**Microservices**

**https://github.com/in28minutes/spring-microservices-v3**

**Microservices** is an architectural style used in software development where an application is composed of small, independently deployable services. Each service implements a specific business capability, runs in its own process, and typically communicates with other services through lightweight mechanisms—often HTTP-based APIs or messaging.

Below is an overview of the key aspects and benefits of a Microservices architecture:

**1. Core Principles of Microservices**

1. **Single Responsibility**  
   Each microservice does one thing and does it well (i.e., implements exactly one business capability or a small set of closely related functions).
2. **Autonomy**  
   Each microservice is developed, deployed, and managed independently of others. Teams can choose their own programming languages, frameworks, and data stores, as each service can have its own technology stack.
3. **Communication via APIs**  
   Microservices communicate with each other using lightweight protocols such as REST (HTTP/JSON), gRPC, or messaging (e.g., RabbitMQ, Apache Kafka). This enables loose coupling.
4. **Decentralized Data Management**  
   Each microservice can own its own database. Decentralization avoids shared database coupling, allowing each service to evolve and scale independently.
5. **Continuous Delivery and Deployment**  
   Because of their small size and clear boundaries, microservices can be tested and deployed more frequently and reliably.
6. **Resilience and Fault Isolation**  
   If one service fails or experiences an error, the rest of the application can continue running, minimizing overall downtime.

**2. Advantages of Microservices**

1. **Scalability**  
   Each service can be scaled independently. If one service needs more resources, you can allocate them without affecting the rest of the application.
2. **Agility**  
   Smaller, autonomous teams can iterate faster. This promotes quicker time-to-market and the ability to pivot or adopt new technologies.
3. **Maintainability**  
   Because each service focuses on a small set of functionalities, the codebase is easier to maintain, test, and evolve.
4. **Technology Freedom**  
   Different teams can choose the most suitable technology stack for each service, which can help adopt the best tool for a particular job.
5. **Fault Tolerance**  
   The application can remain partially functional even if certain microservices fail.

**3. Challenges and Considerations**

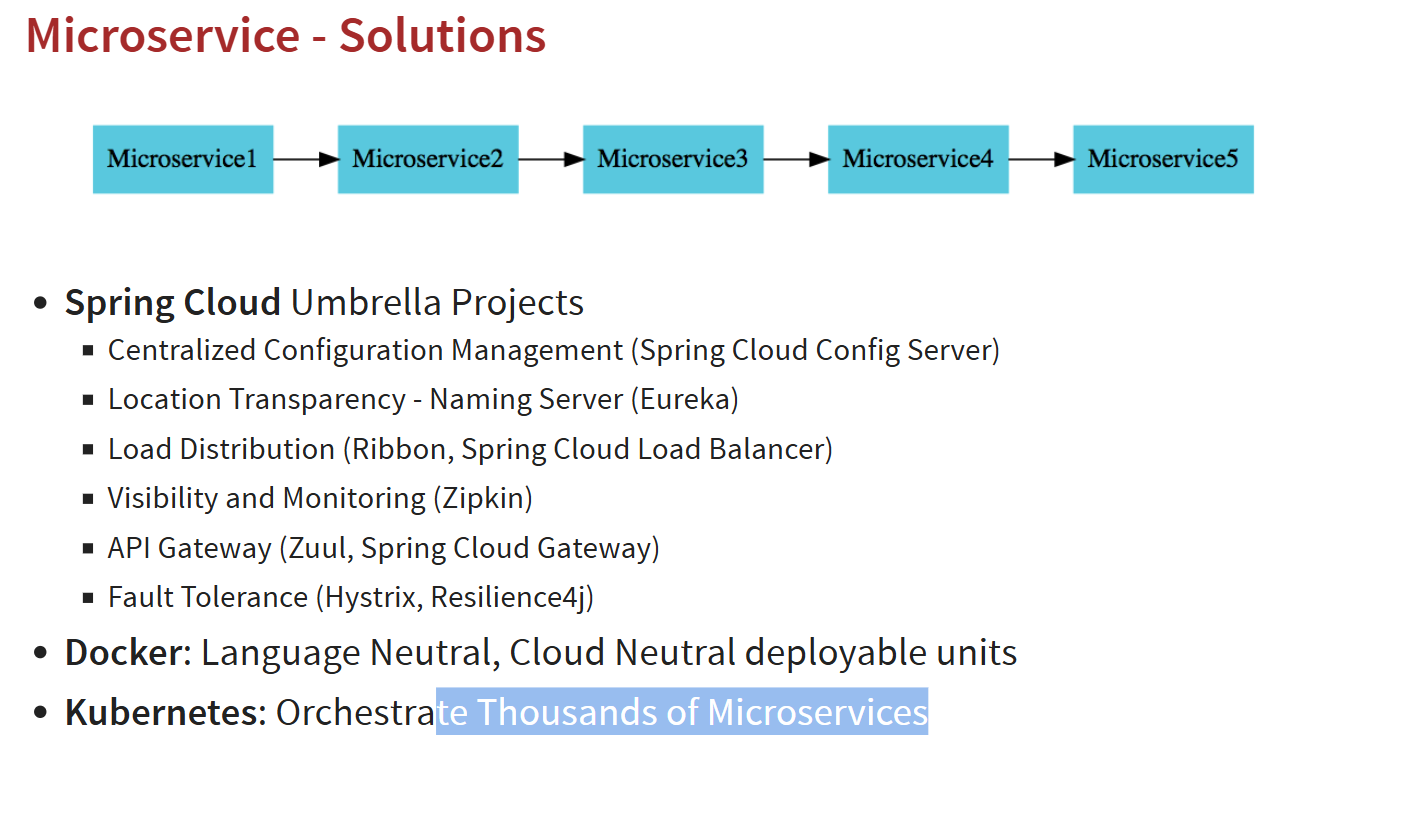
1. **Complexity of Distributed Systems**  
   The network communication between many small services adds complexity in terms of monitoring, logging, load balancing, and handling failures.
2. **Deployment and Orchestration**  
   With many services, managing deployments, updates, and rollbacks can become complex. Container orchestration tools like **Kubernetes** or **Docker Swarm** are commonly used to handle these challenges.
3. **Data Consistency**  
   Decentralized data management can make transactions across multiple services more complex. You may need to use patterns like **Saga** or **Event Sourcing** to maintain consistency.
4. **Observability**  
   Monitoring and tracing the flow of requests across multiple services requires sophisticated logging, metrics, and distributed tracing solutions (e.g., **Prometheus**, **Grafana**, **Jaeger**, **Zipkin**).
5. **Service Discovery**  
   You need a mechanism to let services find and communicate with each other. Tools such as **Eureka**, **Consul**, or **Zookeeper** are often used for service discovery.
6. **Security**  
   Each microservice must handle authentication, authorization, and secure communication (TLS/SSL). This can significantly increase the effort required to implement a secure system.

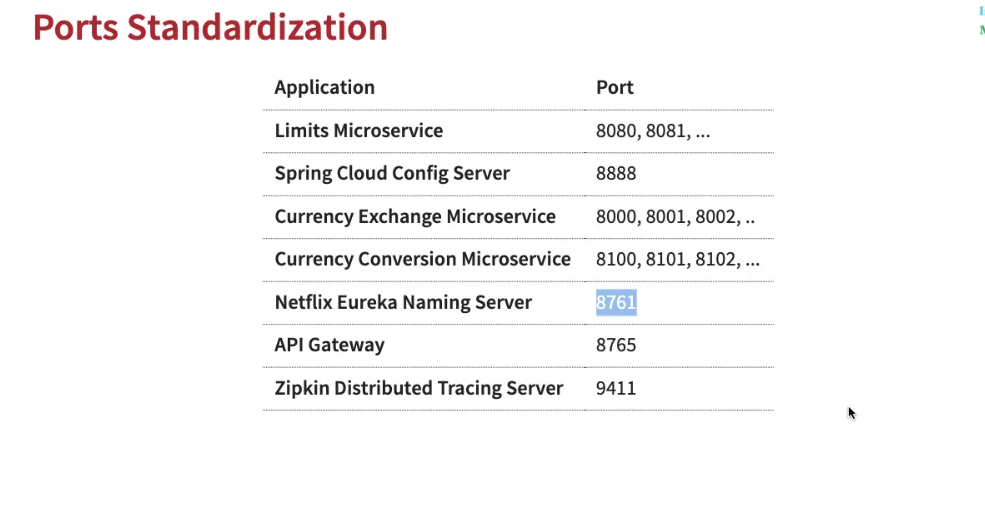
**4. Common Best Practices**

1. **API Gateways**  
   Use an API Gateway (e.g., **Kong**, **NGINX**, **Amazon API Gateway**) to provide a single entry point to the microservices. This helps with routing, throttling, caching, and security.
2. **CI/CD Pipelines**  
   Automate build, test, and deployment processes. This ensures faster releases and rapid feedback loops.
3. **Use Domain-Driven Design (DDD)**  
   Align microservices with **bounded contexts** in DDD. This mapping between business domains and services helps maintain clear service boundaries.
4. **Implement Resilience Patterns**  
   Use **circuit breakers**, **bulkheads**, **timeouts**, and **retries** to handle failures gracefully and prevent cascading failures.
5. **Containerization**  
   Package each service in its own container with all its dependencies to ensure reproducible environments and easier deployment.
6. **Automated Testing Strategy**  
   Employ a layered testing approach, including unit testing for each microservice, integration testing for related services, and end-to-end tests for the entire system.

**5. Example Technology Stack**

* **Runtime**: Java (Spring Boot), Node.js (Express), Python (FastAPI), Go, etc.
* **Communication**: REST over HTTP/JSON, gRPC, Kafka, RabbitMQ.
* **Containers & Orchestration**: Docker, Kubernetes, Docker Swarm.
* **Service Discovery**: Netflix Eureka, HashiCorp Consul.
* **API Gateway**: Kong, NGINX, AWS API Gateway.
* **Observability**: Prometheus, Grafana, Elastic Stack, Zipkin, Jaeger.
* **Security**: JWT (JSON Web Tokens), OAuth 2.0, TLS/SSL, mutual TLS.
* **CI/CD**: Jenkins, GitLab CI, GitHub Actions, Azure DevOps.

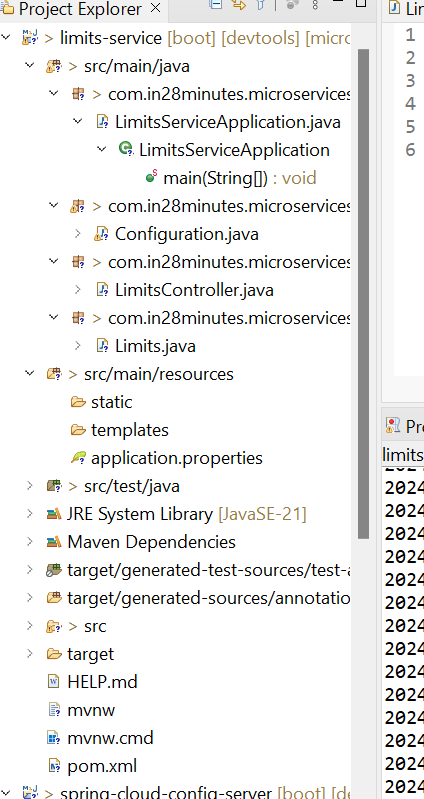




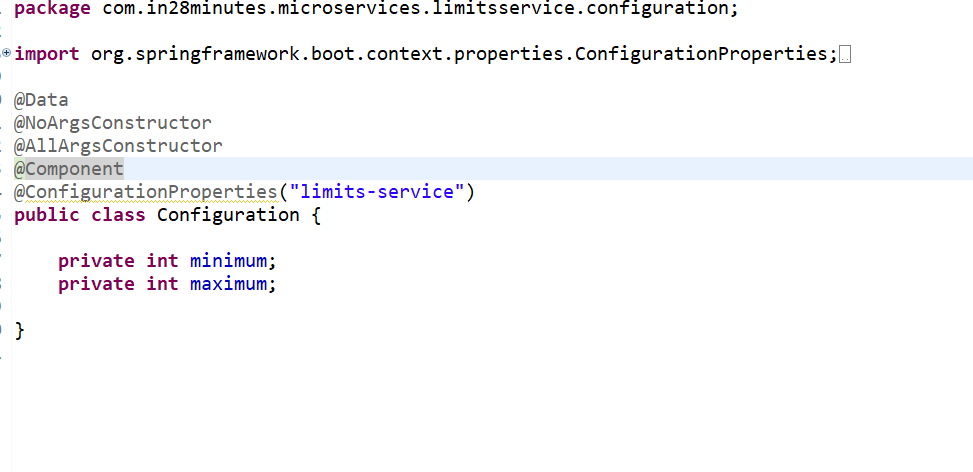
**Project Creation**

* 1. Limit-service project or service

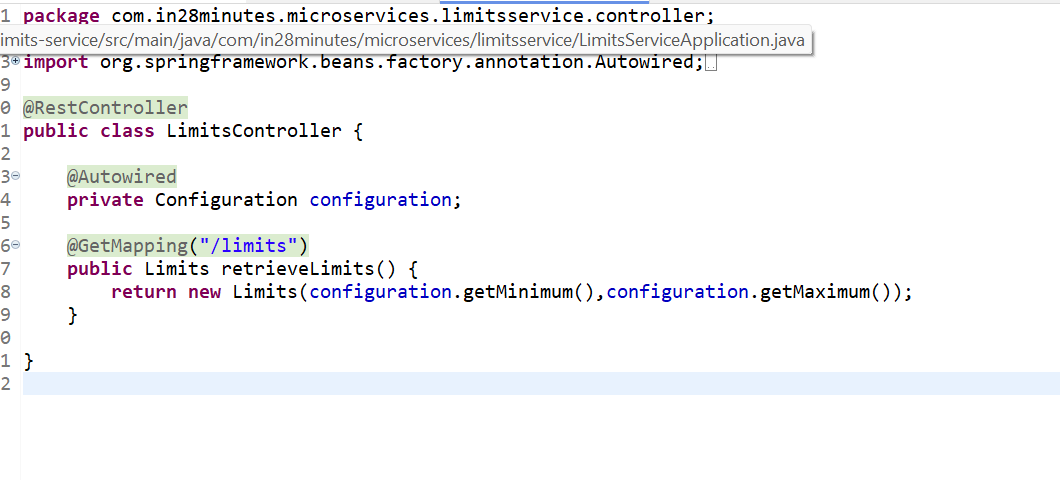
Project Structure

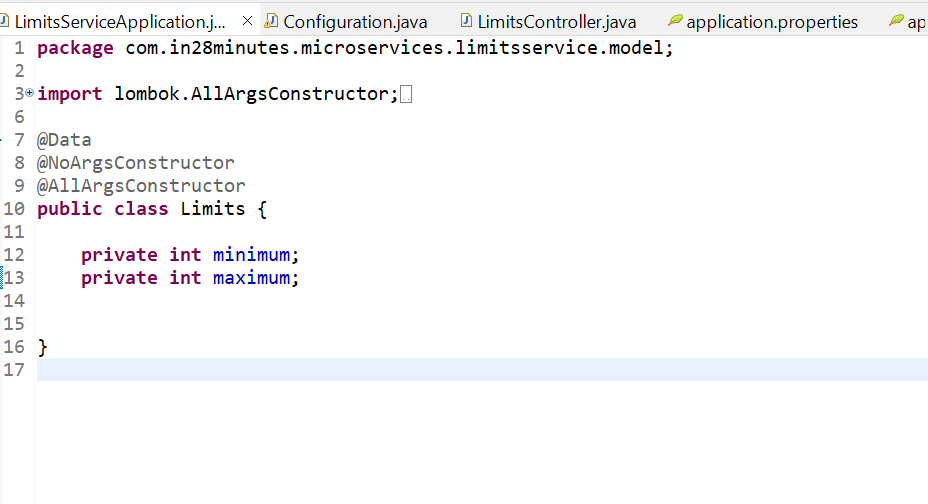


Configuration class



Controller class LimitController.java



Bean class  


Application.properties



Run SpringBoot Application and hit below request

<http://localhost:8080/limits>

Response

{

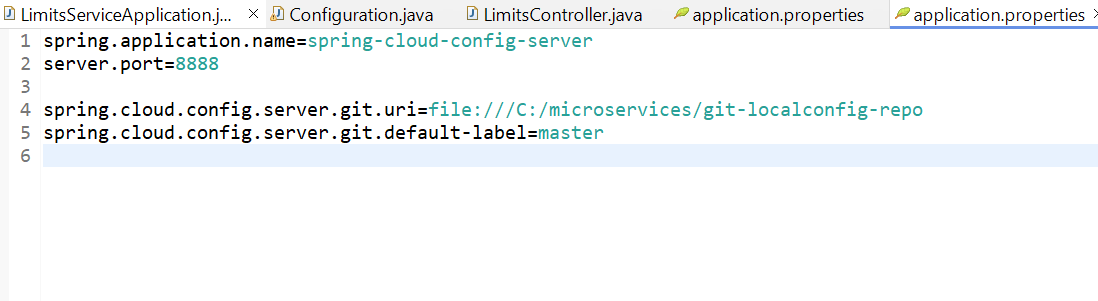
"minimum": 3,

"maximum": 997

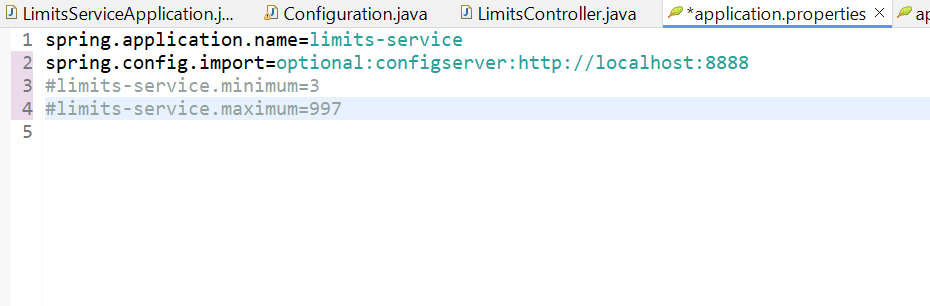
}

Means application is working fine.

* 1. Spring-cloud-config-server



Update the application.properties of limit-service service

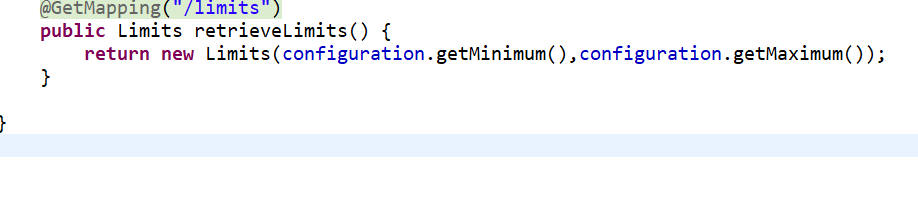


Run the Spring-cloud-config-server and it should run at 8888

**Code Connections**

**LimitsController**

* Calls Configuration to get values:

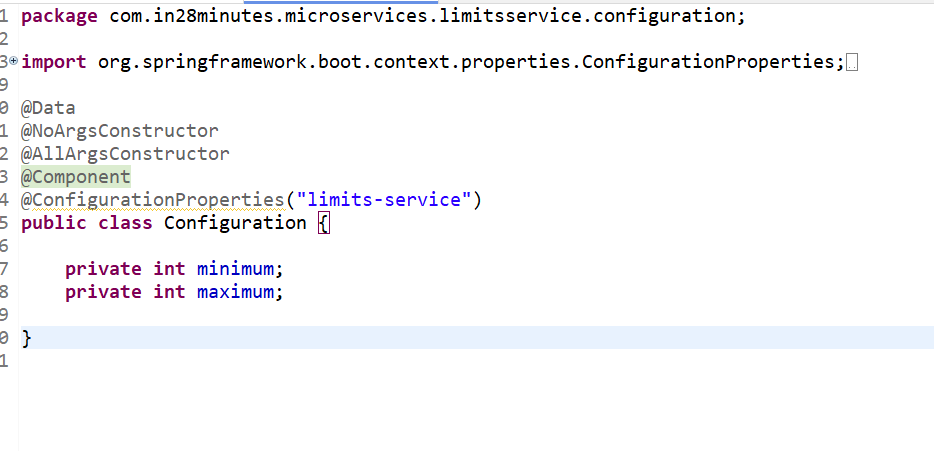


**Configuration**

* Retrieves values from the Config Server:

java

Copy code



**Spring Cloud Config Server**

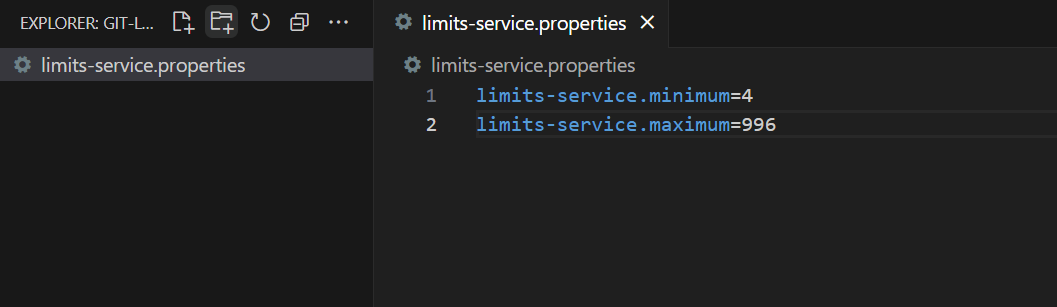
* Fetches properties from Git repository:

spring.cloud.config.server.git.uri=file:///C:/microservices/git-localconfig-repo

spring.cloud.config.server.git.default-label=master

**Git Repository**

* Stores limits-service.properties:



**Project Explanation**

I understand everything you've shared. You have created two microservices:

1. **limits-service**:
   * Fetches configuration values (minimum and maximum) dynamically.
   * Values are fetched from a configuration server.
2. **spring-cloud-config-service**:
   * Acts as a central configuration server.
   * Fetches configuration from a Git repository (C:/microservices/git-localconfig-repo).

Both services are running successfully, and you're able to see the expected outputs. You're essentially exploring **Spring Cloud Config Server** and learning how microservices can fetch configurations dynamically from a centralized configuration repository.

**Classes in limits-service**

1. **LimitsController**
   * This is a REST controller that exposes the /limits endpoint to the client.
   * It interacts with the Configuration class to fetch configuration values.
2. **Configuration**
   * This is a Spring-managed component that maps configuration properties (limits-service.minimum and limits-service.maximum) from the configuration source.
   * It fetches these values dynamically from the Spring Cloud Config Server.
3. **Limits**
   * This is a simple POJO (Plain Old Java Object) with fields minimum and maximum.
   * It acts as the response model for the /limits endpoint.

**How They Are Connected**

**1. Connection Between LimitsController and Configuration**

* **Dependency Injection**:
  + LimitsController has an @Autowired field for Configuration.
  + Spring automatically injects an instance of Configuration into LimitsController during runtime.

Ex

@Autowired private Configuration configuration;

When the /limits endpoint is called, LimitsController uses the Configuration class to fetch the minimum and maximum values:

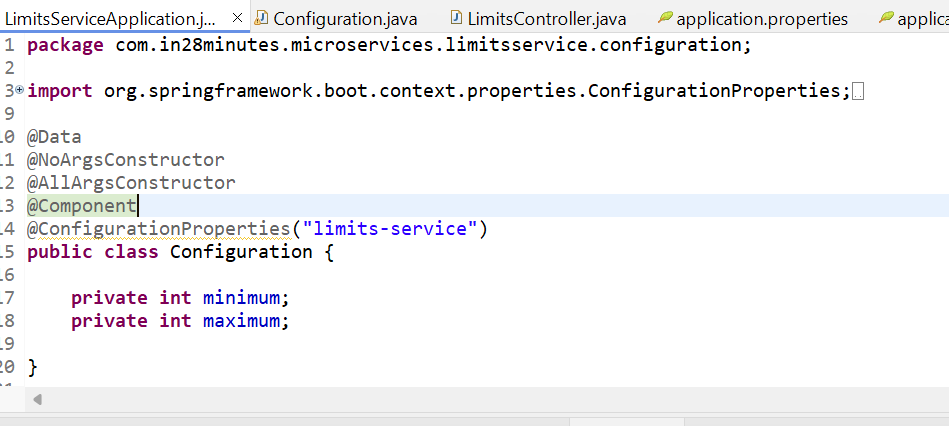
@GetMapping("/limits") public Limits retrieveLimits() {

return new Limits(configuration.getMinimum(), configuration.getMaximum());

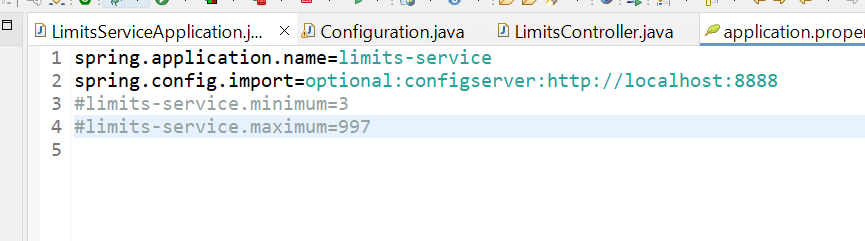
}

**2. Connection Between Configuration and Spring Cloud Config Server**

* **Spring Cloud Configuration Binding**:
  + The @ConfigurationProperties("limits-service") annotation in the Configuration class tells Spring to bind properties prefixed with limits-service to this class.
  + The actual values for these properties (limits-service.minimum and limits-service.maximum) are fetched from the Spring Cloud Config Server at runtime.



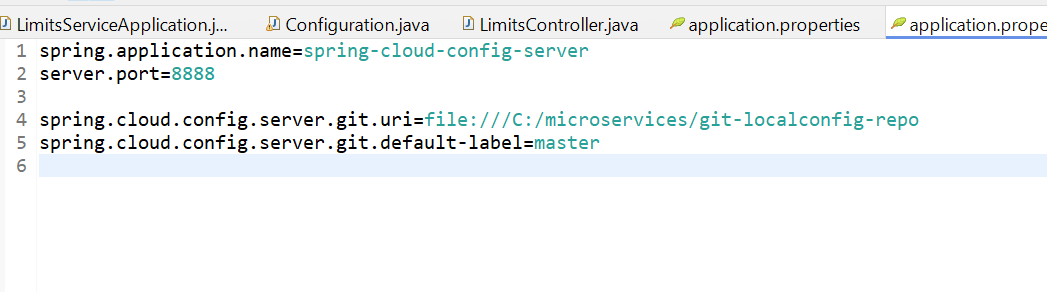
The spring.config.import property in application.properties of limits-service specifies the location of the Config Server:



When limits-service starts, it sends a request to the Config Server (http://localhost:8888) to retrieve configuration values.

**3. Connection Between Spring Cloud Config Server and Git Repository**

* **Spring Cloud Config Server** retrieves the property files from the Git repository. This is configured in application.properties of the spring-cloud-config-service project:

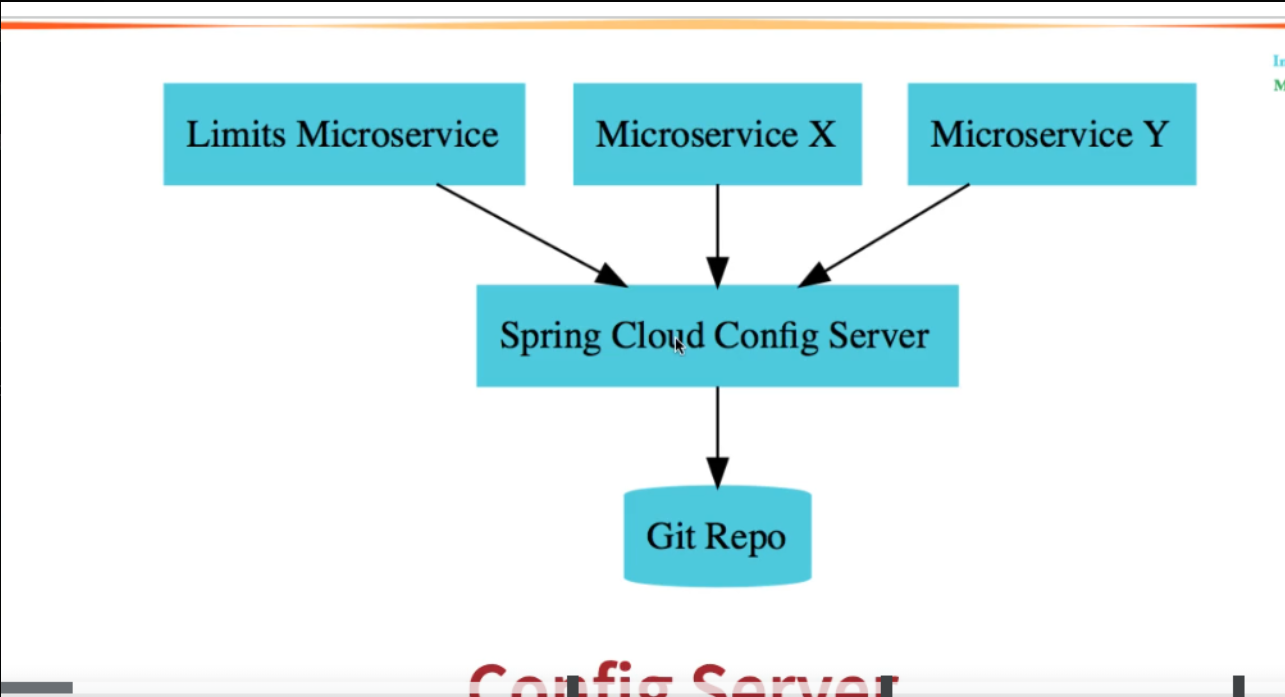


or example, when limits-service starts, it requests its configuration from the Config Server:

* URL: http://localhost:8888/limits-service/default
* The Config Server fetches the limits-service.properties file from the Git repository (C:/microservices/git-localconfig-repo) and returns the properties to limits-service.

**Request Flow**

1. **Client Sends Request to /limits**
   * When you hit http://localhost:8080/limits, the request is handled by LimitsController.
2. **LimitsController Fetches Configuration**
   * The controller calls Configuration to get the minimum and maximum values.
3. **Configuration Fetches Properties**
   * The Configuration class fetches the limits-service.minimum and limits-service.maximum properties from the Spring Cloud Config Server.
4. **Config Server Fetches Properties from Git**
   * The Spring Cloud Config Server retrieves the limits-service.properties file from the Git repository and provides the values to Configuration.
5. **Response is Returned**
   * LimitsController creates a Limits object with the minimum and maximum values and returns it to the client.



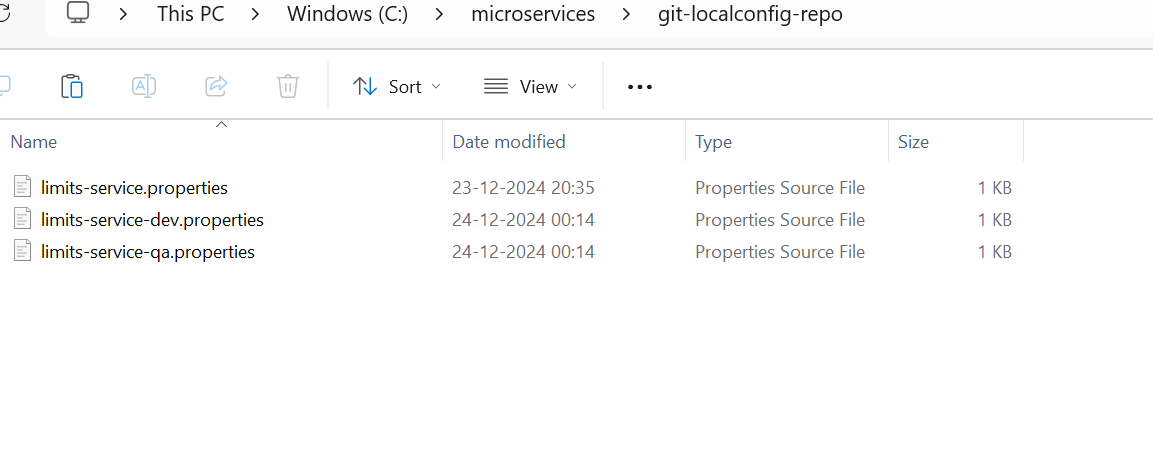
**Diagram of Connections**

1. **Client** → **LimitsController**
   * Client makes a request to /limits.
2. **LimitsController** → **Configuration**
   * Controller fetches configuration properties.
3. **Configuration** → **Spring Cloud Config Server**
   * Fetches properties dynamically.
4. **Spring Cloud Config Server** → **Git Repository**
   * Retrieves limits-service.properties.

**Key Benefits of this Connection**

1. **Dynamic Configuration Updates**: Any changes in the limits-service.properties file in Git will be reflected in the microservice after a restart or refresh.
2. **Centralized Configuration Management**: All microservices fetch their configurations from a single source, simplifying management.
3. **Loose Coupling**: limits-service does not need to know where the configuration resides; it simply fetches it from the Config Server.

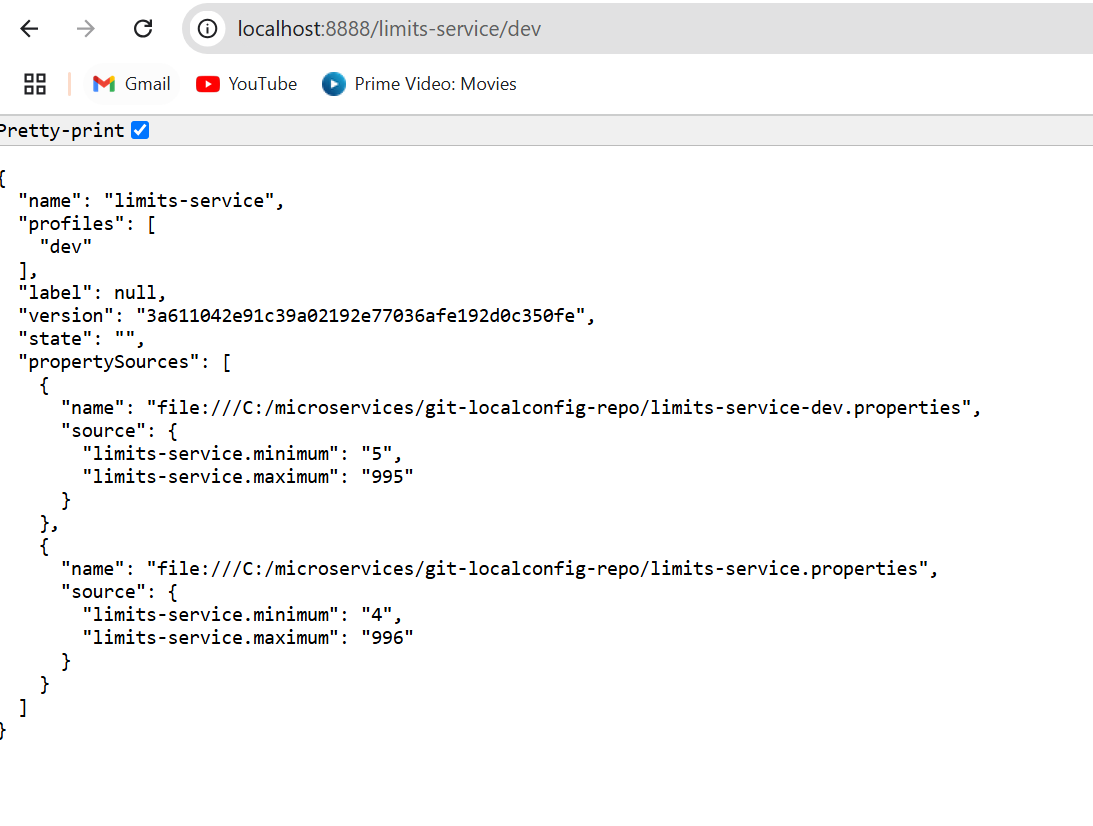
**Configuration Profile for limits-service**

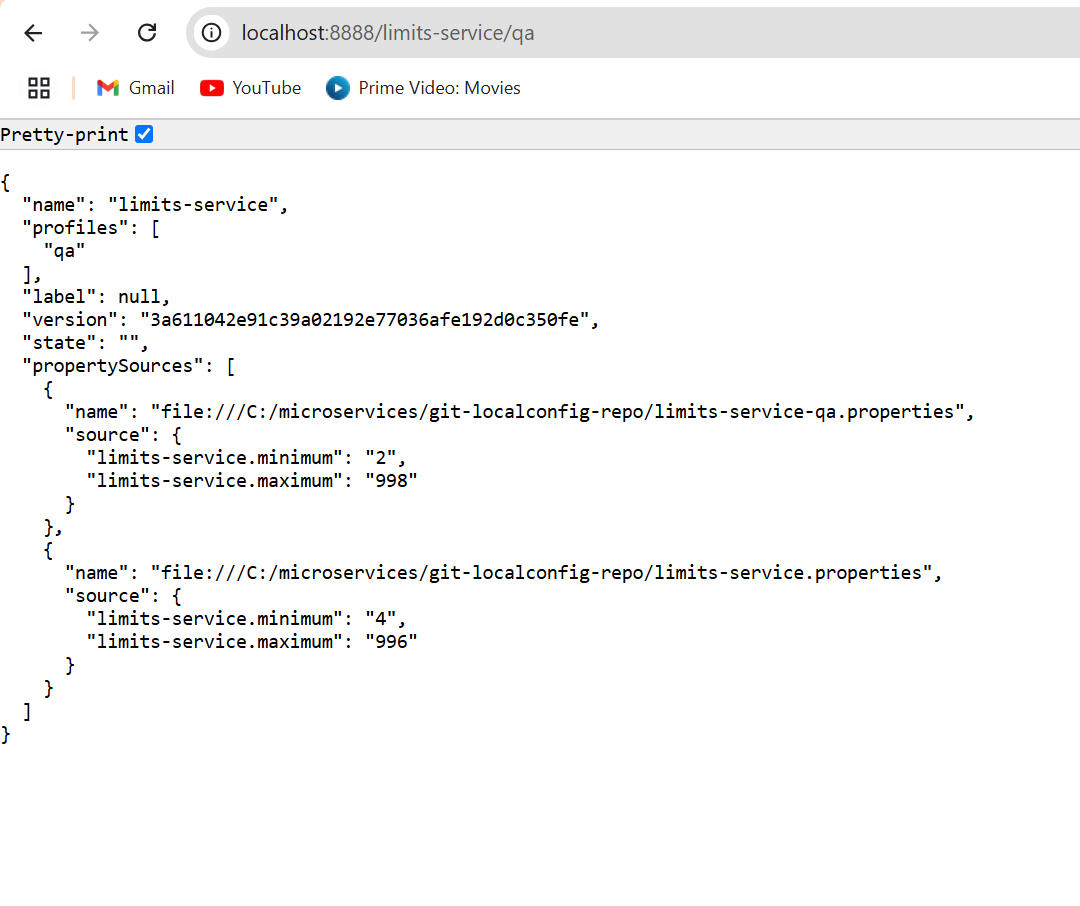
****

Create dev and qa properties file in git local repo

And hit the below urls

<http://localhost:8888/limits-service/dev>





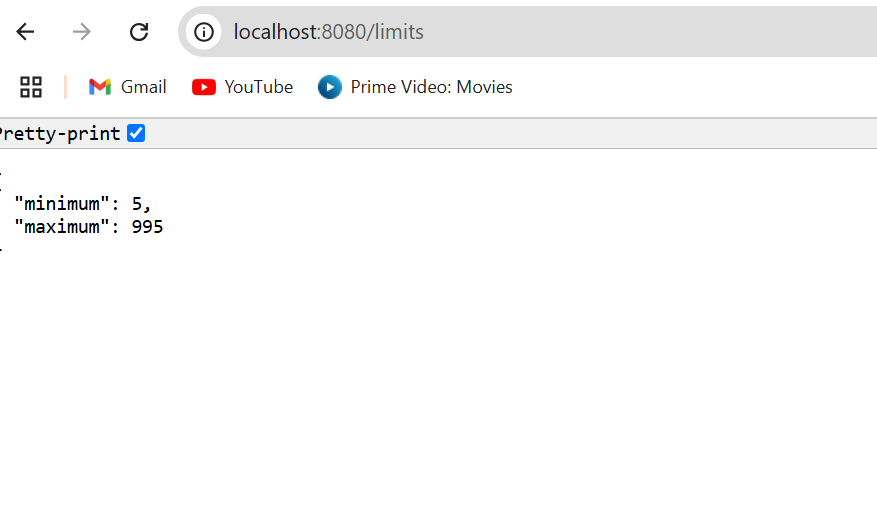
How Application Fetch data from Application means from limits-service

Update the application.properties field

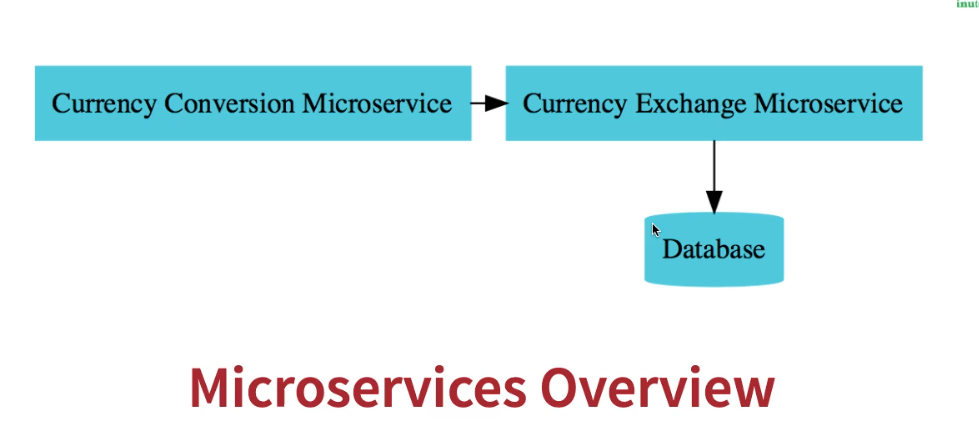


Run the below URL <http://localhost:8080/limits>

Fetching the data from dev profile.



**Second Example**



1.**Create a simple hard coded currency exchange service - V3**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-actuator</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework.cloud</groupId>**

**<artifactId>spring-cloud-starter-config</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-devtools</artifactId>**

**<scope>runtime</scope>**

**<optional>true</optional>**

**</dependency>**

**<dependency>**

**<groupId>org.projectlombok</groupId>**

**<artifactId>lombok</artifactId>**

**<optional>true</optional>**

**</dependency>**

**<dependency>**

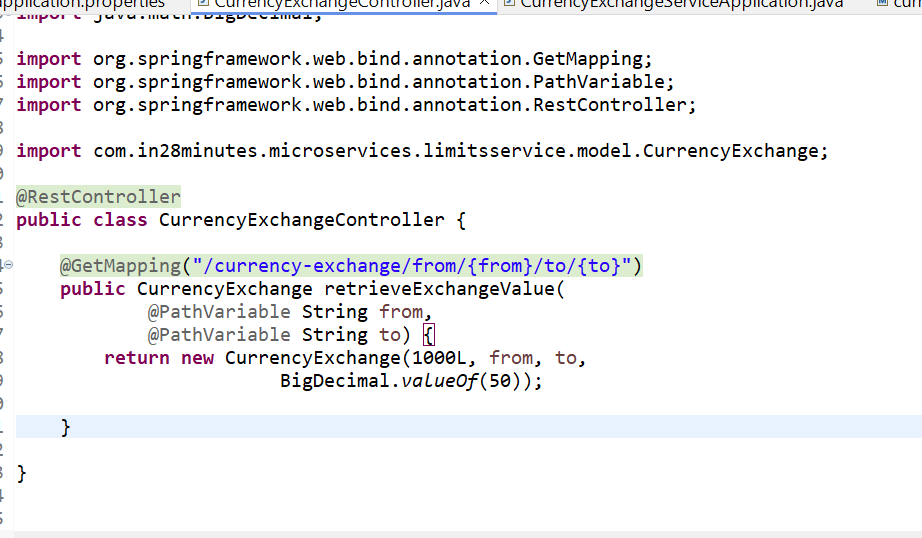
**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-test</artifactId>**

**<scope>test</scope>**

**</dependency>**

**CurrentExchangeController.java**

****

<http://localhost:8000/currency-exchange/from/USD/to/INR>

Response

{

"id": 1000,

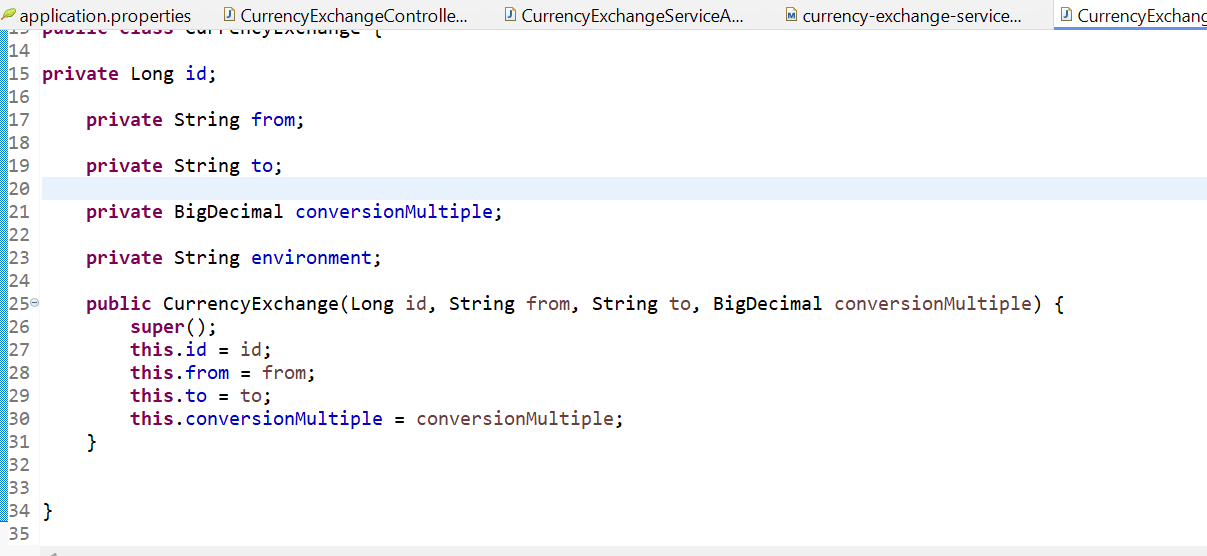
"from": "USD",

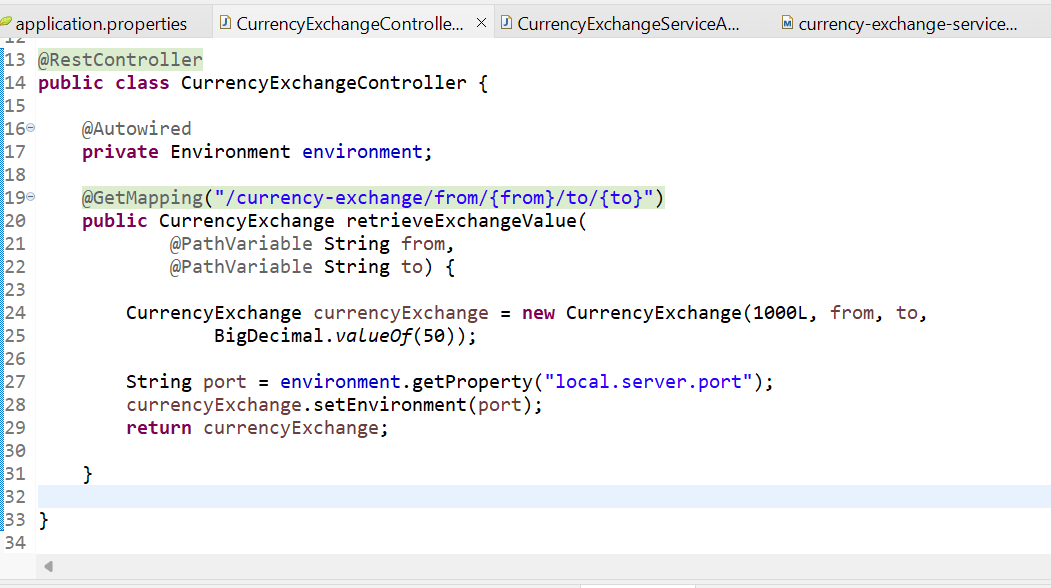
"to": "INR",

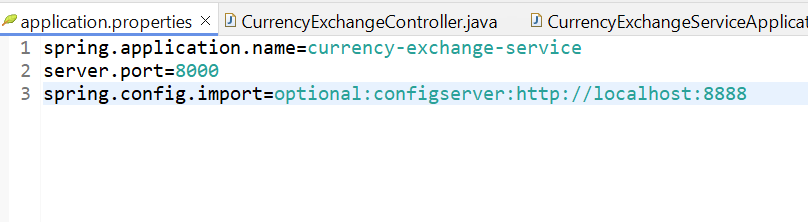
"conversionMultiple": 50,

}

**2.Setting up Dynamic Port in the the Response - V3**

****

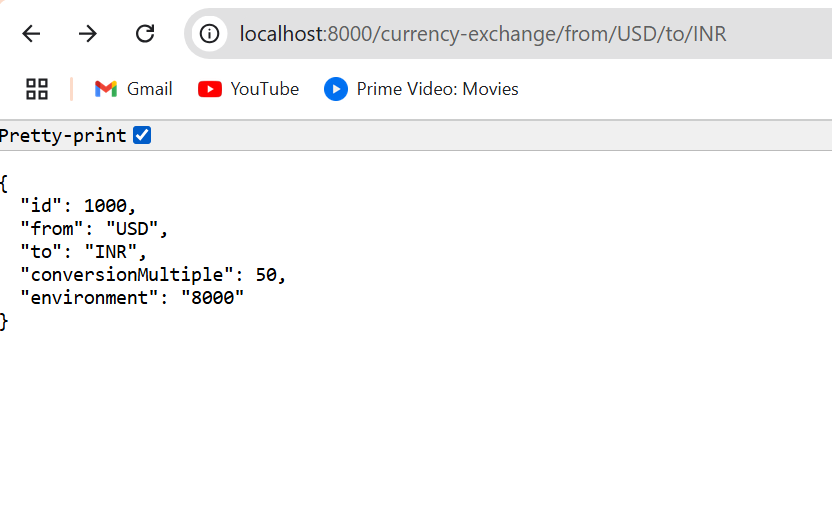
****

****

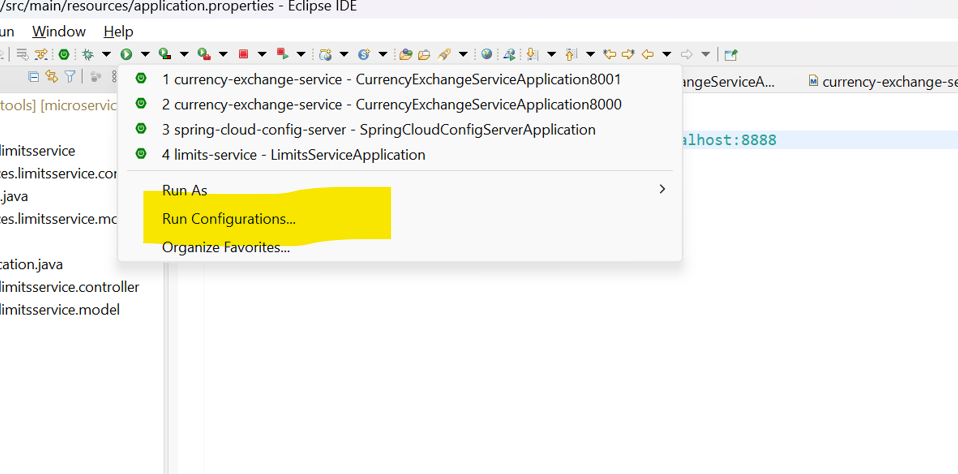
**Hit the URL**

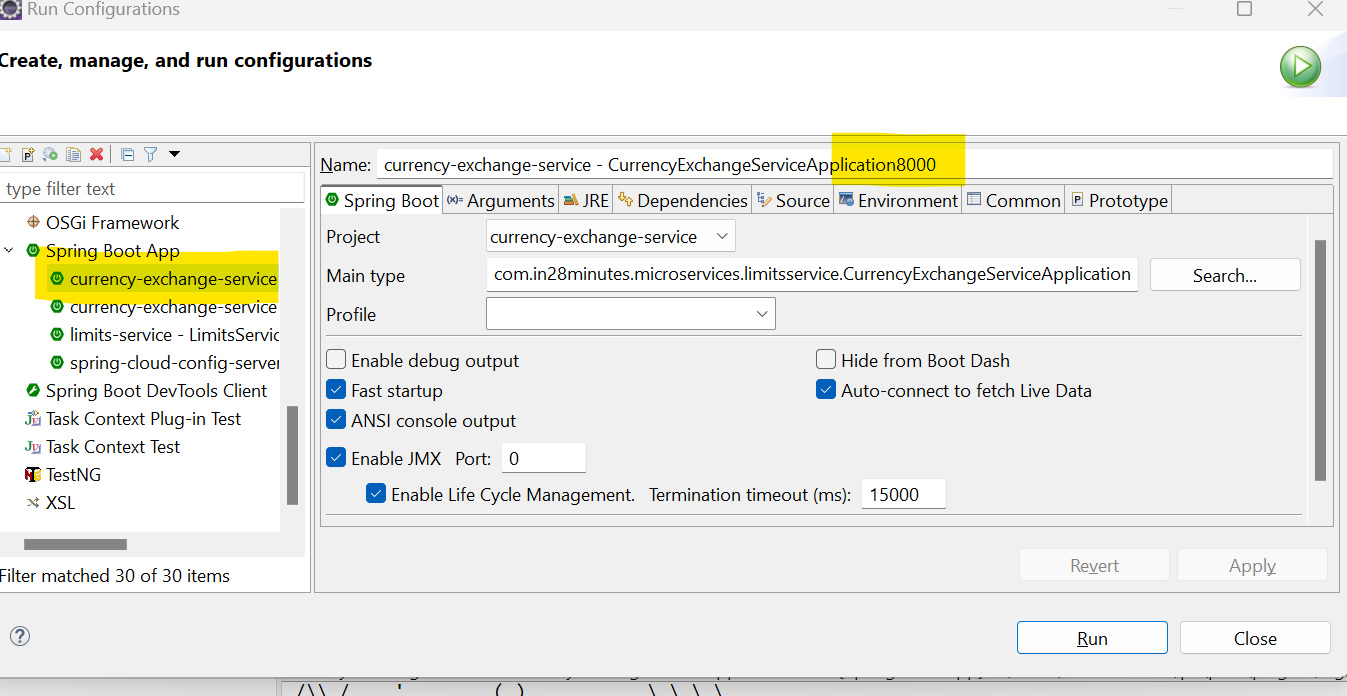
<http://localhost:8000/currency-exchange/from/USD/to/INR>

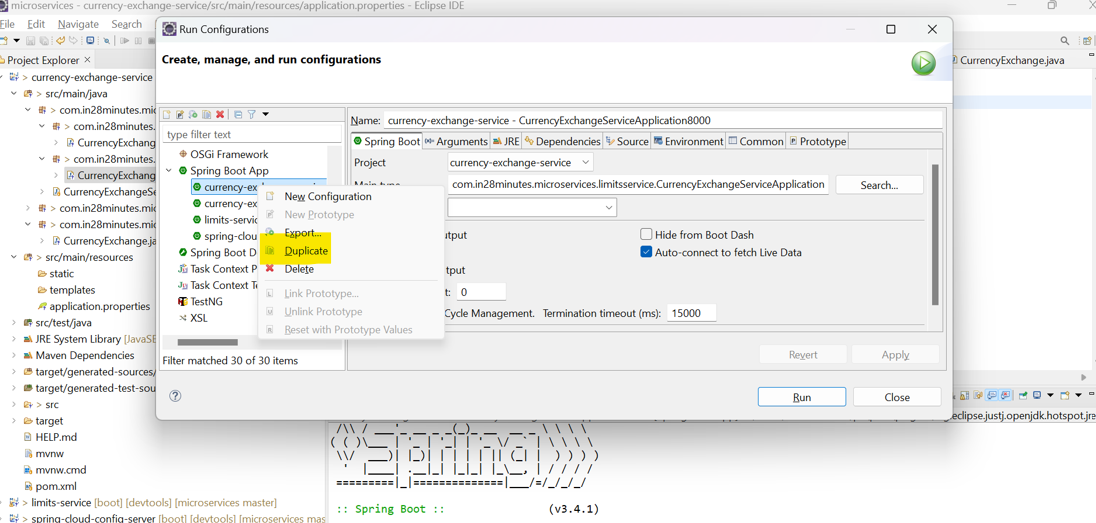
Response:

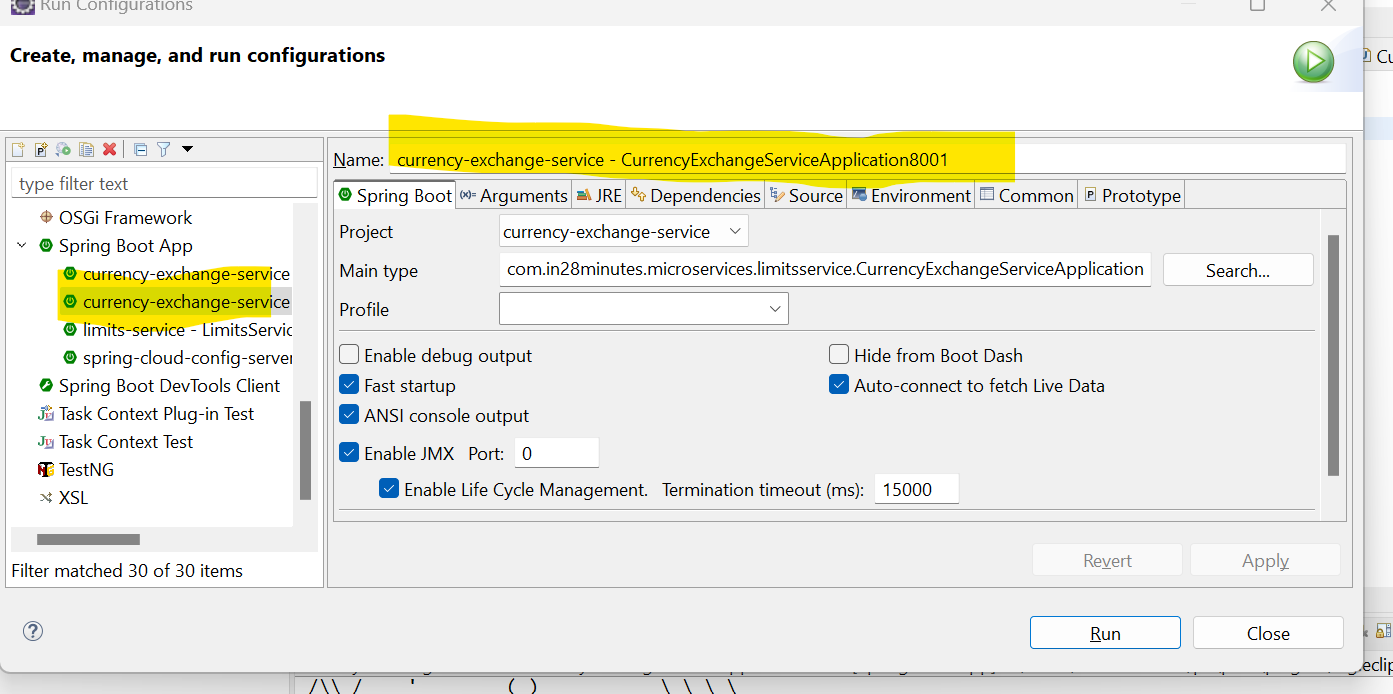


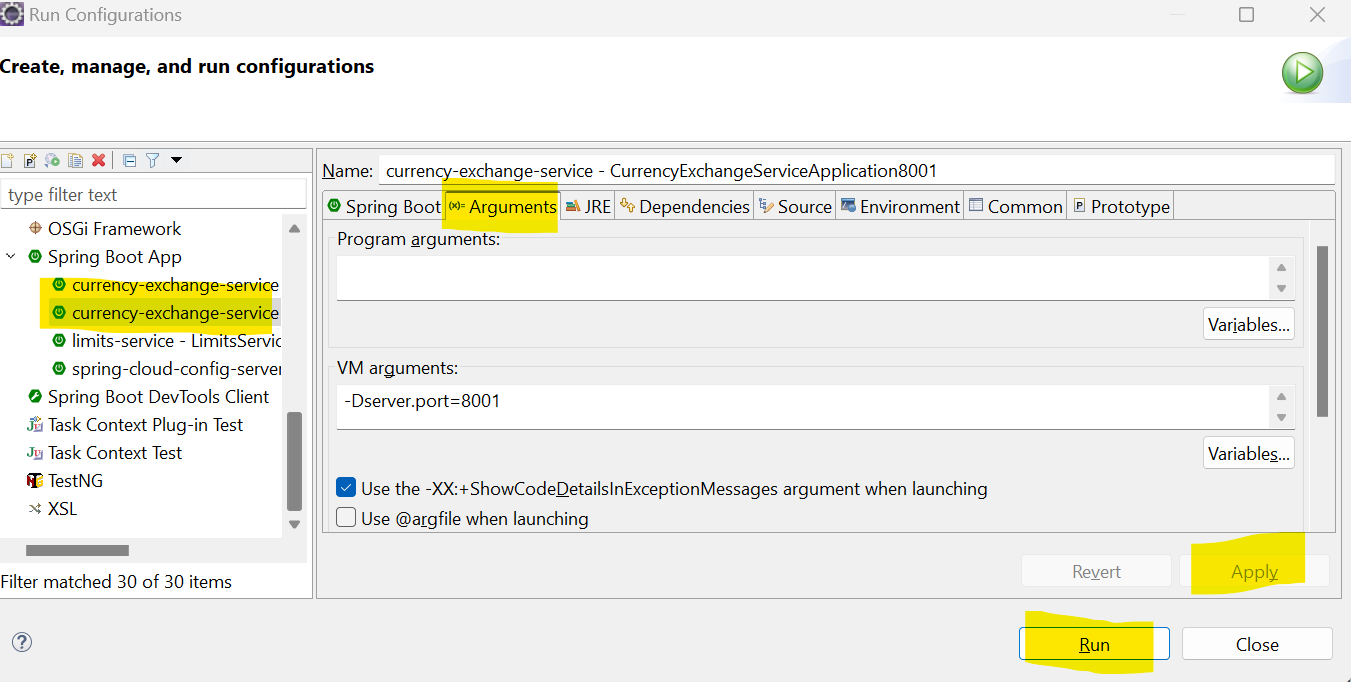
Now

****

****

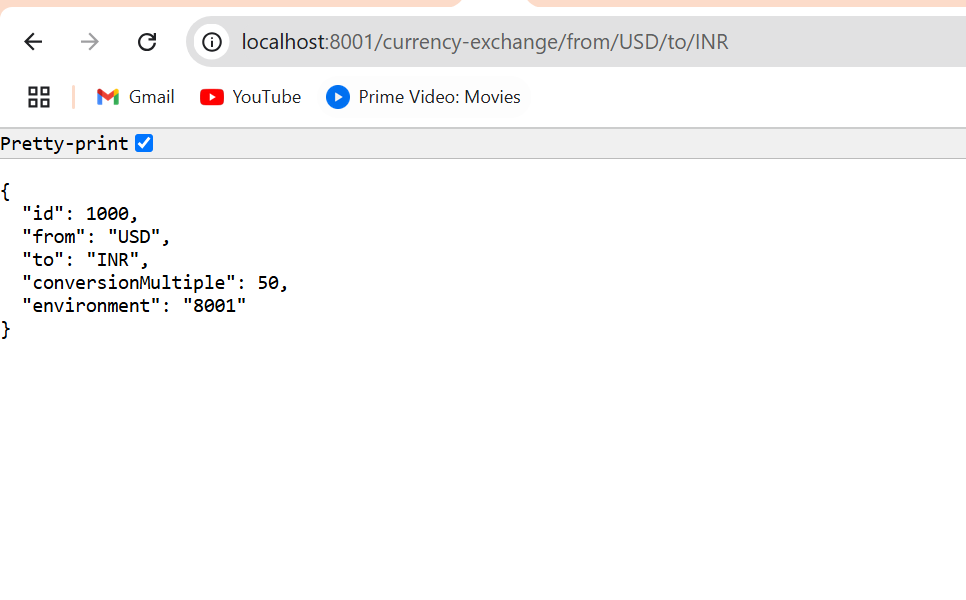
****

****

****

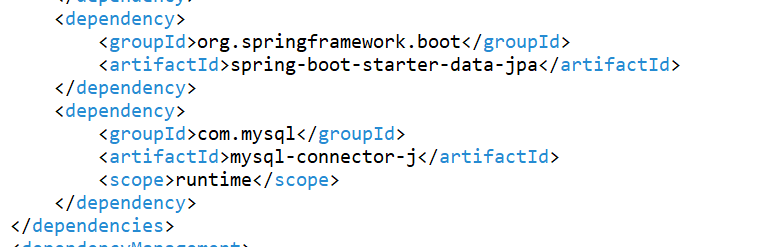
**Hit the URL**

[**http://localhost:8001/currency-exchange/from/USD/to/INR**](http://localhost:8001/currency-exchange/from/USD/to/INR)

****

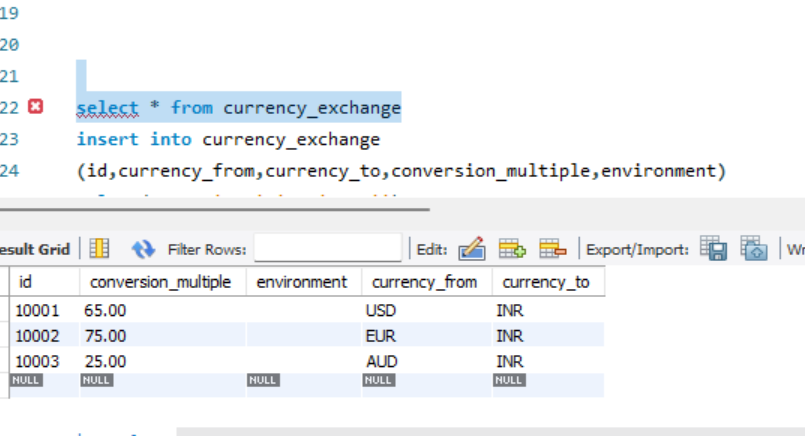
**Configure JPA and Initialized Data - V3**

**Add these two dependency in POM.xml**

****

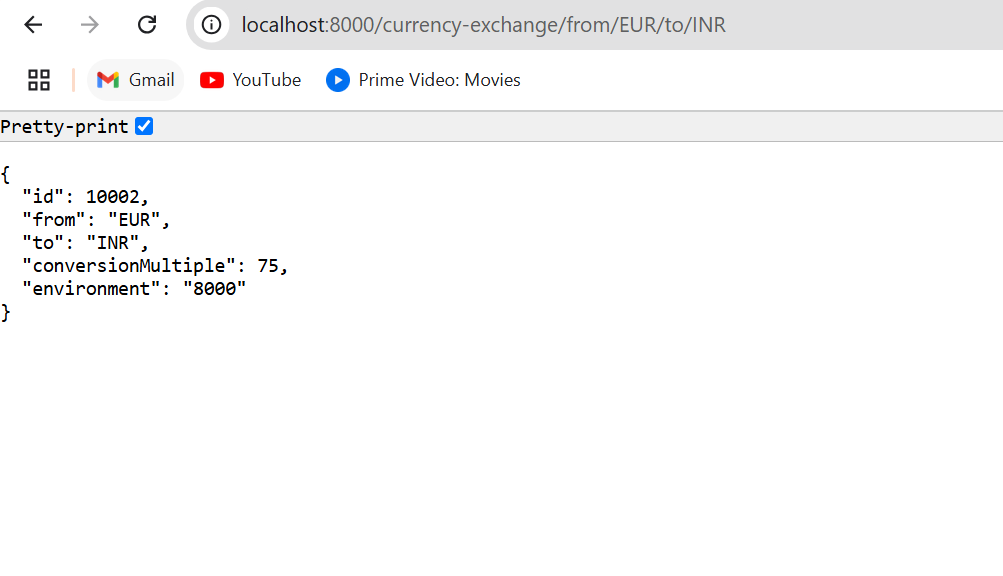
****

**Insert Data**

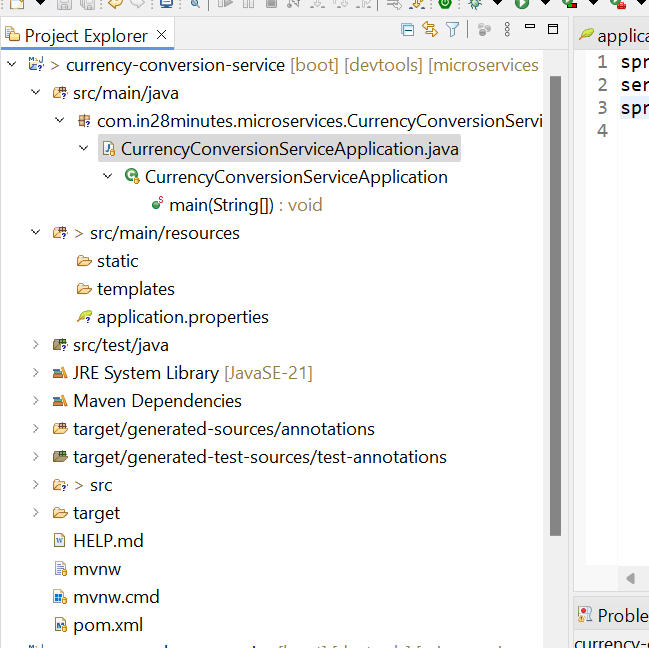
****

**Stop all the servers and rerun**

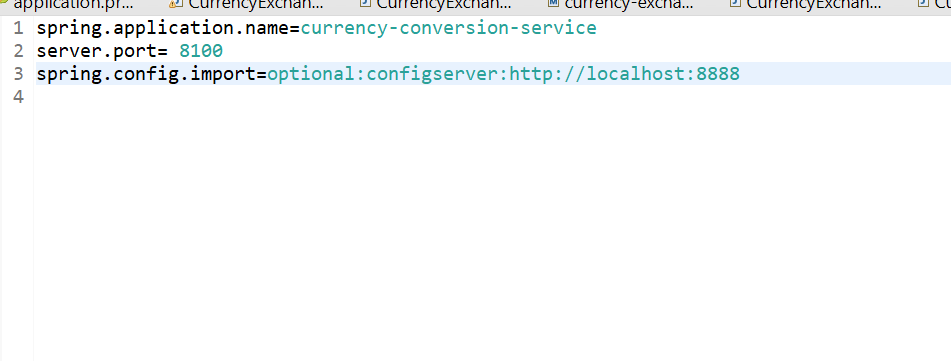
[**http://localhost:8000/currency-exchange/from/EUR/to/INR**](http://localhost:8000/currency-exchange/from/EUR/to/INR)

****

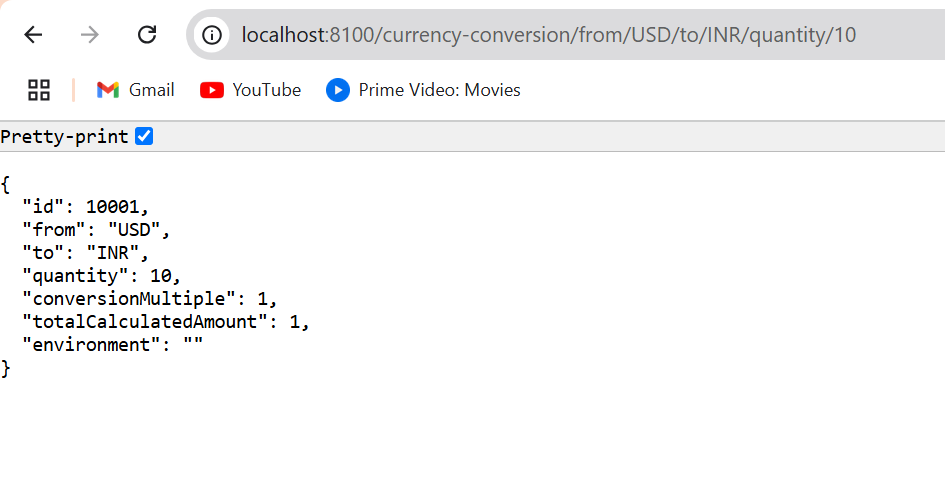
**Creating Currency Conversion service**

****

**Application.properties**

****

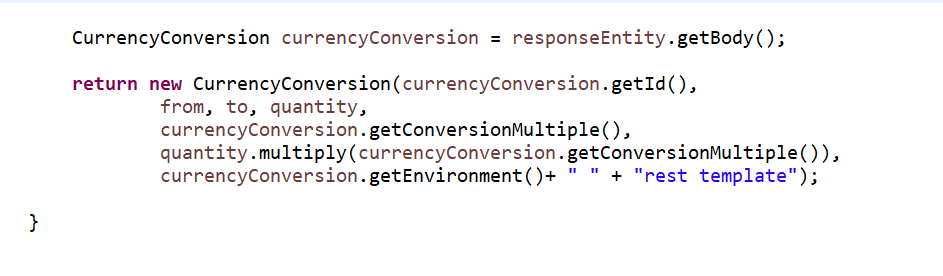
**Hit the below URL**[**http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/10**](http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/10)

****

**Invoking Currency Exchange from Currency Conversion Microservice**

**Updated Controller class**

****

****

**Flow of Execution**

**1. Client Request:**

* When the client calls http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/100, it hits the CurrencyConversionController.

**2. Dynamic Path Variables:**

* The from, to, and quantity values are extracted from the URL using @PathVariable.

**3. Building the Request:**

* A HashMap (uriVariables) is created to hold the path variables (from and to) for dynamic substitution in the URL.
* These variables are then used to replace the placeholders {from} and {to} in the **Currency Exchange Service** URL.

**4. Calling Currency Exchange Service:**

* The RestTemplate's getForEntity() method is used to send an HTTP GET request to the **Currency Exchange Service** running at http://localhost:8000/currency-exchange/from/{from}/to/{to}.
* The {from} and {to} placeholders are dynamically replaced with the values provided by the client (e.g., USD and INR).

**5. Receiving Response:**

* The getForEntity() method sends the request and receives the response from the **Currency Exchange Service**, mapping it to the CurrencyConversion class.

**6. Processing the Response:**

* The response (exchange rate and environment info) is extracted and used to calculate the total converted amount (quantity \* conversionMultiple).

**7. Returning the Result:**

* The final result, including the calculated amount and the environment details, is returned to the client.

**Using Feign REST Client for Service Invocation**

Add dependency in POM.xmk file   
  
<dependency>

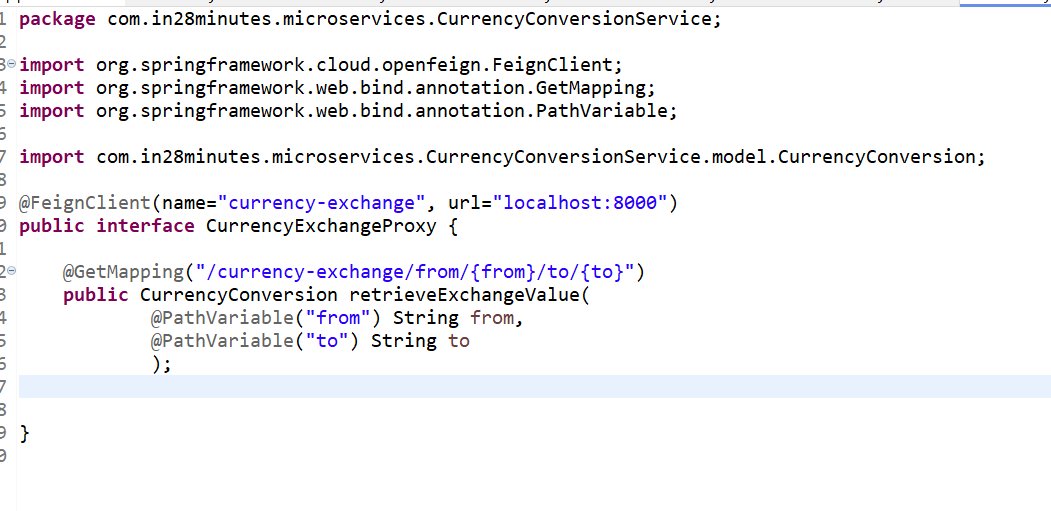
<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

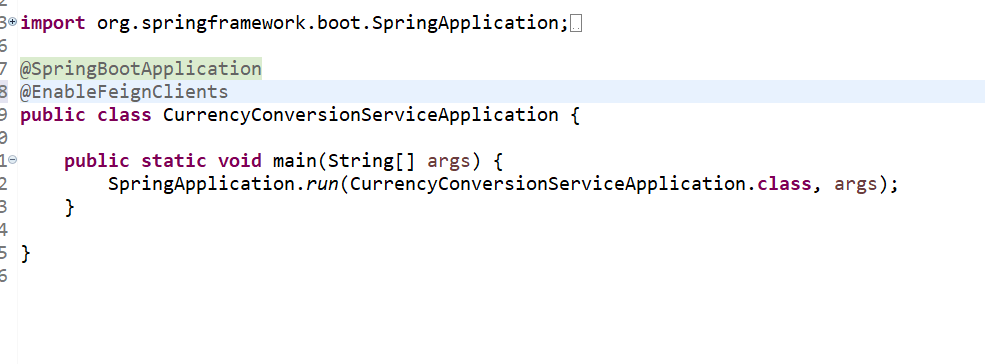
</dependency>



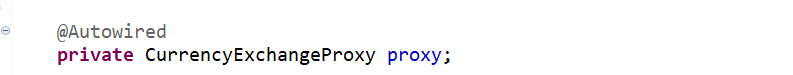
**Create Proxy class**

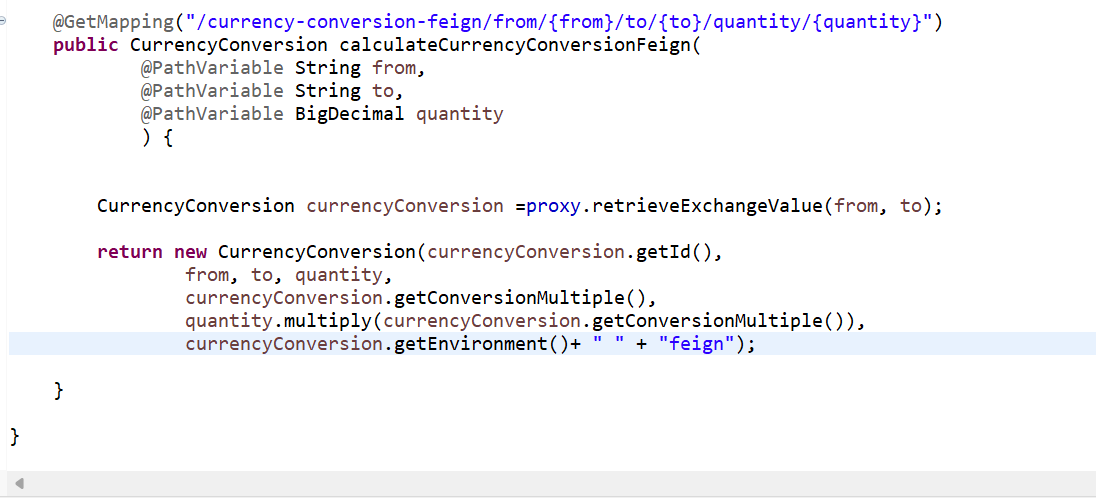
****

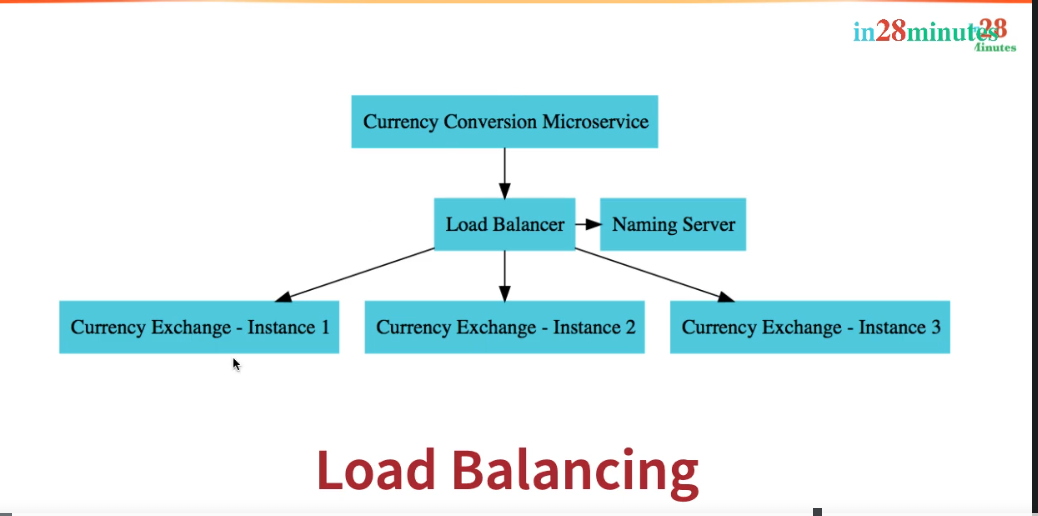
Enable Feign Clients in Main Application Class: Add the @EnableFeignClients annotation to your main class:



**Use the Proxy in Your Controller**: Inject the CurrencyExchangeProxy into your CurrencyConversionController and call the retrieveExchangeValue method.

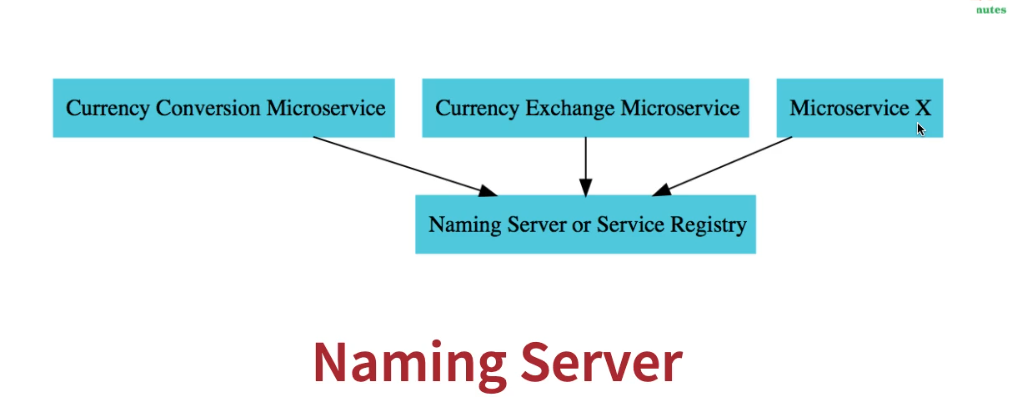


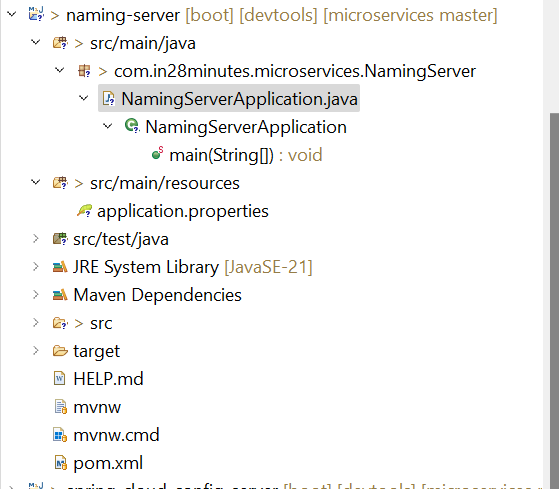


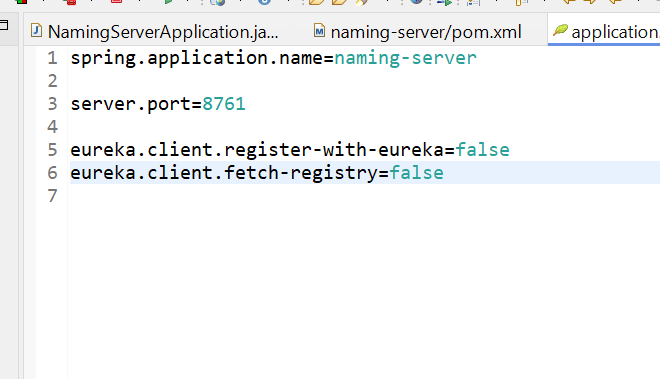


**Service registry and naming Server**

**Understand Naming Server and Setting up Eureka Naming Server**

****

****

****

Run the application and hit this url ‘http://localhost:8761/”. You can see Instances currently registered with Eureka will be **N/A**

**Add below dependency in Pom.xml in Currency-exchange service and Currency-conversion**

<dependency>

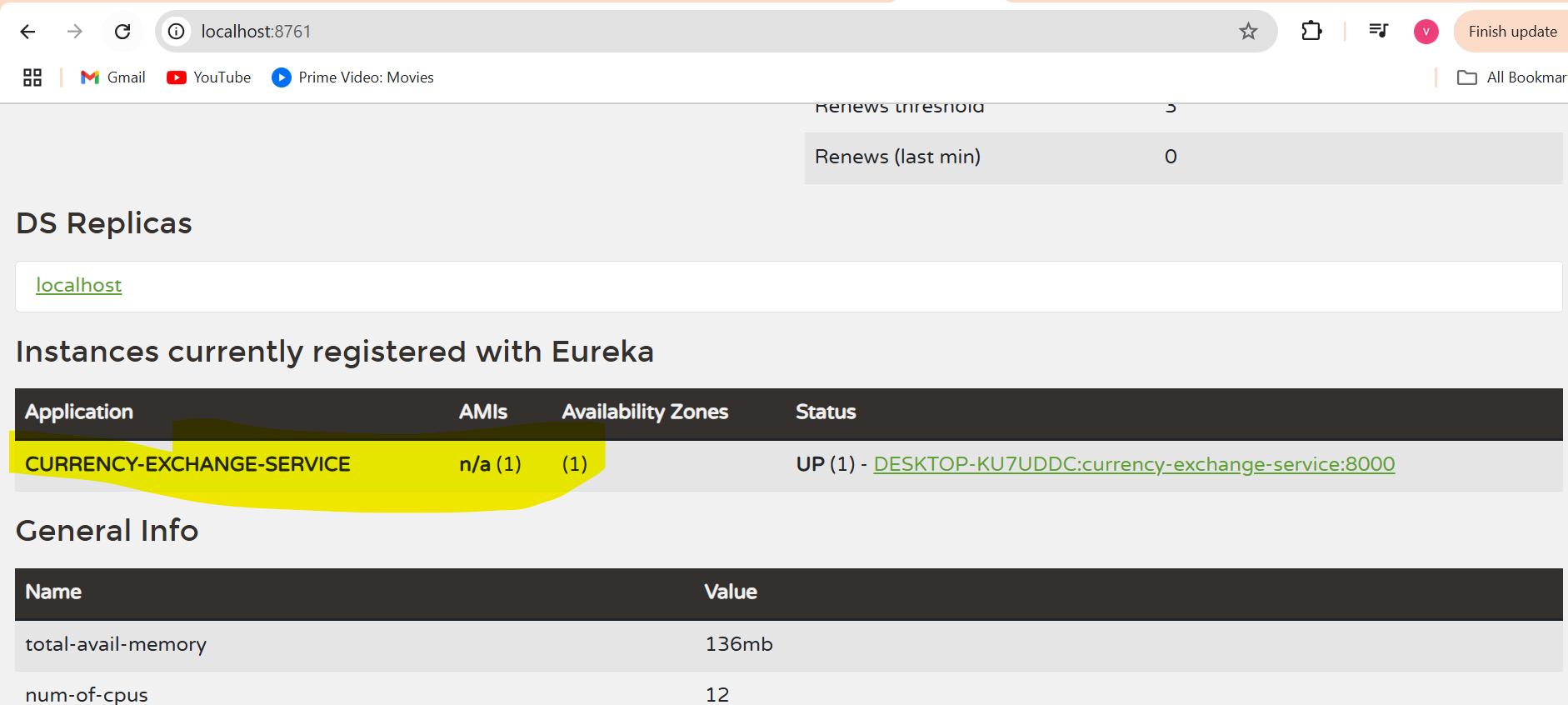
<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Stop all the application and only run naming-server service and currency-Exchange service

**Hit this url ‘http://localhost:8761/”**

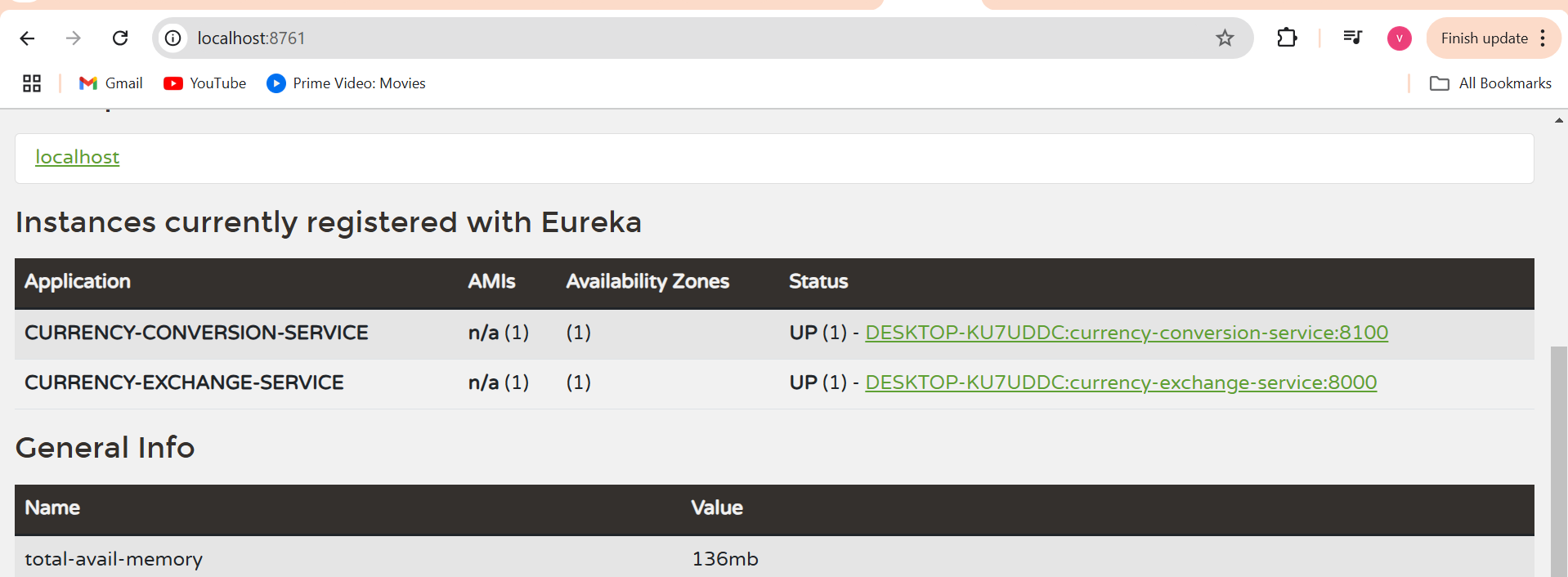
****

**Connect Currency Conversion & Currency Exchange Microservices**

Add below line in application.properties of both currency-exchange and currency-conversion

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

And run the cuurency-conversion application as well and hit this url <http://localhost:8761/>

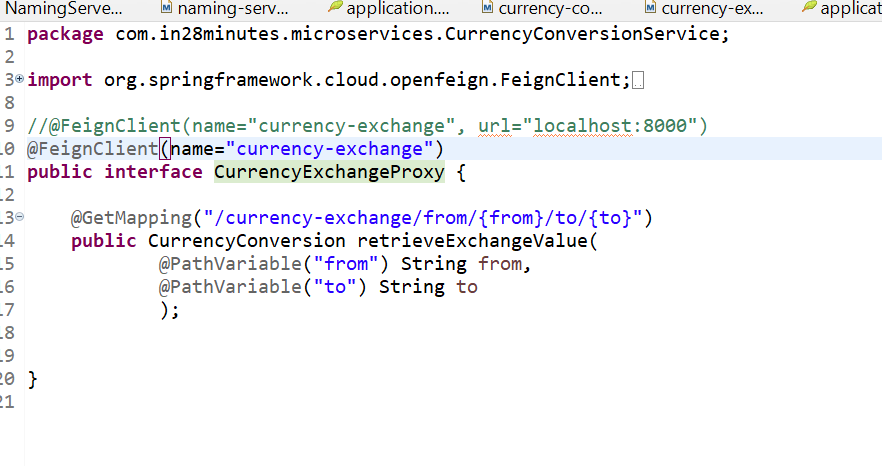


You can see both services are registered with the Eureka server.

**Concept: Service Discovery with Eureka and Load Balancing using Feign**

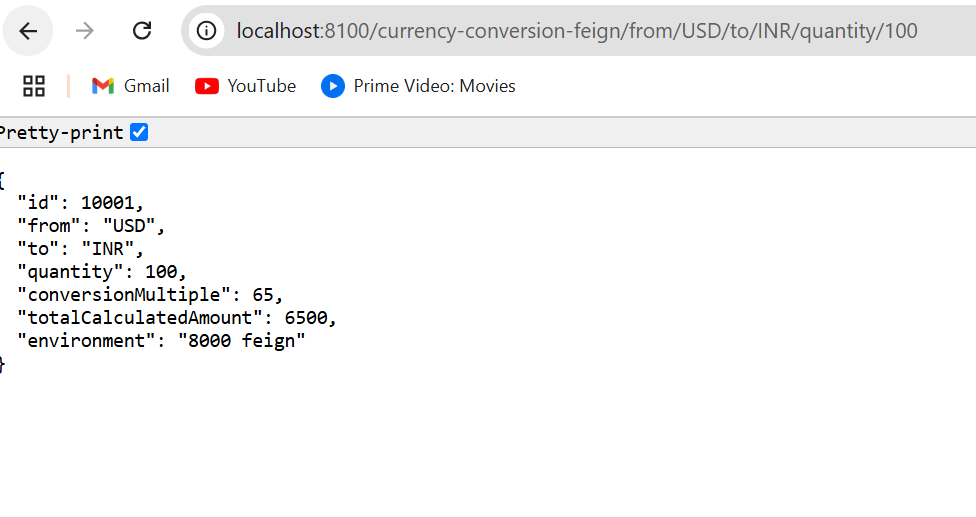
**Load Balancing with Eureka, Feign & Spring Cloud LoadBalancer**

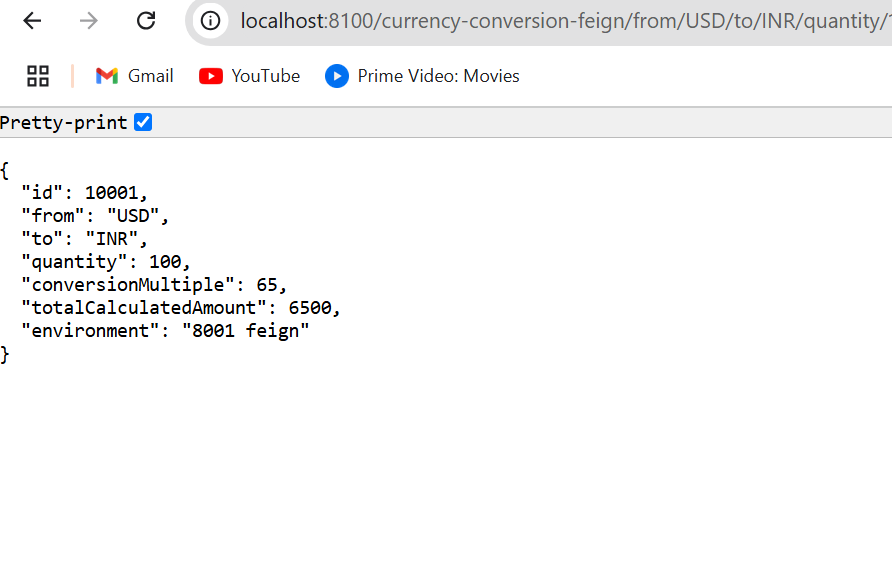
Updated Proxy class

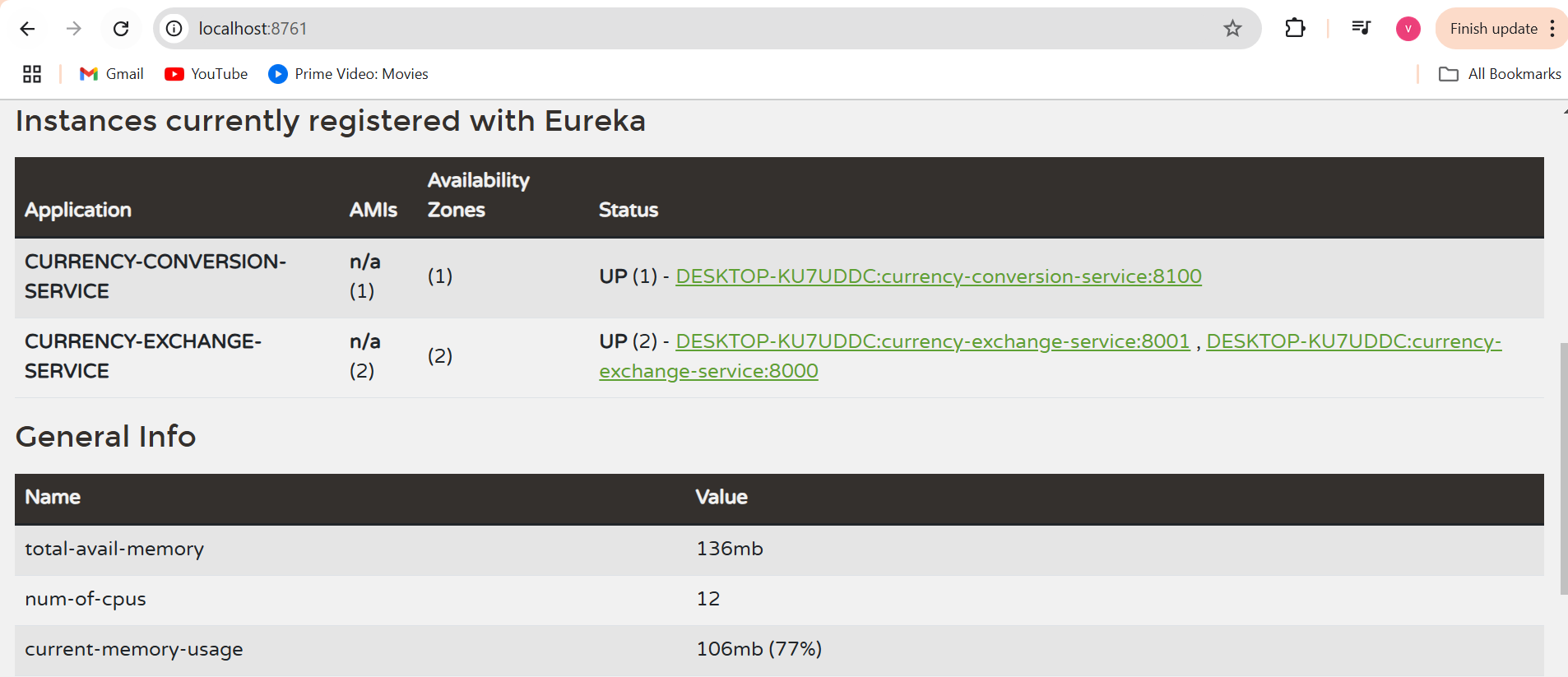


Hi this url <http://localhost:8761/> And we can see there are two instences are running on currency-exchange service.

Hi the below URL   
[**http://localhost:8100/currency-conversion-feign/from/USD/to/INR/quantity/100**](http://localhost:8100/currency-conversion-feign/from/USD/to/INR/quantity/100) **again and again   
it will throw u below response**

****

****



**1. Service Discovery**

**Definition:**

Service Discovery is a design pattern in microservices architecture where services register themselves with a centralized registry (like Eureka) so that other services can dynamically discover their locations.

**How It Works:**

* **Eureka Server**:
  + Acts as a centralized service registry where all microservices register themselves.
  + Services periodically send heartbeat signals to Eureka to indicate they are alive.
* **Eureka Clients**:
  + Services (like Currency Conversion and Currency Exchange) register themselves with Eureka.
  + They use Eureka to discover the locations (IP, port) of other services dynamically.

**Why It’s Used:**

* Avoids hardcoding service URLs in microservices.
* Enables scalability by allowing multiple instances of a service to register with Eureka.
* Provides resilience by dynamically updating service availability (e.g., if one instance goes down, Eureka will route requests to other instances).

**2. Feign Client**

**Definition:**

Feign is a declarative HTTP client in Spring Cloud that simplifies service-to-service communication by allowing developers to call other microservices using Java interfaces.

**How It Works:**

* Feign integrates with Eureka for **service discovery**.
* You define an interface with @FeignClient to map a remote service's REST endpoints.
* Feign uses the service name to dynamically resolve the service's location through Eureka.

**Why Feign?**

* Simplifies service communication by abstracting HTTP calls.
* No need to use RestTemplate or manually handle URLs, headers, and response mappings.
* Supports load balancing automatically when combined with Eureka.

**3. Load Balancing**

**Definition:**

Load Balancing is the process of distributing incoming requests across multiple instances of a service to ensure even utilization and high availability.

**How It Works:**

* Spring Cloud Load Balancer (used by Feign) communicates with Eureka to fetch all instances of a service.
* It distributes requests among available instances based on a load-balancing strategy (e.g., round-robin).

**Why It’s Used:**

* Enhances scalability by using multiple service instances.
* Provides fault tolerance: If one instance is down, requests are routed to others.

**Flow Explanation (Combining All Concepts)**

1. **Eureka Server**:
   * Runs on http://localhost:8761.
   * Serves as the service registry where services like currency-exchange and currency-conversion register themselves.
2. **Service Registration**:
   * **Currency Exchange Service** registers itself with Eureka (spring.application.name=currency-exchange).
   * **Currency Conversion Service** registers itself with Eureka (spring.application.name=currency-conversion).
3. **Feign with Eureka**:
   * CurrencyConversionController in the **Currency Conversion Service** uses CurrencyExchangeProxy (a Feign client) to call the Currency Exchange Service.
   * Feign resolves the currency-exchange service name using Eureka and dynamically fetches the instance URL (e.g., http://localhost:8001 or http://localhost:8000).
4. **Load Balancing**:
   * If multiple instances of the Currency Exchange Service are running, Feign (backed by Spring Cloud Load Balancer) distributes requests across those instances.

**Example Workflow**

* **Client Request**:

plaintext

Copy code

<http://localhost:8100/currency-conversion-feign/from/USD/to/INR/quantity/100>

**Step 1**: CurrencyConversionController uses CurrencyExchangeProxy to call the currency-exchange service:

@FeignClient(name = "currency-exchange")

**Step 2**: Feign queries Eureka for the currency-exchange service's location.

**Step 3**: Eureka responds with the available instances (e.g., http://localhost:8001 and http://localhost:8000).

**Step 4**: Feign (with load balancer) selects an instance (e.g., http://localhost:8001) and makes the API call:

<http://localhost:8001/currency-exchange/from/USD/to/INR>

**Step 5**: The Currency Exchange Service returns the exchange rate:

{

"id": 10001,

"from": "USD",

"to": "INR",

"conversionMultiple": 65,

"environment": "8001"

}

**Step 6**: Currency Conversion Service performs the calculation and returns:

{

"id": 10001,

"from": "USD",

"to": "INR",

"quantity": 100,

"conversionMultiple": 65,

"totalCalculatedAmount": 6500,

"environment": "8001 feign"

}

**Key Benefits**

1. **Dynamic Service Discovery**:
   * No hardcoded URLs; services locate each other dynamically.
2. **Simplified Communication**:
   * Feign provides a declarative approach to calling other microservices.
3. **Load Balancing**:
   * Automatically distributes requests among multiple instances for scalability and fault tolerance.

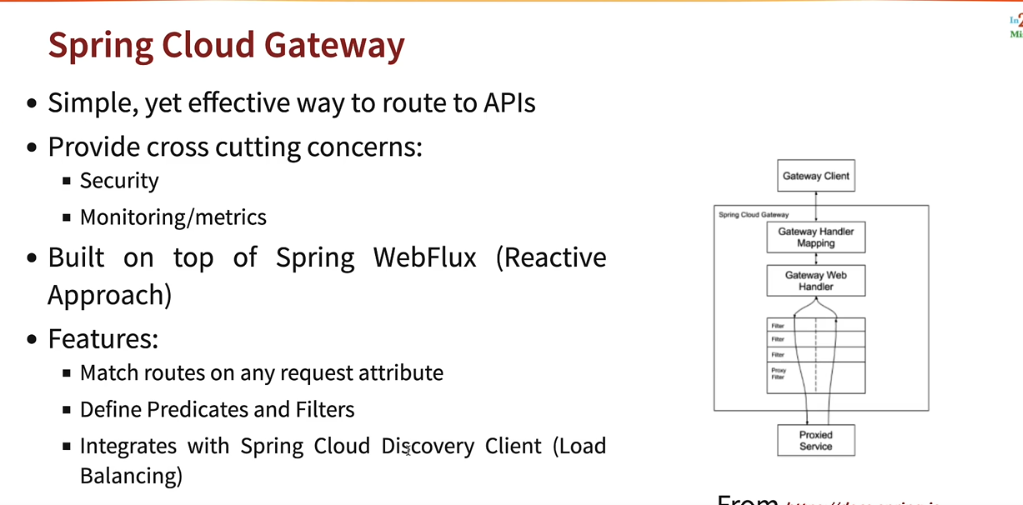
**Why Proxy Class in Currency Conversion Service Only?**

* The Currency Conversion Service needs to fetch exchange rates from the Currency Exchange Service, requiring a Feign Proxy for inter-service communication. The Currency Exchange Service has no such dependency.

**Why eureka.client.serviceUrl.defaultZone in Both Services?**

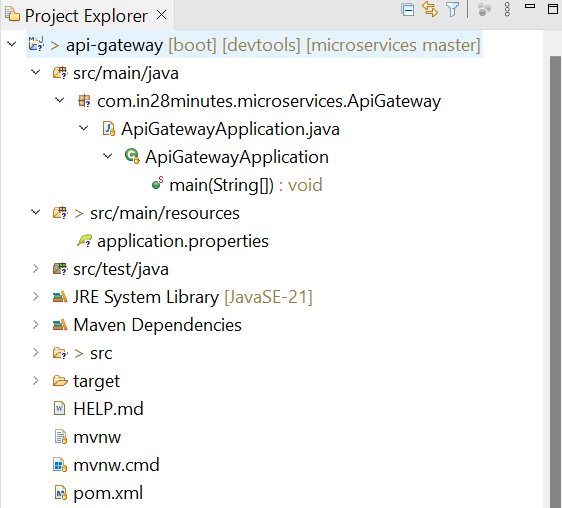
* This ensures both services register with the Eureka Naming Server and can discover each other dynamically for communication, avoiding hardcoding service URLs.

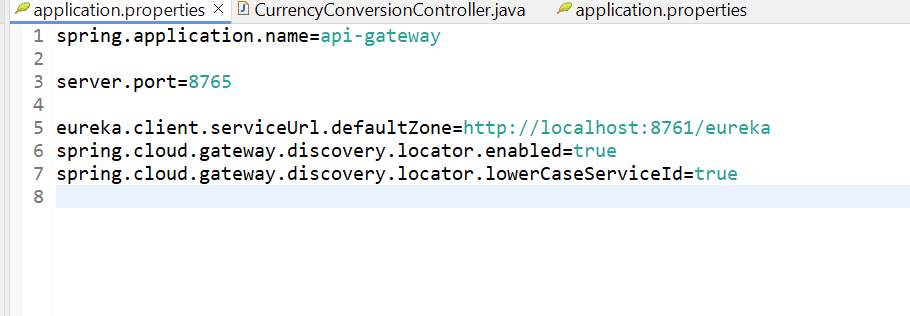
**Setting Up Spring Cloud Api Gateway**

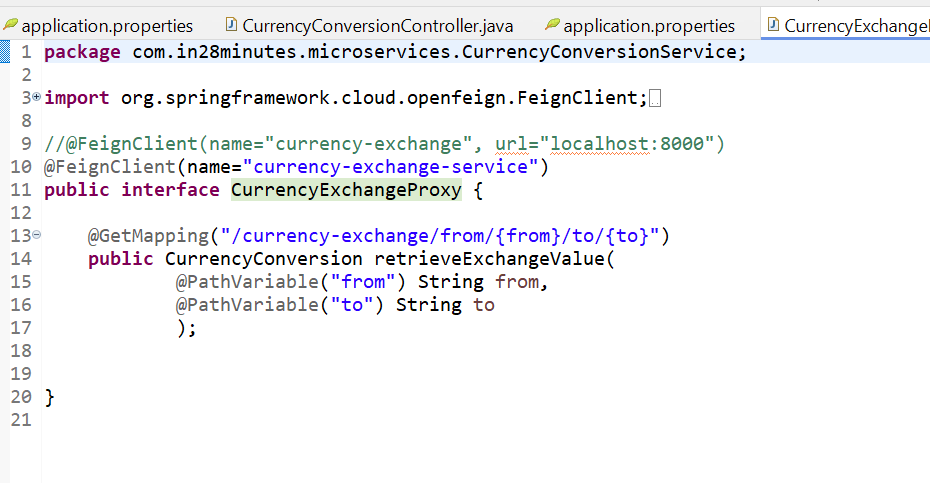
****

**Enabling Discovery Locator with Eureka for Spring Cloud Gateway**

Updated Proxy class in currency-conversion

****

****

****

**@FeignClient(name="currency-exchange-service") this name should match woith the servce name of exchange service in application.properties**

**Hit below urls**

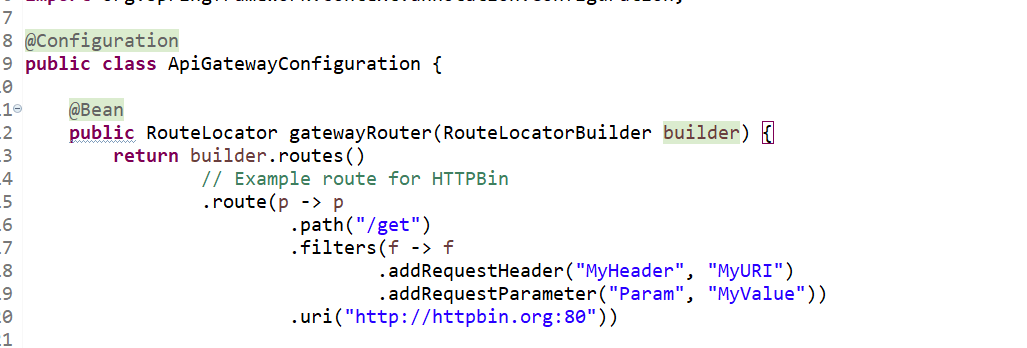
[**http://localhost:8765/currency-exchange-service/currency-exchange/from/USD/to/INR**](http://localhost:8765/currency-exchange-service/currency-exchange/from/USD/to/INR)

[**http://localhost:8765/currency-conversion-service/currency-conversion/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion-service/currency-conversion/from/USD/to/INR/quantity/10)

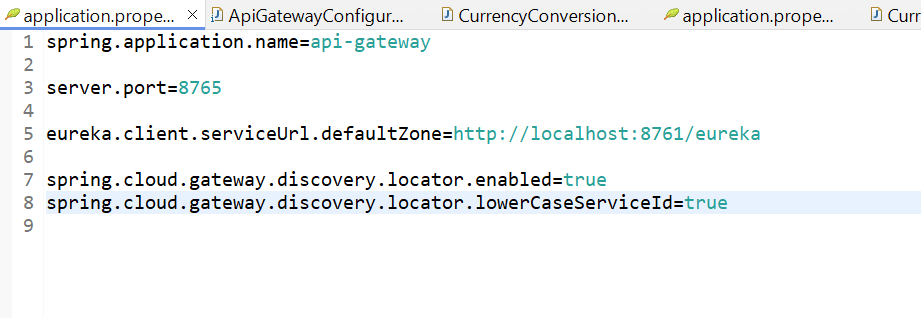
[**http://localhost:8765/currency-conversion-service/currency-conversion-feign/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion-service/currency-conversion-feign/from/USD/to/INR/quantity/10)

This is the new url after api gateway implementation

Create ApiGatewayConfiguration bean class





Update Apigateway applicartion.property file   
  


* <http://localhost:8765/currency-exchange/from/USD/to/INR>
* <http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10>
* <http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10>
* <http://localhost:8765/currency-conversion-new/from/USD/to/INR/quantity/10>

**Exploring Routes with Spring Cloud Gateway**

**Concept Name: API Gateway with Routing, Load Balancing, and Path Rewriting**

This concept involves building a centralized **API Gateway** to handle incoming client requests and forward them to the appropriate microservices. The API Gateway acts as a reverse proxy and provides additional functionality like dynamic routing, request filtering, load balancing, and path rewriting.

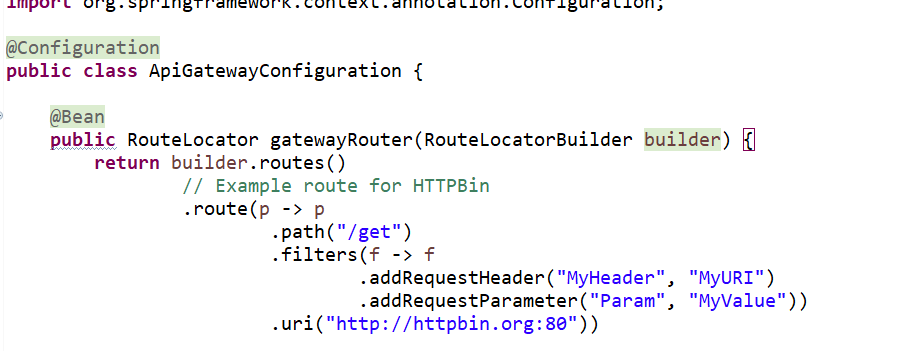
**Flow Explanation**

**1. API Gateway Overview**

* The API Gateway is the single entry point for all client requests.
* It forwards the requests to different microservices (e.g., currency-exchange-service, currency-conversion-service).
* It hides the internal details of microservices, like their ports and locations, from the client.

**2. Route Definitions**

In the ApiGatewayConfiguration class, we define routes that match incoming paths and forward them to the appropriate services using Spring Cloud Gateway.



**What It Does**:

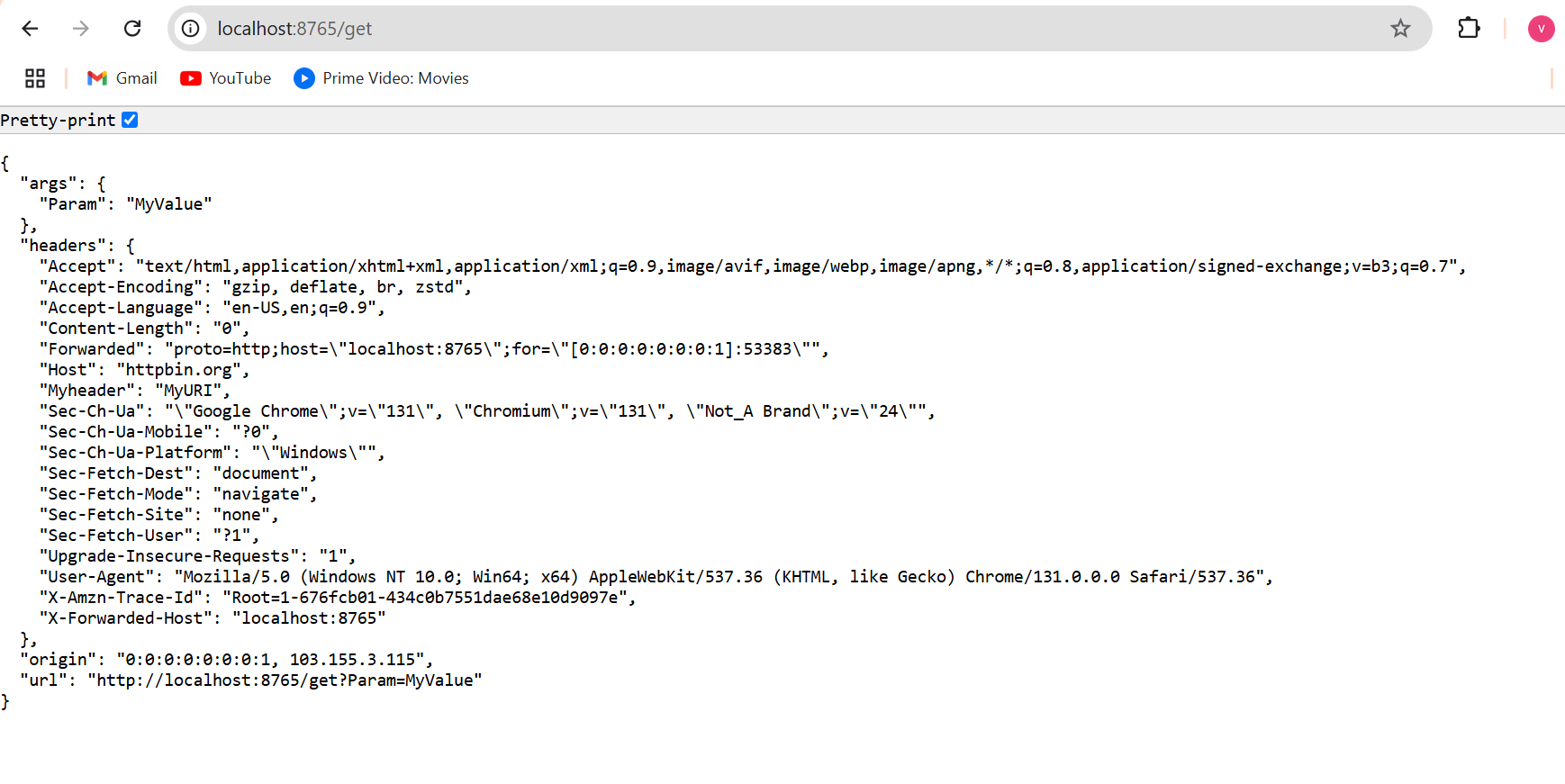
* Matches requests with the /get path.
* Adds a custom header MyHeader with value MyURI and a query parameter Param with value MyValue.
* Forwards the request to the http://httpbin.org:80 endpoint for demonstration purposes.

**Example Request**

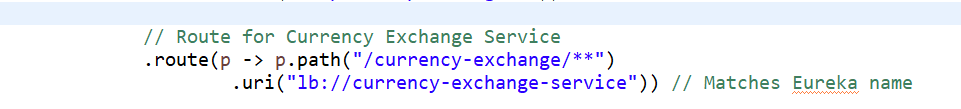
<http://localhost:8765/get>

This forwards the request to <http://httpbin.org:80>.

Response



Route for currency-exchange-service



**What It Does**:

* Matches requests starting with /currency-exchange/\*\*.
* Forwards the request to the currency-exchange-service registered in Eureka (using lb:// for load balancing).

**Flow**:

* Example request: /currency-exchange/from/USD/to/INR
* API Gateway resolves the service currency-exchange-service via Eureka and forwards the request.

**Result**: The CurrencyExchangeController handles the request and returns the exchange rate.

**Route for currency-conversion-service**

.route(p -> p.path("/currency-conversion/\*\*")

.uri("lb://currency-conversion-service"))

* **What It Does**:
  + Matches requests starting with /currency-conversion/\*\*.
  + Forwards the request to the currency-conversion-service.
* **Flow**:
  + Example request: /currency-conversion/from/USD/to/INR/quantity/10
  + The Gateway forwards the request to the CurrencyConversionController in the currency-conversion-service.
* **Result**: The conversion logic calculates the total converted amount and returns the result.

**Route for Feign-based currency-conversion-service**

.route(p -> p.path("/currency-conversion-feign/\*\*")

.uri("lb://currency-conversion-service"))

**What It Does**:

* Matches requests starting with /currency-conversion-feign/\*\*.
* Forwards the request to the currency-conversion-service.

**Flow**:

* Example request: /currency-conversion-feign/from/USD/to/INR/quantity/10
* The Feign client in currency-conversion-service calls the currency-exchange-service to fetch the exchange rate.
* The response is returned to the client.

**Route with Path Rewriting**



**What It Does**:

* Matches requests starting with /currency-conversion-new/\*\*.
* Uses the rewritePath filter to transform the path:
  + Removes /currency-conversion-new and replaces it with /currency-conversion-feign.
* Forwards the updated path to the currency-conversion-service.

**Flow**:

* Example request: /currency-conversion-new/from/USD/to/INR/quantity/10
* The Gateway rewrites it to /currency-conversion-feign/from/USD/to/INR/quantity/10 and forwards it to the currency-conversion-service.

**Flow of Execution**

**1. Client Request**

* A client sends a request to the API Gateway

<http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10>

**2. Route Matching**

* The Gateway matches the request path /currency-conversion/\*\* to the defined route

.route(p -> p.path("/currency-conversion/\*\*")

.uri("lb://currency-conversion-service"))

**Eureka Resolution**

* The Gateway uses Eureka to locate the currency-conversion-service and forwards the request.

**Service Processing**

* The currency-conversion-service processes the request, possibly calling the currency-exchange-service to get the exchange rate.

**5. Response Returned**

* The processed response is returned to the API Gateway, which forwards it back to the client.

**Common Test Cases**

1. Test all routes to ensure they forward correctly:
   * /currency-exchange/from/USD/to/INR
   * /currency-conversion/from/USD/to/INR/quantity/10
   * /currency-conversion-feign/from/USD/to/INR/quantity/10
   * /currency-conversion-new/from/USD/to/INR/quantity/10
2. Verify load balancing:
   * Start multiple instances of currency-exchange-service or currency-conversion-service and check if requests are distributed evenly.

**Role of Eureka in This Setup**

Eureka is acting as the **Service Registry and Discovery Mechanism** in your microservices architecture. Here’s how Eureka is involved and how it works in this setup:

**1. Service Registration**

* Each microservice registers itself with the Eureka Server on startup using the configuration:

properties

Copy code

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

spring.application.name=<service-name>

In your case:

* **API Gateway** registers as api-gateway.
* **Currency Exchange Service** registers as currency-exchange-service.
* **Currency Conversion Service** registers as currency-conversion-service.

This registration makes the services discoverable by other services via Eureka.

**2. Service Discovery**

* When the API Gateway receives a request (e.g., /currency-exchange/from/USD/to/INR), it forwards the request to the corresponding service.
* The Gateway uses Eureka to dynamically locate the service instance by name (lb://currency-exchange-service).
  + **lb://**: This prefix tells Spring Cloud Gateway to use the Spring Load Balancer and resolve the service name to an actual running instance.
* Eureka returns the hostname and port of the currency-exchange-service instance.

**3. Dynamic Routing**

If you enable **Dynamic Route Discovery** in the API Gateway

spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true

The Gateway automatically discovers all registered services in Eureka and creates routes for them. For example:

* A service registered as currency-exchange-service would automatically map to:

[http://localhost:8765/currency-exchange-service/\*\*](http://localhost:8765/currency-exchange-service/**)

This eliminates the need for explicitly defining routes in ApiGatewayConfiguration.

**4. Load Balancing**

* Eureka also helps with **load balancing**. For example:
  + If you have two instances of currency-exchange-service running on ports 8000 and 8001, Eureka keeps track of both instances.
  + When the API Gateway sends a request to lb://currency-exchange-service, the Spring Load Balancer fetches the available instances from Eureka and forwards the request to one of them, distributing the load.

**End-to-End Flow Involving Eureka**

1. **Service Registration**:
   * currency-exchange-service registers with Eureka as currency-exchange-service.
   * currency-conversion-service registers with Eureka as currency-conversion-service.
   * api-gateway registers with Eureka as api-gateway.
2. **Client Request**:
   * A client sends a request to the API Gateway

<http://localhost:8765/currency-exchange/from/USD/to/INR>

1. **Gateway Routing**:
   * The Gateway matches the path /currency-exchange/\*\* in its route configuration and determines the destination as lb://currency-exchange-service.
2. **Eureka Discovery**:
   * The Gateway queries Eureka to locate the currency-exchange-service.
   * Eureka responds with the hostname and port of an available instance (e.g., localhost:8000).
3. **Request Forwarding**:
   * The Gateway forwards the request to http://localhost:8000/currency-exchange/from/USD/to/INR.
4. **Service Processing**:
   * The currency-exchange-service processes the request and returns the exchange rate (e.g., 65 for USD to INR).
5. **Response Returned**:
   * The Gateway receives the response from currency-exchange-service and forwards it back to the client.

**Key Benefits of Using Eureka Here**

1. **Dynamic Service Discovery**:
   * You don’t need to hardcode service locations (like hostnames and ports). Eureka handles this dynamically.
2. **High Availability**:
   * If one instance of a service goes down, Eureka automatically directs traffic to the remaining healthy instances.
3. **Load Balancing**:
   * By integrating with Spring Cloud Load Balancer (lb://), Eureka ensures requests are evenly distributed among multiple instances of a service.
4. **Simplified API Gateway Configuration**:
   * Instead of explicitly defining routes for every service, you can use dynamic discovery for automatic routing.

**Conclusion**

Eureka plays a crucial role in dynamically discovering and routing requests in your microservices architecture. It simplifies service interaction by decoupling services from their locations (hostnames and ports) and enables features like load balancing and fault tolerance.

Let me know if you’d like to dive deeper into any specific aspect! 😊

**What is Service Discovery?**

**Service Discovery** is a design pattern in microservices architecture that allows different services to locate and communicate with each other dynamically. Instead of hardcoding the locations (IP addresses and ports) of services, Service Discovery enables services to register themselves and be discovered by other services at runtime.

**Key Components of Service Discovery**

1. **Service Registry**:
   * A centralized database where all running services register themselves with their metadata, such as service name, IP address, port, and health status.
   * Example: **Eureka** (Netflix OSS), **Consul**, or **Zookeeper**.
2. **Service Registration**:
   * Each service registers itself with the registry when it starts up and deregisters when it shuts down.
3. **Service Discovery**:
   * Other services query the registry to discover the location of services they need to interact with.
   * There are two types:
     + **Client-Side Discovery**: The client queries the registry to discover a service and load balances the requests. (Example: Ribbon with Eureka)
     + **Server-Side Discovery**: The client sends the request to a load balancer, which queries the registry and forwards the request to the appropriate service. (Example: Kubernetes, AWS Elastic Load Balancer)
4. **Health Monitoring**:
   * The registry periodically checks the health of services to ensure only healthy instances are discoverable.

**How Service Discovery Works**

1. **Service Registration**:
   * When a service (e.g., currency-exchange-service) starts, it registers its instance (e.g., localhost:8000) with the service registry under a unique service name (currency-exchange-service).
2. **Registry Stores Metadata**:
   * The registry keeps track of all running instances of each service.
3. **Discovery by Clients**:
   * Another service (e.g., currency-conversion-service) or an API Gateway queries the registry to find the instances of the currency-exchange-service.
4. **Dynamic Routing**:
   * The client or gateway uses the metadata from the registry to forward the request to one of the service instances.

**Why Service Discovery is Important**

1. **Dynamic Scaling**:
   * Services can scale up or down (e.g., adding or removing instances), and the registry dynamically updates the list of available instances.
2. **Fault Tolerance**:
   * If an instance of a service goes down, the registry removes it, and the requests are routed to healthy instances.
3. **Simplifies Configuration**:
   * Clients do not need to know the IP addresses and ports of services; they only need the service name.
4. **Load Balancing**:
   * Service Discovery works with client-side or server-side load balancers to distribute requests across multiple instances.

**Real-World Example**

Imagine a microservices architecture with three services:

1. **Currency Exchange Service** (handles exchange rates).
2. **Currency Conversion Service** (converts amounts using the exchange rates).
3. **API Gateway** (acts as a centralized entry point for clients).

**Without Service Discovery:**

* The currency-conversion-service must know the IP address and port of the currency-exchange-service. If the IP or port changes (e.g., due to scaling or a restart), the configuration must be manually updated.

**With Service Discovery:**

1. Both services register with Eureka (or another registry).
2. The currency-conversion-service queries Eureka to dynamically discover the currency-exchange-service.
3. Eureka responds with the list of available instances of the currency-exchange-service.
4. The request is routed to one of the available instances.

**Types of Service Discovery**

1. **Client-Side Discovery**:
   * The client queries the service registry directly and chooses an instance to call.
   * Example: Eureka with Ribbon (Netflix OSS).
2. **Server-Side Discovery**:
   * A load balancer (e.g., API Gateway) queries the service registry and forwards the request to a service instance.
   * Example: Spring Cloud Gateway, Kubernetes Service.

**Example with Eureka**

1. **Eureka Server**:
   * Acts as the service registry.
   * All services register with it.
2. **Currency Exchange Service**:

properties

Copy code

spring.application.name=currency-exchange-service

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

**Currency Conversion Service**:

spring.application.name=currency-conversion-service

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

**API Gateway**:

* Instead of hardcoding service URLs, the Gateway uses Eureka for service discovery

spring.cloud.gateway.discovery.locator.enabled=true

spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true

When a request is made to the API Gateway for currency-conversion-service, the Gateway queries Eureka for the service's available instances and forwards the request dynamically.

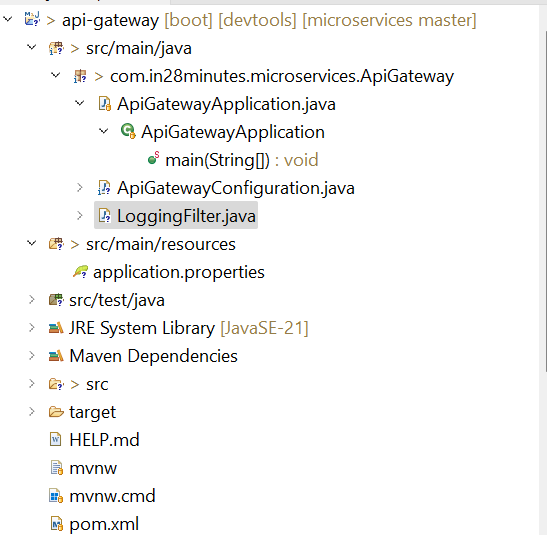
**Benefits of Using Eureka for Service Discovery**

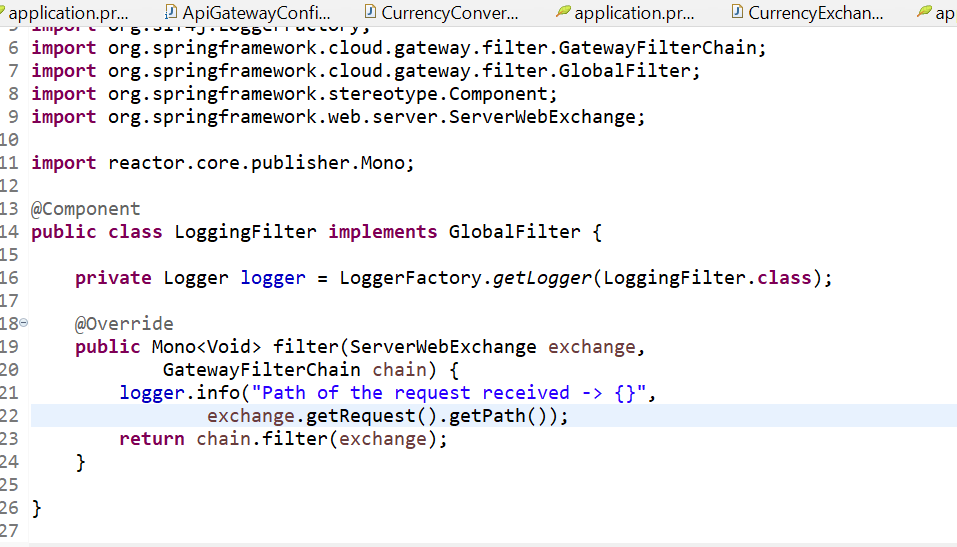
1. **Dynamic Routing**:
   * Services can move between IPs and ports without breaking communication.
2. **Load Balancing**:
   * Requests are distributed among multiple instances of a service.
3. **Self-Healing**:
   * If a service instance crashes, it is automatically deregistered, ensuring requests are not sent to unhealthy instances.
4. **Simplified Architecture**:
   * No need to maintain a static configuration of service URLs.

**Common Tools for Service Discovery**

* **Eureka** (Netflix OSS)
* **Consul** (HashiCorp)
* **Zookeeper** (Apache)
* **Kubernetes** (Built-in Service Discovery)

**Implementing Spring Cloud Gateway Logging Filter**





**Getting started with Circuit Breaker - Resilience4j**

**What is Circuit Breaker in Microservices?**

A **Circuit Breaker** is a design pattern used in microservices to handle failures gracefully and prevent cascading failures in a distributed system. It monitors requests between services and interrupts the flow of requests if a service is unavailable or experiencing high latency.

In the context of Java microservices, **Resilience4j** is a library that implements the Circuit Breaker pattern, among other fault tolerance mechanisms like rate limiting, retries, and bulkheads.

**How Circuit Breaker Works**

1. **Closed State**:
   * All requests are allowed to flow to the service.
   * If failures occur, the Circuit Breaker tracks them and transitions to the **Open State** if the failure threshold is exceeded.
2. **Open State**:
   * Requests are short-circuited (rejected) without calling the service.
   * This prevents further strain on the failing service.
   * After a configurable wait time, it transitions to the **Half-Open State**.
3. **Half-Open State**:
   * Allows a limited number of requests to test if the service has recovered.
   * If successful, transitions back to **Closed State**. If failures persist, it reopens.

**What is Resilience4j?**

**Resilience4j** is a lightweight, easy-to-use Java library designed to handle fault tolerance in microservices. It provides features like:

* **Circuit Breaker**: Stops calling a failing service temporarily.
* **Rate Limiter**: Limits the number of requests sent to a service.
* **Retry**: Automatically retries failed requests.
* **Bulkhead**: Isolates critical services to prevent resource exhaustion.

**How Resilience4j Implements Circuit Breaker**

1. **Monitoring Requests**:
   * Tracks the success or failure of requests to a service.
   * Measures latency and error rates.
2. **Breaking the Circuit**:
   * If the error rate exceeds a defined threshold, the Circuit Breaker trips and starts rejecting requests.
3. **Recovery**:
   * After a defined wait time, it transitions to **Half-Open** to test the service’s recovery.

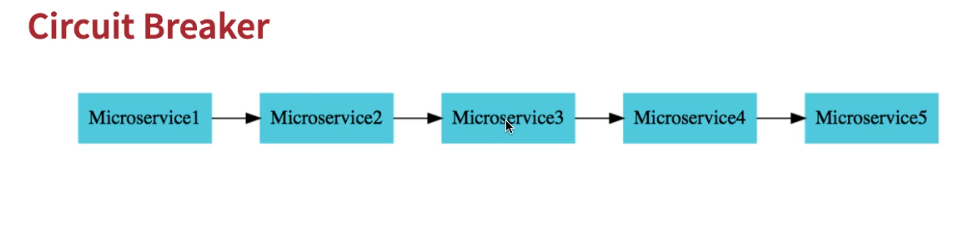
**Use Case for Circuit Breaker**

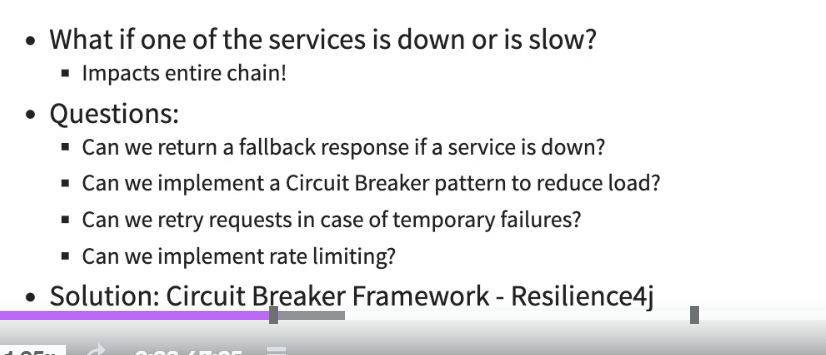
Imagine a microservices architecture where:

* **Service A** calls **Service B** to fetch data.
* If **Service B** is down or slow, the Circuit Breaker prevents **Service A** from being overloaded with failed requests.
* This ensures **Service A** remains responsive and provides a fallback response or default behavior.

**Advantages of Circuit Breaker**

1. **Improves Fault Tolerance**:
   * Prevents cascading failures in distributed systems.
2. **Enhances System Stability**:
   * Protects critical services from being overwhelmed.
3. **Graceful Degradation**:
   * Allows fallback mechanisms to handle requests when services fail.

****

****

**Answer to the Questions:**

**1. Can we return a fallback response if a service is down?**

* Yes, with Resilience4j, you can define a fallback method. When the main service fails, the Circuit Breaker will invoke this fallback method to return a default response or handle the failure gracefully.

**2. Can we implement a Circuit Breaker pattern to reduce load?**

* Yes, Resilience4j implements the Circuit Breaker pattern. When a service starts failing (e.g., exceeds a failure rate threshold), the Circuit Breaker "trips" into the Open State, rejecting further requests until the service recovers. This reduces the load on the failing service and prevents cascading failures.
* **How it reduces load:**
  + By short-circuiting requests, it avoids sending unnecessary traffic to a failing service.
  + It retries **only after a specified waitDurationInOpenState in the Half-Open State.**

**3. Can we retry requests in case of temporary failures?**

* Yes, Resilience4j supports the Retry pattern. You can configure retries to handle transient failures (e.g., network issues or temporary unavailability).

**4. Can we implement rate limiting?**

* Yes, Resilience4j provides a Rate Limiter module to control the number of requests sent to a service within a specified time frame.

**How it works:**

* Limits the number of requests (limitForPeriod) in a given time (limitRefreshPeriod).
* Requests exceeding this limit are rejected with a RateLimiterException.

**Solution: Resilience4j Framework**

**Resilience4j addresses all the above questions with its robust fault tolerance features:**

* Circuit Breaker: Handles failures gracefully.
* Retry: Automatically retries failed requests for transient errors.
* Rate Limiter: Controls the request rate to avoid overloading services.
* Fallback: Provides default responses or alternative flows in case of service failures.

**Explanation of @Retry(name = "default") in Circuit Breaker**

@Retry is an annotation provided by **Resilience4j** that enables retry logic for methods that fail due to transient errors, such as network issues or timeouts. When the annotated method throws an exception, the @Retry mechanism retries the method execution based on the configuration provided in the application.

**Key Concepts**

**1. name in @Retry**

* The name specifies the configuration instance for the retry logic defined in your application.properties or application.yml.
* The default configuration is applied if no custom instance is specified.

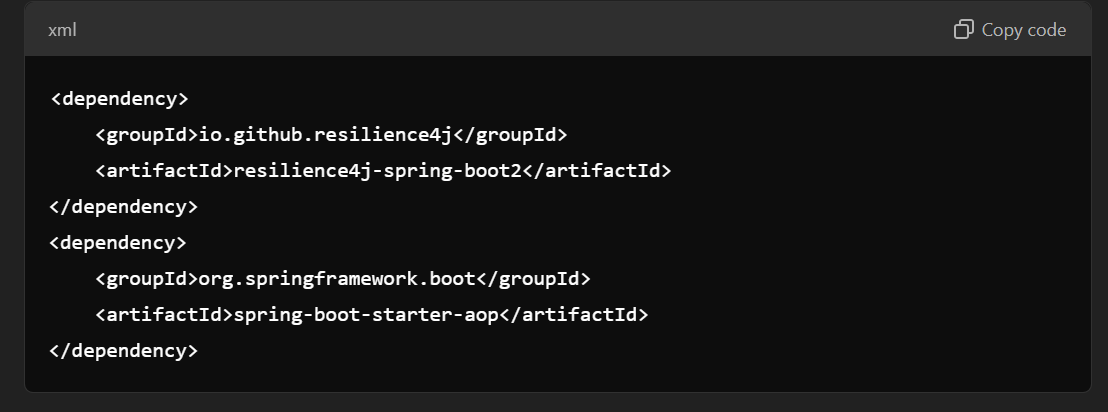
**2. Fallback Method**

* A fallback method is a backup method invoked when all retries fail.
* It helps handle failures gracefully by providing an alternative response.

**Example Implementation**

**1. Add Dependencies**

Add the following dependencies to your pom.xml:



1. **Sample Service with Retry Logic**

import io.github.resilience4j.retry.annotation.Retry;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate**;**

@Service

public class MyService {

@Retry(name = "default", fallbackMethod = "fallbackResponse")

public String callExternalService() {

// Simulate a REST call to an external service

RestTemplate restTemplate = new RestTemplate();

return restTemplate.getForObject("http://localhost:8081/dummy-api", String.class);

}

// Fallback method

public String fallbackResponse(Exception ex) {

return "Fallback response: Service is currently unavailable. Please try again later.";

}

}

1. **Controller to Test Retry**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class MyController {

@Autowired

private MyService myService;

@GetMapping("/test-retry")

public String testRetry() {

return myService.callExternalService();

}

}

1. **Add Configuration in application.properties**

# Resilience4j Retry Configuration

resilience4j.retry.instances.default.maxAttempts=3

resilience4j.retry.instances.default.waitDuration=200ms

 maxAttempts: Number of retry attempts before considering the request failed.

 waitDuration: Wait time between retries

**Flow of Execution**

1. **First Attempt:**
   * The callExternalService method is executed.
   * If it fails (e.g., an exception is thrown), it retries based on the maxAttempts and waitDuration.
2. **Retry Attempts:**
   * Resilience4j retries the method execution up to maxAttempts times**.**
3. **Fallback Execution:**
   * If all retries fail, the fallbackResponse method is invoked.
   * The fallback method provides an alternative response to the client.

**Use Cases of @Retry**

1. Transient Failures: Retry failed calls due to temporary issues like network glitches or timeouts.
2. Rate-Limited APIs: Retry requests after a brief wait when the external API throttles requests.
3. Database Connections: Retry failed database queries during a short-term downtime.

**Benefits of Fallback Method**

1. Graceful Degradation: Provide a meaningful response instead of exposing internal errors to the client.
2. Improved User Experience: Inform users about service unavailability in a user-friendly way.
3. Fault Tolerance: Ensure system stability by avoiding cascading failures**.**

**Circuit Breaker**

A **Circuit Breaker** in **Resilience4j** prevents a system from repeatedly trying a failing service. It works in three states:

1. **Closed**: Requests flow as usual.
   * If failures exceed a threshold, it transitions to **Open** state.
2. **Open**: Requests are blocked for a specific time.
   * After a timeout, it transitions to **Half-Open**.
3. **Half-Open**: A limited number of requests are allowed.
   * If successful, it transitions back to **Closed**. Otherwise, it moves to **Open**.

**Steps to Implement Circuit Breaker**

**1. Add Dependencies**

Add the following dependencies in your pom.xml:

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

1. **Sample Service with Circuit Breaker**

import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

@Service

public class MyService {

@CircuitBreaker(name = "default", fallbackMethod = "fallbackResponse")

public String callExternalService() {

// Simulating an external API call

RestTemplate restTemplate = new RestTemplate();

return restTemplate.getForObject("http://localhost:8081/dummy-api", String.class);

}

// Fallback Method

public String fallbackResponse(Exception ex) {

return "Fallback response: External service is down!";

}

}

1. **Controller to Test Circuit Breaker**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class MyController {

@Autowired

private MyService myService;

@GetMapping("/test-circuit-breaker")

public String testCircuitBreaker() {

return myService.callExternalService();

}

}

1. **Add Configuration in application.properties**

# Resilience4j Circuit Breaker Configuration

resilience4j.circuitbreaker.instances.default.register-health-indicator=true

resilience4j.circuitbreaker.instances.default.sliding-window-type=COUNT\_BASED

resilience4j.circuitbreaker.instances.default.sliding-window-size=5

resilience4j.circuitbreaker.instances.default.failure-rate-threshold=50

resilience4j.circuitbreaker.instances.default.wait-duration-in-open-state=10s

resilience4j.circuitbreaker.instances.default.minimum-number-of-calls=5

**Key Properties:**

1. **sliding-window-type:**
   * **COUNT\_BASED: Tracks a fixed number of calls.**
   * **TIME\_BASED: Tracks calls within a time window.**
2. **sliding-window-size: Number of calls to evaluate.**
3. **failure-rate-threshold: Percentage of failures to trigger the Circuit Breaker.**
4. **wait-duration-in-open-state: Time the Circuit Breaker remains in the Open state.**
5. **minimum-number-of-calls: Minimum calls before Circuit Breaker evaluates failures.**

**Key Properties:**

1. **sliding-window-type**:
   * COUNT\_BASED: Tracks a fixed number of calls.
   * TIME\_BASED: Tracks calls within a time window.
2. **sliding-window-size**: Number of calls to evaluate.
3. **failure-rate-threshold**: Percentage of failures to trigger the Circuit Breaker.
4. **wait-duration-in-open-state**: Time the Circuit Breaker remains in the **Open** state.
5. **minimum-number-of-calls**: Minimum calls before Circuit Breaker evaluates failures.

**Features in Resilience4j**

**1. Rate Limiting**

* Controls the number of calls to a service in a given time.
* Use **Rate Limiter** for APIs with rate restrictions.

**Implementation Example**

import io.github.resilience4j.ratelimiter.annotation.RateLimiter;

import org.springframework.stereotype.Service;

@Service

public class RateLimitingService {

@RateLimiter(name = "rateLimiterExample")

public String callService() {

return "Service response";

}

}

**Configuration**

resilience4j.ratelimiter.instances.rateLimiterExample.limit-for-period=5

resilience4j.ratelimiter.instances.rateLimiterExample.limit-refresh-period=10s

resilience4j.ratelimiter.instances.rateLimiterExample.timeout-duration=2s

**limit-for-period**: Max calls in the period.

**limit-refresh-period**: Time window for the limit.

**timeout-duration**: Timeout if limit is exceeded.

**Bulkhead**

* Limits concurrent calls to a service to prevent overloading.

**Implementation Example**

import io.github.resilience4j.bulkhead.annotation.Bulkhead;

import org.springframework.stereotype.Service;

@Service

public class BulkheadService {

@Bulkhead(name = "bulkheadExample", fallbackMethod = "fallbackResponse")

public String callService() {

return "Service response";

}

public String fallbackResponse(Throwable t) {

return "Service is overloaded. Please try again later.";

}

}

**Configuration**

**resilience4j.bulkhead.instances.bulkheadExample.max-concurrent-calls=3**

**resilience4j.bulkhead.instances.bulkheadExample.max-wait-duration=2s**

**max-concurrent-calls**: Maximum simultaneous requests.

**max-wait-duration**: Wait time for a thread.

**Flow of Execution**

1. **Circuit Breaker**:
   * Calls the method.
   * If failures exceed failure-rate-threshold, Circuit Breaker **opens**.
   * Requests fail immediately during **Open** state.
   * After wait-duration-in-open-state, moves to **Half-Open**.
2. **Rate Limiting**:
   * Limits number of calls per limit-refresh-period.
   * If exceeded, requests are blocked.
3. **Bulkhead**:
   * Allows up to max-concurrent-calls.
   * Additional calls are queued or fail based on max-wait-duration.

**When to Use**

1. **Circuit Breaker**: For unreliable services to prevent cascading failures.
2. **Rate Limiting**: To protect rate-limited APIs or maintain stability.
3. **Bulkhead**: For resource-intensive services to ensure fair resource sharing