Service Discovery & Eureka

order-service now finds inventory-service by NAME instead of a hardcoded URL: the Eureka Server (a central registry), the Eureka Client (spring.application.name becoming the discovery key), direct queries with DiscoveryClient, calling by name with a @LoadBalanced RestClient, heartbeats/eviction/self-preservation mode, and why Eureka picks the AP side of the CAP theorem.

Intermediate 24 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/service-discovery-eureka

---

Service Discovery & Eureka

Microservices' "wave 1" (`microservices-fundamentals`, `spring-boot-microservice-basics`, `inter-service-communication`) had `order-service` FIND `inventory-service` through a hardcoded URL (`@Value("${services.inventory-service.url}")`, see the "Inter-Service Communication" lesson's "From order-service to inventory-service: A Synchronous Call with RestClient" section). That works fine for TWO fixed services -- but in the real world, services MULTIPLY (multiple copies of the same service, for load balancing), MOVE (IPs change when containers restart), and SCALE. This lesson introduces the Java/Spring ecosystem's classic answer to that problem: Service Discovery and Netflix Eureka.

What Is Service Discovery?

Service Discovery is a pattern that lets services find each other by NAME instead of a FIXED address. At its center is a REGISTRY: every service instance REGISTERS itself with this registry when it starts ("I'm `inventory-service`, currently running at this host:port"), and when a service wants to CALL another service, instead of a fixed address it ASKS the registry "where is `inventory-service` right now?"

Why Does It Exist?

A hardcoded URL via `@Value` is fine as long as `inventory-service` runs as a SINGLE instance at a SINGLE fixed address. But in real production environments: (1) MULTIPLE copies (instances) of the SAME service run to handle increased load -- which one to go to can't be hardcoded; (2) containers/cloud environments change IP addresses FREQUENTLY -- manually updating `application.yml` on every restart isn't practical; (3) when a new service

is added, EVERY service that will call it needs its configuration updated. Service Discovery solves all three problems with a SINGLE central registry -- services find each other by NAME, not by fixed address.

History

Eureka is a service discovery tool Netflix built around 2012 for its own massive microservices infrastructure and open-sourced (part of Netflix OSS -- around the same time Netflix also open-sourced other microservices tools like Hystrix/Ribbon). Spring Cloud Netflix (2015) made it possible to add Eureka to a Spring Boot application with just a few lines of configuration -- this lesson uses exactly that integration. An important honesty note: Netflix STOPPED using Eureka 2.0 internally around 2018 and no longer uses its own tools -- but Eureka 1.x is still WIDELY used and actively maintained within the Spring Cloud ecosystem. Since platforms like Kubernetes have their OWN built-in service discovery mechanism, Eureka usually ISN'T needed for projects running on Kubernetes -- Eureka is most valuable for Spring applications running OUTSIDE Kubernetes (on VMs, classic servers).

Eureka Server: A Central Registry

The Eureka Server is a COMPLETELY separate, INDEPENDENT Spring Boot application from `order-service` / `inventory-service` -- it contains no business logic, connects to no database. Its only job: knowing which services are registered and at what address they run. `@EnableEurekaServer` is the ONE annotation that makes it a Eureka Server.

```
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` and `fetch-registry: false` PREVENT the Eureka Server from acting like a client of ITSELF -- in a single-node setup, the server doesn't need to register itself with, or "fetch" a registry from, another server. (In production, multiple Eureka Server nodes can replicate each other -- in that case these settings would be `true`, but this lesson focuses on a single-node setup.)

Eureka Client: Registering Services

`order-service` and `inventory-service` become Eureka CLIENTS by adding the `spring-cloud-starter-netflix-eureka-client` dependency and writing the Eureka Server's address (`eureka.client.service-url.defaultZone`) into their `application.yml`. A critical point: `spring.application.name` (see the "Microservice Configuration" lesson's "Its Own `application.yml`: Port, Application Name, and Database" section) is NO LONGER just a name that shows up in logs -- it's now the ACTUAL key other services will use to find it.

```
# 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")
```

Discovering Services with DiscoveryClient

`DiscoveryClient` is the low-level way to ask the registry a question DIRECTLY -- "which instances are currently registered under the name `inventory-service`?" Spring Cloud's Eureka client dependency AUTOMATICALLY provides a `DiscoveryClient` bean with no extra configuration.

```
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` is NOT the right tool for everyday inter-service calls -- its real use case is diagnostics and understanding what the registry currently sees. For ACTUAL inter-service calls, the approach in "Calling a Service by Name with a Load-Balanced RestClient" should be used.

Calling a Service by Name with a Load-Balanced RestClient

The `@LoadBalanced` annotation makes a `RestClient.Builder` bean REGISTRY-AWARE -- a `RestClient` built from it interprets an "address" like `http://inventory-service` NOT as a real hostname, but as a SERVICE NAME; Spring Cloud LoadBalancer INTERCEPTS that call, asks `DiscoveryClient`, and picks ONE of the currently registered instances.

```
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();
    }
}
```

This REPLACES the "Inter-Service Communication" lesson's `StockClient` and its hardcoded `@Value` URL -- ALL the rest of the logic (telling a 404 apart from a connection failure, translating it into `InventoryServiceUnavailableException`) stays UNCHANGED:

```
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) {
}
```

> When multiple instances of `inventory-service` are running, `@LoadBalanced RestClient` picks WHICH instance to go to FOR you -- this choice happens without changing a SINGLE line in `StockClientWithDiscovery` itself. The load balancing itself is Spring Cloud LoadBalancer's job, NOT Eureka's -- Eureka only TELLS you which instances exist, it doesn't CHOOSE between them.

Heartbeats, Eviction, and Self-Preservation Mode

A Eureka client sends a "heartbeat" to the Eureka Server at regular intervals (default 30 seconds) to STAY registered -- this is called "lease renewal". If a client fails to send a heartbeat for a certain period (default 90 seconds), the Eureka Server REMOVES it from the registry ("eviction"). But here's an interesting behavior: if the Eureka Server sees a LARGE number of clients stop sending heartbeats AT THE SAME TIME (the heartbeat rate drops significantly below expected), it does NOT interpret this as "multiple services genuinely crashed at once" -- it interprets it as "there's probably something wrong with MY OWN network connectivity" -- and enters "self-preservation mode", where it STOPS evicting any instance.

> Self-preservation mode can be especially CONFUSING in LOCAL development environments (running several services on one machine) -- even after you shut a service down, the Eureka Server may keep SHOWING it as "registered" for a while (sometimes minutes). This is NOT a bug -- it's a deliberate design decision so Eureka doesn't MISTAKENLY evict HEALTHY instances during a network partition.

Where Eureka Sits in the CAP Theorem: An AP System

Recall the "Microservices Fundamentals" lesson's "A Quick Look at the CAP Theorem" section: a distributed system must CHOOSE between Consistency and Availability during a network partition. Eureka deliberately picks the AP side: every Eureka Server node prefers to ALWAYS give an answer (even from a partially stale registry), rather than guaranteeing it has the MOST UP-TO-DATE information. Self-preservation mode is a CONSEQUENCE of this philosophy -- Eureka prefers the risk of "a few entries might be stale" over the risk of "mistakenly evict healthy services".

Best Practices

- **Call other services by NAME (with `@LoadBalanced RestClient`) instead of a hardcoded host:port** -- horizontal scaling and IP changes don't require code CHANGES.
- **Use `DiscoveryClient` only for diagnostics/observability**, NOT for everyday inter-service calls -- `@LoadBalanced RestClient` already does that job FOR you.
- **EXPECT self-preservation mode's confusing behavior in local development** (a shut-down service still appearing "registered" for a while) -- this isn't a bug, it's a natural consequence of Eureka's AP design.
- **Name the Eureka Server itself with a `spring.application.name` too**, even though it won't show up in the registry -- keeps logs and future observability tooling consistent.

Common Mistakes

- **Leaving `register-with-eureka` / `fetch-registry` as `true` in the Eureka Server's own `application.yml`.** In a single-node setup, the server tries to connect to itself, producing unnecessary errors/logs.
- **Using `DiscoveryClient` directly in inter-service business logic.** This means manually reimplementing load-balancing logic -- `@LoadBalanced RestClient` already provides it.
- **Expecting Eureka to instantly remove a service from the registry the moment it's shut down.** Eviction depends on the heartbeat timeout (default 90 seconds) and self-preservation mode -- it isn't instant.
- **Using Eureka UNNECESSARILY on a platform like Kubernetes that has its own service discovery.** Kubernetes' own Service/DNS mechanism already covers this -- Eureka adds an extra layer of complexity on top of it.

Summary, Cheat Sheet, and Glossary

Service Discovery is a pattern that lets services find each other by NAME instead of a fixed address. The Eureka Server, enabled with `@EnableEurekaServer`, is a central registry; every Eureka Client (`order-service`, `inventory-service`) registers itself with it and stays registered via regular heartbeats. `DiscoveryClient` provides low-level, direct queries; `@LoadBalanced RestClient` is the idiomatic way for everyday inter-service calls -- it automatically turns a service name into a real host:port. Eureka picks the AP side of the CAP theorem -- self-preservation mode is a consequence of that.

Quick reference:

```
@SpringBootApplication
@EnableEurekaServer                          // Eureka Server -- a separate application
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                                    // enables calling by name
RestClient.Builder loadBalancedRestClientBuilder() {
    return RestClient.builder();
}

restClient.get().uri("http://inventory-service/inventory/{name}", name)  // a NAME, not
host:port
```

Glossary

Service Discovery — A pattern that lets services find each other by name instead of a fixed address.

Eureka Server — The central registry, enabled with `@EnableEurekaServer`, that knows which services are registered and where they run.

Eureka Client — A microservice that registers itself with the Eureka Server and finds other services through it.

DiscoveryClient — The Spring Cloud interface that provides low-level, direct queries against the registry.

Self-Preservation Mode — The protective mode where the Eureka Server, seeing a large number of missed heartbeats, interprets it as its own network issue rather than a real service crash, and stops evicting instances.