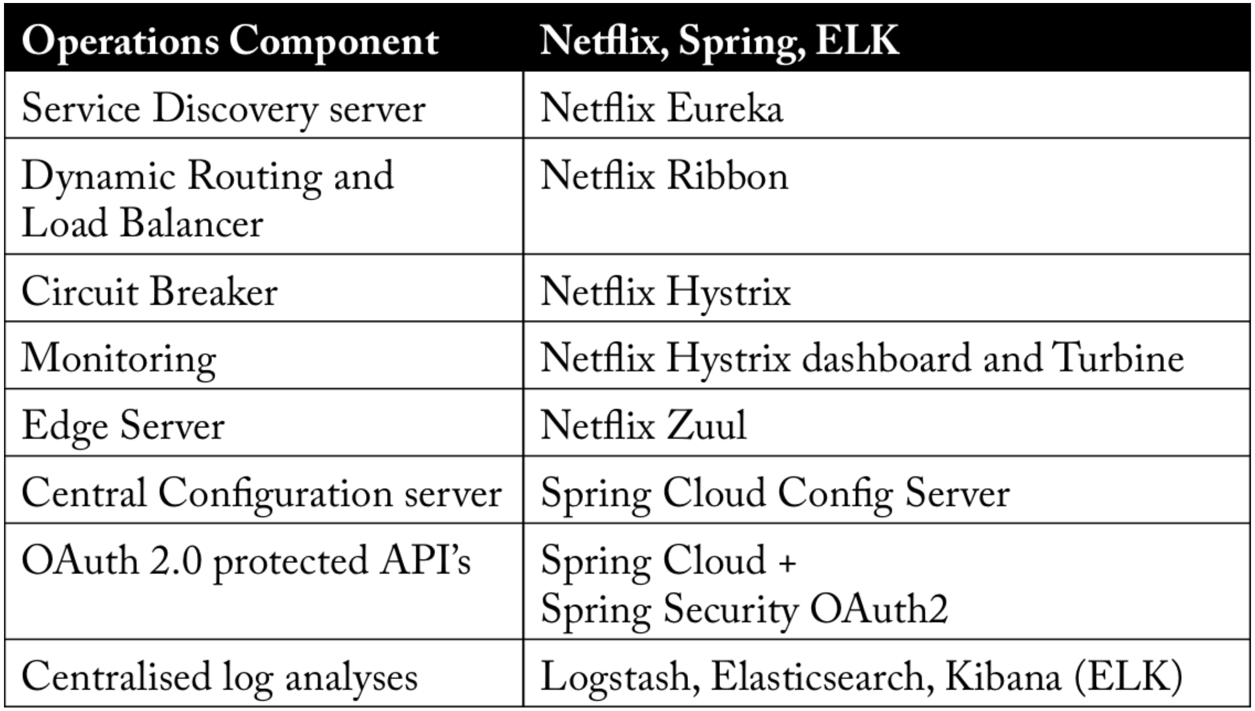
Spring Cloud 1

by Vlad Ungureanu

Eureka and Feign Clients

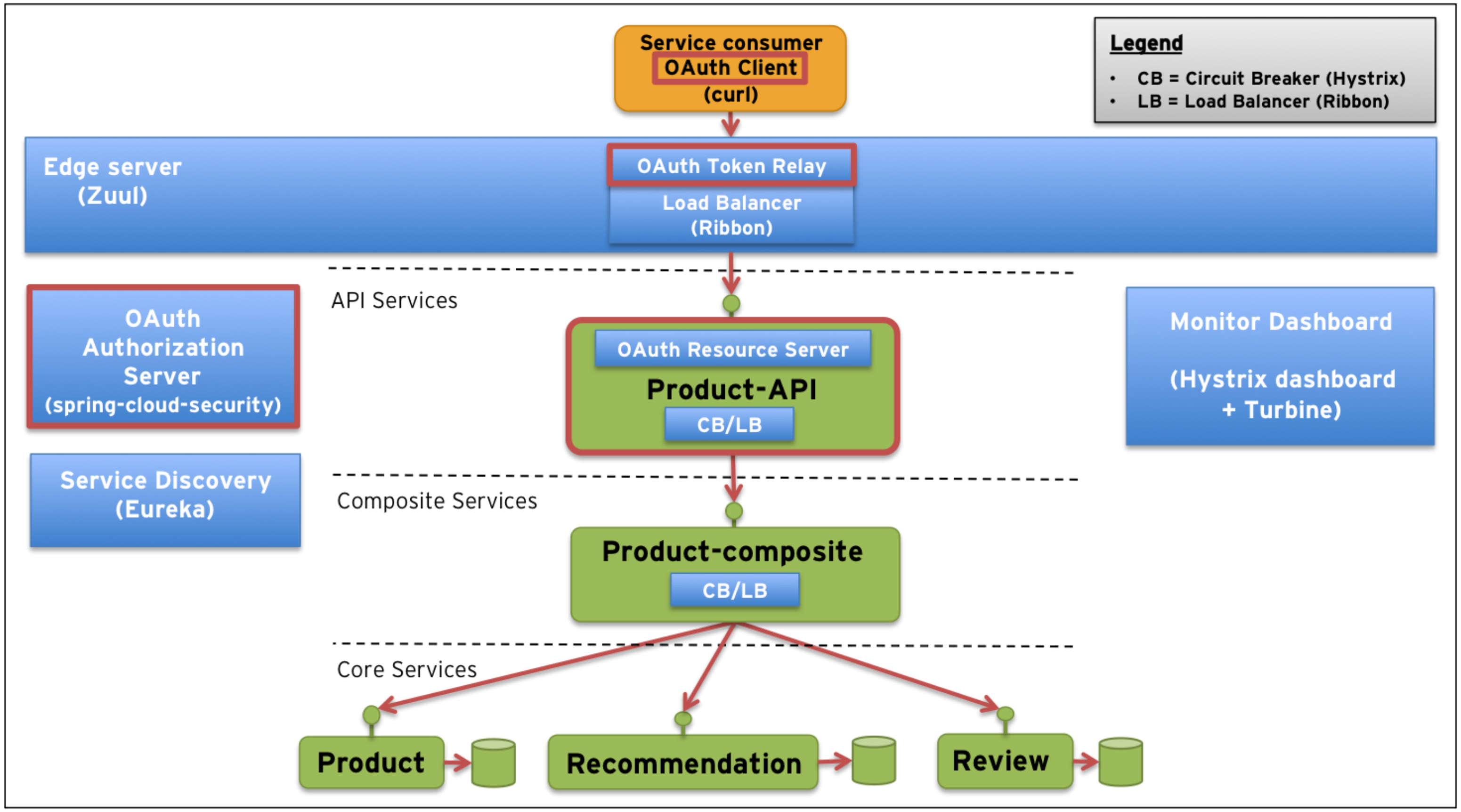
- Spring Cloud Overview
- Eureka
- Spring Eureka
- Ribbon
- Spring Ribbon
- Feign Clients

Overview



Operations Component	Netflix, Spring, ELK
Service Discovery server	Netflix Eureka
Dynamic Routing and Load Balancer	Netflix Ribbon
Circuit Breaker	Netflix Hystrix
Monitoring	Netflix Hystrix dashboard and Turbine
Edge Server	Netflix Zuul
Central Configuration server	Spring Cloud Config Server
OAuth 2.0 protected API's	Spring Cloud + Spring Security OAuth2
Centralised log analyses	Logstash, Elasticsearch, Kibana (ELK)

Overview



Service consumer
OAuth Client
(curl)
Legend
CB = Circuit Breaker (Hystrix)
LB = Load Balancer (Ribbon)
Edge server
(Zuul)
OAuth Token Relay
Load Balancer
(Ribbon)
OAuth
Authorization
Server
(spring-cloud-security)
API Services
OAuth Resource Server
Product-API
CB/LB
Monitor Dashboard
(Hystrix dashboard
+ Turbine)
Service Discovery
(Eureka)
Composite Services
Product-composite
CB/LB
Core Services
Product
Recommendation
Review

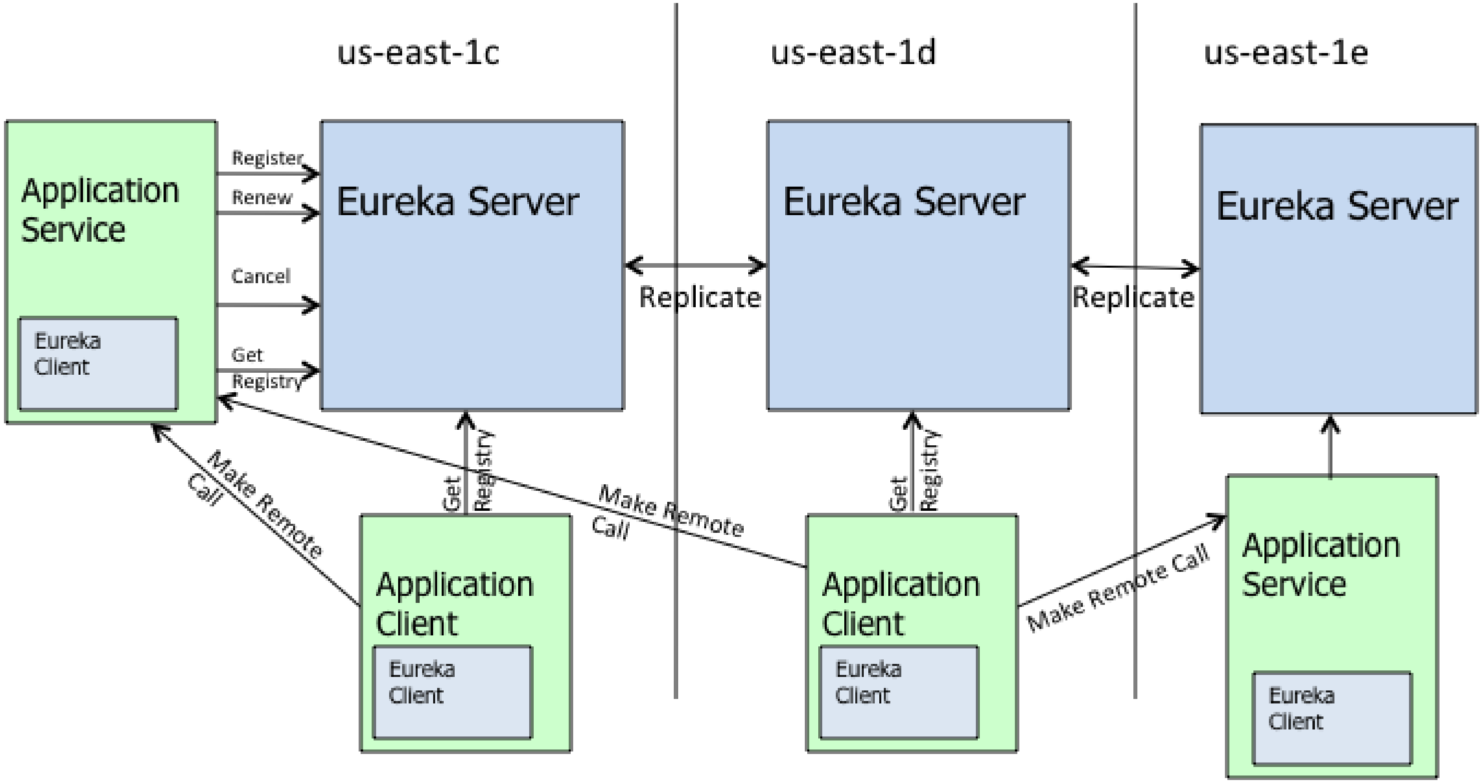
Eureka

- The Server is a standalone application and is responsible for:
  - managing a registry of Service Instances,
  - provide means to register, de-register and query Instances with the registry,
  - registry propagation to other Eureka Instances (Servers or Clients).
- The Client is part of the Service Instance ecosystem and has responsibilities like:
  - register and unregister a Service Instance with Eureka Server,
  - keep alive the connection with Eureka Server,
  - retrieve and cache discovery information from the Eureka Server.

Eureka Overview

- Services have no prior knowledge about the physical location of other Service Instances
- Services advertise their existence and disappearance to the Eureka Servers
- Services are able to find instances of another Service based on advertised metadata
- Instance failures are detected and they become invalid discovery results
- Service Discovery is not a single point of failure by itself due to replicated Eureka Servers

Overview


us-east-1c
us-east-1d
us-east-1e
Application Service
Eureka Client
Register
Renew
Cancel
Get Registry
Eureka Server
Replicate
Eureka Server
Replicate
Eureka Server
Make Remote Call
Get Registry
Make Remote Call
Get Registry
Application Client
Eureka Client
Application Client
Eureka Client
Make Remote Call
Application Service
Eureka Client

Eureka Mechanics

- Client Registration
- The first heartbeat happens 30s (described earlier) after startup so the instance doesn’t appear in the Eureka registry before this interval.

- Server Response Cache
- The server maintains a response cache that is updated every 30s (configurable by eureka.server.responseCacheUpdateIntervalMs). Even if the instance is just registered, it won’t appear in the result of a call to the /eureka/apps REST endpoint. However, the instance may appear on the Eureka Dashboard just after registration. This is because the Dashboard bypasses the response cache used by the REST API.

- So, it may take up to another 30s for other clients to discover the newly registered instance.

Eureka Mechanics

- Client Cache Refresh

Eureka client maintain a cache of the registry information. This cache is refreshed every 30s (described earlier). So, it may take another 30s before a client decides to refresh its local cache and discover other newly registered instances.

- Load Balancer Refresh
- The load balancer used by Ribbon gets its information from the local Eureka client. Ribbon also maintains a local cache to avoid calling the client for every request. This cache is refreshed every 30s (configurable by ribbon.ServerListRefreshInterval). So, it may take another 30s before Ribbon can make use of the newly registered instance.

- In the end, it may take up to 2min before a newly registered instance starts receiving traffic from the other instances.

Eureka Replication

- When the Eureka server comes up, it tries to get all of the instance registry information from a neighboring node.

- If there is a problem getting the information from a node, the server tries all of the peers before it gives up.

- If the server is able to successfully get all of the instances, it sets the renewal threshold that it should be receiving based on that information.

- If any time, the renewals falls below the percent configured for that value, the server stops expiring instances to protect the current instance registry information.

Eureka Reliability

- The HA strategy seems to be one main Eureka server (**server1**) with backup(s) (**server2**).

- A client is provided with a list of Eureka servers through config (or DNS or /etc/hosts)

- Client attempts to connect to **server1**; at this point, **server2** is sitting idle.
- I
- n case **server1** is unavailable, the client tries the next one from the list.

- When **server1** comes back online, client goes back to using server1.

Eureka Reliability

- The HA strategy seems to be one main Eureka server (**server1**) with backup(s) (**server2**).

- A client is provided with a list of Eureka servers through config (or DNS or /etc/hosts)

- Client attempts to connect to **server1**; at this point, **server2** is sitting idle.
- I
- n case **server1** is unavailable, the client tries the next one from the list.

- When **server1** comes back online, client goes back to using server1.

Spring Eureka

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
// note that this will also start the Eureka dashboard
// dashboard is accessible at the specified application server port
@EnableConfigServer
public class SpringMicroservicesConfigServerApplication {

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

Spring Eureka

- Configuration in application.properties:

```
// server port
server.port=8761

// do not register with other Eureka servers
eureka.client.register-with-eureka=false

// do not try to fetch data from other Eureka servers
eureka.client.fetch-registry=false
```

Spring Eureka

- Fault Tolerance by adding multiple instances in the “hosts” file for 127.0.0.1:

```
// in hosts
127.0.0.1 reduntantserver1
127.0.0.1 reduntantserver2
// first instance in application-reduntantserver1.properties
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.instance.hostname=reduntantserver1
eureka.client.serviceUrl.defaultZone=http://reduntantserver2:8762/eureka
// second instance in application-reduntantserver2.properties
server.port=8762
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.instance.hostname=reduntantserver2
eureka.client.serviceUrl.defaultZone=http://reduntantserver1:8761/eureka
```

Spring Eureka

```
@SpringBootApplication
@EnableEurekaClient
@RestController
public class SpringMicroservicesEurekaClient2Application {
    @Autowired
    private EurekaClient client;
    @RequestMapping("/serviceinfo")
    public String serviceInfo(){
        InstanceInfo instance = client.getNextServerFromEureka(“myFirstClient", false);
        return instance.getHomePageUrl();
    }
    public static void main(String[] args) {
        SpringApplication.run(SpringMicroservicesEurekaClient2Application.class, args);
    }
}
```

Spring Eureka

- Configuration in application.properties:

```
// client port
server.port=8888

// the name of the application
spring.application.name=mySecondClient

 // first client also needs to have a name specified (“myFirstClient” in this example)
```

Ribbon

- Ribbon provides software load balancers to communicate with cluster of servers. The load balancers provide the following basic functionalities:
  - Supply the public DNS name or IP of individual servers to communication client
  - Rotate among a list of servers according to certain logic

- Certain load balancers can also provide advanced features like
  - Establishing affinity between clients and servers by dividing them into zones (like racks in a data center) and favor servers in the same zone to reduce latency
  - Keeping statistics of servers and avoid servers with high latency or frequent failures
  - Keeping statistics of zones and avoid zones that might be in outage

Ribbon

- Ribbon Components:
  - Rule - a logic component to determine which server to return from a list
  - Ping - a component running in background to ensure liveness of servers
  - ServerList - this can be static or dynamic. If it is dynamic (as used by DynamicServerListLoadBalancer), a background thread will refresh and filter the list at certain intervals

Spring Ribbon

```
@SpringBootApplication
 @RestController
 public class SpringMicroservicesMyServiceApplication  {
 @RequestMapping("/execute")
    public String execute(){
       return "Hello from the port " + this.port;
    }
  // required by ribbon to evaluate service status
 @RequestMapping("/")
    public String status(){
       return "Up";
    }
  public static void main(String[] args) {
       SpringApplication.run(SpringMicroservicesMyServiceApplication.class, args);
    }
 }
```

Ribbon

- We start a server instance several times
- The service that will be load balanced needs to be named so that the Ribbon instances can find it based on the name
- We will a balanced rest template that calls the service by name
- Calls will be distributed throughout all the registered instances

Spring Ribbon

```
@RestController
@RibbonClient(name="my-service",configuration=SimpleServiceConfiguration.class)
public class SpringMicroservicesRibbonApplication {
    @Autowired
    public RestTemplate restTemplate;
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
    @RequestMapping("/startClient")
    public String startClient(){
        return this.restTemplate.getForObject("http://my-service/execute",String.class);
    }
}
```

Spring Ribbon

```
public class SimpleServiceConfiguration {

  @Autowired
  public IClientConfig ribbonClientConfig;

  @Bean
  public IPing ping(IClientConfig config){
    return new PingUrl();
  }

  @Bean
  public IRule rule(IClientConfig config){
    return new AvailabilityFilteringRule();
  }
}
```

Spring Ribbon

- Configuration in application.properties:

```
my-service.ribbon.eureka.enable=false

my-service.ribbon.ServerListRefreshInterval=15000

my-service.ribbon.listOfServers=localhost:7777,localhost:8888,localhost:9999
```

Spring Feign Client

- Instead of using RestTemplate, we can the Feign client which is basically a wrapper over the rest client.

- As we specified the application name that should be balanced in the URL we can make the same configuration for Feign client to automatically call the specified instance by name

```
 // in main
@EnableFeignClients


 // in an service interface
@FeignClient(name  = " my-service ", fallback = MyServiceRequestFallback.class)
 public interface ProductRequest {


    @RequestMapping(method = RequestMethod.GET, path = "/my-service-execute")
    MyServiceResult getMyService();


}
```

THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com

This is a free course from LearnStuff.io
– not for commercial use –