired

private RestTemplate restTemplate;

@GetMapping("/consumeMicroservice2")

public String consumeMicroservice2() {

String microservice2Response = restTemplate.getForObject("http://microservice2/api/info", String.class);

return "Response from Microservice 2: " + microservice2Response;

}

}

2.4. Configure RestTemplate Bean in Microservice 1:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class RestTemplateConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

3. Microservice 2:

3.1. Create a Spring Boot Application (Microservice 2):

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

public class Microservice2Application {

public static void main(String[] args) {

SpringApplication.run(Microservice2Application.class, args);

}

}

3.2. Configure Microservice 2 in application.properties or application.yml:

server.port=8082

spring.application.name=microservice2

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

3.3. Create a Controller in Microservice 2:

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/api")

public class Microservice2Controller {

@GetMapping("/info")

public String getInfo() {

return "Information from Microservice 2";

}

}

4. Run the Applications:

Run the Eureka Server application.

Run Microservice 1 and Microservice 2 applications.

Access the Eureka Server dashboard at http://localhost:8761 to see registered microservices.

Access the Microservice 1 endpoint at http://localhost:8081/api/consumeMicroservice2 to see it making a call to Microservice 2.

**Ribbon With Eureka:** To get load Balanced service using Eureka.

Steps:-

1. Remove DiscoveryClient Autowire.

2. Add Autowired RestTemplate.

3. PlanDTO planDTO = template.getForObject("http://PLANMS"+"/plans/"+custDTO.getCurrentPlan().getPlanId(), PlanDTO.class);

List<Long> friends = template.getForObject("http://FRIENDFAMILYMS"+"/customers/"+phoneNo+"/friends", List.class);

NOTE: When we have used only ribbon, we used ribbonClient as http://custRibbon/ but now we will use ServiceName, which is registered with Eureka like http://PLANMS etc.

4. By default ribbon is always enabled with eureka, so we have disabled ribbon, when we were using Eureka only, but now ribbon is require. So, we will remove the properties from application.properties file which we written to disable ribbon. Now the list of servers(services) properties also removed from properties file.

5. Now, One service have many instances running, but when we hit it, it will redirect to only one, Bcoz the id created for every instance is same, so its get overridden, So, now we have to provide different ids to different instance. So in GIT SERVICE.properties file like CUTOMERMS.properties, we will declare one more property,

**eureka.instance.instance-id= $(spring.cloud.client.hostname):$(spring.application.name):$(spring.application.instance\_id):$(random.value)**

**Eureka-Cluster:**

Steps:

1. Open the hosts file in C:\Windows\System32\drivers\etc

2. Add the below hostnames:

127.0.0.1 Eurl

127.0.0.1 Eur2

127.0.0.1 Eur3

3. Use the below yml file in the infytel-eureka server

spring:

profiles: Eurekal

application:

name: Eureka

server:

port: 2222

eureka:

instance:

hostname: Eur1

client:

registerWithEureka: true

fetchRegistry: true

serviceUrl:

defaultZone: http://Eur2:2223/eureka/, http://Eur3:2224/eureka/

spring:

profiles: Eureka2

application:

name: Eureka

server:

port: 2223

eureka:

instance:

hostname: Eur2

client:

registerWithEureka: true

fetchRegistry: true

serviceUrl:

defaultZone: http://Eur1:2222/eureka/, http://Eur3:2224/eureka/

spring:

profiles: Eureka3

application:

name: Eureka

server:

port: 2224

eureka:

instance:

hostname: Eur3

client:

registerWithEureka: true

fetchRegistry: true

serviceUrl:

defaultZone: http://Eur1:2222/eureka/,http://Eur2:2223/eureka/

3. Comma separated values in the defaultZone indicate peer awareness Eureka1, Eureka2, and Eureka3 are peers of each other and hence will replicate the details across each other.

4. Update the application properties file in GIT for the below property:

eureka.client.service-url.defaultZone=http://Eur1:2222/eureka,http://Eur3 2223/eureka,http://Eur3:2224/eureka

5. Run all the three profiles of the Eureka server and restart all the micro- services

6. You will get three dashboards in three different Eureka ports. Since we have a cluster, each dashboard will have the same details of micro- services as the other two Eureka Servers in the cluster.

7. Bring down a microservice. You will find that since each Eureka server in a cluster replicates itself, all Eureka servers in the cluster will now have the same updated information.

**---------------: Declarative Client (Feign Client) :-----------------**

Feign is a declarative web service client developed by Netflix and integrated into the Spring Cloud ecosystem. It simplifies the process of making HTTP requests to other microservices in a declarative and easily configurable manner. Below, I'll provide a detailed explanation of Feign clients with examples.

**Declarative Feign Client Basics:**

Add Feign Dependency:

1. To use Feign in your Spring Boot project, add the following dependency:

<dependency>

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

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

</dependency>

1. Enable Feign Client in Application:

Annotate your main application class with @EnableFeignClients to enable Feign client support:

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableFeignClients

public class YourApplication {

public static void main(String[] args) {

SpringApplication.run(YourApplication.class, args);

}

}

1. Create Feign Client Interface:

Define a Feign client interface with the @FeignClient annotation specifying the name of the target microservice.

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.GetMapping;

@FeignClient(name = "target-microservice")

public interface MyFeignClient {

@GetMapping("/api/data")

String getData();

}

1. Feign Client Configuration:

Feign clients provide various configuration options, such as specifying request interceptors, error handling, and timeouts. You can create a configuration class annotated with @Configuration and use @Bean to customize the Feign client.

import feign.Logger;

import feign.Request;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class FeignConfig {

@Bean

public Logger.Level feignLoggerLevel() {

return Logger.Level.FULL; // Set the logging level

}

@Bean

public Request.Options options() {

return new Request.Options(5000, 10000); // Set connection and read timeouts

}

}

1. Using Feign Client in Service:

Now, you can inject the Feign client interface into your service or controller and use it to make declarative HTTP requests.

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class MyService {

private final MyFeignClient feignClient;

@Autowired

public MyService(MyFeignClient feignClient) {

this.feignClient = feignClient;

}

public String fetchDataFromMicroservice() {

return feignClient.getData();

}

}

**Example with Request Parameters and Headers:**

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RequestParam;

@FeignClient(name = "target-microservice")

public interface MyFeignClient {

@GetMapping("/api/data")

String getData();

@GetMapping("/api/data-with-params")

String getDataWithParams(@RequestParam("param1") String param1, @RequestParam("param2") int param2);

@GetMapping("/api/data-with-headers")

String getDataWithHeaders(@RequestHeader("Authorization") String token);

}

In this example, the MyFeignClient interface includes methods for making HTTP GET requests. The @FeignClient annotation specifies the name of the target microservice. You can then inject this interface into your services or controllers and use it as if it were a local service.

Feign clients provide a powerful and declarative way to interact with other microservices in a Spring Cloud environment. They handle many aspects of HTTP communication, making it easier to build resilient and scalable microservices architectures.

**-------------:Resilience (Circuit Breaker, Fallback, Hystrix):-------------**

Suppose one service tries to connect to another service. Due to some network issue or the server is slow or down, there is some error and service 1 not able to contact service 2.

What should we do in this scenario?

If a particular service is taking more time than usual, then don't send any- more requests to that service. This prevents an increase in slowing down of other services. As like electrical fuse. This is called circuit breaker pattern.

**Circuit Breaker(like electrical fuse):** In Microservices Communication, when the number of errors in a given time frame is beyond an acceptable limit, the circuit opens, thereby preventing further flow and protecting other parts of the application.

Using Netflix Hystrix, we can apply a circuit breaker pattern in our application. Hystrix opens the circuit, When number of failures are more, Hystrix uses the fail fast approach.

After opening the circuit, Hystrix will attempt to close the circuit again and send one request to check, is there any change in the status quo, if fails, then again opens the circuit and waits.

The error threshold, waiting time, retry attempts etc are all configurable in Hystrix.

**Fallback: Alternative Plan**

Alternate arrangement, when the circuit is opened by Hystrix is called a Fallback Pattern. Hystrix allows us to mention any alternate piece of code

that we wish to run if a service is down. We don't get the same result, but providing some form of data instead of an error is better.

Fallback executes when:

* An error occurs
* A timeout occurs
* Circuit opens

Step 1: Add Hystrix dependency into pom.xml.

<dependency>

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

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

</dependency>

Step 2: Add @EnableCircuit Breaker on top of EmployeeDashBoardService (Spring boot main method class or boot- strap class) Microservice, to enable Hystrix for this service.

@EnableDiscoveryClient and may be @Enable Eureka Client

@EnableCircuit Breaker

@EnableFeignClients

@SpringBootApplication

public class EmployeeDashBoardService (

public static void main(String[] args)

SpringApplication.run (Employee DashBoardService.class,

@Bean

public Rest Template restTemplate (RestTemplateBuilder builder)

return builder.build();

}

}

Step 3: Now we will change the Employee InfoController.java so it can be Hystrix enabled.

@RefreshScope

@RestController

public class EmployeeInfoController {

@Autowired

private RestTemplate restTemplate;

@Autowired

private EurekaClient eurekaClient;

@Value ("$(service.employyesearch.serviceId}")

private String employeeSearchServiceId;

@RequestMapping("/dashboard/(myself)")

@HystrixCommand (fallbackMethod="defaultMe")

public EmployeeInfo findme (@PathVariable Long myself){

Application application = eurekaClient.getApplication (employeeSearchServiceId);

InstanceInfo instanceInfo = application.getInstances().get(0);

String url = "http://"+instanceInfo.getIPAddr()+":"+instanceInfo.getPort()+"/"+"employee/find/"+mys elf;

System.out.println("URL" + url);

Employee Info emp = restTemplate.getForObject(url, EmployeeInfo.class);

System.out.println("RESPONSE " + emp);

return emp;

}

private EmployeeInfo defaultMe (Long id) {

Employee Info info = new EmployeeInfo();

info.setEmployeeId(id);

info.setName("Hystrix fallback");

info.setCompanyInfo("Netfilx");

info.setDesignation ("Fallback");

return info;

}

@RequestMapping("/dashboard/peers")

public Collection<Employee Info> Peeza(){

Application application = eurekaClient.getApplication (employeeSearchServiceId);

InstanceInfo instanceInfo = application.getInstances().get(0);

String url="http://"+instanceInfo.getIPAddr() + ":"+instanceInfo.getPort() +"/"+"employee/findall";

System.out.println("URL" + url);

Collection<Employee Info> list = restTemplate.getForObject (url, Collection.class);

System.out.println("RESPONSE" list);

return list;

}

}

Step4: Add the below properties in the EmployeeMs.properties file

hystrix.command.default.circuitBreaker.requestVolumeThreshold = 4 // to apply circuit breaker minimum 4 requests should be sent.

hystrix.metrics.rollingstats.timeInMilliseconds=10000 // within 10 seconds minimum of 4 request should be sent and if

hystrix.command.default.circuit Breaker.errorThresholdPer- centage=50 // if 50 percent request fails within 10 sec- onds then apply circuit breaker

@override

Public PlanDto invoke(){

return template.getForObject(url, PlanDto.class);

}

};

}

And in controller where this method getSpecificPlans is called, suppose

Future<PlanDto> planDtoFuture = xyzService.getSpecificPlans(id); //This is asynchronous call

xyzxyzxyz;

xyzxyzxyz;

planDtoFuture.get(); // here it waits to get the value, so this line added in the last and allows others line to get completed.

Xyzxyz;

xyzxyz;

add here planDtoFuture.get();

Failure in one microservice can result in cascading effect. So, failure in one microser. Should not bring down entire app. Hystrix uses circuit breaker pat- tern to take care of latency issues and failures. When it detects that the threshold of failures has breached, it opens the circuit and prevents from reaching it(Fail first approach).

Fail silent approach using Fallback is adopted to discard Fail First approach.

**------: Resilience using Resilience4J:------**

1. Dependency:

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<!-- https://mvnrepository.com/artifact/io.github.resilience4j/resilience4j-spring-boot3 -->

<dependency>

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

<artifactId>resilience4j-spring-boot3</artifactId>

<version>2.1.0</version>

</dependency>

1. Properties:

resilience4j:

circuitbreaker:

instances:

ratingHotelBreaker:

registerHealthIndicator: true

eventConsumerBufferSize: 10

failureRateThreshold: 50

minimumNumberOfCalls: 5

automaticTransitionFromOpenToHalfOpenEnabled: true

waitDurationInOpenState: 6s

permittedNumberOfCallsInHalfOpenState: 3

slidingWindowSize: 10

slidingWindowType: COUNT\_BASED

1. Code:

@GetMapping("/{id}")

@CircuitBreaker(name="ratingHotelBreaker", fallbackMethod = "ratingHotelFallback")

public ResponseEntity<UserEntity> getUser(@PathVariable String id){

logger.info("RetryCount: {}",retryCount);

retryCount++;

UserEntity userEntity = userService.getUser(id);

return ResponseEntity.ok(userEntity);

}

// Create fallback method for circuitbreaker, return type should be same as main method(getUser)

public ResponseEntity<UserEntity> ratingHotelFallback(String id, Exception ex){

logger.info("Fallback method called because service id down: ",ex.getMessage());

UserEntity user = UserEntity.builder().email("fallback@dummy.com").name("Fallback dummy user")

.about("Fallback user").userId("121212").build();

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

}

**-----:Retry Mechanism:------**

If we don’t want to use CircuitBreaker and use retry mechanism:

1. Properties:

resilience4j:

retry:

instances:

ratingHotelService:

maxAttempts: 3

waitDuration: 5s

1. Code:

@GetMapping("/{id}")

@Retry(name="ratingHotelService", fallbackMethod = "ratingHotelFallback")

public ResponseEntity<UserEntity> getUser(@PathVariable String id){

logger.info("RetryCount: {}",retryCount);

retryCount++;

UserEntity userEntity = userService.getUser(id);

return ResponseEntity.ok(userEntity);

}

// Create fallback method for circuitbreaker, return type should be same as main method(getUser)

public ResponseEntity<UserEntity> ratingHotelFallback(String id, Exception ex){

logger.info("Fallback method called because service id down: ",ex.getMessage());

UserEntity user = UserEntity.builder().email("fallback@dummy.com").name("Fallback dummy user")

.about("Fallback user").userId("121212").build();

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

}

TEST: make system down and test. We can see status close,open,half\_open in actuators health.

<http://192.168.43.157:8191/actuator/health> -- userservice url.

-----: Rate Limiter :------

To Limit the number of requests per user in a specified time to improve avalability, security and performance.

1. Properties:

resilience4j:

ratelimiter:

instances:

userRateLimiter:

limitRefreshPeriod: 4s

limitForPeriod: 2

timeoutDuration: 0

These properties signifies, a user can send max 2 request in 4 seconds.

1. Code:

@GetMapping("/{id}")

@RateLimiter(name="userRateLimiter", fallbackMethod = "ratingHotelFallback")

public ResponseEntity<UserEntity> getUser(@PathVariable String id){

logger.info("RetryCount: {}",retryCount);

retryCount++;

UserEntity userEntity = userService.getUser(id);

return ResponseEntity.ok(userEntity);

}

Above mentioned fallback method also require.

**----: Api Gateway (Zuul, Zuul And ribbon,Zuul and Hystrix):----**

Note: Provides Middleware Application to get URL of required micro- service to hit by the client.

In cloud the port and host for micro services changes frequently. So we need a proxy server between client and required micro service.

Reverse proxy--> When UI application sends its request to a proxy server which then forwards the request to appropriate micro services. This type of proxy is known as Reverse proxy. Zuul works as a proxy server which talks to eureka to get required micro service.

In Microservices architecture, the client may have to send requests to multiple services directly to get the final data. This Approach have many problems:

• Client is responsible for both gathering, aggregating and formatting the content.

• Client must know the port and host of the services. If they change client code also needs to change. Etc

The solution for these problems is API Gateway Pattern. Here API Gateway Server comes in between the client and the services. When a request comes it intelligently routes it to the appropriate services.

It also does:

* Request Aggregation. Based on a single request from the client, it invokes multiple services, aggregates the result and sends it back.
* Protocol translation. It will be responsible for taking data from a service through let's say AMQP and sends the data to the client over http.
* Security
* Load Balancing. (Using Netflix Zuul library.)

Dependency we need to add for this:

<dependency>

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

<artifactId>spring-cloud-starter-netflix-zuul</artifactId>

</dependency>

Steps to involve API Gateway in Boot application:

1. Create spring boot starter application (for middleware application).

2. Apart from eureka,cloud config dependencies, Add the above mentioned dependency in porn.xml.

3. Add the below mentioned annotations in the application file (main class) of the application.

@EnableZuulProxy

@EnableDiscoveryClient

Create bootstrap properties file to load config server details. 4.Routes can be added in the properties file like:

zuul.routes.<routeName>.path = /<URI> and

zuul.routes. <routeName>.service-id = <ServiceName> // by using strip-prefix= false we can avoid

// repition in the path

NOTE: if we don't add strip-prefix the url will be

http://localhost:3333/plans/plans/2 //this localhost:3333 is url of ZuulServer,

so, to get url as http://localhost:3333/plans/2 we have to add property strip-prefix as false.

5. /<URI>/\*\* will match anything after the given URI. Sometimes a single can be used as a wildcard character : /URI/\*/hello.

6. Add the below routes to the properties file of ZuulServer(Middleware application)

**----------: Api Gateway without Zuul:---------------**

A microservices architecture often involves multiple services working together to provide a complete application. In such architectures, an API Gateway plays a crucial role by serving as the entry point for client requests, handling various cross-cutting concerns, and routing requests to the appropriate microservices. Let's explore the concept of an API Gateway with descriptive examples.

What is an API Gateway?

An API Gateway is a server that acts as an API front-end, receiving API requests, enforcing throttling and security policies, passing requests to the back-end services, and then passing the response back to the requester. It is a key component in microservices architectures that provides a centralized point for managing, securing, and optimizing API calls.

**Key Responsibilities of an API Gateway:**

Routing and Composition:

An API Gateway routes incoming requests to the appropriate microservice based on the request's URI or other attributes. It can also aggregate multiple microservices into a single response.

Request Transformation:

It can modify incoming requests, such as transforming the request payload or headers before forwarding them to the microservices.

Response Transformation:

Similarly, the API Gateway can transform the responses from microservices, making them suitable for the client or applying consistent formatting.

Authentication and Authorization:

Handles user authentication and authorization, ensuring that only authorized users or systems can access certain resources.

Load Balancing:

Distributes incoming requests across multiple instances of a microservice to ensure high availability and load distribution.

Rate Limiting and Throttling:

Enforces rate limiting and throttling to prevent abuse or overuse of microservices.

Logging and Monitoring:

Captures logs and metrics for incoming and outgoing requests, aiding in monitoring and debugging.

Example:

Api- Gateway module contains following code:

1. Register itself to ServiceRegistry:

package com.lcwd.gateway;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

public class ApiGatewayApplication {

public static void main(String[] args) {

SpringApplication.run(ApiGatewayApplication.class, args);

}

}

1. Dependencies needs to be added:

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

1. Application.yml file:

server:

port: 8195

spring:

application:

name: API-GATEWAY

cloud:

gateway:

routes:

- id: USER-SERVICE

uri: lb://USER-SERVICE

predicates:

- Path=/users/\*\*

- id: HOTEL-SERVICE

uri: lb://HOTEL-SERVICE

predicates:

- Path=/hotel/\*\*,/staffs/\*\*

- id: RATING-SERVICE

uri: lb://RATING-SERVICE

predicates:

- Path=/rating/\*\*

eureka:

instance:

prefer-ip-address: true

client:

fetch-registry: true

register-with-eureka: true

service-url:

defaultZone: <http://localhost:8194/eureka>

Now call User-Service from postman through Api-Gateway:

**<http://localhost:8195/users/601eaffa-e870-41cd-958d-09631310700c>**

**-----------------: Security :----------------**

For Security we will use OKTA.

**PART-1: Steps to create OKTA account and create Application , Groups and Peoples(users) in OKTA.**

1. Go to OKTA and sign-In with google.
2. The URL after our login is the domain.

domain - <https://dev-54760333-admin.okta.com/admin/getting-started> till .com is the domain.

1. In People We can add more peoples. No need to do anything now.
2. Groups - we can create different user group like Admin, User, Customer etc.
3. Main this is the Application. Which we will create here in OKTA.
4. Create App Integration --> OIDC(OpenID Connect)-->Web Application-->Give name to application

-->Select Grant Types--Client Credentials, Authorization Code and Refresh Token--> Sign-In Redirect url will be <http://localhost:8195/login/oauth2/code/okta> give API Gateway IP and port.--> Sign\_out also change Ip and port.--> Skip Group Assignment for now. --> Save

1. Now note down Client ID, which is -- 0oadjzmdbtPbaNmX15d7
2. And secret , which is -- V5E5gNVrgdEoIPGuqfC7zxCtTOU2btY7iKHgn3DXuaNBH9rpOyyoWndEA0ys29Eu
3. Now Go To Assignments Tab. And assign two things. Groups and People.
4. Now to sidebar Directory--> Groups --> Add Group --> Give name

Ex. i) Group Name - Normal or Admin , Description - This Group Belongs to Normal user.

Now Groups are not assigned to Any Application and there are no peoples in the group.

1. Now Application --> Select our Application -->Go to assignments tab --> Groups and then assign groups to the app. We can select Admin and Normal Group.
2. Now Assign Peoples to the Application. We can assign to ourself.
3. If require we can create multiple Peoples.
4. In Security -->API we will get issuer details,scopes, claims.

**PART 2: We will configure OKTA in API-Gateway and then in Other Services:**

1. Now go to OKTA--> API--> Scopes--> Add Scope, Name it as internal.

Scope is used to determine that only internally API call will happen to hotel or rating App. But no one directly access Hotel or Rating from outside.

1. Go To Claims--> Name it like myclaim--> Include in Token Type will be Access Token -->Value type will be Groups-->Filter= Matches Regex--> .\* -->Save.
2. Dependency:

<!-- Security and Okta related dependency -->

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.okta.spring</groupId>

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

<version>3.0.5</version>

</dependency>

Properties:

okta:

oauth2:

issuer: https://dev-54760333.okta.com/oauth2/default

audience: api://default

client-id: 0oadjzmdbtPbaNmX15d7

client-secret: V5E5gNVrgdEoIPGuqfC7zxCtTOU2btY7iKHgn3DXuaNBH9rpOyyoWndEA0ys29Eu

scopes: openid, profile, email, offline\_access

Code 1 :

package com.subh.gateway.config;

import org.springframework.context.annotation.Bean;

import org.springframework.security.config.web.server.ServerHttpSecurity;

import org.springframework.security.web.server.SecurityWebFilterChain;

@Configuration

@EnableWebFluxSecurity // Here we have used EbFlux because in dependency we have used web flux in // place of normal web.

public class SecurityConfig {

@Bean

public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity httpSecurity){

httpSecurity

.authorizeExchange()

.anyExchange()

.authenticated()

.and()

.oauth2Client()

.and()

.oauth2ResourceServer()

.jwt();

return httpSecurity.build();

}

}

Till here code is enough, now if we try to access any Url through Api-Gateway it will redirect to login page.

Below code is not required, but we have written this controller method to see login details like userid, token etc. This method will provide token which we can use to call other methods.

Code 2 :

package com.subh.gateway.controllers;

import com.subh.gateway.models.AuthResponse;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.security.core.annotation.AuthenticationPrincipal;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClient;

import org.springframework.security.oauth2.client.annotation.RegisteredOAuth2AuthorizedClient;

import org.springframework.security.oauth2.core.oidc.user.OidcUser;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import java.util.List;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/auth")

public class AuthController {

private Logger logger = LoggerFactory.getLogger(AuthController.class);

@GetMapping("/health")

public ResponseEntity<String> getHealth(){

logger.info("Api Gateway Health Check");

return new ResponseEntity<>("Health OK", HttpStatus.OK);

}

@GetMapping("/login")

public ResponseEntity<AuthResponse> login(

@RegisteredOAuth2AuthorizedClient("okta") OAuth2AuthorizedClient client,

@AuthenticationPrincipal OidcUser user,

Model model){

logger.info("user email id :{}",user.getEmail());

AuthResponse authResponse = new AuthResponse();

authResponse.setUserId(user.getEmail());

authResponse.setAccessToken(client.getAccessToken().getTokenValue());

authResponse.setRefreshToken(client.getRefreshToken().getTokenValue());

authResponse.setExpireAt(client.getAccessToken().getExpiresAt().getEpochSecond());

List<String> authorities = user.getAuthorities().stream().map(grantedAuthority -> {

return grantedAuthority.getAuthority();

}).collect(Collectors.toList());

authResponse.setAuthorities(authorities);

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

}

}

Now whatever url of API Gateway we will try to access it will redirect to OKTA login page. Then we have to provide creds, then it will login and go to the actual page.

Like - <http://localhost:8195/auth/login> or <http://localhost:8195/auth/health> will redirect to OKTA login page.

After login we will get the Access Token , we need to take the token and in POSTMAN where we are calling API’s through gateway, in Authorization-->Type OAuth 2.0--> Add Authorization Data to - Request header --> In Token paste the token and call API.

**Part 5: Now we have to apply security in UserService :**

Here, UserService will work as client also, because it’s calling hotel and rating service.

Steps:

1. Add Dependency in UserService. From API Gateway after login request will come to UserService.

Secured request will come to User from API Gateway. UserService will behave as Client.

<!-- Spring Security dependencies -->

<dependency>

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

<artifactId>spring-boot-starter-oauth2-client</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.okta.spring</groupId>

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

<version>3.0.5</version>

</dependency>

1. Properties:

# Spring security configuration to make userservice a client, which will call hotel and rating

spring:

security:

oauth2:

resourceserver:

jwt:

issuer-uri: https://dev-54760333.okta.com/oauth2/default # from okta

client:

registration:

my-internal-custom-client:

provider: okta

authorizationGrantType: client\_credentials

scope: internal # we created in okta

clientId: 0oadjzmdbtPbaNmX15d7

clientSecret: V5E5gNVrgdEoIPGuqfC7zxCtTOU2btY7iKHgn3DXuaNBH9rpOyyoWndEA0ys29Eu

provider:

okta:

issuerUri: <https://dev-54760333.okta.com/oauth2/default>

## Okta configurations no need of client-id and secret because it's not going to open login page

okta:

oauth2:

issuer: https://dev-54760333.okta.com/oauth2/default

audience: api://default

1. Now first we have to create Security Configurations for UserService.

package com.subh.user.service.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class WebSecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity security) throws Exception {

security.authorizeHttpRequests()

.anyRequest()

.authenticated()

.and()

.oauth2ResourceServer()

.jwt();

return security.build();

}

}

1. Now Create Feign Client Interceptor:

package com.subh.user.service.config.interceptor;

import feign.RequestInterceptor;

import feign.RequestTemplate;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.oauth2.client.OAuth2AuthorizeRequest;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClientManager;

import org.springframework.stereotype.Component;

@Configuration

@Component

public class FeignClientInterceptor implements RequestInterceptor {

private OAuth2AuthorizedClientManager manager; // this manager bean will be created in next step

@Override

public void apply(RequestTemplate requestTemplate) {

// "my-internal-custom-client" we declared in yml file, "internal" we declared in OKTA Scope or claim

String token = manager.authorize(OAuth2AuthorizeRequest.withClientRegistrationId("my-internal-custom-client")

.principal("internal").build()).getAccessToken().getTokenValue();

requestTemplate.header("Authorization", "Bearer "+token);

}

}

1. Create manager:

package com.subh.user.service.config;

import com.subh.user.service.config.interceptor.RestTemplateInterceptor;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.client.ClientHttpRequestInterceptor;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClientManager;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClientProvider;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClientProviderBuilder;

import org.springframework.security.oauth2.client.registration.ClientRegistrationRepository;

import org.springframework.security.oauth2.client.web.DefaultOAuth2AuthorizedClientManager;

import org.springframework.security.oauth2.client.web.OAuth2AuthorizedClientRepository;

import org.springframework.web.client.RestTemplate;

import java.util.ArrayList;

import java.util.List;

@Configuration

public class UserConfigurations {

@Bean

@LoadBalanced

public RestTemplate restTemplate(){

return new RestTemplate();

}

// Bean for Spring Security

@Bean

public OAuth2AuthorizedClientManager manager(ClientRegistrationRepository clientRegistrationRepository,

OAuth2AuthorizedClientRepository oAuth2AuthorizedClientRepository){

OAuth2AuthorizedClientProvider provider = OAuth2AuthorizedClientProviderBuilder.builder()

.clientCredentials().build();

DefaultOAuth2AuthorizedClientManager defaultOAuth2AuthorizedClientManager =

new DefaultOAuth2AuthorizedClientManager(clientRegistrationRepository, oAuth2AuthorizedClientRepository);

defaultOAuth2AuthorizedClientManager.setAuthorizedClientProvider(provider);

return defaultOAuth2AuthorizedClientManager;

}

}

1. Now Create RestTemplate Interceptor: First Make change in RetTemplate Bean

package com.subh.user.service.config;

import com.subh.user.service.config.interceptor.RestTemplateInterceptor;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.client.ClientHttpRequestInterceptor;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClientManager;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClientProvider;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClientProviderBuilder;

import org.springframework.security.oauth2.client.registration.ClientRegistrationRepository;

import org.springframework.security.oauth2.client.web.DefaultOAuth2AuthorizedClientManager;

import org.springframework.security.oauth2.client.web.OAuth2AuthorizedClientRepository;

import org.springframework.web.client.RestTemplate;

import java.util.ArrayList;

import java.util.List;

@Configuration

public class UserConfigurations {

@Autowired

private ClientRegistrationRepository clientRegistrationRepository;

@Autowired

private OAuth2AuthorizedClientRepository oAuth2AuthorizedClientRepository;

@Bean

@LoadBalanced

public RestTemplate restTemplate(){

// now will comment this and create resttemplate with interceptors

//return new RestTemplate();

RestTemplate restTemplate = new RestTemplate();

List<ClientHttpRequestInterceptor> interceptors = new ArrayList<>();

interceptors.add(new RestTemplateInterceptor(manager(clientRegistrationRepository,

oAuth2AuthorizedClientRepository)));

restTemplate.setInterceptors(interceptors);

return restTemplate;

}

// Bean for Spring Security

@Bean

public OAuth2AuthorizedClientManager manager(ClientRegistrationRepository clientRegistrationRepository,

OAuth2AuthorizedClientRepository oAuth2AuthorizedClientRepository){

OAuth2AuthorizedClientProvider provider = OAuth2AuthorizedClientProviderBuilder.builder()

.clientCredentials().build();

DefaultOAuth2AuthorizedClientManager defaultOAuth2AuthorizedClientManager =

new DefaultOAuth2AuthorizedClientManager(clientRegistrationRepository, oAuth2AuthorizedClientRepository);

defaultOAuth2AuthorizedClientManager.setAuthorizedClientProvider(provider);

return defaultOAuth2AuthorizedClientManager;

}

}

1. Now Create Interceptor:

package com.subh.user.service.config.interceptor;

import org.springframework.http.HttpRequest;

import org.springframework.http.client.ClientHttpRequestExecution;

import org.springframework.http.client.ClientHttpRequestInterceptor;

import org.springframework.http.client.ClientHttpResponse;

import org.springframework.security.oauth2.client.OAuth2AuthorizeRequest;

import org.springframework.security.oauth2.client.OAuth2AuthorizedClientManager;

import java.io.IOException;

public class RestTemplateInterceptor implements ClientHttpRequestInterceptor {

private OAuth2AuthorizedClientManager manager;

public RestTemplateInterceptor(OAuth2AuthorizedClientManager manager) {

this.manager = manager;

}

@Override

public ClientHttpResponse intercept(HttpRequest request, byte[] body, ClientHttpRequestExecution execution) throws IOException {

String token = manager.authorize(OAuth2AuthorizeRequest.withClientRegistrationId("my-internal-custom-client")

.principal("internal").build()).getAccessToken().getTokenValue();

request.getHeaders().add("Authorization", "Bearer "+token);

return execution.execute(request,body);

}

}

**Part 6: Now we have to apply security in Hotel and Rating :**

Hotel and Service Presently not calling any other service. So, it will not be configured as client. But in any case we want to call any service from Hotel or Rating, then client related changes are needs to be done. As done in User Service.

Steps:

1. Dependencies

<!-- Security and Okta related dependency -->

<dependency>

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

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

</dependency>

<dependency>

<groupId>com.okta.spring</groupId>

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

<version>3.0.5</version>

</dependency>

1. Properties:

## Okta configurations no need of client-id and secret because it's not going to open login page

okta:

oauth2:

issuer: https://dev-54760333.okta.com/oauth2/default

audience: api://default

1. SecurityConfig Code:

package com.subh.hotel.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class SecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity security) throws Exception {

security

.authorizeHttpRequests()

.anyRequest()

.authenticated()

.and()

.oauth2ResourceServer()

.jwt();

return security.build();

}

}

Now if we want to put some access limitations, means we want api’s to be called by only admins or internally it will be called. we can do in two ways. Either in SecurityConfig class we write methods and put there antmatchers and restrictions or as we used annotation

@EnableGlobalMethodSecurity(prePostEnabled = true)

We can put restrictions in controller API’s itself.

Examples:

In RatingService:

/\*\*\* Only User with access as Admin can create hotel, internally(UserService then HotelService)

\* or directly accessing hotel service through Api Gateway or direct Hotel service call \*\*\*/

@PreAuthorize("hasAuthority('Admin')")

@PostMapping()

public ResponseEntity<HotelEntity> createHotel(@RequestBody HotelEntity hotel){

HotelEntity hotelEntity = hotelService.saveHotel(hotel);

return ResponseEntity.status(HttpStatus.CREATED).body(hotelEntity);

}

/\*\*\* This api will only called by internal Api. But we can't call it directly.

In OkTA we have created one SCOPE as internal, that we used here \*\*\*/

@PreAuthorize("hasAuthority('SCOPE\_internal')")

@GetMapping("/{id}")

public ResponseEntity<HotelEntity> getHotel(@PathVariable String id){

HotelEntity hotelEntity = hotelService.getHotel(id);

return ResponseEntity.status(HttpStatus.OK).body(hotelEntity);

}

/\*\*\* This api will called by internal Api or user with Admin access, Normal user will not be able

\* to call it. But we can call it directly,

\* If it's Admin.In OkTA we have created one SCOPE as internal, that we used here \*\*\*/

@PreAuthorize("hasAuthority('SCOPE\_internal') || hasAuthority('Admin')")

@GetMapping()

public ResponseEntity<List<HotelEntity>> getAllHotel(){

List<HotelEntity> hotelEntity = hotelService.getAllHotel();

return ResponseEntity.ok(hotelEntity);

}

In HotelService:

@PreAuthorize("hasAuthority('Admin')")

@PostMapping()

public ResponseEntity<Rating> createRating(@RequestBody Rating rating){

Rating ratingEntity = ratingService.saveRating(rating);

return ResponseEntity.status(HttpStatus.CREATED).body(ratingEntity);

}

@PreAuthorize("hasAuthority('SCOPE\_internal') || hasAuthority('Admin')")

@GetMapping("/user/{userId}")

public ResponseEntity<List<Rating>> getRatingByUser(@PathVariable String userId){

List<Rating> ratingEntity = ratingService.getRatingByUserId(userId);

return ResponseEntity.status(HttpStatus.OK).body(ratingEntity);

}

In UserService:

@PreAuthorize("hasAuthority('Admin')")

@PostMapping()

public ResponseEntity<UserEntity> createUser(@RequestBody UserEntity user){

UserEntity userEntity = userService.saveUser(user);

return ResponseEntity.status(HttpStatus.CREATED).body(userEntity);

}

int retryCount = 1;

/\*\*\* We can either use circuit-breaker or retry mechanism\*\*\*/

/\*\*\* This api will called by internal Api or user with Admin access, Normal user will not be able

\* to call it. But we can call it directly,

\* If it's Admin.In OkTA we have created one SCOPE as internal, that we used here \*\*\*/

@PreAuthorize("hasAuthority('SCOPE\_internal') || hasAuthority('Admin')")

@GetMapping("/{id}")

//@CircuitBreaker(name="ratingHotelBreaker", fallbackMethod = "ratingHotelFallback")

//@Retry(name="ratingHotelService", fallbackMethod = "ratingHotelFallback")

@RateLimiter(name="userRateLimiter", fallbackMethod = "ratingHotelFallback")

public ResponseEntity<UserEntity> getUser(@PathVariable String id){

logger.info("RetryCount: {}",retryCount);

retryCount++;

UserEntity userEntity = userService.getUser(id);

return ResponseEntity.ok(userEntity);

}