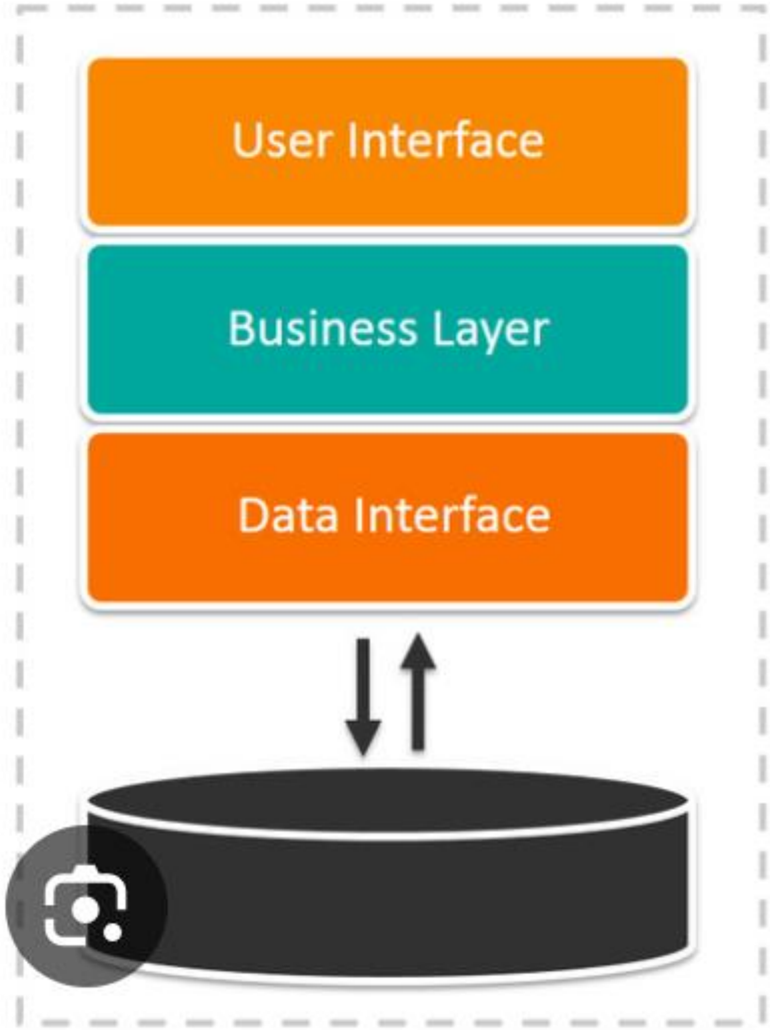
Microservices

Monolithic Architecture

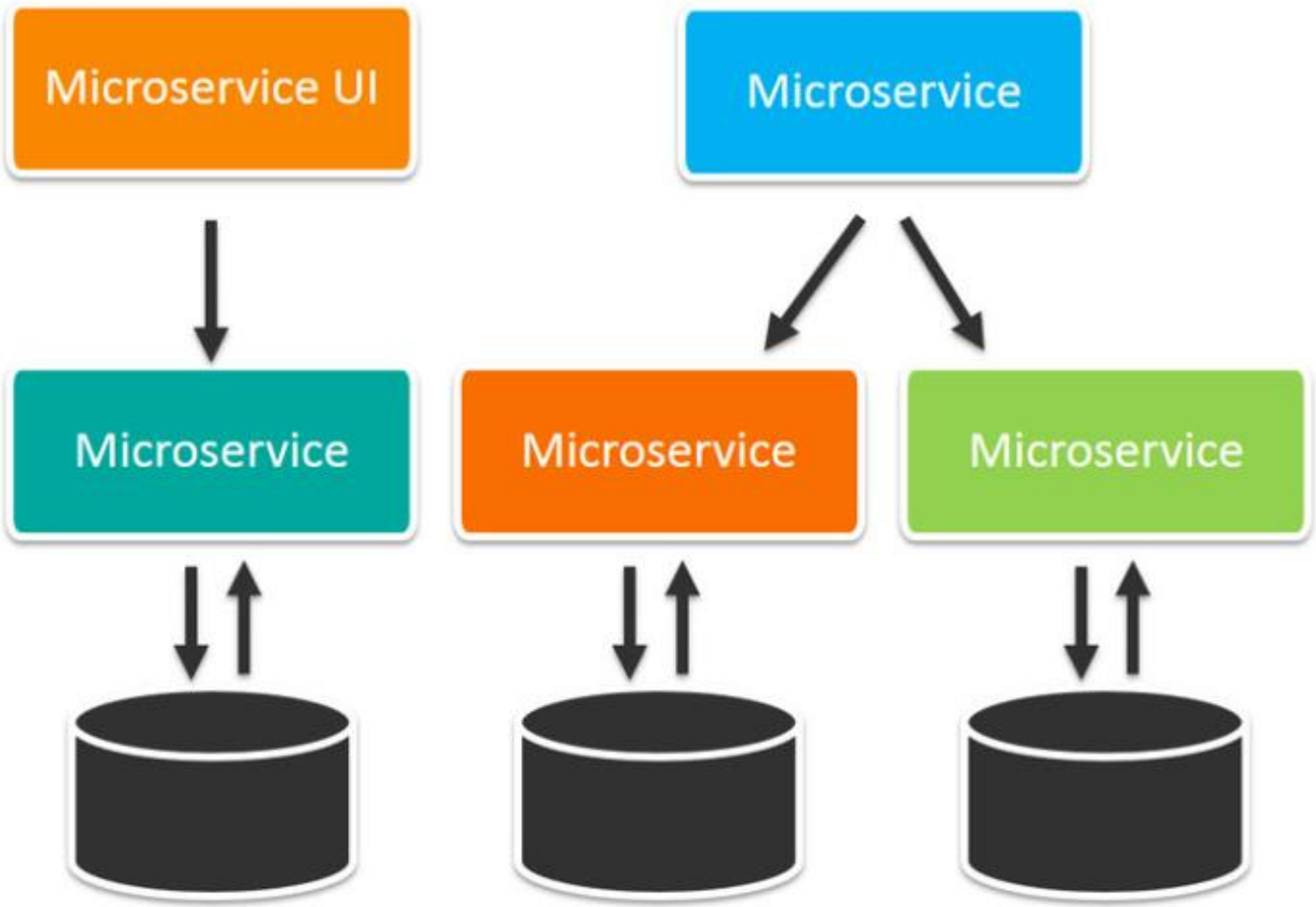


User Interface

Business Layer

Data Interface

Microservices Architecture



Microservice UI

Microservice

Microservice

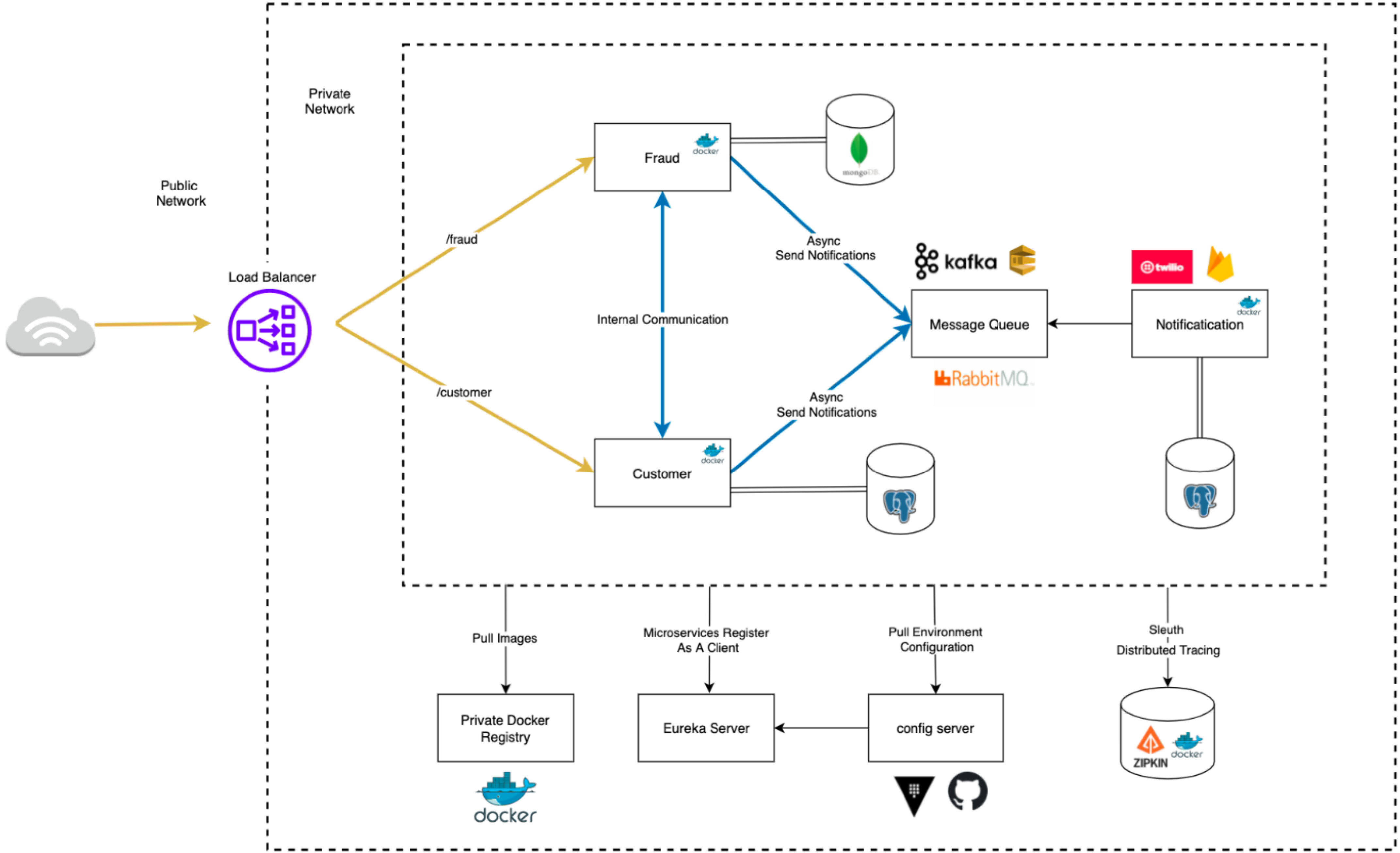
Microservice

Microservice

Monolithic

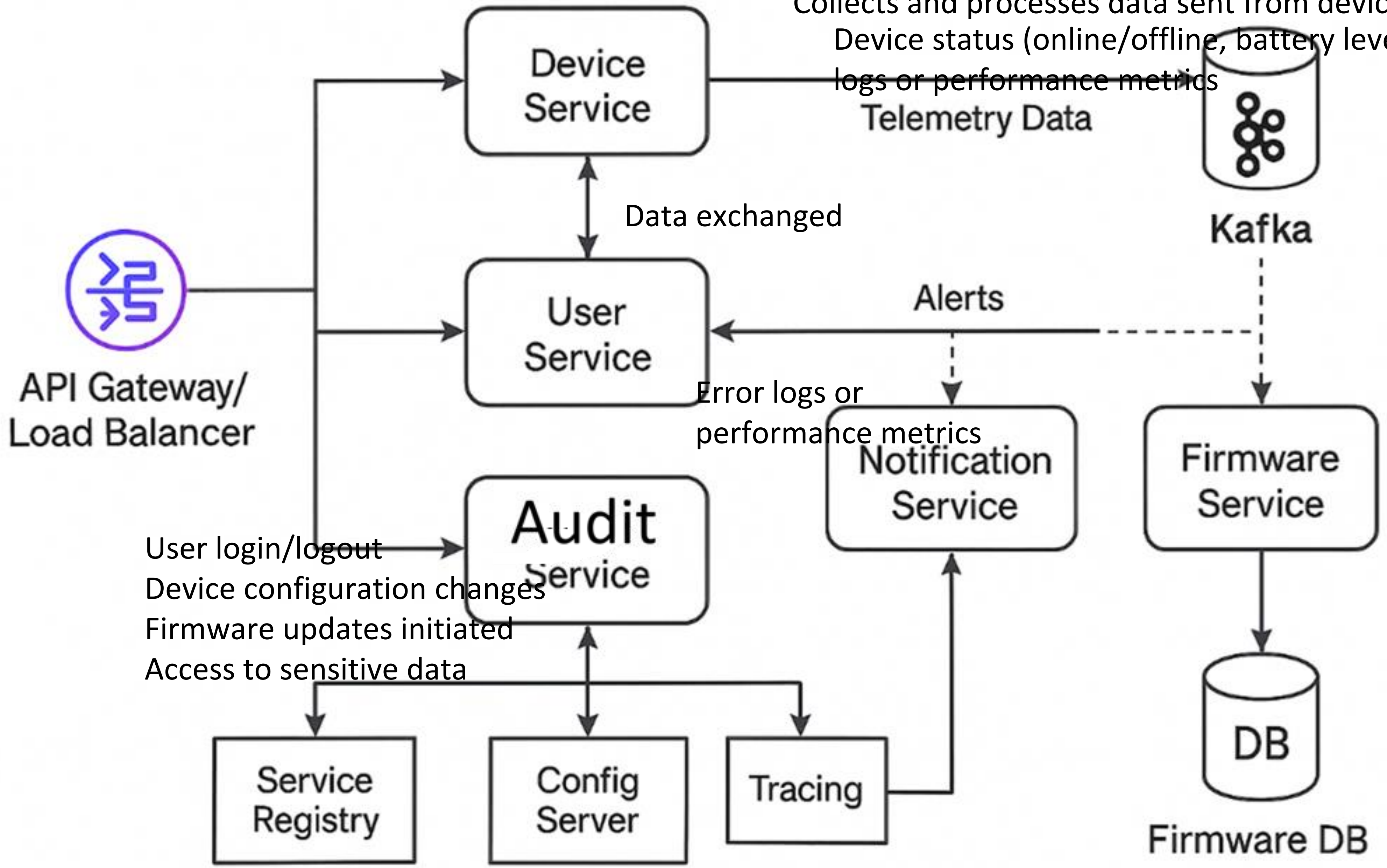
- **Hard to Adopt New Technologies**: Migrating to new frameworks or languages is difficult.
- **Low Fault Isolation**: A failure in one module can crash the entire application.
- **Tight Coupling**: All components are interconnected, making it hard to isolate and modify parts independently.
- **Scalability Issues**: You can only scale the entire application, even if only one part needs more resources.
- **Limited Technology Flexibility**: You’re often stuck with one tech stack across the whole application.

- **Hard to Maintain**: As the codebase grows, it becomes more complex and harder to manage.
- **Onboarding Difficulty**: New developers may struggle to understand the entire system before contributing.
- **Complex Testing**: Testing the whole application is time-consuming and error-prone.
- **Risky Deployments**: A single bug can bring down the entire system during deployment.
- **Poor Separation of Concerns**: Teams working on different features may interfere with each other’s code.
- **Limited Parallel Development**: Hard to work on multiple features simultaneously without conflicts.


Public Network
Private Network
Load Balancer
/fraud
/customer
Fraud
docker
mongoDB
Internal Communication
Async Send Notifications
kafka
twilio
Message Queue
Notificaticatio
docker
RabbitMQ
Async Send Notifications
Customer
docker
Pull Images
Microservices Register As A Client
Pull Environment Configuration
Sleuth Distributed Tracing
Private Docker Registry
Eureka Server
config server
docker
ZIPKIN
docker

Core Components

- **Device Service**: Handles registration, updates, and status of devices.
- **User Service**: Manages users who own or interact with devices.
- **Telemetry Service**: Collects and stores data from devices.
- **Notification Service**: Sends alerts or updates based on device events.
- **Firmware Service**: Manages firmware updates and versioning.



Collects and processes data sent from devices in real-time or at

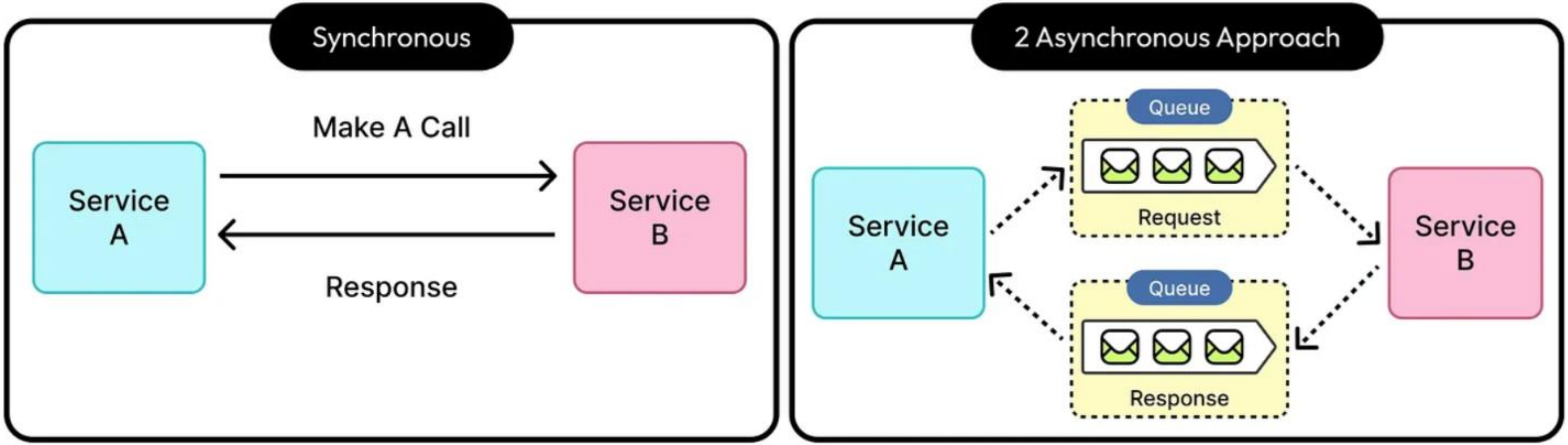
Device status (online/offline, battery level), Error logs or performance metrics

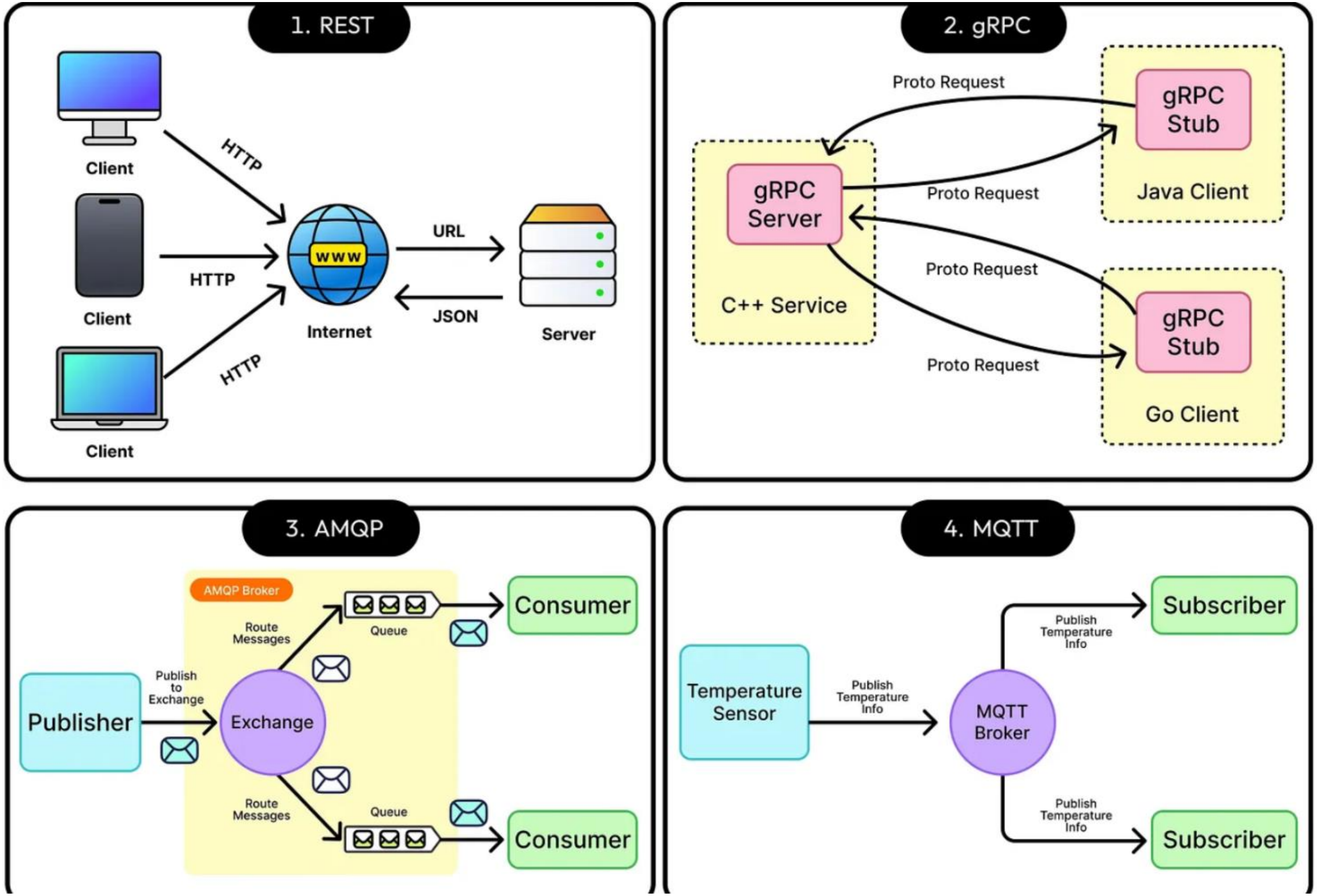
Data exchanged

Error logs or performance metrics

User login/logout
Device configuration changes
Firmware updates initiated
Access to sensitive data

Synchronous vs Asynchronous Communication



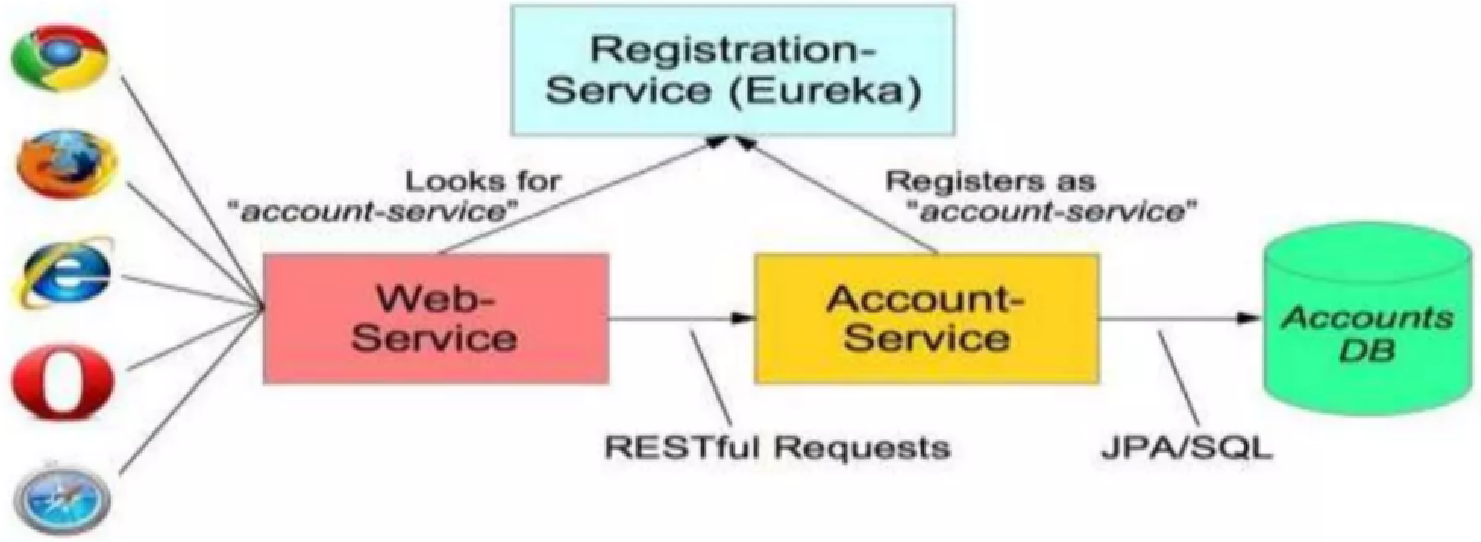

1. REST
Client
Client
Client
HTTP
HTTP
HTTP
www
Internet
URL
JSON
Server
2. gRPC
Proto Request
gRPC Stub
Java Client
gRPC Server
Proto Request
C++ Service
Proto Request
gRPC Stub
Proto Request
Go Client
3. AMQP
AMQP Broker
Publisher
Publish to Exchange
Exchange
Route Messages
Queue
Consumer
Route Messages
Queue
Consumer
4. MQTT
Temperature Sensor
Publish Temperature Info
MQTT Broker
Publish Temperature Info
Subscriber
Publish Temperature Info
Subscriber

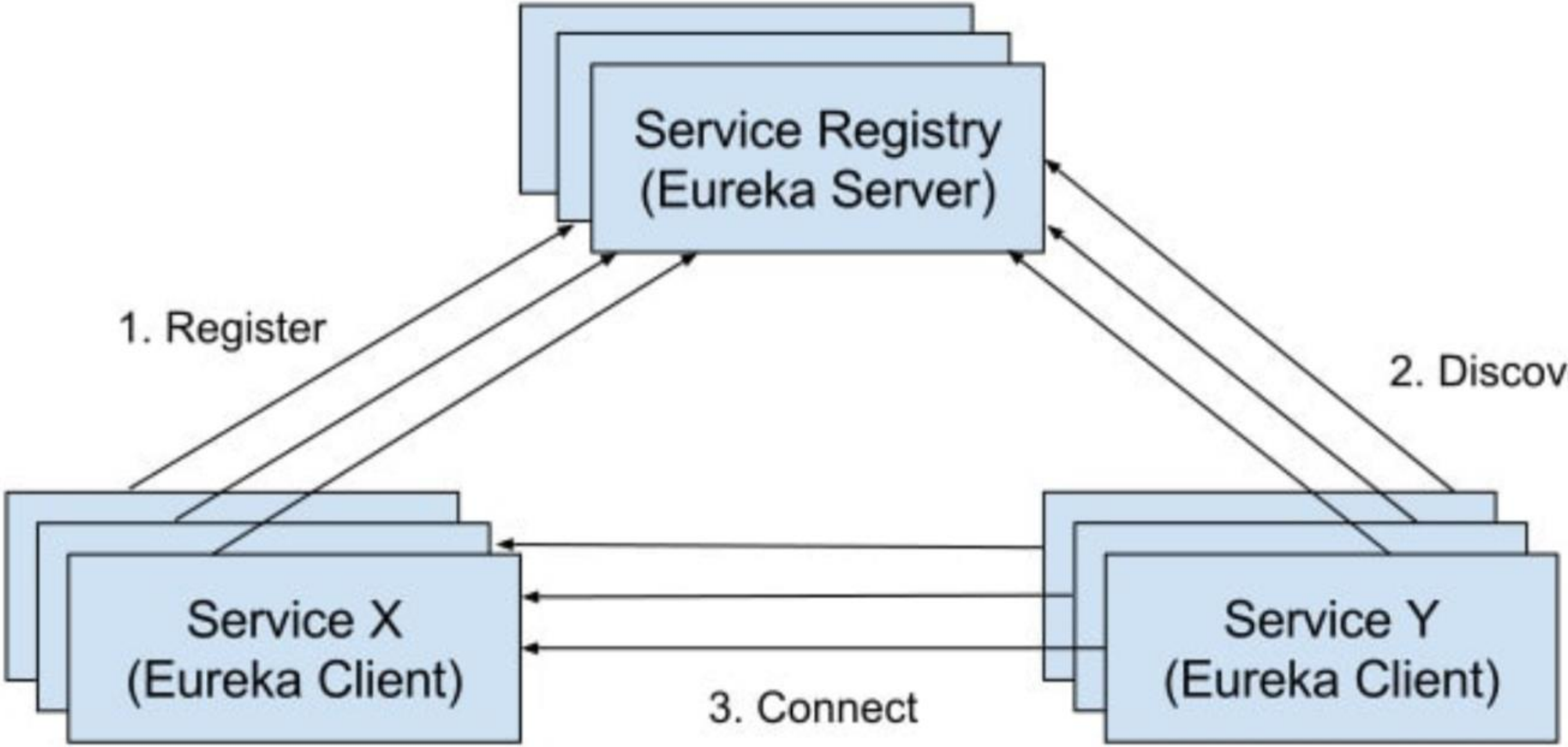
Infrastructure & Communication

- **API Gateway / Load Balancer**: Entry point for external requests.
- **Service Registry (Eureka)**: For service discovery.
- **Config Server**: Centralized configuration management.
- **Message Broker (Kafka/RabbitMQ)**: For asynchronous communication (e.g., telemetry, alerts).
- **Tracing (Zipkin)**: For monitoring service interactions.

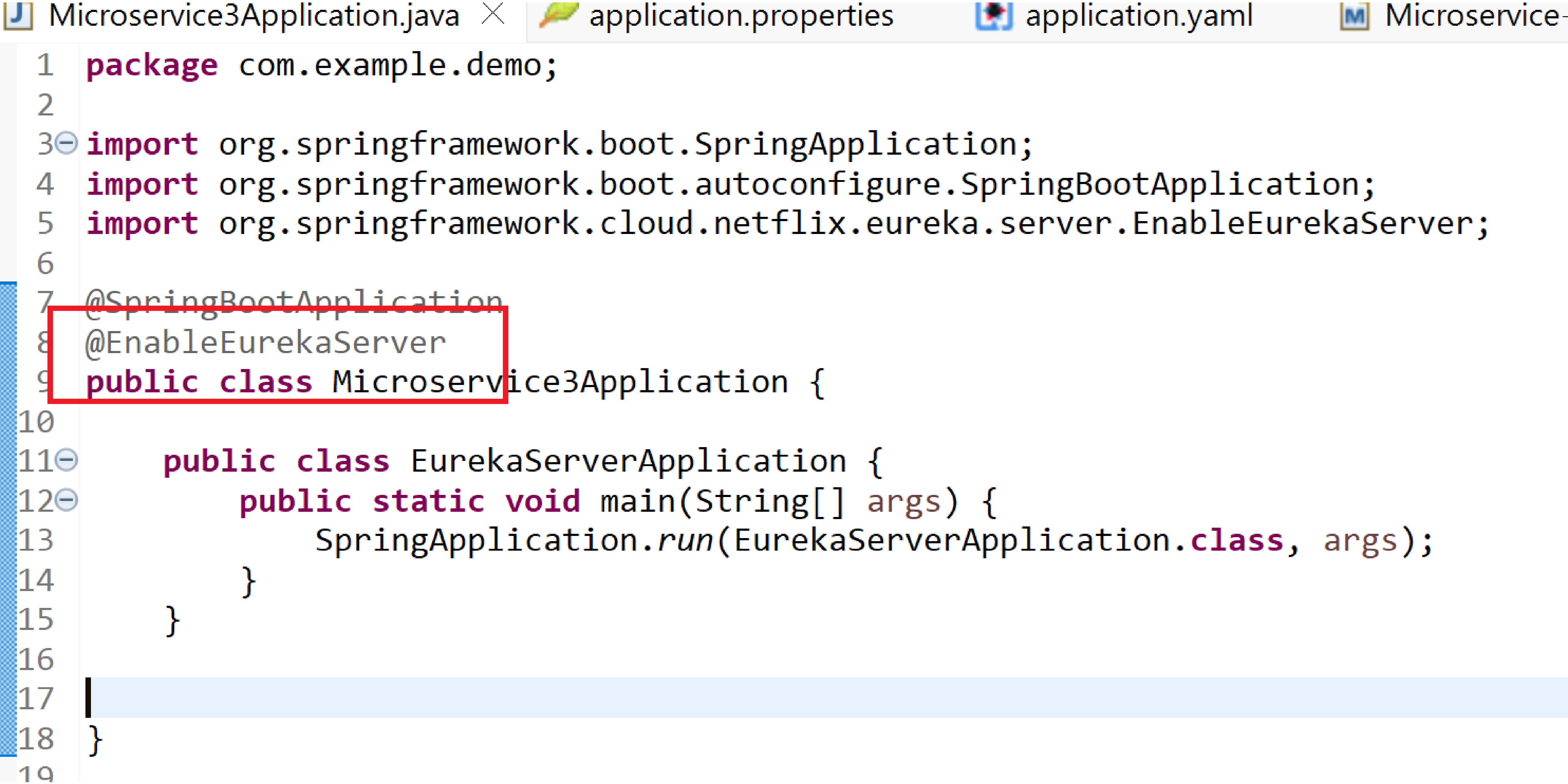
Eureka Server

When you have multiple processes working together they need to find each other. The developers at Netflix had this problem when building their systems and created a registration server called Eureka ("I have found it" in Greek). Fortunately for us, they made their discovery server open-source and Spring has incorporated into Spring Cloud, making it even easier to run up a Eureka server.

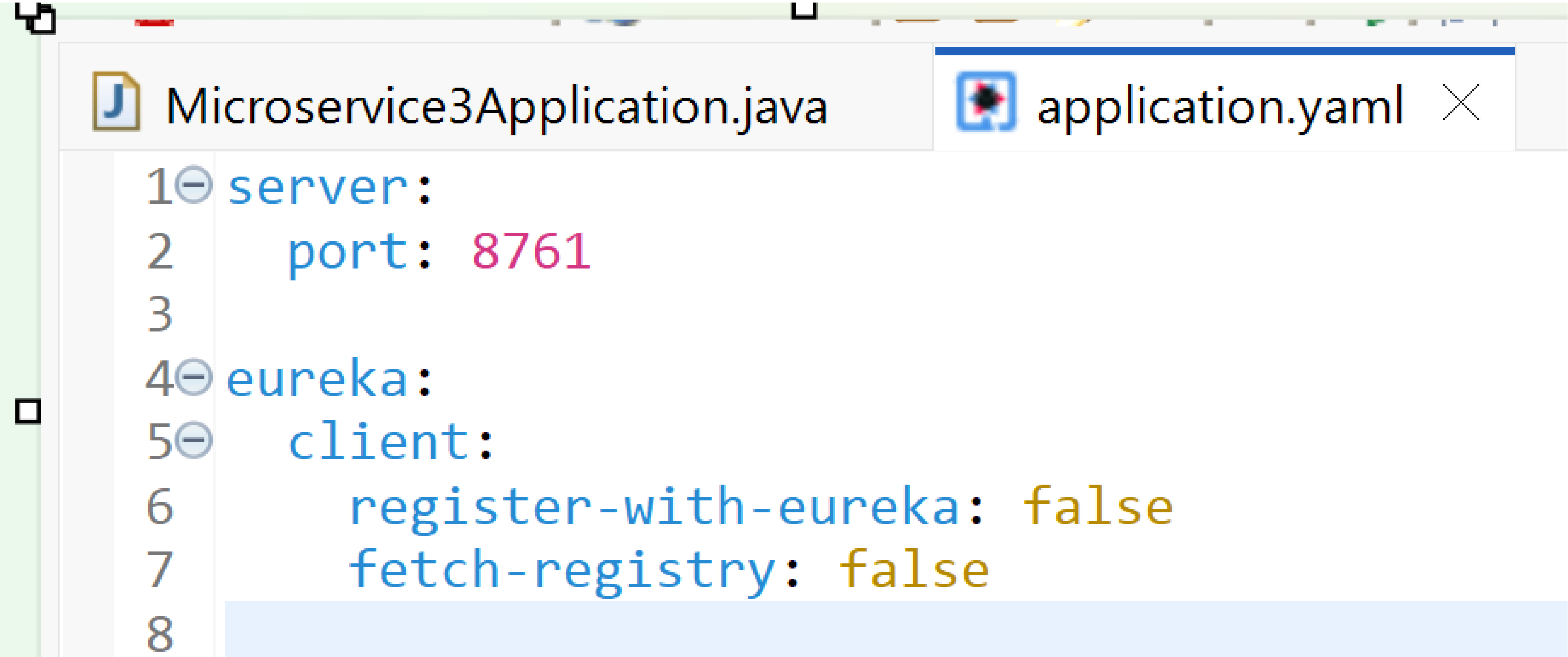

Registration-Service (Eureka)
Looks for "account-service"
Registers as "account-service"
Web-Service
Account-Service
Accounts DB
RESTful Requests
JPA/SQL



Microservice Registry with Eureka



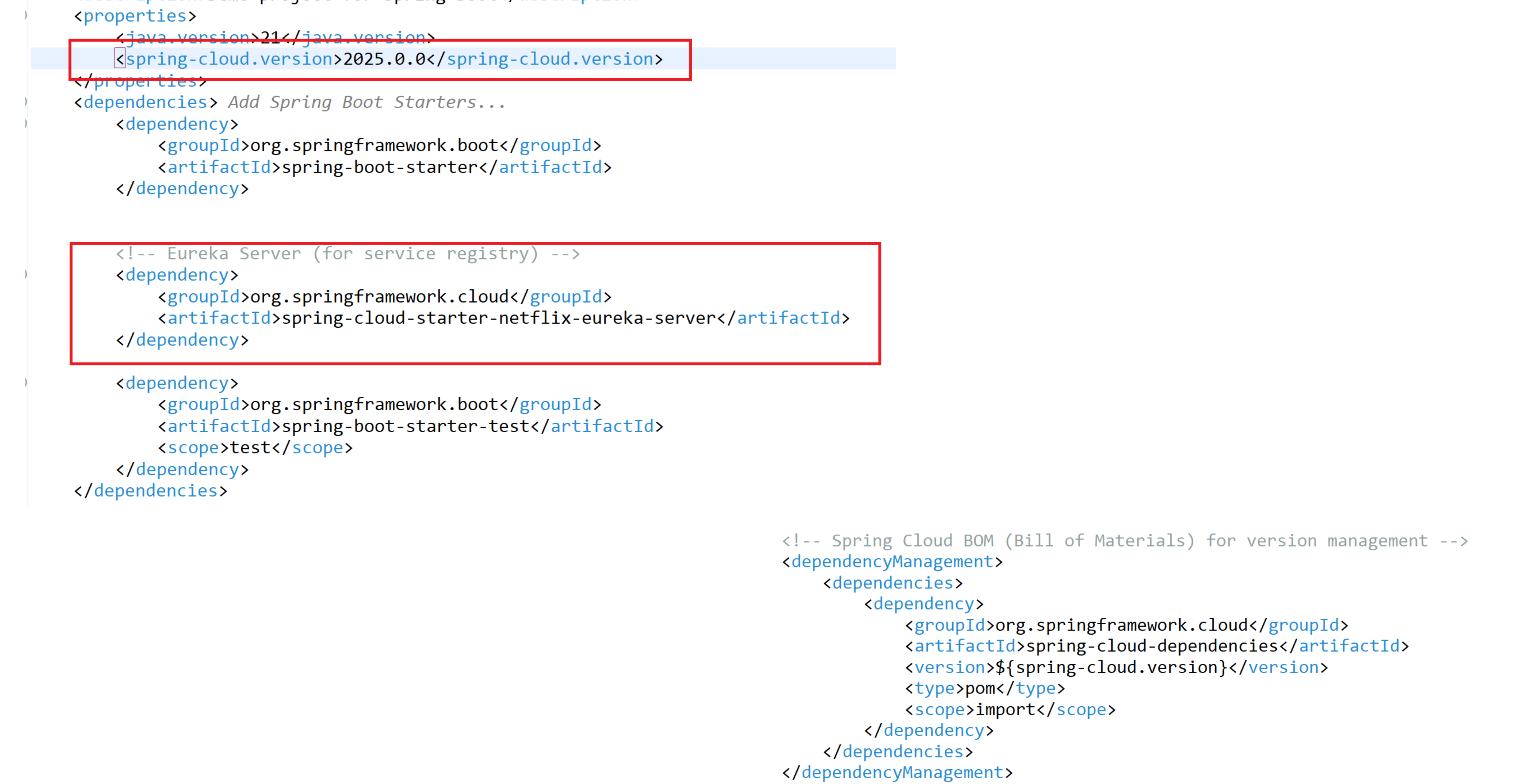
```java
package com.example.demo;

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

@SpringBootApplication
@EnableEurekaServer
public class Microservice3Application {

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


}
```



```yaml
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false

```

☑ register-with-eureka: false
This tells the Eureka Server not to register itself as a client.
Normally, microservices register themselves with Eureka so they can be discovered.
But the Eureka Server is the registry itself, so it doesn’t need to register.

☑ fetch-registry: false
This tells the Eureka Server not to fetch the list of services from other Eureka servers.
This is useful when you have only one Eureka Server (not a cluster).
In a cluster setup, you might set this to true so servers sync with each other.

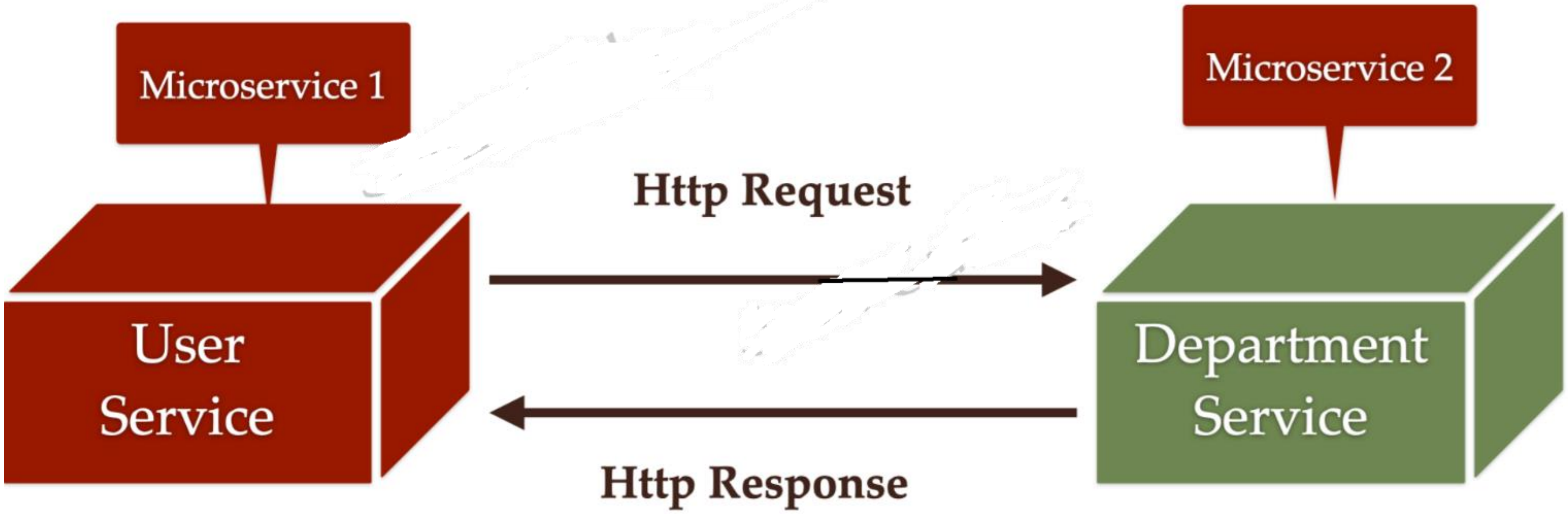


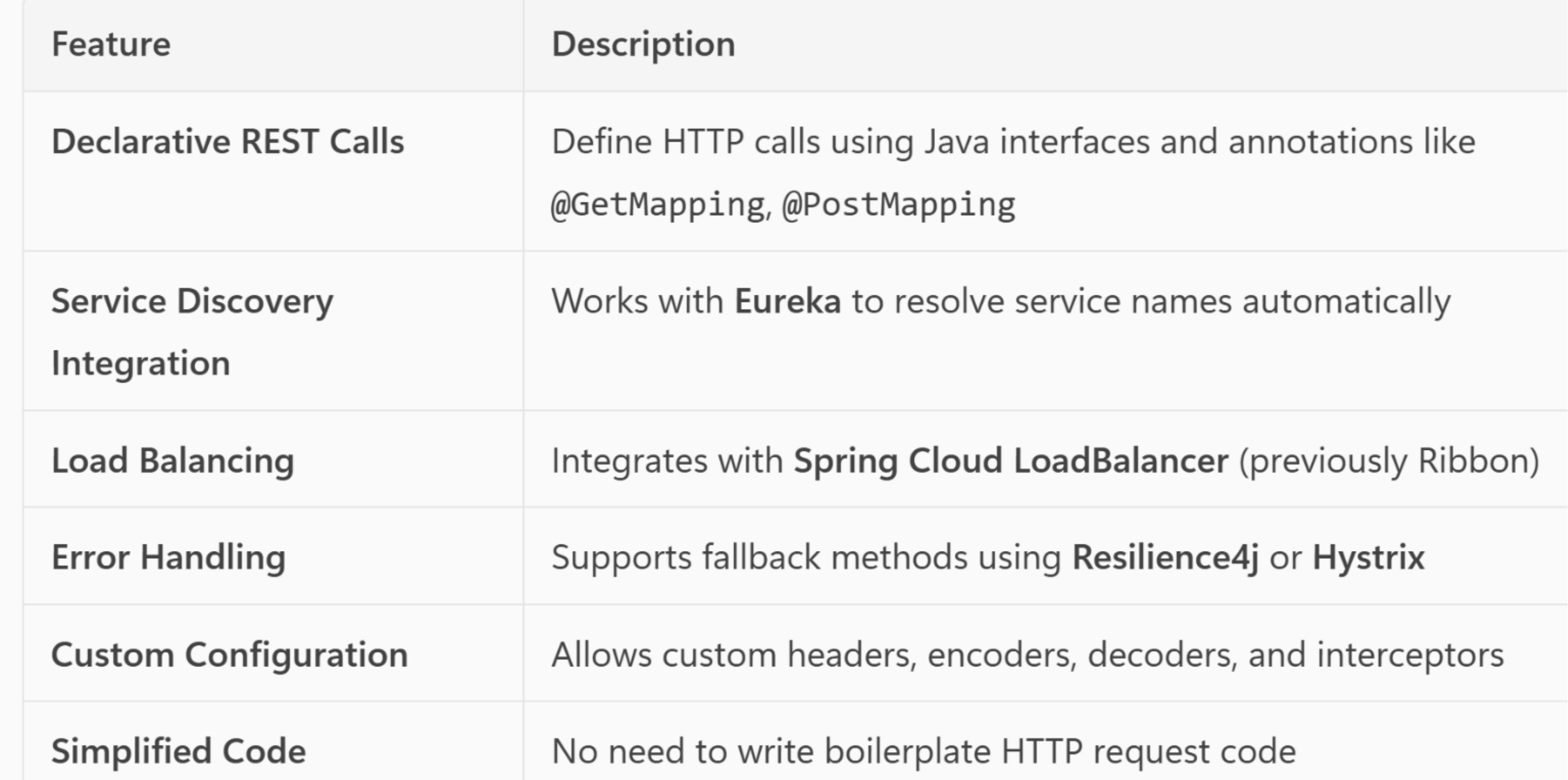
```
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>21</java.version>
        <spring-cloud.version>2025.0.0</spring-cloud.version>
    </properties>
    <dependencies> Add Spring Boot Starters...
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>


        <!-- Eureka Server (for service registry) -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

```
    <!-- Spring Cloud BOM (Bill of Materials) for version management -->
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

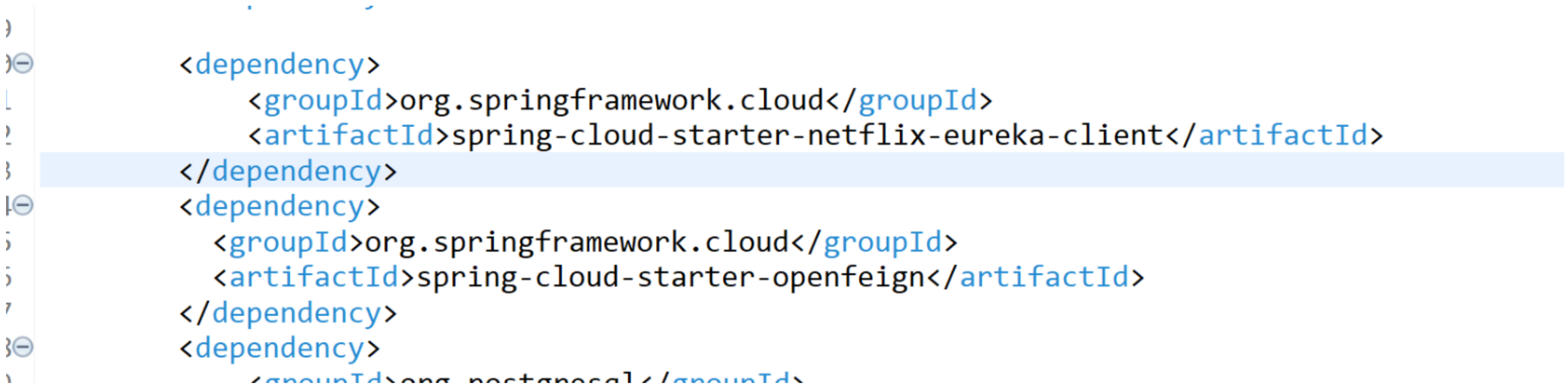
Interservice communication:

Feign Client is a declarative web service client provided by Spring Cloud. It simplifies HTTP communication between microservices by allowing you to write Java interfaces to call REST APIs, instead of manually using RestTemplate or WebClient.

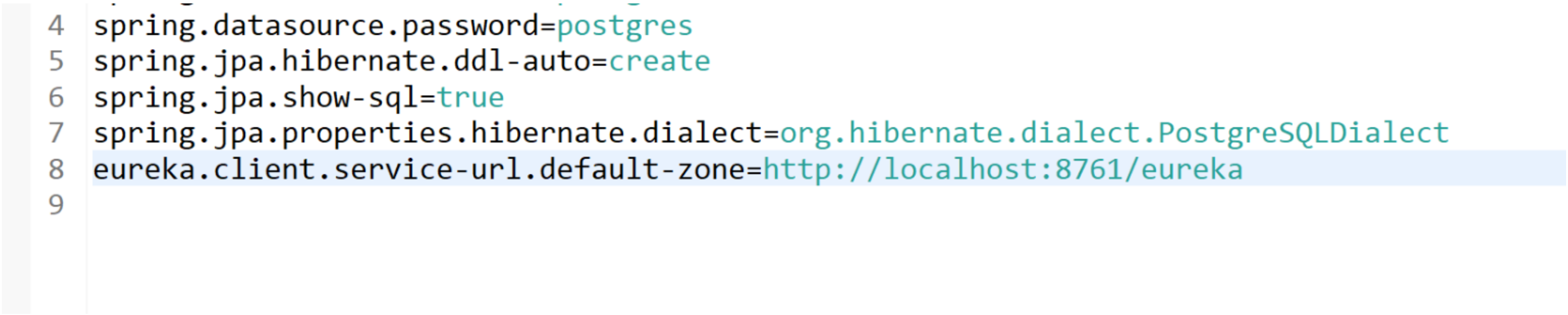




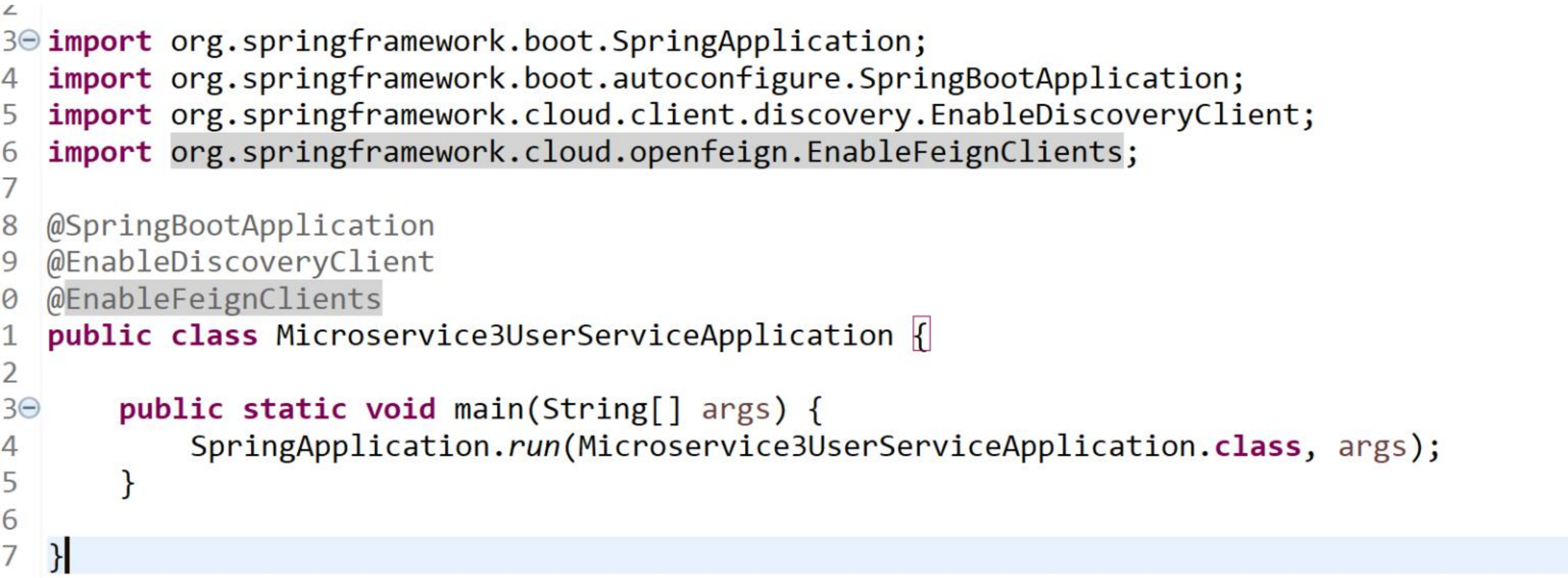
Feature	Description
Declarative REST Calls	Define HTTP calls using Java interfaces and annotations like `@GetMapping`, `@PostMapping`
Service Discovery Integration	Works with **Eureka** to resolve service names automatically
Load Balancing	Integrates with **Spring Cloud LoadBalancer** (previously Ribbon)
Error Handling	Supports fallback methods using **Resilience4j** or **Hystrix**
Custom Configuration	Allows custom headers, encoders, decoders, and interceptors
Simplified Code	No need to write boilerplate HTTP request code



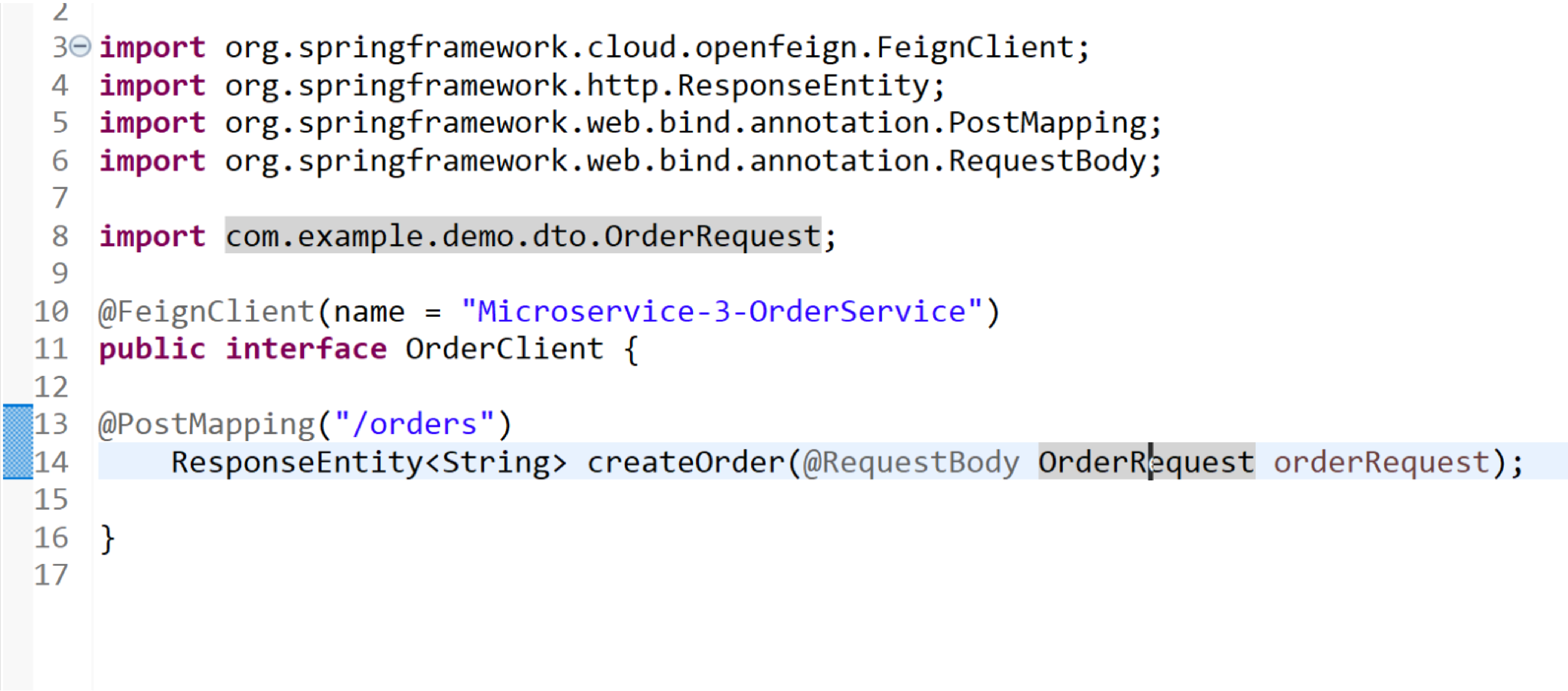
```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-openfeign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



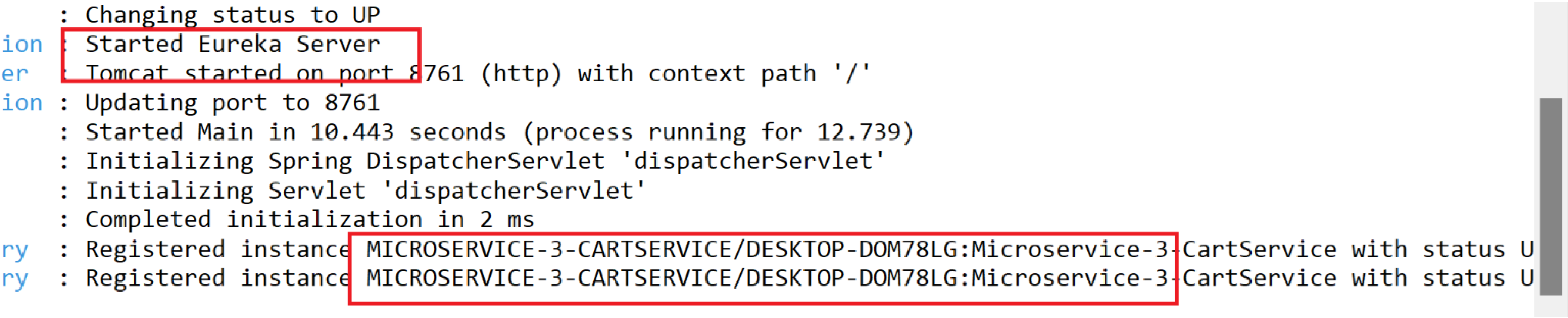
```
spring.datasource.password=postgres
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```



```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Microservice3UserServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(Microservice3UserServiceApplication.class, args);
    }

}
```

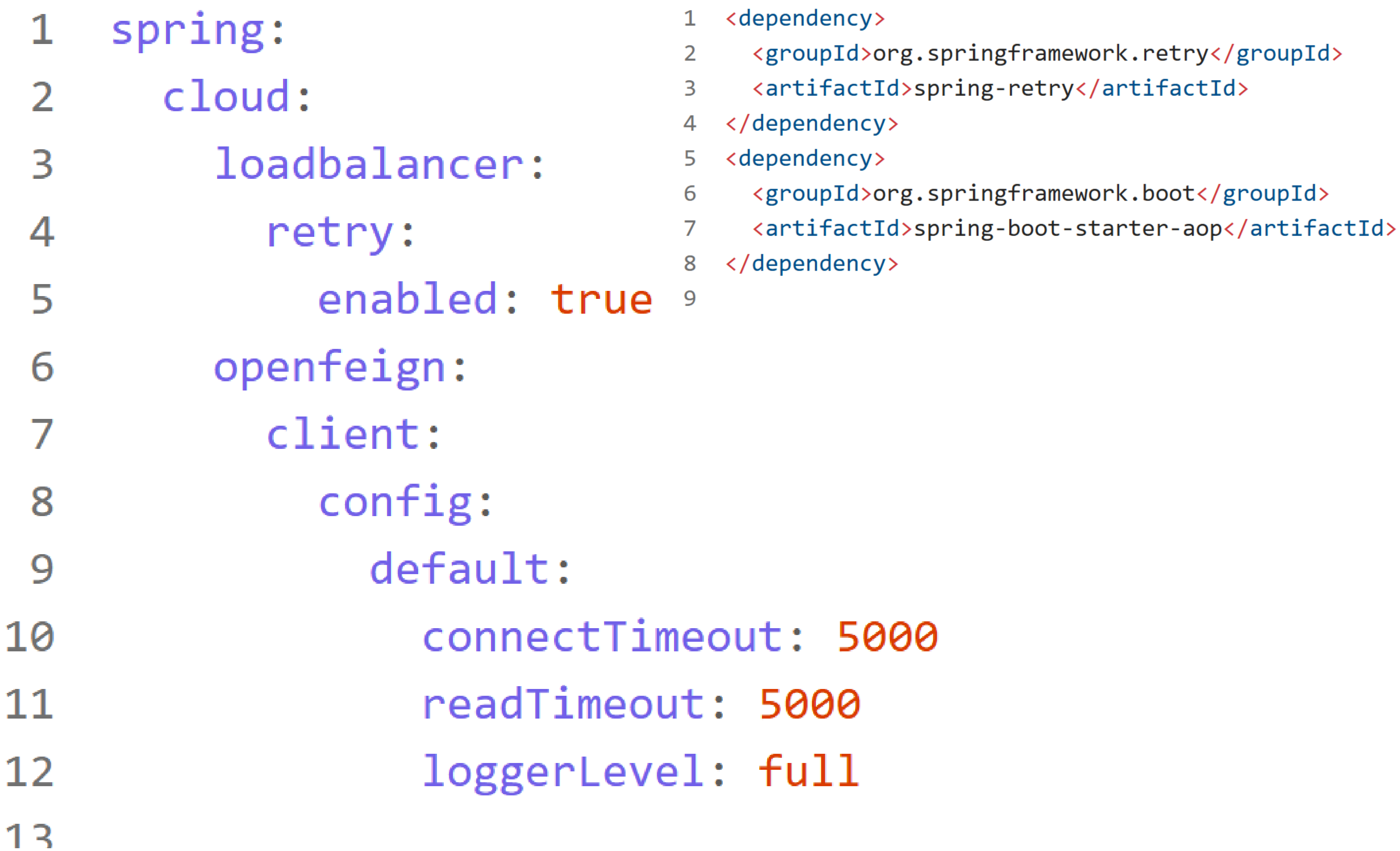


```

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;

import com.example.demo.dto.OrderRequest;

@FeignClient(name = "Microservice-3-OrderService")
public interface OrderClient {

@PostMapping("/orders")
    ResponseEntity<String> createOrder(@RequestBody OrderRequest orderRequest);

}

```



```
    : Changing status to UP
ion : Started Eureka Server
er  : Tomcat started on port 8761 (http) with context path '/'
ion : Updating port to 8761
    : Started Main in 10.443 seconds (process running for 12.739)
    : Initializing Spring DispatcherServlet 'dispatcherServlet'
    : Initializing Servlet 'dispatcherServlet'
    : Completed initialization in 2 ms
ry  : Registered instance MICROSERVICE-3-CARTSERVICE/DESKTOP-DOM78LG:Microservice-3-CartService with status U
ry  : Registered instance MICROSERVICE-3-CARTSERVICE/DESKTOP-DOM78LG:Microservice-3-CartService with status U
```

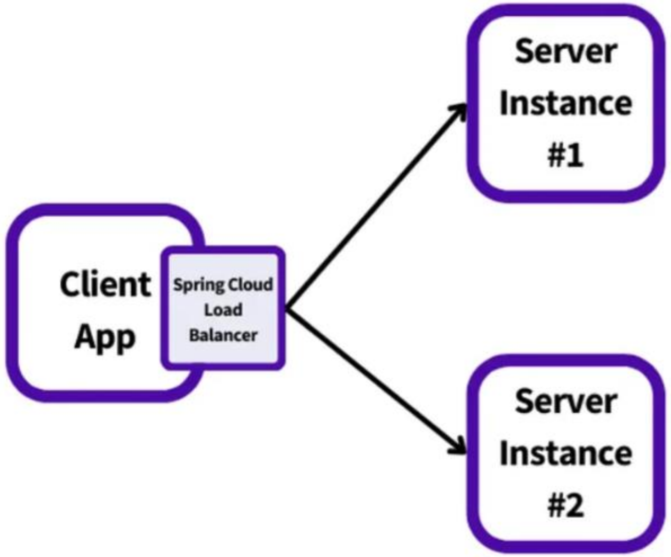
Optional:



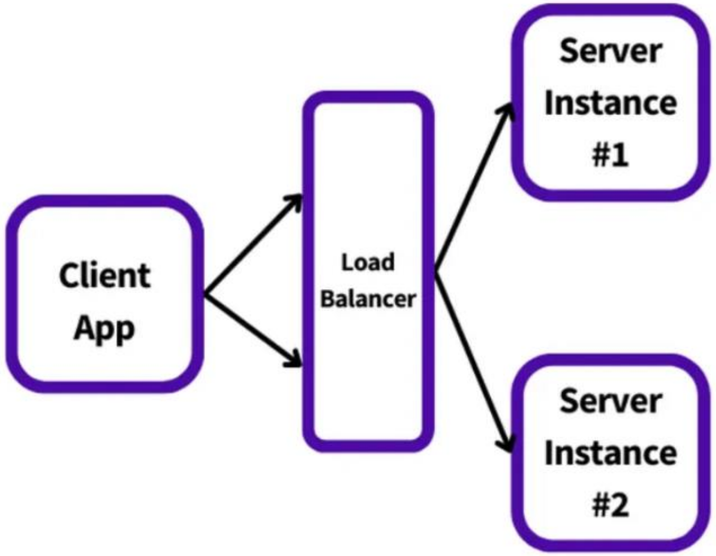
```yaml
spring:
  cloud:
    loadbalancer:
      retry:
        enabled: true
    openfeign:
      client:
        config:
          default:
            connectTimeout: 5000
            readTimeout: 5000
            loggerLevel: full

```

```xml
<dependency>
  <groupId>org.springframework.retry</groupId>
  <artifactId>spring-retry</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>

```

Client Side Load Balancing



Client Side Load Balancing



Server Side Load Balancing

Feign uses **Spring Cloud LoadBalancer** for client-side load balancing.

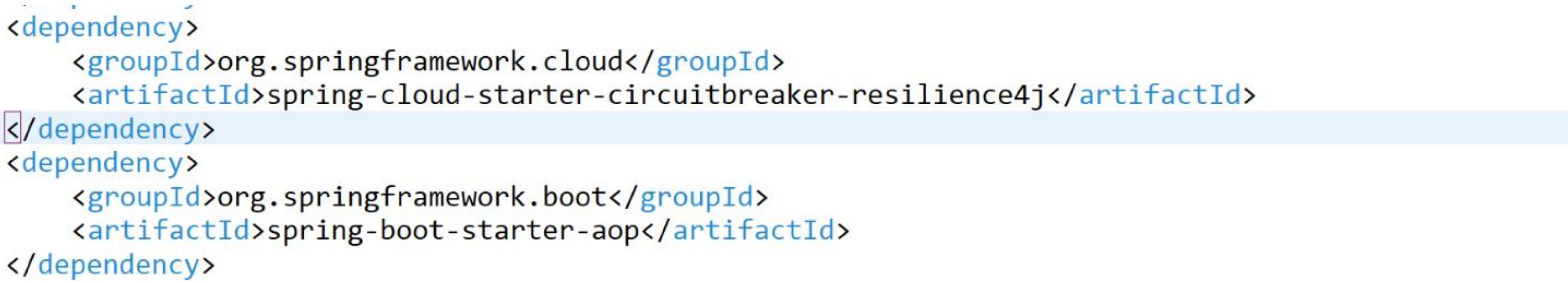
- You call a service by its name (e.g., "order-service").
- Feign + Eureka + LoadBalancer will:
- Discover all instances of order-service
- Choose one instance based on a load balancing strategy (e.g., round-robin)
- Send the request to that instance
- Client-side load balancing removes that extra hop. Each client receives a list of healthy backend servers. It stores this list locally and decides where to send requests. This makes routing faster, more reliable, and easier to scale.
- Instead of one router doing all the work, each client shares the load.

Advantages of Client-Side Load Balancing

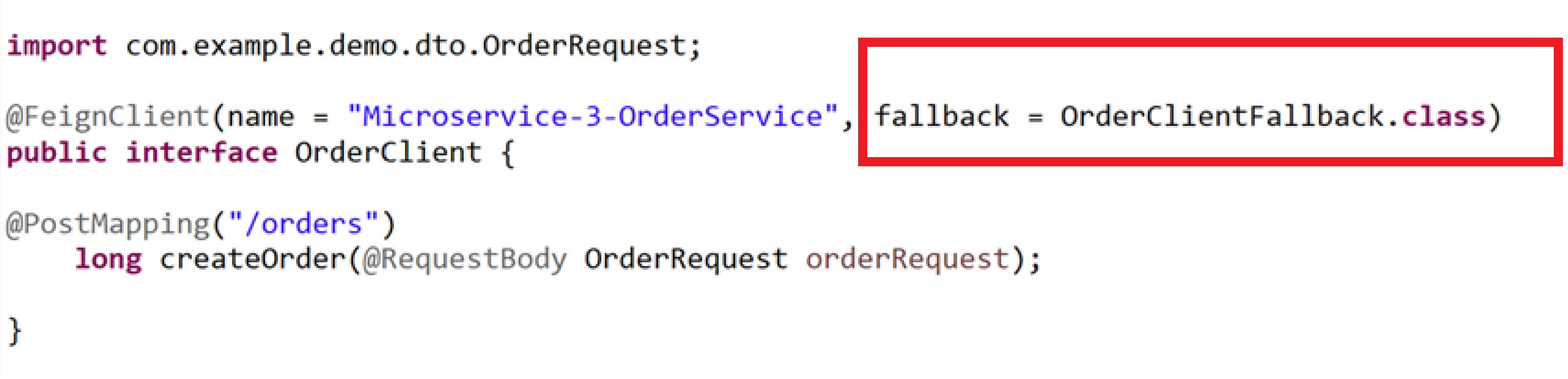
- **Reduced Latency**
- Requests go directly from client to server. There’s no central router to slow things down.
- **2. No Single Point of Failure**
- If one client fails, others keep working. There’s no central piece to bring the system down.
- **3. Better Traffic Distribution**
- Each client spreads its own traffic. This balances the load across all servers.
- **4. Local Control**
- Clients make routing decisions based on real-time feedback. They can retry quickly or back off when needed.
- **5. Easy to Scale**
- As services grow, adding new clients or servers doesn’t overload a central router. The system scales naturally.

3. Add Circuit Breaker with Fallback

Use Resilience4j with Feign:

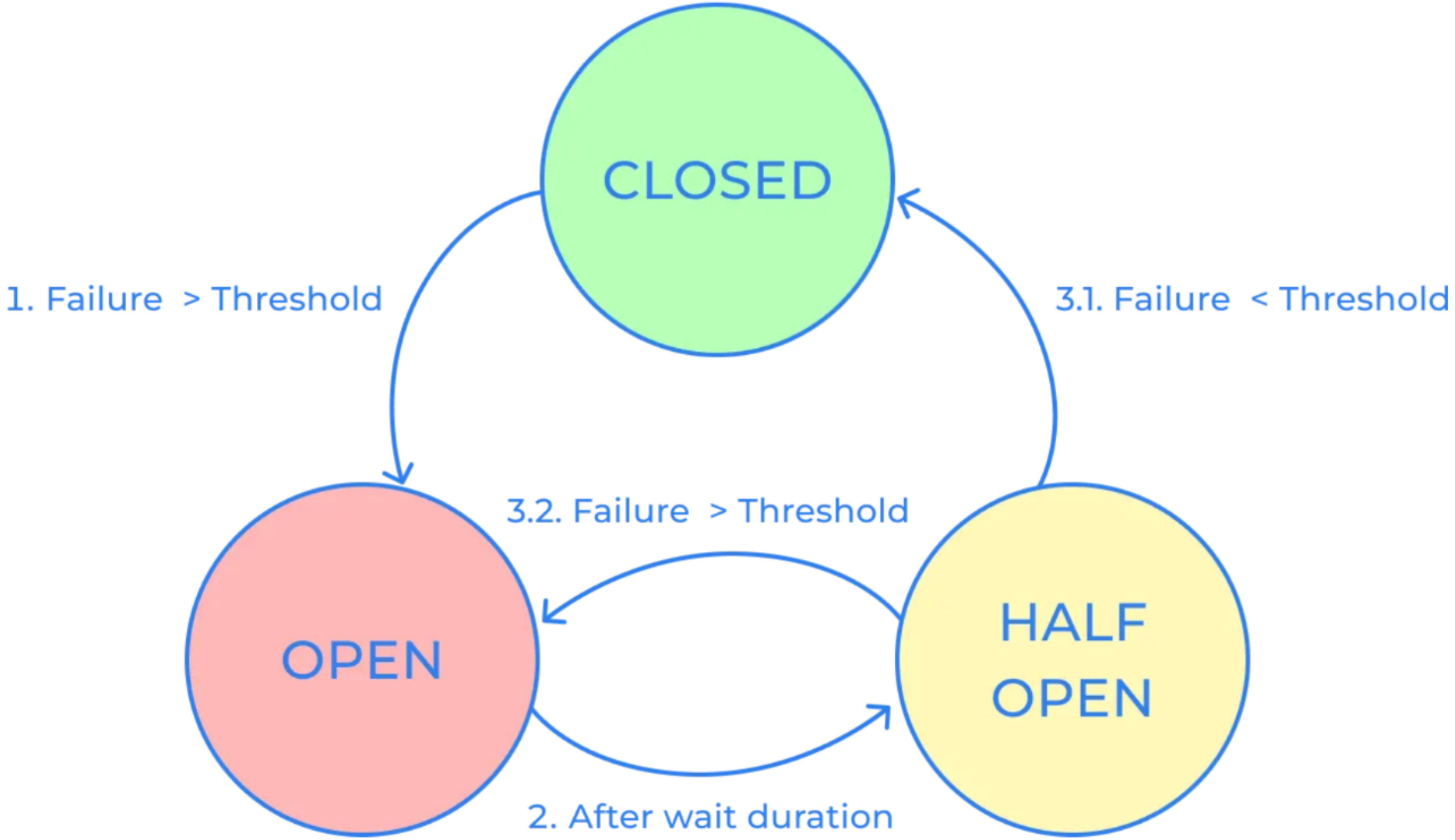


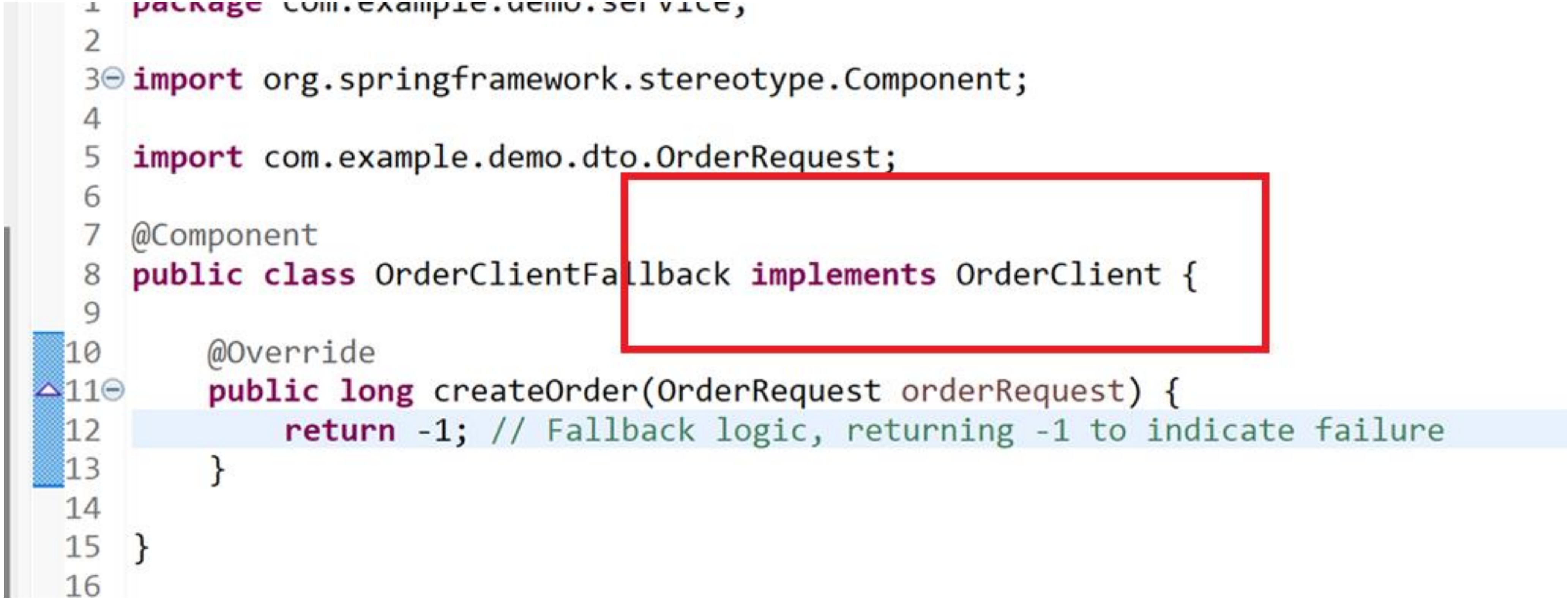
```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```



```java
import com.example.demo.dto.OrderRequest;

@FeignClient(name = "Microservice-3-OrderService", fallback = OrderClientFallback.class)
public interface OrderClient {

@PostMapping("/orders")
    long createOrder(@RequestBody OrderRequest orderRequest);

}
```

When the “circuit” is *CLOSED*, requests can reach the service, when it is *OPEN* it fails immediately. After a period of time *OPEN* it moves to *HALF_OPEN,* if the failures are below the threshold it moves to *CLOSED*, or returns to OPEN otherwise.





```
package com.example.demo.service;

import org.springframework.stereotype.Component;

import com.example.demo.dto.OrderRequest;

@Component
public class OrderClientFallback implements OrderClient {

    @Override
    public long createOrder(OrderRequest orderRequest) {
        return -1; // Fallback logic, returning -1 to indicate failure
    }

}

```

Storage

- Device DB
- User DB
- Telemetry DB
- Firmware DB