

Add instructor notes
here.

Locating Services at Runtime using Service Discovery



Objective

- How to implement client-side load balancing with Ribbon
- How to implement a Naming Server (Eureka Naming Server)
- How to connect the micro services with the Naming Server and Ribbon
- Fault tolerance & Resilience
- Circuit Breaker Pattern-**Hystrix**

What is Eureka?

- Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.
- Eureka also comes with a Java-based client component, the **Eureka Client**, which makes interactions with the service much easier.
- The client also has a built-in load balancer that does basic round-robin load balancing. At Netflix, a much more sophisticated load balancer wraps Eureka to provide weighted load balancing based on several factors like traffic, resource usage, error conditions etc to provide superior resiliency.

Netflix Eureka

- **Netflix Eureka** is a lookup server (also called a registry). All the microservices in the cluster register themselves to this server.
- When making a REST call to another service, instead of providing a hostname and port, they just provide the service name.
- The actual routing is done at runtime along with equally distributing the load among the end services. There are other service discovery clients like Consul, Zookeeper etc
- **Eureka Server**: acts as a service registry.
- **Product Service**: a simple REST service that provides product information.
- **Recommendation Service**: a simple REST service but it internally calls the product Service to complete its requests.

Eureka-Server

← → ↻ 🔒 https://start.spring.io

☆ 🔄 📄 📁 📌 ⌚ ⚙️

Project Metadata

Group

com.cg

Artifact

Eureka-Server

> Options

Dependencies

🔍 ☰

1 selected

Search dependencies to add

Web, Security, JPA, Actuator, Devtools...

Selected dependencies

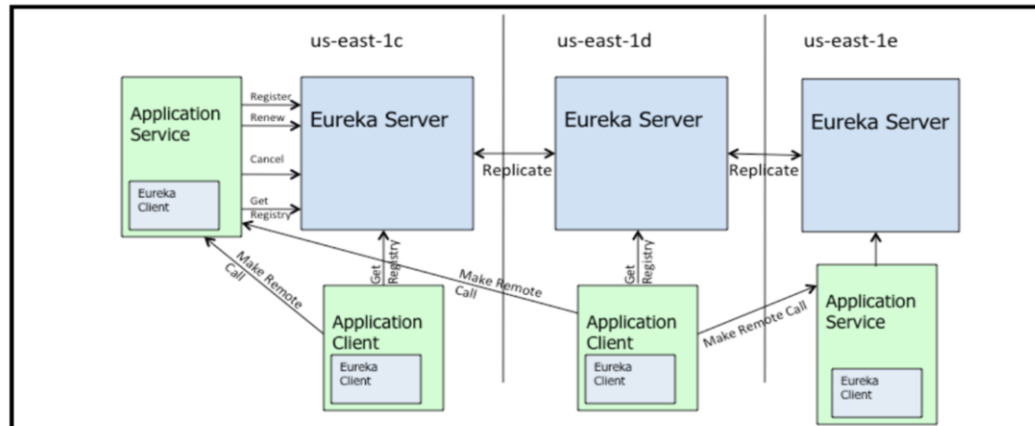
Eureka Server

spring-cloud-netflix Eureka Server

✓

5

Eureka-Server



Eureka-Server

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }

}
```

Eureka-Server-Properties

server.port=8761 //We can change port number

spring.application.name=eureka-server


#server.port=8761---default port

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

Eureka-Server-UP

← → ↻ localhost:8761 🔍 ☆ 🌐 📄 📱 🕒 🌐

 [HOME](#) [LAST 1000 SINCE STARTUP](#)

System Status

Environment	test
Data center	default

Current time	2019-07-19T11:55:05 +0530
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

Eureka-Client

Dependencies

Product-EuClient

> Options



2 selected

Search dependencies to add

Web, Security, JPA, Actuator, Devtools...

Selected dependencies

Eureka Discovery Client

a REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.



Spring Web Starter

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.



Eureka-Client

```
@SpringBootApplication
@EnableDiscoveryClient
public class ProductInfoClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductInfoClientApplication.class, args);
    }

}
```

Eureka-Client

`server.port=9091`

`spring.application.name=product-info-service`

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

Service Name

Registry
Server

Service Discovery -Eureka

← → ↻ localhost:8761 🔍 ☆ 🌐 📄 🗑️ 🔄 🛑

Environment	test	Current time	2019-07-20T16:56:45 +0530
Data center	default	Uptime	00:01
		Lease expiration enabled	false
		Renews threshold	3
		Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PRODUCT-INFO-SERVICE	n/a (1)	(1)	UP (1) - localhost:product-info-service:9091

Accessing Service from Rest Template

- We also need to create a RestTemplate bean and mark it as @LoadBalanced.
- By Ribbon we want to take advantage of client-side load balancing.
- Ribbon was developed by Netflix and later open sourced. It's dependency automatically comes with the Eureka Discovery dependency. It automatically integrates with Spring and distributes loads based on server health, performance, region, etc.
- We won't be required to use Ribbon directly as it automatically integrates RestTemplate, Zuul, Feign, etc. Using @LoadBalanced we made RestTemplate ribbon aware

- To call Client

```
resttemplate.getForObject("http://product-info-service/info/list", Product[].class);
```

Client Service Setup

```
@SpringBootApplication
@EnableDiscoveryClient
@ComponentScan("com.cg.productinfofront")
public class ProductinfofrontApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductinfofrontApplication.class, args);
    }

    @LoadBalanced
    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }

}
```

Demo

Productinfoclient
Productinfofront
Eurekaserver

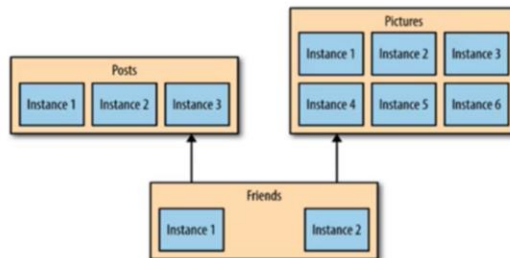
Fault tolerance & Resilience

- If Service A calls Service B, which in turn calls Service C, what happens when Service B is down? What is our fallback plan in such a scenario?
 1. Can you return a pre-decided error message to the user?
 2. Can you call another service to fetch the information?
 3. Can you return values from cache instead?
 4. Can you return a default value?
- There are a number of things we can do to make sure that the entire chain of microservices does not fail with the failure of a single component.
- There are a number of moving components in a microservice architecture, hence it has more points of failures. Failures can be caused by a variety of reasons – errors and exceptions in code, release of new code, bad deployments, hardware failures, datacenter failure, poor architecture, lack of unit tests, communication over the unreliable network, dependent services, etc.

In a monolithic application, a single error has the potential of bringing down the entire application. This can be avoided in a microservice architecture since it contains smaller independently deployable units, which won't effect the entire system. But does that mean a microservice architecture is resilient to failures? No, not at all. Converting your monolith into microservices does not resolve the issues automatically. In fact, working with a distributed system has its own challenges! While architecting distributed cloud applications, you should assume that failures will happen and design your applications for resiliency. A microservice ecosystem is going to fail at some point or another and hence you need to embrace failures. Don't design systems with the assumption that its going to go smoothly throughout the year. Be realistic and account for the chances of having rain, snow, thunderstorms, and other adverse conditions (if I may be metaphorical). In short, design your microservices with failure in mind. Things don't always go according plan and you need to be prepared for the worst case scenario

Resilience

- The problem of one component failure which can be isolated and the rest of the system can carry on working. This is a key concept in resilience engineering.
- With smaller services, we can only scale services that need scaling.
 - This way other parts of the system can still run on less powerful hardware.
- When embracing on-demand provisioning systems like those provided by Amazon Web Services, Azure, we can apply this scaling on demand for the component that need it.



Why Do We Need to Make Services Resilient?

- A problem with distributed applications is that they communicate over a network – which is unreliable. Hence we need to design our microservices so that they are fault tolerant and handle failures gracefully. In our microservice architecture, there might be a dozen of services talking with each other. We need to ensure that one failed service does not bring down the entire architecture.

How to Make our Services Resilient?

- Identify Failure Scenarios
- Avoid Cascading Failures
- Avoid Single Points of Failure
- Handle Failures Gracefully and Allow for Fast Degradation
- Design for Failures

Identify Failure Scenarios

Before releasing your new microservice to production, make sure you have tested it good enough. Strange things might happen though and you should be ready for the worst case scenario. This means you should prepare to recover from all sort of failures gracefully and in a short duration of time. This gives confidence on the system's ability to withstand failures and recover quickly with minimal impact. Hence it is important to identify failure scenarios in your architecture.

One way to achieve this is by making your microservices to fail and then try to recover from the failure. This process is commonly termed as Chaos Testing. Think about scenarios like below and find out how the system behaves:

Service A is not able to communicate with Service B.

Database is not accessible.

Your application is not able to connect to the file system.

Server is down or not responding.

Inject faults/delays into the services.

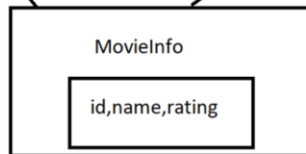
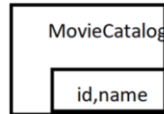
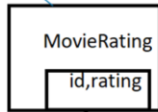
Avoid Cascading Failures

When you have service dependencies built inside your architecture, you need to ensure that one failed service does not cause ripple effect among the entire chain. By avoiding cascading failures, you will be able to save network resources, make optimal

use of threads and also allow the failed service to recover!

Fault tolerance & Resilience

If its down what
is the solution



- Scenarios
1. If its down what is the solution
 2. Suppose Any of the service is slow

Called by Front End-
Angular, React, etc

Lab

Lab 2