

Practice 8: RESTful Webservices, Spring Cloud

Service discovery and Fault tolerance

Open projects config-server, discount-service and subscription

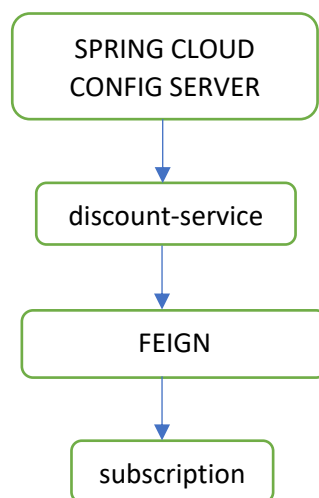
1.

Run project config-server on port 8070, discount-service on port 8081 and subscription on port 8080. Test request with Swagger or Postman.

<http://localhost:8080/swagger-ui.html>

<http://localhost:8080/subscription/coach/James/sport/tennis>

<http://localhost:8081/discount>



config-service configures service *discount-service* (sets the properties *discount.month* and *discount.year*). *subscription* services calls *discount-service* using FEIGN -proxy.

Service discovery and registration pattern

- Microservices locate each other and share information between them.
- Dynamically new instances are started or stopped, new instances register and share the host and port they are running on.
- A load-balancer distribute workload between instances.

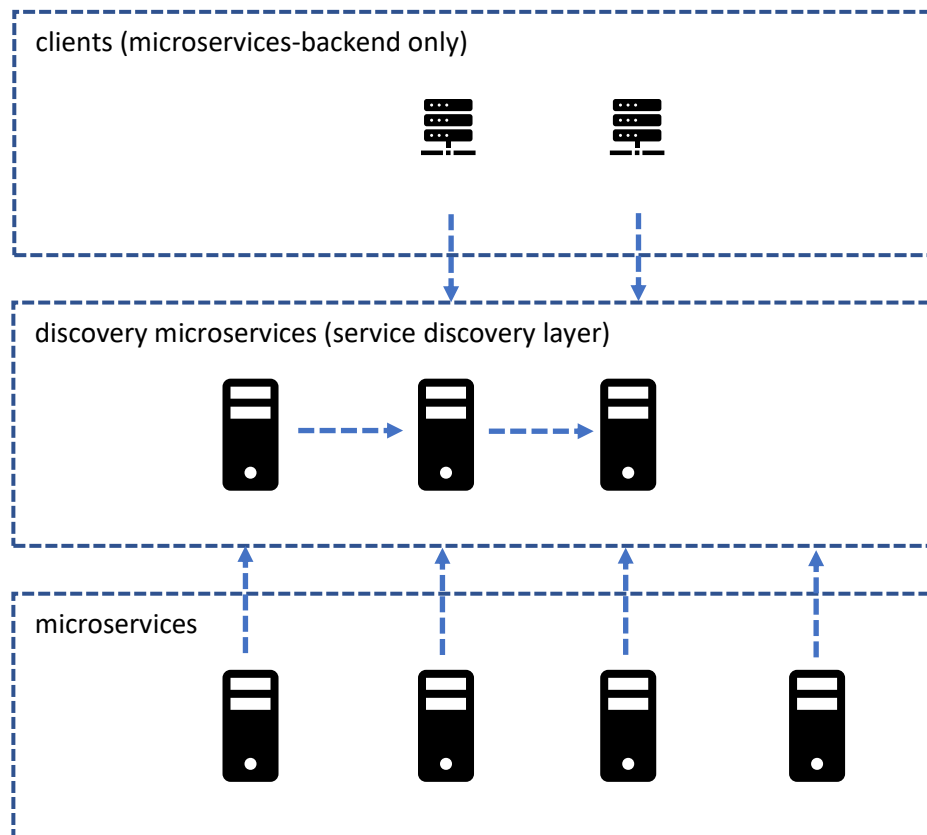
Traditional approach: Addresses are maintained by a **central server**. New instances are **registered** at startup and periodically send information about their health. If for a given timeframe an instance registered on the server is not available, the corresponding address entry will be **removed**.

Exposing logical service names instead of IPs ensures security and flexibility, one may easily move microservices on other hosts, without the need to change client requests.

Load balancers (with different strategies, round-robin, geographical location etc.) are responsible with receiving requests from clients and distribute requests to microservices using microservices names and addresses. It is common to run a primary load-balancer and a secondary load-balancer prepared to serve the requests in case of failure of the primary load-balancer. **[1]**

Disadvantages in maintain a primary/secondary load balancer:

- Difficult to scale
- Single point of failure
- Manually manage IP addresses.



Service discovery and registration:

- Key-value store for addresses, API to query and update addresses.
- P2P communication between service discovery agents, Gossip communication protocol
- Resilient, Fault-tolerant
- Dynamic configuration
- Server load-balancing, client service interacts with the service discovery layer.

Client load balancing

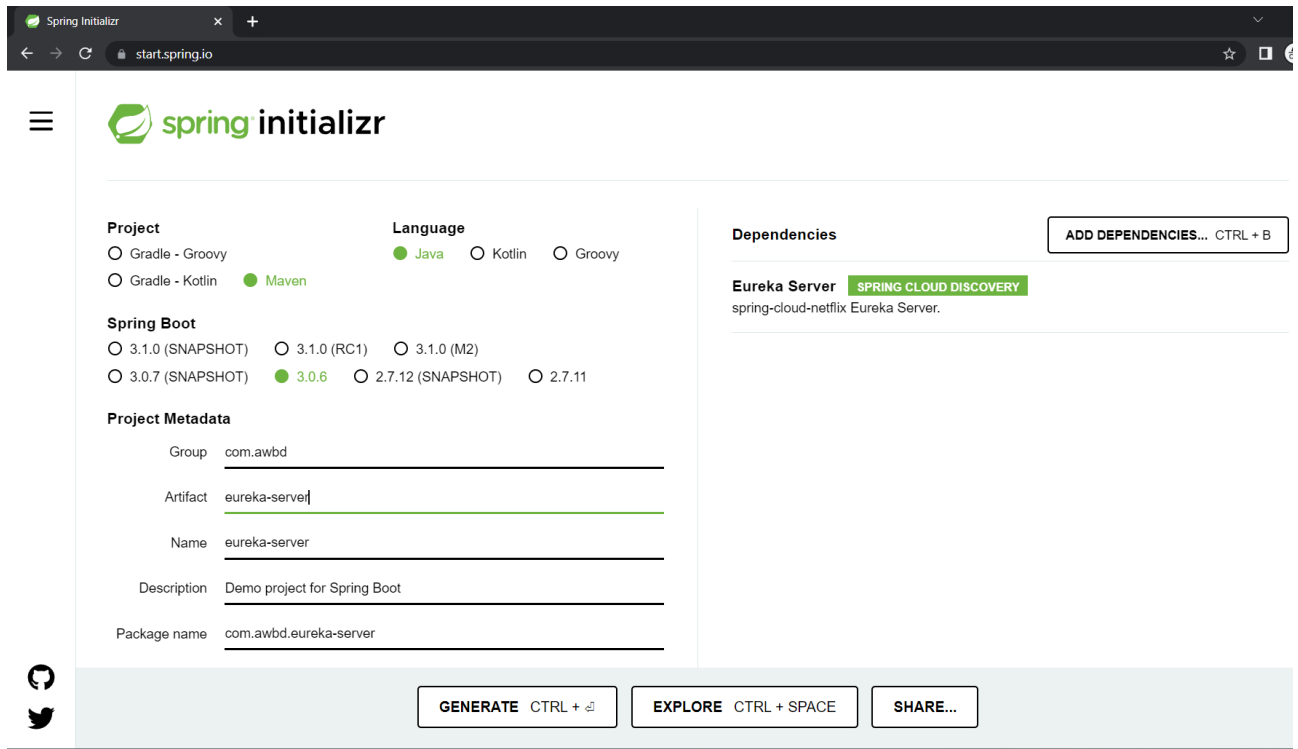
- Scenario: **subscription** microservice is requesting a **discount** microservice instance from service discovery layer. It then finds that there are two instances available for microservice with logical name discount. **subscription** microservice may cache the addresses of **discount** microservice instances and apply its own load-balancing strategy.
- Periodically client-side cache is updated invoking service discovery layer.
- If one of the addresses cached by the client is not available, the client invokes service discovery layer and updates its address list.

Service discovery agents: Netflix Eureka, Apache Zookeeper.

Load-balancer: Spring cloud cloud-balancer, replacement for Ribbon

Microservice-client: Netflix Feign

2. Create a new project with Group Id: *com.awbd* and Artifact Id: *eureka-namingserver*. Add dependencies: Eureka Server and Actuator.



The screenshot shows the Spring Initializr web application interface. The 'Project' section has 'Gradle - Maven' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '3.0.6' selected. The 'Project Metadata' section has the following values: Group: *com.awbd*, Artifact: *eureka-server*, Name: *eureka-server*, Description: *Demo project for Spring Boot*, and Package name: *com.awbd.eureka-server*. The 'Dependencies' section has 'Eureka Server' and 'SPRING CLOUD DISCOVERY' selected. The 'ADD DEPENDENCIES... CTRL + B' button is visible. At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

3. Add annotation *@EnableEurekaServer* on Spring Boot main application.

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaNamingserverApplication {

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

}
```

4. Add application name and config server url in application.properties

```
spring.application.name=eurekaserver
spring.config.import=optional:configserver:http://localhost:8070/
```

5. Add dependency *spring-cloud-starter-config* to load eurekaserver configuration from a config server:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

6.

Make sure the file `eureka-server.properties` is present on config server store (git or local directory):

```
server.port=8761
eureka.instance.host=localhost
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.service-
url.defaultZone=http://${eureka.instance.host}:${server.port}/eureka/
```

Eureka [3][4][5]

Eureka is a service registry (or *Discovery Server*) that holds the information about client-service applications. Every micro service will register into the Eureka server and Eureka server registers all the client applications running on each port and IP address.

Eureka server is not fetching the addresses registry as it itself holds the registry, also Eureka server is not adding itself into the registry.

Start Eureka server and test:

<http://localhost:8761/>

7.

In project discount add dependency for Eureka client:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

8.

Annotate class `DiscountServiceApplication` with `@EnableDiscoveryClient` annotation to peek up the implementation of the discovery client on the classpath.

```
@SpringBootApplication
@RefreshScope
@EnableDiscoveryClient
public class DiscountApplication {

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

}
```

9.

Modify discount-dev.properties and discount-prod.properties on config server files:

```
server.port=8081
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.prefer-ip-address=true

management.info.env.enabled=true
info.app.name=discount
info.app.description=subscription discount
info.app.version=1
```

```
server.port=8082
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.prefer-ip-address=true

management.info.env.enabled=true
info.app.name=discount
info.app.description=subscription discount
info.app.version=2
```

10.

Test that dev and prod instances are registered with Eureka

Add Actuator shutdown endpoint on discount service:

```
management.endpoint.shutdown.enabled=true
```

Shutdown on instance and check again Eureka registry.

Add necessary dependencies/annotations to register subscription service with Eureka. Add in application.properties:

```
spring.application.name=subscription
```

Add Actuator dependency and mappings.

11.

Add field versionId in class Discount, in project discount-service:

```
@Setter
@Getter
public class Discount {

    private String versionId;
    private int month;
    private int year;
}
```

12. Instantiate field `versionId` with environment variable `info.app.version`

```
@Autowired
Environment environment;

@GetMapping("/discount")
public Discount getDiscount() {

    return new
Discount(configuration.getMonth(), configuration.getYear(),
environment.getProperty("info.app.version"));
}
```

13. Add class `Discount` in project `Subscription`:

```
@Data
public class Discount {
    private String versionId;
    private int month;
    private int year;
}
```

14. Add feign dependency [6] and `@EnableFeignClients` annotation:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableHypermediaSupport(type =
EnableHypermediaSupport.HypermediaType.HAL)
@EnableEurekaClient
@EnableFeignClients
public class SubscriptionApplication {

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

15. Modify class `DiscountServiceProxy` in service `Subscription`:

```
@FeignClient(value = "discount")
public interface DiscountServiceProxy {
    @GetMapping(value="/discount", consumes = "application/json")
    Discount findDiscount();
}
```

16.

In *SubscriptionController* log versionId of *Discount* object.

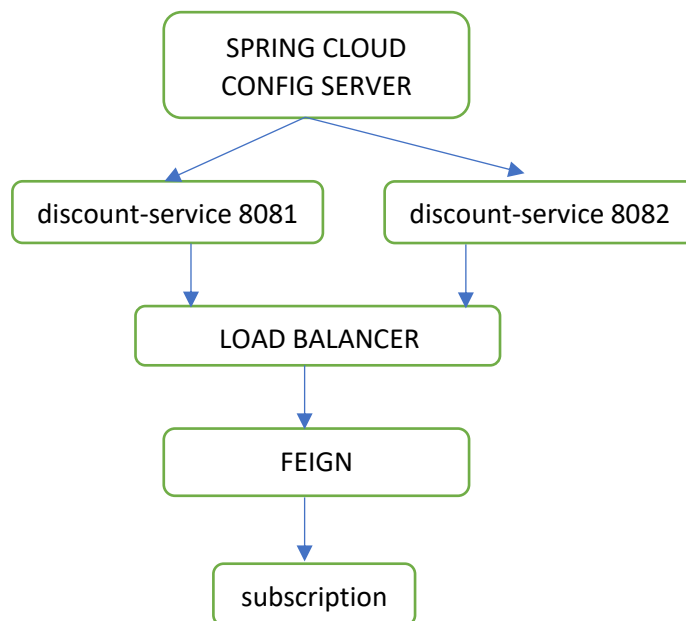
```
@RestController
@Slf4j
public class SubscriptionController {
    @Autowired
    SubscriptionService subscriptionService;

    @Autowired
    DiscountServiceProxy discountServiceProxy;

    @GetMapping("/subscription/coach/{coach}/sport/{sport}")
    Subscription findByCoachAndSport(@PathVariable String coach,
                                     @PathVariable String sport){
        Subscription subscription =
        subscriptionService.findByCoachAndSport(coach, sport);

        Discount discount = discountServiceProxy.findDiscount();
        log.info(discount.getVersionId());
        subscription.setPrice(subscription.getPrice() * (100 -
        discount.getMonth())/100);

        return subscription;
    }
}
```

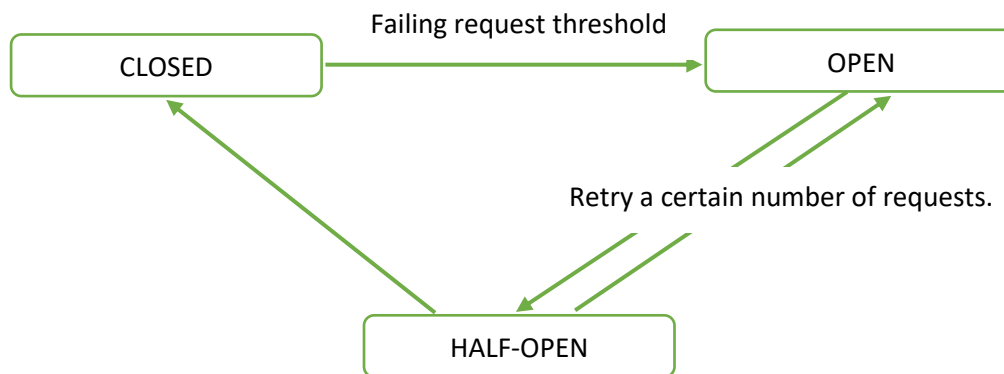


Resilience

- Avoid cascading failures Circuit Breaker Pattern.
- Add fallback behavior.
- Add self-healing capabilities (Retry and limit the number of calls).

Hystrix or Resilience4j [7]

In Circuit Breaker Pattern, Circuit Breaker is monitoring request. When reaching a threshold of failing request it decides to prevent further calls to failing microservices and replace response with a fallback method. Failing microservices gain time to recover, since for a certain amount of time no incoming request pass the circuit breaker. Periodically, circuit breaker checks temporarily unavailable microservices.



20.

Add Resilience4j dependencies:

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-circuitbreaker</artifactId>
</dependency>
```

21.

Add @CircuitBreaker annotation on *getSubscription* method.

```
@GetMapping("/subscription/{subscriptionId}")
@CircuitBreaker(name="discountById", fallbackMethod =
"getSubscriptionFallback")
public Subscription getSubscription(@PathVariable Long subscriptionId) {

    Subscription subscription =
    subscriptionService.findById(subscriptionId);

    Discount discount = discountServiceProxy.findDiscount();
    log.info(discount.getInstanceId());
    subscription.setPrice(subscription.getPrice() * (100 -
discount.getMonth())/100);

    return subscription;
}
```


21.

Add Circuit Breaker properties:

```
resilience4j.circuitbreaker.instances.discountById.minimum-number-of-calls=5
resilience4j.circuitbreaker.instances.discountById.failure-rate-threshold=70
resilience4j.circuitbreaker.instances.discountById.wait-duration-in-open-state=10000
resilience4j.circuitbreaker.instances.discountById.permitted-number-of-calls-in-half-open-state=1
resilience4j.circuitbreaker.configs.default.register-health-indicator=true
```

22.

Add fallback method:

```
private Subscription getSubscriptionFallback(Long subscriptionId,
    Throwable throwable) {

    Subscription subscription =
    subscriptionService.findById(subscriptionId);
    return subscription;

}
```

B

- [1] <https://spring.io/guides/gs/service-registration-and-discovery/>
- [2] <https://www.baeldung.com/spring-cloud-rest-client-with-netflix-ribbon>
- [3] <https://spring.io/guides/gs/service-registration-and-discovery/>
- [4] <https://www.baeldung.com/spring-cloud-netflix-eureka>
- [5] <https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka>
- [6] <https://www.baeldung.com/spring-cloud-openfeign>
- [7] <https://resilience4j.readme.io/docs>