Step 1: Create PaymentApp and OrderApp

Create Payment method: return status as success

OrderApp: create order: save order ->// Call payment service and get payment status: Will implement later

Step 2: To talk to each other, they need a name:

Use eurekaserver: discovery service
Services register themselves with unique name
spring.application.name=

Step 3: Spin up eureka service

Annotate main class with @EnableEurekaServer

Add dependency for server

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

application.properties:

```
server:
 port: 8761
eureka:
 client:
   register-with-eureka: false
   fetch-registry: false
```

Step 4: make other services client for eureka.

Add eureka client library. Set eureka-client dependency

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

```
        </dependency>
    </dependencies>
      <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>
```

```
<spring-cloud.version>2025.0.0</spring-cloud.version>
```

Specify eureka url in application.properties

```
eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

Check services got registered with eureka at url:

@EnableDiscoveryClient: to register as client with eureka

Step5: Now we will make OrderService: call payment service

We will use feign client for it

Declarative way of calling other service using http

In order service, add feign client dependency

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

In orderservice class where u want to make a call to paymentservice, inject paymentservice interface and call the method

Create the paymentService interface and declare the method there.

In PaymentService, add @FeignClient annotation , name is name of application u r trying to call
Annotate method with proper get/ post mapping..

```
@FeignClient(name=”payment-service”) - name of calling service

public interface PaymentClient {

@PostMapping

//copy method definition, include @RequestBody/ @RequestParam/
@PathVariable etc

}


On main class, annotate with @EnableFeignClients(basePackages =
"com.example.demo.service")
```

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableJpaRepositories(basePackages = "com.example.demo.dao")
@EnableFeignClients(basePackages = "com.example.demo.service")
public class Microservice3UserServiceApplication {
      public static void main(String[] args) {
            SpringApplication.run(Microservice3UserServiceApplication.class,
args);
      }

}


 }
```

Step 6: Add circuit breaker:

Think of an electrical circuit breaker in your home:

- If there’s too much current (problem), the breaker trips to prevent damage.

- After some time, it allows current again to see if things are back to normal.
- If everything is fine, it resets; if not, it trips again.
-

Circuit breaker in microservices works exactly the same way — but for service calls instead of electricity.

In microservices, one service calls another over the network.
But what if:

- The downstream service is **slow**,
- Or it's **unavailable**,
- Or keeps **failing**?

Without protection, your service keeps waiting, retrying, and eventually **fails too** — this is called a **cascading failure**.

Circuit breaker protects against this.

🔹 States of a Circuit Breaker

1. **Closed (Normal)**
   - ○ Requests flow as usual.
   - ○ The breaker is "closed" → calls go through.
   - ○ Failures are counted.
2. **Open (Tripped)**
   - ○ If failures exceed a threshold (e.g., 50% errors in last 10 calls), the breaker "opens".
   - ○ Now, **no requests are sent** to the failing service → they fail **fast** with a fallback.
   - ○ This prevents wasting resources waiting for a broken service.
3. **Half-Open (Testing)**
   - ○ After a "wait time" (say 5 seconds), the breaker allows **limited test calls**.

- If those succeed → breaker closes again (service recovered).
- If they fail → breaker goes back to Open.

https://blog.devgenius.io/circuit-breaker-and-feign-client-implementation-in-spring-boot-3-1-zipkin-opentelemetry-46606aaded0c

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

```
public record ExceptionMessage(String timestamp,
                              int status,
                              String error,
                              String message,
                              String path) {
}
```

```
public class MessageErrorDecoder implements ErrorDecoder {

   private final ErrorDecoder errorDecoder = new Default();

   @Override
   public Exception decode(String methodKey, Response response) {
       ExceptionMessage message = null;
       try (InputStream body = response.body().asInputStream()){
           message = new ExceptionMessage((String)
response.headers().get("date").toArray()[0],
                   response.status(),

HttpStatus.resolve(response.status()).getReasonPhrase(),
                   IOUtils.toString(body, StandardCharsets.UTF_8),
                   response.request().url());

       } catch (IOException exception) {
           return new Exception(exception.getMessage());
       }
       switch (response.status()) {
           case 404:
               return new Exception(message.message());
```

```
            default:
                return errorDecoder.decode(methodKey, response);
        }
    }
}
```

Note:

In **Spring Cloud OpenFeign**, the ErrorDecoder.decode(String methodKey, Response response) method is part of the ErrorDecoder contract.

Here's what **methodKey** means:

- methodKey is a **unique identifier** for the Feign client method that triggered the HTTP call.
- Its format is usually:

```
<FeignClientClassName>#<methodName>(<parameterTypes>)
```

For example:

If you have a Feign client:

```
@FeignClient(name = "order-service")
public interface OrderClient {
   @GetMapping("/orders/{id}")
   Order getOrder(@PathVariable("id") Long id);
}
```

and you call:

```
orderClient.getOrder(123L);
```

Then in your ErrorDecoder, the methodKey would look like:

```
OrderClient#getOrder(Long)
```

Why is this useful?

- You can use methodKey to differentiate **which Feign client method** caused the error.

- That way, you can apply different error-handling logic depending on which API call failed.

Example:

```
@Override
public Exception decode(String methodKey, Response response) {
    if (methodKey.contains("getOrder")) {
        return new OrderNotFoundException("Order not found");
    }
    return errorDecoder.decode(methodKey, response);
}
```

Step 7: Load Balancing

Client-Side Load Balancing

- Who decides which server instance to call? → The client.
- The client (like Feign, RestTemplate, WebClient) has a list of server instances (from Eureka or config).
- It uses a load-balancing algorithm (round robin, random, weighted, etc.) to choose one instance before sending the request.
- The actual request goes directly to the chosen instance.

Example:
You have 3 instances of order-service:

```
 http://10.0.0.1:8080
http://10.0.0.2:8080
http://10.0.0.3:8080
```

-
- Feign client asks Eureka for all instances.
- Load balancer (client-side, e.g., Ribbon or Spring Cloud LoadBalancer) picks one:
  - 1st call → 10.0.0.1
  - 2nd call → 10.0.0.2
  - 3rd call → 10.0.0.3

So the client decides which server to hit.
👉 Feign uses client-side load balancing.
Earlier via Ribbon, now via Spring Cloud LoadBalancer.

🔹 2. Server-Side Load Balancing

- Who decides which server instance to call? → The server/load balancer.
- The client sends the request to a single endpoint (usually a load balancer like Nginx, HAProxy, AWS ELB, Kubernetes Service, API Gateway).
- That load balancer forwards the request to one of the available service instances.
- The client doesn't know about multiple servers — it just knows the load balancer endpoint.

Example:
Client always calls:

```
http://orders.mycompany.com
```

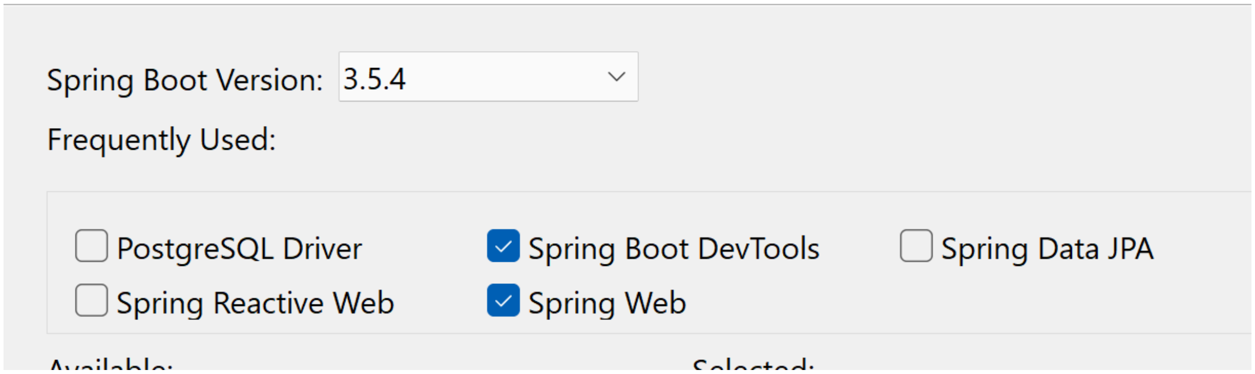
-
- A load balancer (say Nginx) distributes requests to:
  - 10.0.0.1
  - 10.0.0.2
  - 10.0.0.3

👉 Here, the load balancing logic is on the server/gateway, not the client.

Step 8: Config server

Externalize configuration

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

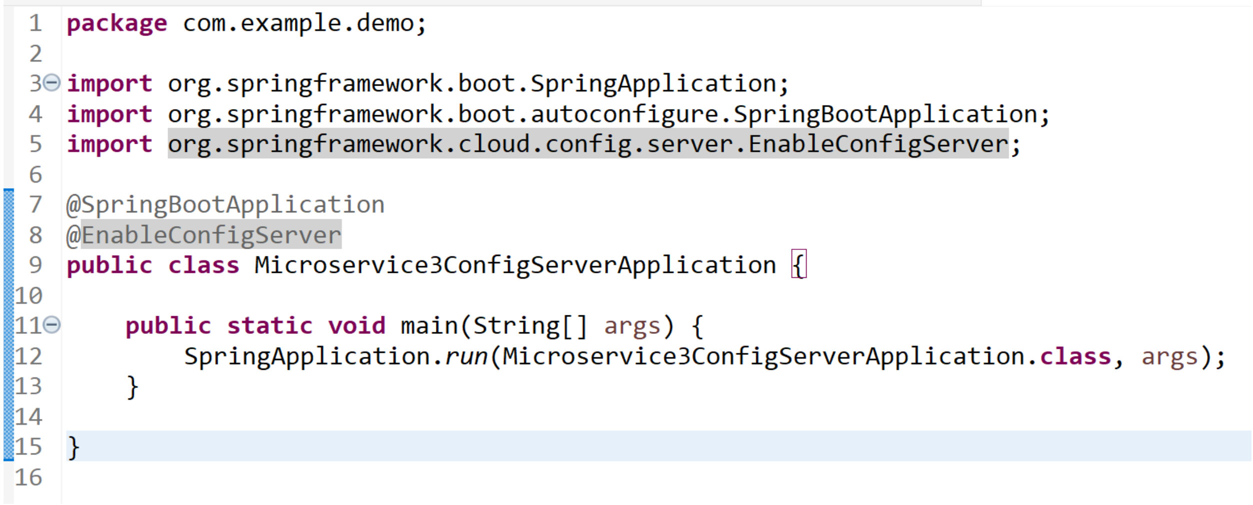


Spring Boot Version: 3.5.4

Frequently Used:

☐ PostgreSQL Driver ☑ Spring Boot DevTools ☐ Spring Data JPA
☐ Spring Reactive Web ☑ Spring Web

Available: Selected:



```java
package com.example.demo;

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

@SpringBootApplication
@EnableConfigServer
public class Microservice3ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(Microservice3ConfigServerApplication.class, args);
    }

}

```

2) Prepare the config Git repository

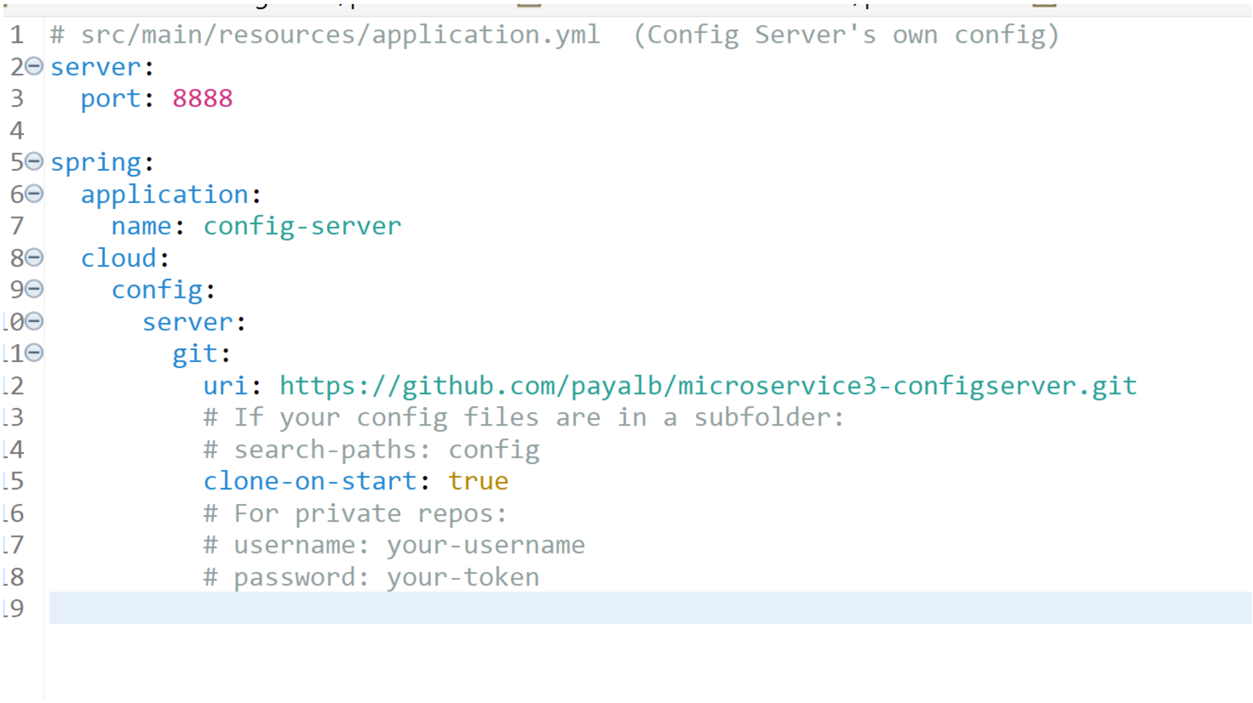
Create a separate repo just for configuration (flat files). Naming rules are important.

Basic naming

- application.yml → defaults for all services/all profiles
- <service-name>.yml → service-specific defaults
- <service-name>-<profile>.yml → service-specific + profile (e.g., orderservice-dev.yml)

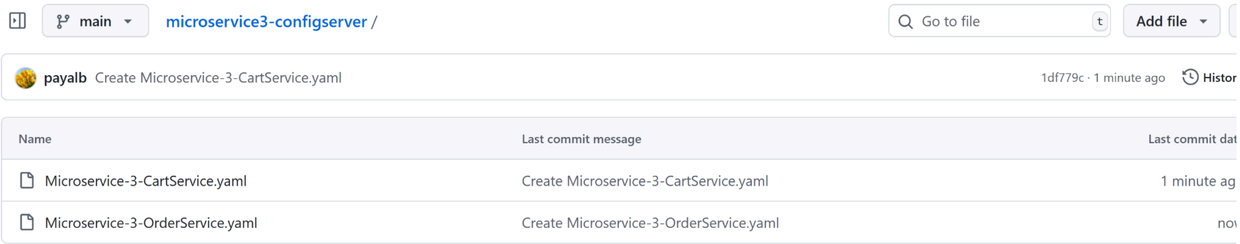
Example layout

application.yml
orderservice.yml
orderservice-dev.yml
paymentservice.yml



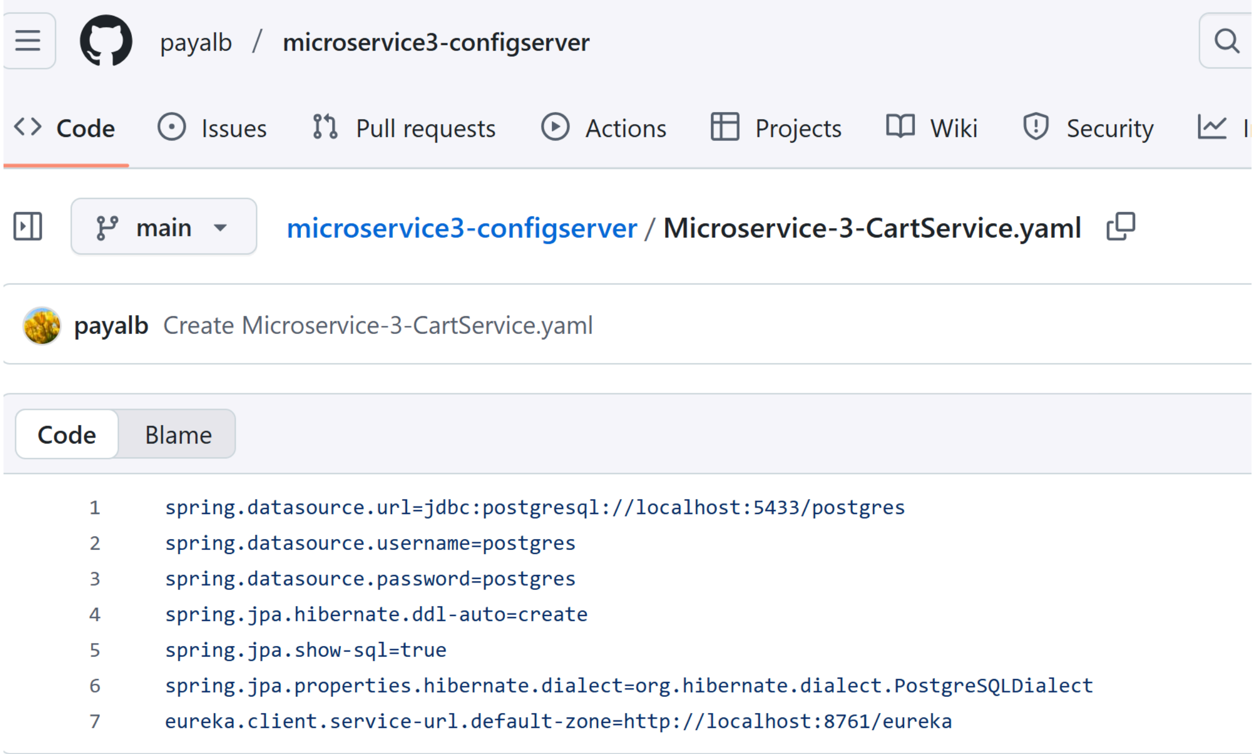
```yaml
# src/main/resources/application.yml  (Config Server's own config)
server:
  port: 8888

spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/payalb/microservice3-configserver.git
          # If your config files are in a subfolder:
          # search-paths: config
          clone-on-start: true
          # For private repos:
          # username: your-username
          # password: your-token

```

```yaml
# src/main/resources/application.yml  (Config Server's own config)
server:
 port: 8888
spring:
 application:
   name: config-server
 cloud:
   config:
     server:
       git:
         uri: https://github.com/payalb/microservice3-configserver.git
         # If your config files are in a subfolder:
         # search-paths: config
         clone-on-start: true
         # For private repos:
         # username: your-username
         # password: your-token
```



payalb Create Microservice-3-CartService.yaml 1df779c · 1 minute ago Histor

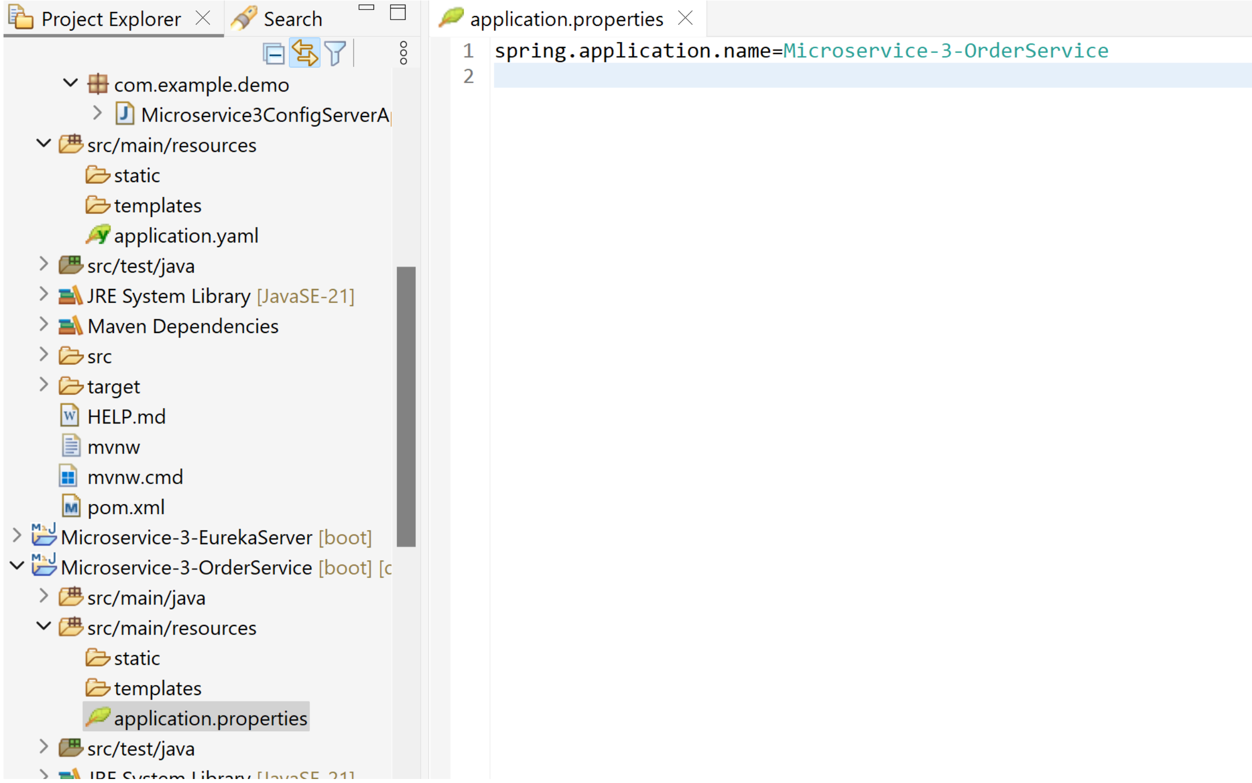
Name	Last commit message	Last commit dat
Microservice-3-CartService.yaml	Create Microservice-3-CartService.yaml	1 minute ag
Microservice-3-OrderService.yaml	Create Microservice-3-OrderService.yaml	nov



microservice3-configserver / Microservice-3-CartService.yaml

payalb Create Microservice-3-CartService.yaml

```
spring.datasource.url=jdbc:postgresql://localhost:5433/postgres
spring.datasource.username=postgres
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
```



```
spring.application.name=Microservice-3-OrderService
```

4) Refresh config at runtime (no restart)

a) Mark beans as refreshable

```
@RefreshScope
@RestController
public class HelloController {
  @Value("${app.greeting:Hello default}")  //Hello default → is the fallback (default) value.
  private String greeting;

  @GetMapping("/hello")
  public String hello() { return greeting; }
}
```

b) Expose the refresh endpoint

```
# client application's application.yml
management:
  endpoints:
    web:
      exposure:
        include: health,info,env,refresh
```

Now, when you change the Git file and the Config Server sees it, you can trigger a refresh on the client:

```
POST http://localhost:8080/actuator/refresh
```

The next call to /hello should return the updated value.

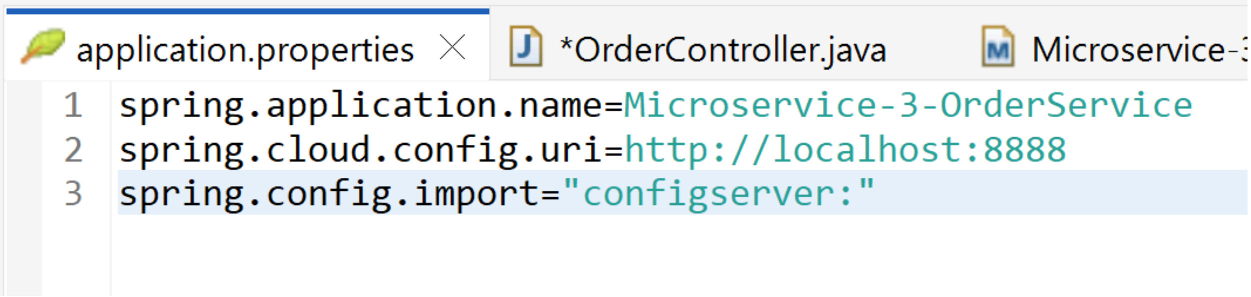
Test the Config Server endpoints

- GET http://localhost:8888/orderservice/default
- GET http://localhost:8888/orderservice/dev

You should see JSON showing the merged property sources.

Read these properties in microservices:

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



```
spring.application.name=Microservice-3-OrderService
spring.cloud.config.uri=http://localhost:8888
spring.config.import="configserver:"
```

spring.application.name=Microservice-3-OrderService
spring.cloud.config.uri=http://localhost:8888
spring.config.import="configserver:"

http://localhost:8888/application/default

Actuator:

Get health of service:

```
<dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-actuator</artifactId>
                </dependency>
```

2. Default Endpoints

After adding the dependency and starting your app, you get endpoints like:

- http://localhost:8080/actuator → lists available actuator endpoints
- http://localhost:8080/actuator/health → shows app health
- http://localhost:8080/actuator/info → shows app info

a) Expose all endpoints

By default, only health and info are exposed. To expose all:

management.endpoints.web.exposure.include=*

Or selectively:

management.endpoints.web.exposure.include=health,info,metrics

b) Change Actuator base path

management.endpoints.web.base-path=/manage

Now endpoints will be under /manage/*.

c) Customize health details

management.endpoint.health.show-details=always

Step 9: Spring Cloud Gateway

It sits between clients and your microservices and handles:

- Routing (forwarding requests to services)
  Cross-cutting concerns (logging, security, rate limiting, monitoring, etc.)

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

2. Configure Routes in application.yml

Here’s an example for routing requests:

```
spring:
  application:
    name: api-gateway

  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8081   # your user service URL
          predicates:
            - Path=/users/**

        - id: order-service
          uri: http://localhost:8082   # your order service URL
          predicates:
            - Path=/orders/**

        - id: product-service
          uri: http://localhost:8083
          predicates:
            - Path=/products/**
```

👉 Now:

- http://localhost:8080/users/** → goes to User Service (8081)

- http://localhost:8080/orders/** → goes to Order Service (8082)
- http://localhost:8080/products/** → goes to Product Service (8083)

🔹 3. Enable Discovery (Optional, if using Eureka)

If you use **Eureka** or another discovery server, you can make the Gateway dynamic:

```
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lower-case-service-id: true
```

Now routes are automatically created from registered services, e.g.:
http://localhost:8080/USER-SERVICE/users/1

🔹 4. Filters

Filters allow you to modify requests/responses.

Example: Add a Request Header

```
spring:
  cloud:
    gateway:
      routes:
        - id: order-service
          uri: http://localhost:8082
          predicates:
            - Path=/orders/**
          filters:
            - AddRequestHeader=X-Request-Source, Gateway
```

Example: Strip Prefix

```
filters:
 - StripPrefix=1
```

If client calls /api/orders/1, it becomes /orders/1 when forwarded.

🔹 5. Security (Optional)

You can add Spring Security + JWT/OAuth2 to secure APIs. Example dependencies:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

This lets the Gateway handle **authentication/authorization** in one place.

🔹 6. Run Gateway

- Run the API Gateway on port 8080.
- All requests go through the gateway and get routed to microservices.