Servis Keşfi ve Eureka

order-service'in inventory-service'i artık sabit kodlanmış bir URL yerine bir İSİMLE bulması: Eureka Server (merkezi kayıt defteri), Eureka Client (spring.application.name'in keşif anahtarına dönüşmesi), DiscoveryClient ile doğrudan sorgu, @LoadBalanced RestClient ile isimle çağrı yapma, heartbeat/eviction/self-preservation modu, ve Eureka'nın CAP teoreminin AP tarafını seçmesi.

Orta 24 dk

Son güncelleme: 6 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/service-discovery-eureka

---

Service Discovery ve Eureka

Mikroservisler kategorisinin "wave 1"i (`microservices-fundamentals` , `spring-boot-microservice-basics` , `inter-service-communication`) `order-service` 'in `inventory-service` 'i BULMASI için sabit kodlanmış bir URL kullandı (`@Value("${services.inventory-service.url}")` , bkz. "Servisler Arası İletişim" dersinin "order-service'ten inventory-service'e: RestClient ile Senkron Çağrı" bölümü). Bu, İKİ sabit servis için gayet iyi çalışır -- ama gerçek dünyada servisler ÇOĞALIR (aynı servisin birden fazla kopyası, yük dengeleme için), TAŞINIR (konteynerler yeniden başladığında IP'ler değişir), ve ÖLÇEKLENİR. Bu ders, o probleme Java /Spring ekosisteminin klasik cevabını -- Service Discovery ve Netflix Eureka'yı -- tanıtıyor.

Service Discovery Nedir?

Service Discovery (servis keşfi), servislerin birbirini SABİT bir adresle değil, bir İSİMLE bulmasını sağlayan bir desendir. Merkezinde bir KAYIT DEFTERİ (registry) vardır: her servis instance'ı başladığında kendini bu deftere KAYDEDER ("ben `inventory-service` 'im, şu anda şu host:port'ta çalışıyorum"), ve bir servis başka bir servisi ÇAĞIRMAK istediğinde, sabit bir adres yerine deftere " `inventory-service` şu anda nerede?" diye SORAR.

Neden Var?

`@Value` ile sabit kodlanmış bir URL, `inventory-service` TEK bir instance olarak TEK bir sabit adreste çalıştığı sürece sorunsuzdur. Ama gerçek üretim ortamlarında: (1) yük artışına karşı AYNI servisin BİRDEN FAZLA kopyası (instance) çalışır -- hangisine gidileceği sabit kodlanamaz; (2) konteynerler/bulut ortamları IP adreslerini SIK SIK değiştirir -- her yeniden başlatmada `application.yml` 'i elle güncellemek pratik değildir; (3) yeni bir servis

eklendiğinde, ONU çağıracak HER servisin yapılandırmasının güncellenmesi gerekir. Service Discovery, bu üç sorunu da TEK bir merkezi kayıt defteriyle çözer -- servisler birbirini İSİMLE bulur, sabit adresle değil.

Tarihçe

Eureka, Netflix'in kendi devasa mikroservis altyapısı için 2012 civarında geliştirdiği ve açık kaynağa aktardığı bir servis keşif aracıdır (Netflix OSS'in bir parçası -- aynı dönemde Netflix, Hystrix/Ribbon gibi başka mikroservis araçlarını da açtı). Spring Cloud Netflix (2015), Eureka'yı Spring Boot uygulamalarına birkaç satır yapılandırmayla eklemeyi mümkün kıldı -- bu ders, tam olarak o entegrasyonu kullanıyor. Önemli bir dürüstlük notu: Netflix, kendi iç altyapısında Eureka 2.0'ı 2018 civarında DURDURDU ve artık kendi araçlarını kullanmıyor -- ama Eureka 1.x, Spring Cloud ekosisteminde hâlâ YAYGIN ŞEKİLDE kullanılıyor ve aktif olarak desteklenmeye devam ediyor. Kubernetes gibi platformların KENDİ yerleşik servis keşif mekanizması olduğu için, Kubernetes üzerinde çalışan projelerde Eureka genellikle GEREKMEZ -- Eureka en çok, Kubernetes'in dışında (VM'lerde, klasik sunucularda) çalışan Spring uygulamaları için değerlidir.

Eureka Server: Merkezi Kayıt Defteri

Eureka Server, `order-service` / `inventory-service`'ten TAMAMEN ayrı, kendi başına çalışan BAĞIMSIZ bir Spring Boot uygulamasıdır -- ne bir iş kuralı içerir, ne bir veritabanına bağlanır. Tek görevi: hangi servislerin kayıtlı olduğunu ve hangi adreste çalıştıklarını bilmektir. `@EnableEurekaServer` ONU Eureka Server yapan TEK anotasyondur.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

// A THIRD Spring Boot application in this course, next to order-service and
// inventory-service -- but this one isn't a business microservice at all. It's the
// central registry (see "Eureka Server: A Central Registry") that order-service and
// inventory-service will register themselves with, and look each other up through.
//
// @EnableEurekaServer is the ONLY thing that makes this a Eureka server -- everything
// else (dependency: spring-cloud-starter-netflix-eureka-server, its own
// EurekaServerConfig.yml) is standard Spring Boot, exactly like order-service's own
// @SpringBootApplication entry point (see the Spring Boot Microservice Basics lesson's
// "A Microservice's Entry Point: @SpringBootApplication" section).
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

```
# eureka-server's own application.yml -- runs as its own independent process, on its
# own well-known port (8761 is Eureka's traditional default, used by convention across
# almost every Spring Cloud tutorial and this course follows it).

server:
  port: 8761                          # never 8080 (learning-platform), 8081
                                       # (order-service), or 8082 (inventory-service)

spring:
  application:
    name: eureka-server               # this service's own identity too -- even the
                                       # registry has a name, though it doesn't register
                                       # itself with anyone (see below)

eureka:
  client:
    register-with-eureka: false       # the server does NOT register itself as a client
                                       # of itself -- in a single-node setup, there is no
                                       # "another" Eureka server to talk to
    fetch-registry: false             # and it doesn't need to fetch a registry either,
                                       # since IT IS the registry
  server:
    enable-self-preservation: true    # default -- see "Heartbeats, Eviction, and
                                       # Self-Preservation Mode" for what this protects
                                       # against
```

> `register-with-eureka: false` ve `fetch-registry: false`, Eureka Server'ın KENDİ KENDİNE bir client gibi davranmasını ENGELLER -- tek düğümlü (single-node) bir kurulumda sunucu, kendini kaydedecek ya da kendi kayıt defterini "fetch" edecek başka bir sunucuya ihtiyaç duymaz. (Üretimde birden fazla Eureka Server düğümü birbirini replike edebilir -- bu durumda bu ayarlar `true` olurdu, ama bu ders tek düğümlü kuruluma odaklanıyor.)

Eureka Client: Servisleri Kaydetmek

`order-service` ve `inventory-service`, `spring-cloud-starter-netflix-eureka-client` bağımlılığını ekleyip `application.yml`'lerine Eureka Server'ın adresini (`eureka.client.service-url.defaultZone`) yazarak Eureka CLIENT'ına dönüşür. Kritik bir nokta: `spring.application.name` (bkz. "Mikroservis Yapılandırma" dersinin "Kendi `application.yml`'i: Port, Uygulama Adı ve Veritabanı" bölümü) ARTIK yalnızca loglarda görünen bir isim DEĞİL -- bu, diğer servislerin ONU bulmak için kullanacağı GERÇEK anahtardır.

```
# order-service's application.yml, with the Eureka CLIENT settings added on top of
# everything from the Spring Boot Microservice Basics lesson's "Its Own
# `application.yml`: Port, Application Name, and Database" section -- nothing already
# there changes, this is purely additive. inventory-service gets the exact same
# `eureka.client` block in its own application.yml (with its own
# spring.application.name, unchanged).

server:
  port: 8081

spring:
  application:
    name: order-service               # UNCHANGED -- but now this name is what OTHER
                                      # services will look order-service up BY, via
                                      # Eureka, instead of a hardcoded host:port (see
                                      # "Discovering Services with DiscoveryClient")
  datasource:
    url: jdbc:postgresql://localhost:5432/orders_db
    username: orders_user
    password: ${ORDERS_DB_PASSWORD}

management:
  endpoints:
    web:
      exposure:
        include: health

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/   # where the Eureka SERVER lives --
                                                       # the ONLY new piece of information
                                                       # order-service needs to start
                                                       # registering itself
    # register-with-eureka and fetch-registry default to true for a regular client --
    # unlike eureka-server's own config above, order-service DOES want both: it
    # registers ITSELF, and it fetches the registry to look up OTHER services.
  instance:
    lease-renewal-interval-in-seconds: 30    # default -- how often order-service sends
                                             # a heartbeat (see "Heartbeats, Eviction,
                                             # and Self-Preservation Mode")
```

Servisleri Dinamik Olarak Keşfetmek: DiscoveryClient

`DiscoveryClient`, kayıt defterine DOĞRUDAN soru sormanın düşük seviyeli yoludur -- "şu anda `inventory-service` adı altında hangi instance'lar kayıtlı?" `Spring Cloud`'un Eureka client bağımlılığı, hiçbir ekstra yapılandırma olmadan bir `DiscoveryClient` bean'ini OTOMATİK olarak sağlar.

```
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Map;

// DiscoveryClient is the LOW-LEVEL way to ask the registry a question directly: "which
// instances are currently registered under this service name?" Spring Boot autoconfigures
// a DiscoveryClient bean automatically once eureka.client dependencies are on the
// classpath -- nothing else needs to be wired up.
//
// This is NOT how order-service will normally talk to inventory-service (see "Calling a
// Service by Name with a Load-Balanced RestClient" for the idiomatic way) -- but it's
// useful for diagnostics, and for understanding exactly what Eureka is tracking under the
// hood: host, port, and a small metadata map per instance.
@RestController
class DiscoveryClientExample {

    private final DiscoveryClient discoveryClient;

    DiscoveryClientExample(DiscoveryClient discoveryClient) {
        this.discoveryClient = discoveryClient;
    }

    // GET /discovered-services/{serviceName} -- lists every currently-registered
    // instance of a given service name, straight from the Eureka registry.
    @GetMapping("/discovered-services/{serviceName}")
    List<Map<String, Object>> listInstances(@PathVariable String serviceName) {
        List<ServiceInstance> instances = discoveryClient.getInstances(serviceName);
        return instances.stream()
                .map(instance -> Map.<String, Object>of(
                        "host", instance.getHost(),
                        "port", instance.getPort(),
                        "uri", instance.getUri().toString(),
                        "metadata", instance.getMetadata()
                ))
                .toList();
    }

    // GET /discovered-services -- lists every service NAME currently registered with
    // Eureka, regardless of instance count (useful to see the whole registry at a
    // glance).
    @GetMapping("/discovered-services")
    List<String> listServiceNames() {
        return discoveryClient.getServices();
    }
}
```

> `DiscoveryClient`, günlük servisler arası çağrılar için DOĞRU araç DEĞİLDİR -- asıl kullanım alanı teşhis (diagnostics) ve kayıt defterinin o an ne gördüğünü anlamaktır. Servisler arası GERÇEK çağrılar için "Load-Balanced RestClient ile İsimle Çağrı Yapmak" bölümündeki yaklaşım kullanılmalıdır.

Load-Balanced RestClient ile İsimle Çağrı Yapmak

`@LoadBalanced` anotasyonu, bir `RestClient.Builder` bean'ini KAYIT DEFTERİNDEN HABERDAR hâle getirir -- bu builder'dan oluşturulan bir `RestClient`, `http://inventory-service` gibi bir "adresi" GERÇEK bir host adı olarak DEĞİL, bir SERVİS İSMİ olarak yorumlar; Spring Cloud LoadBalancer bu çağrıyı YAKALAYIP `DiscoveryClient`'a sorar ve o an kayıtlı instance'lardan BİRİNİ seçer.

```
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestClient;

// This is the ONE change that turns a plain RestClient into a discovery-aware one: a
// RestClient.Builder bean annotated @LoadBalanced. Once this bean exists, ANY RestClient
// built from it can call a service by NAME (e.g. "http://inventory-service") instead of
// a real host and port -- Spring Cloud LoadBalancer intercepts the call, asks the
// DiscoveryClient which instances are registered under that name, and picks one.
//
// Compare this to the plain RestClient.builder() used directly in the Inter-Service
// Communication lesson's StockClient (see its "From order-service to inventory-service:
// A Synchronous Call with RestClient" section) -- the calling code barely changes (see
// StockClientWithDiscovery), but the base URL stops being a real network address and
// becomes a logical service name instead.
@Configuration
class LoadBalancedRestClientConfig {

    @Bean
    @LoadBalanced
    RestClient.Builder loadBalancedRestClientBuilder() {
        return RestClient.builder();
    }
}
```

Bu, "Servisler Arası İletişim" dersindeki `StockClient`'ın sabit kodlanmış `@Value` URL'sinin YERİNİ alıyor -- geri kalan TÜM mantık (404 ile bağlantı hatasının ayrıştırılması, `InventoryServiceUnavailableException`'a çevrilmesi) DEĞİŞMİYOR:

```
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.ResourceAccessException;
import org.springframework.web.client.RestClient;
import org.springframework.stereotype.Component;

// The Eureka-aware rewrite of the Inter-Service Communication lesson's StockClient (see
// its "From order-service to inventory-service: A Synchronous Call with RestClient"
// section). Two things changed, and only two: the constructor now takes the
// @LoadBalanced RestClient.Builder (see LoadBalancedRestClientConfig) instead of reading
// a URL from @Value, and the base URL is now the service NAME ("http://inventory-service"),
// not a real host:port. Every other line -- the 404-vs-connection-failure distinction, the
// InventoryServiceUnavailableException translation -- stays EXACTLY the same, because that
// logic was never about WHERE inventory-service lives, only about HOW to react to what it
// says.
@Component
class StockClientWithDiscovery {

    private final RestClient restClient;

    StockClientWithDiscovery(RestClient.Builder loadBalancedRestClientBuilder) {
        // "inventory-service" here is NOT a hostname the JVM can resolve on its own --
        // Spring Cloud LoadBalancer intercepts it and substitutes a real host:port that
        // DiscoveryClient currently has registered for that name. If inventory-service
        // scales to three instances, this ONE line of code does not change at all.
        this.restClient = loadBalancedRestClientBuilder.baseUrl("http://inventory-service").
build();
    }

    StockCheckResponse checkStock(String productName) {
        try {
            return restClient.get()
                    .uri("/inventory/{productName}", productName)
                    .retrieve()
                    .body(StockCheckResponse.class);
        } catch (HttpClientErrorException.NotFound e) {
            // Same meaning as before: inventory-service answered, it just doesn't know
            // this product.
            return new StockCheckResponse(productName, 0);
        } catch (ResourceAccessException e) {
            // Same meaning as before too -- but now this ALSO covers the case where
            // Eureka has NO instances registered for "inventory-service" at all (the
            // load balancer has nothing to route to), not just a single unreachable
            // host.
            throw new InventoryServiceUnavailableException("inventory-service is not
reachable", e);
        }
    }
}

// Unchanged from the Inter-Service Communication lesson -- what CAN fail about a service
// call didn't change just because we now look the service up by name.
class InventoryServiceUnavailableException extends RuntimeException {
    InventoryServiceUnavailableException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
// Unchanged from the Inter-Service Communication lesson -- discovering inventory-service
// by name doesn't change WHAT it tells order-service, only HOW order-service finds it to
// ask.
record StockCheckResponse(String productName, int quantityInStock) {
}
```

> `inventory-service`'in birden fazla instance'ı çalışırken, `@LoadBalanced RestClient` HANGİ instance'a gideceğini SİZİN İÇİN seçer -- bu seçim, `StockClientWithDiscovery`'nin KENDİSİNDE tek bir satır bile değiştirmeden gerçekleşir. Load balancing'in kendisi, Eureka'nın DEĞİL, Spring Cloud LoadBalancer'ın işidir -- Eureka yalnızca HANGİ instance'ların var olduğunu SÖYLER, aralarında SEÇİM yapmaz.

Heartbeat, Eviction ve Self-Preservation Modu

Bir Eureka client, kayıtlı KALMAK için Eureka Server'a düzenli aralıklarla (varsayılan 30 saniye) bir "heartbeat" (kalp atışı) gönderir -- buna "lease renewal" (kira yenileme) denir. Bir client belirli bir süre (varsayılan 90 saniye) boyunca heartbeat GÖNDEREMEZSE, Eureka Server onu kayıt defterinden SİLER ("eviction"). Ama BURADA ilginç bir davranış var: eğer Eureka Server ÇOK SAYIDA client'ın AYNI ANDA heartbeat göndermeyi kestiğini görürse (beklenen heartbeat oranının belirgin şekilde altına düşerse), bunu "birden fazla servis GERÇEKTEN aynı anda çöktü" olarak DEĞİL, "muhtemelen BENİM ağ bağlantım/kendi durumum sorunlu" olarak YORUMLAR -- ve "self-preservation mode"a (kendini koruma modu) girip HİÇBİR instance'ı SİLMEYİ durdurur.

> Self-preservation modu, özellikle YEREL geliştirme ortamlarında (tek makinede birkaç servis çalıştırılırken) KAFA KARIŞTIRICI olabilir -- bir servisi kapatsanız bile, Eureka Server onu bir süre (bazen dakikalarca) "kayıtlı" gibi GÖSTERMEYE devam edebilir. Bu bir hata DEĞİLDİR -- Eureka'nın, bir ağ bölünmesi sırasında SAĞLIKLI instance'ları YANLIŞLIKLA silmemek için bilinçli bir tasarım kararıdır.

Eureka'nın CAP Teoremindeki Yeri: AP Sistemi

"Microservices Temelleri" dersindeki CAP Teoremine Kısa Bir Bakış bölümünü hatırlayın: dağıtık bir sistem, bir ağ bölünmesi (network partition) sırasında Tutarlılık (Consistency) ile Erişilebilirlik (Availability) arasında SEÇİM yapmak zorundadır. Eureka bilinçli olarak AP tarafını seçer: her Eureka Server düğümü, EN GÜNCEL bilgiye sahip olmayı garanti etmek yerine, HER ZAMAN bir cevap vermeyi (kısmen eski bir kayıt defteriyle bile olsa) tercih eder. Self-preservation modu da bu felsefenin bir SONUCUdur -- Eureka, "belki birkaç kayıt eski" riskini, "sağlıklı servisleri yanlışlıkla sil" riskine TERCİH eder.

Best Practices

- **Servisler arası çağrılarda sabit kodlanmış host:port yerine servis İSMİYLE (`@LoadBalanced RestClient` ile) çağrı yapın** -- yatay ölçeklendirme ve IP değişiklikleri kod DEĞİŞİKLİĞİ gerektirmez.
- **`DiscoveryClient`'ı yalnızca teşhis/gözlemleme amaçlı kullanın**, günlük servisler arası çağrılar için DEĞİL -- `@LoadBalanced RestClient` bu işi zaten SİZİN İÇİN yapar.
- **Yerel geliştirmede self-preservation modunun KAFA KARIŞTIRICI davranışını (kapatılan servisin bir süre "kayıtlı" görünmesi) BEKLEYİN** -- bu bir hata değil, Eureka'nın AP tasarımının doğal bir sonucudur.
- **Eureka Server'ın kendisini de bir `spring.application.name` ile adlandırın**, kayıt defterinde görünmese bile -- loglar ve gelecekteki gözlemlenebilirlik (observability) araçları için tutarlılık sağlar.

Yaygın Hatalar

- **Eureka Server'ın `application.yml`'inde `register-with-eureka` / `fetch-registry`'yi `true` bırakmak.** Tek düğümlü bir kurulumda sunucu, kendi kendine bağlanmaya çalışıp gereksiz hatalar/loglar üretir.
- **`DiscoveryClient`'ı doğrudan servisler arası iş mantığında (business logic) kullanmak.** Bu, yük dengeleme mantığını elle yeniden yazmak anlamına gelir -- `@LoadBalanced RestClient` zaten bunu sağlar.
- **Bir servisi kapatıp Eureka'nın kayıt defterinden ANINDA silmesini beklemek.** Eviction, heartbeat zaman aşımına (varsayılan 90 saniye) ve self-preservation moduna bağlıdır -- anlık değildir.
- **Eureka'yı Kubernetes gibi kendi servis keşfi olan bir platformda GEREKSİZ yere kullanmak.** Kubernetes'in kendi Service/DNS mekanizması varken Eureka fazladan bir karmaşıklık katmanı ekler.

Özet, Cheat Sheet ve Terimler Sözlüğü

Service Discovery, servislerin birbirini sabit bir adresle değil bir İSİMLE bulmasını sağlayan bir desendir. Eureka Server, `@EnableEurekaServer` ile açılan merkezi bir kayıt defteridir; her Eureka Client (`order-service` , `inventory-service`) kendini bu deftere kaydeder ve düzenli heartbeat'lerle kayıtlı kalır. `DiscoveryClient` düşük seviyeli, doğrudan sorgulama sağlar; `@LoadBalanced RestClient` ise günlük servisler arası çağrılar için idiomatik yoldur -- bir servis ismini otomatik olarak gerçek bir host:port'a çevirir. Eureka, CAP teoreminin AP tarafını seçer -- self-preservation modu bunun bir sonucudur.

Hızlı referans:

```
@SpringBootApplication
@EnableEurekaServer                          // Eureka Server -- ayrı bir uygulama
public class EurekaServerApplication { ... }

// eureka-server/application.yml
// eureka.client.register-with-eureka: false
// eureka.client.fetch-registry: false

// order-service/application.yml
// eureka.client.service-url.defaultZone: http://localhost:8761/eureka/

@Bean
@LoadBalanced                                    // isimle çağrı yapmayı etkinleştirir
RestClient.Builder loadBalancedRestClientBuilder() {
    return RestClient.builder();
}

restClient.get().uri("http://inventory-service/inventory/{name}", name)  // İSİM, host:port
değil
```

Terimler Sözlüğü

Service Discovery — Servislerin birbirini sabit bir adres yerine bir isimle bulmasını sağlayan desen.

Eureka Server — Hangi servislerin kayıtlı olduğunu ve nerede çalıştığını bilen, `@EnableEurekaServer` ile açılan merkezi kayıt defteri.

Eureka Client — Kendini Eureka Server'a kaydeden ve diğer servisleri onun üzerinden bulan bir mikroservis.

DiscoveryClient — Kayıt defterine düşük seviyeli, doğrudan sorgu yapmayı sağlayan Spring Cloud arayüzü.

Self-Preservation Mode — Eureka Server'ın, çok sayıda heartbeat kaybını gerçek bir servis çöküşü yerine kendi ağ sorunu olarak yorumlayıp instance silmeyi durdurduğu koruma modu.