Microservices Configuration Management

License

This work by Satish Kumar at Leeds Beckett University is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Contents

License

Contents

Introduction

Service Discovery Implementation

API Gateway Implementation

Config Server Implementation

Introduction

The aim of this week's lab is to gain an understanding of microservices configuration management. Through this lab, we aim to equip you with the knowledge of how to use Spring Cloud Config Server with Git for externalizing microservices configurations efficiently on the central location like Git. Furthermore, we will learn how to configure API Gateway to route all microservice requests through and integrating centralized service discovery for seamless communication between microservices.

Control your microservice configuration with Spring Cloud

We are going to use student-post microservices project, we have implemented in week -5 lab of this module. If you have not implemented **student-post microservices** example then download this project from the GitHub Repo available at https://github.com/LBUSESC/week5 and open **student-post-microservices** project with IntelliJ IDEA.

NOTE: You need to update your local database (MySQL) username and password in the application.yaml file located in the src/main/resources directory of both student-service and post-service.

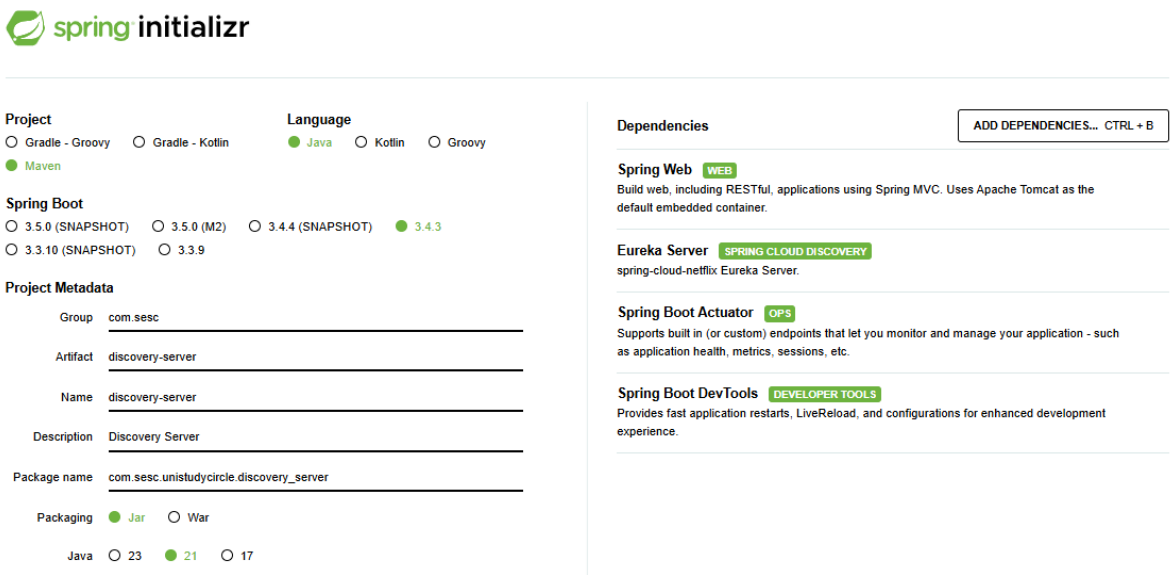
Service Discovery Implementation with Spring Eureka Server

> The principal objective of service discovery is to create an architecture where services can automatically indicate their physical locations rather than requiring manual configuration of their endpoints. This dynamic discovery allows services to register themselves upon startup and deregister when they shut down, making it easier for other services to find and communicate with them.

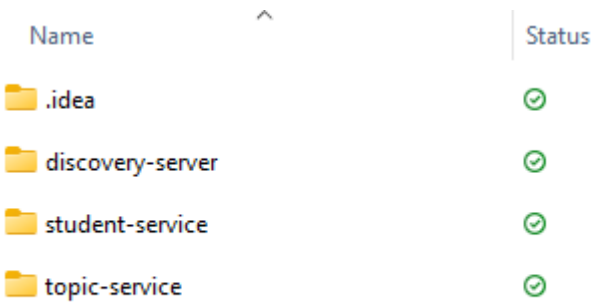
We now have student-service and post-service in the project. These microservices will automatically register themselves in the service discovery upon startup. Let's implement Service Discovery microservice with Spring Cloud Eureka Server.

First, we need to create a **discovery-server** microservice project and add the following dependencies and click on the generate button to download this project. You can access spring initializr at https://start.spring.io/

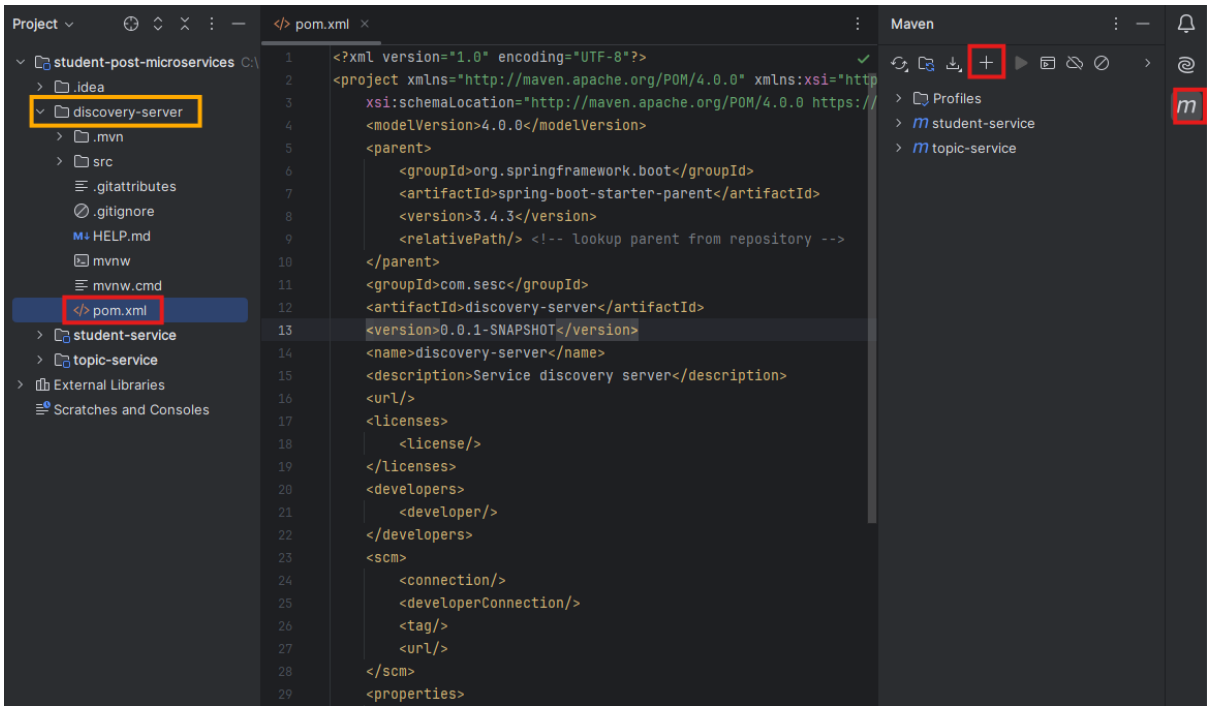
- Spring Web
- Eureka Server
- Spring Boot Actuator
- Spring Boot DevTools



The next step is to unzip and copy the **discovery-server** microservice and paste it inside the student-post-microservice folder (the current working project opened with IntelliJ IDEA) as shown in following figure



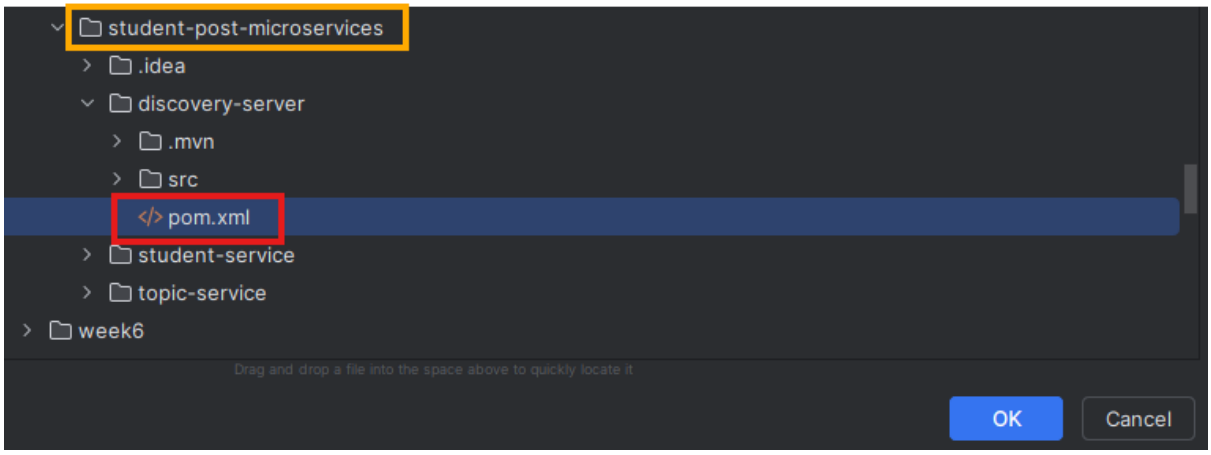
We now see the **discovery-server** microservice appeared in the project explorer as shown below. We need to import (or initialize) **discovery-server** microservice with following steps



Step 1: Click on the maven button sign (m).

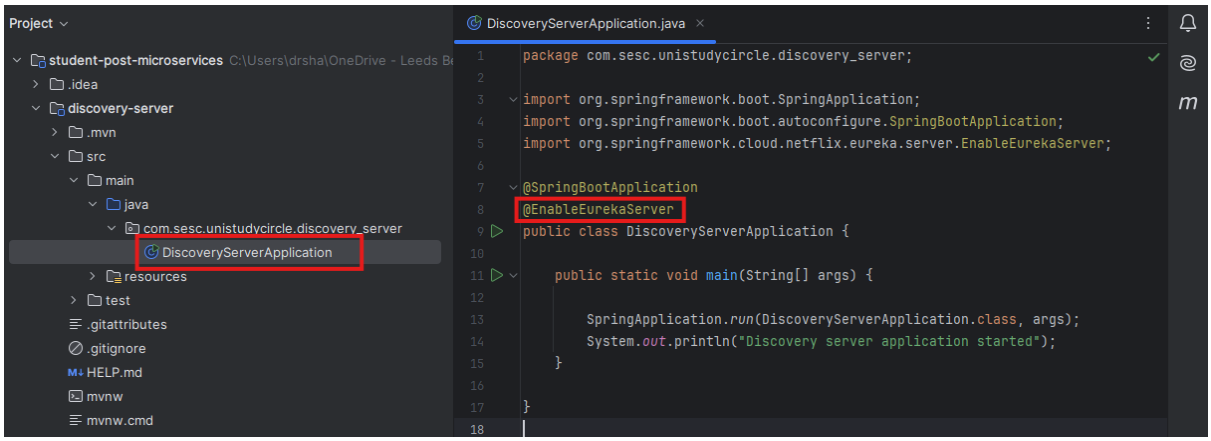
Step 2: Click on the + sign and go to your current working folder (**student-post-microservices**) as shown below.

Step 3: Expand the discovery-server microservice as shown below and select pom.xml file and click on the OK button.



We now ready to use our **discovery-server** microservice as service discovery server.

Let's annotate the DiscoveryServerApplication class with @EnableEurekaServer annotation as shown below

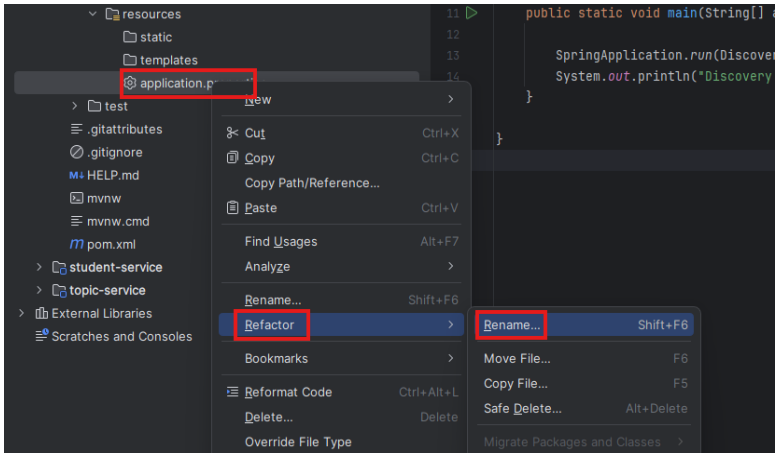


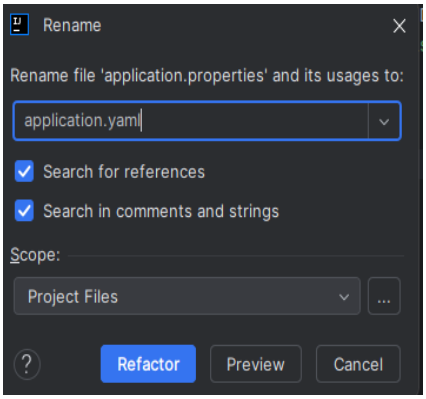
The @EnableEurekaServer annotation is used in a Spring Boot application to make it a Eureka server. By adding this annotation to our main application class, we are enabling the application to act as a service discovery server, which can register and provide information about available microservices.

The next step is to provide necessary configuration to enable service discovery.

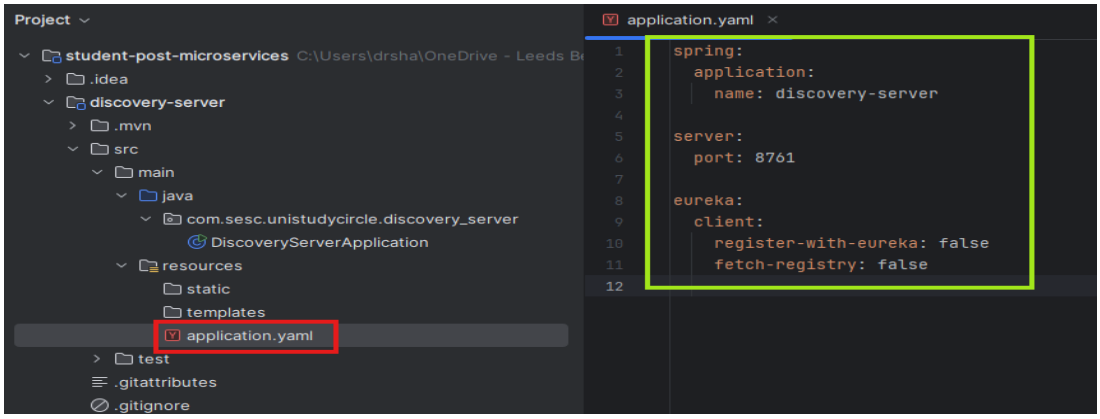
Let's change application.properties to **application.yaml** file by refactoring file name as shown below and add following configuration in the application.yaml file

Note: The **application.properties** file is located in the src/main/resources folder.



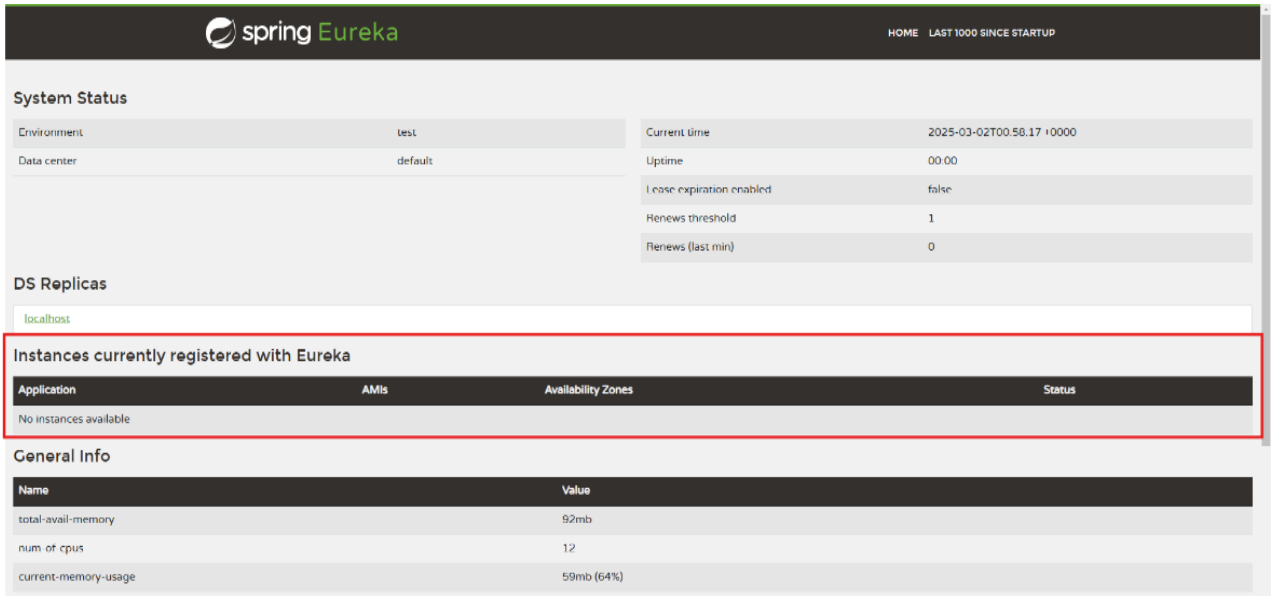



```yaml
spring:
  application:
    name: discovery-server

server:
  port: 8761

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



In above configuration, spring.application.name: discovery-server assigns a unique name to this service, identifying it within the ecosystem. The server.port: 8761 specifies that the Discovery Server will run on port **8761**, which is the default port for Eureka. Under eureka.client, the property register-with-eureka: false ensures that this server does not register itself as a client, as it is solely intended to manage service registrations. Similarly, fetch-registry: false prevents it from retrieving service information from other Eureka instances, reinforcing its role as a dedicated registry.

We now run the application and open the discovery server URL (http://localhost:8761) in a web browser to access the **discovery server** web interface, as shown below. At this point, we notice that no microservice instances are registered with the discovery server.

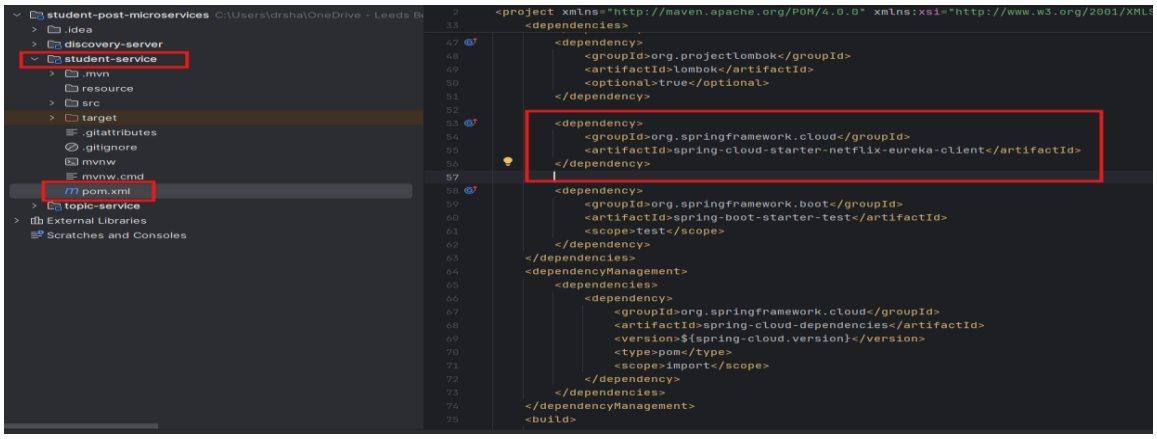


Register Student-Service with Eureka Discovery Server

Now, we are ready to register our **student-service** and **topic-service** microservices with the Eureka Discovery Server. First, we need to add the **Spring Cloud Eureka Client** dependency to the pom.xml file of **student-service**. This dependency enables **student-service** to act as a **Eureka Client**, allowing it to automatically register itself with the **Eureka Discovery Server** upon startup.

Let's open the pom.xml file of student-service and add **Spring Cloud Eureka Client** dependency as follows

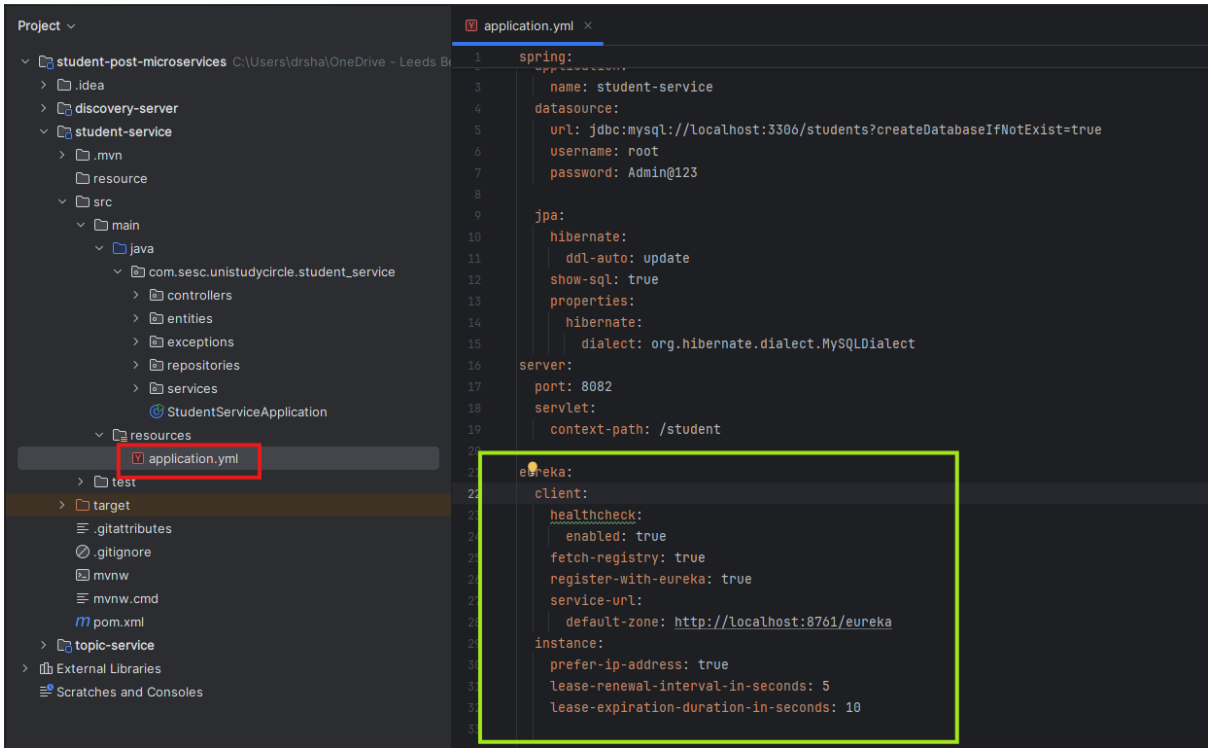
```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```



The next step is to provide necessary configuration to student-service to register with Eureka Server for service discovery.

Let's open the application.yml file of student-service located in src/main/resources folder and add following configuration

```
eureka:
  client:
    healthcheck:
      enabled: true
    fetch-registry: true
    register-with-eureka: true
    service-url:
      default-zone: http://localhost:8761/eureka
  instance:
    prefer-ip-address: true
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 10
```

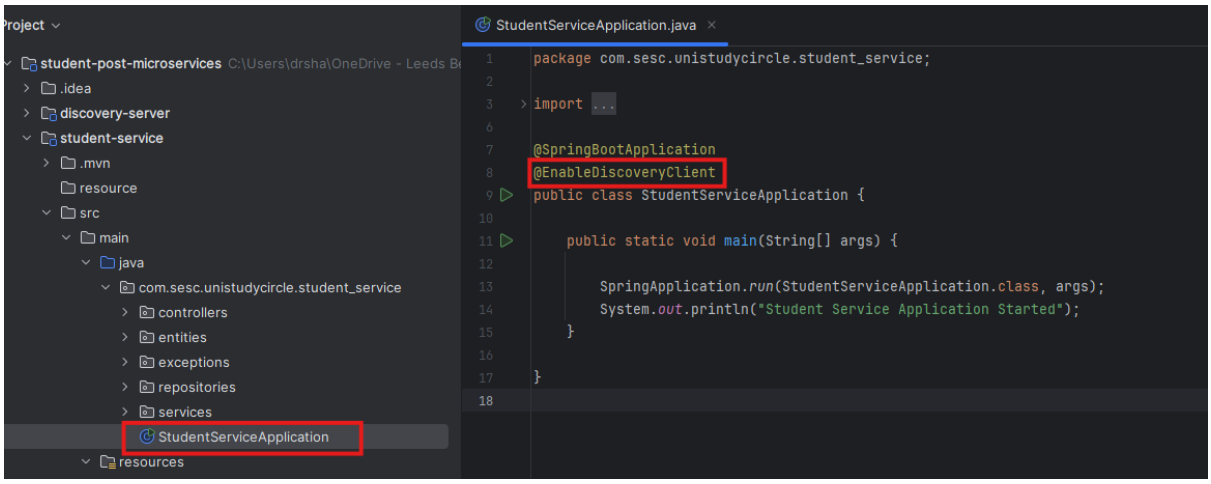


Above configuration enables our student-service to register with the Eureka Server for service discovery. The eureka:client settings ensure the service registers itself (register-with-

eureka: true) and fetches the list of available services (fetch-registry: true). It also enables health checks (healthcheck.enabled: true) to verify the service's status before directing traffic. The Eureka Server is specified at http://localhost:8761/eureka. The eureka.instance settings configure the service to use its IP address instead of its hostname (prefer-ip-address: true). Further, the service sends a heartbeat every 5 seconds (lease-renewal-interval-in-seconds: 5) to notify Eureka that it's alive, and if no heartbeat is received within 10 seconds (lease-expiration-duration-in-seconds: 10), the service may be marked as unavailable.

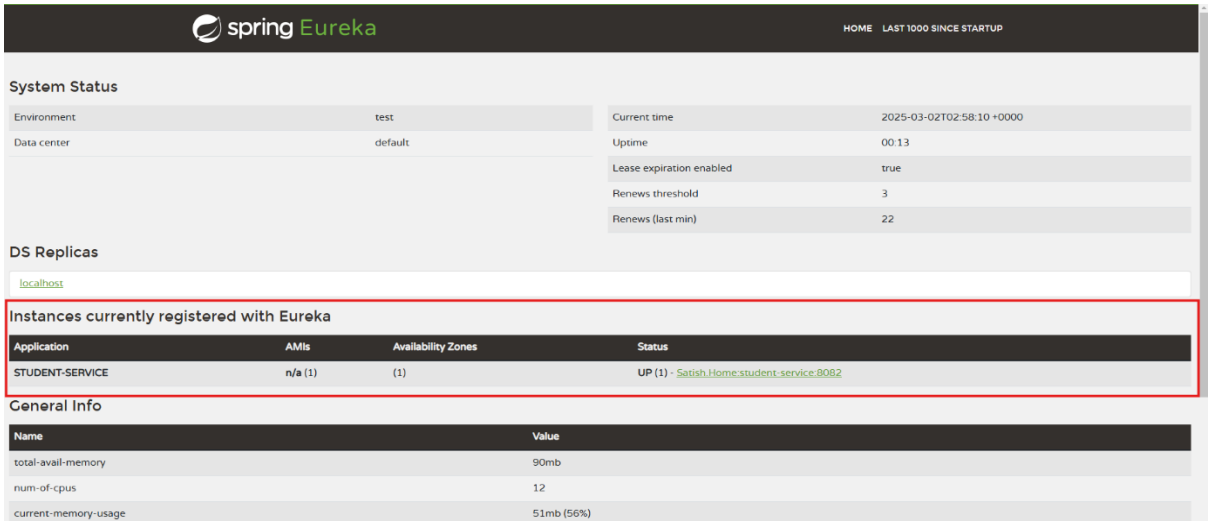
We now ready to start our student service and enable it as eureka client to register with eureka discovery server.

Let's annotate the StudentServiceApplication with @EnableDiscoveryClient annotation.



@EnableDiscoveryClient annotation enables our student-service to register with the service registry and become discoverable by other microservices.

let's start the student-service microservice and verify whether the application successfully registers with the eureka server or not. We now see that the student-service successfully registered with eureka server as shown below

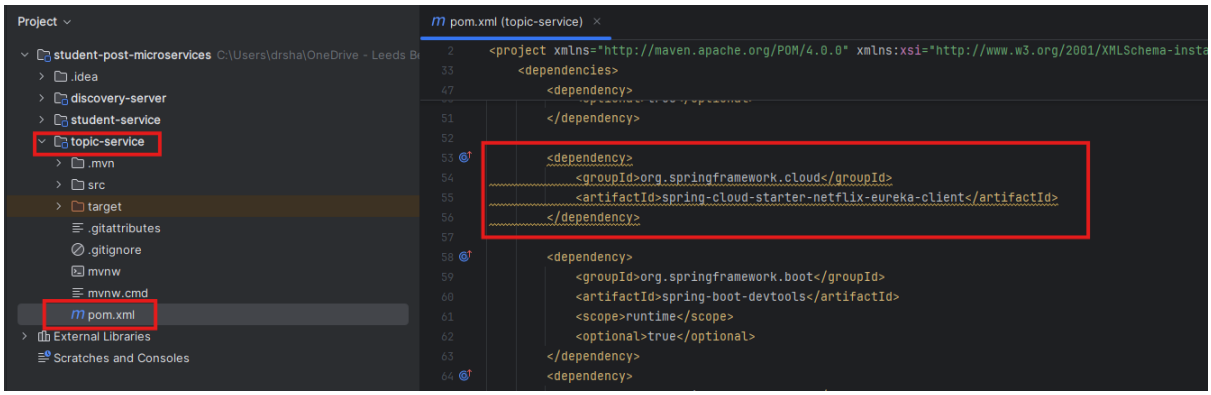


Register topic-service with Eureka Discovery Server

We need to repeat the same steps used for student-service.

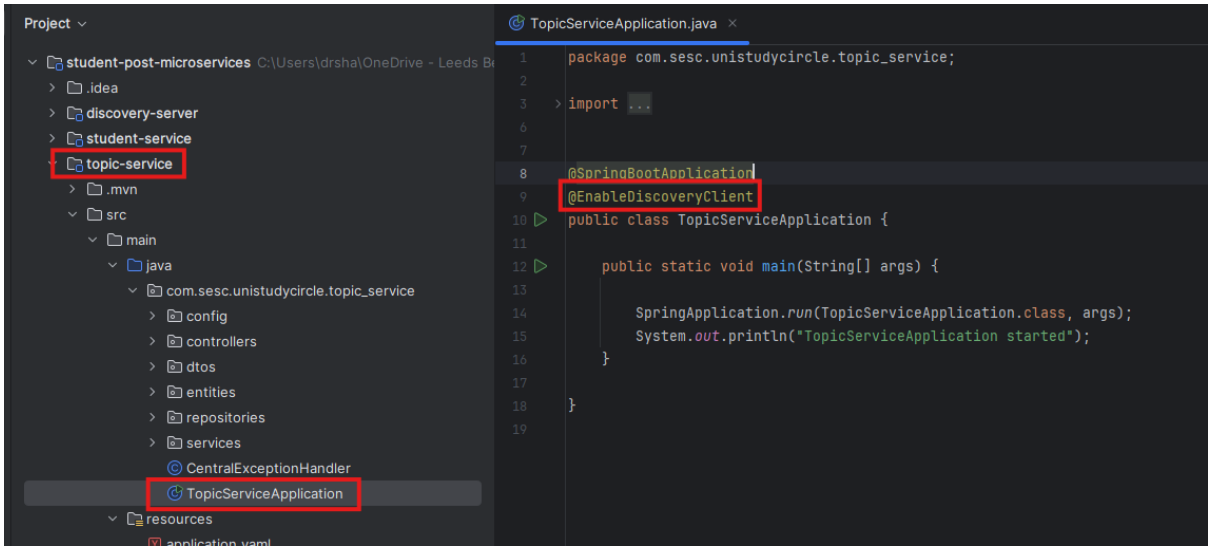
Let's open the pom.xml file of topic-service and add **Spring Cloud Eureka Client** dependency as follows

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



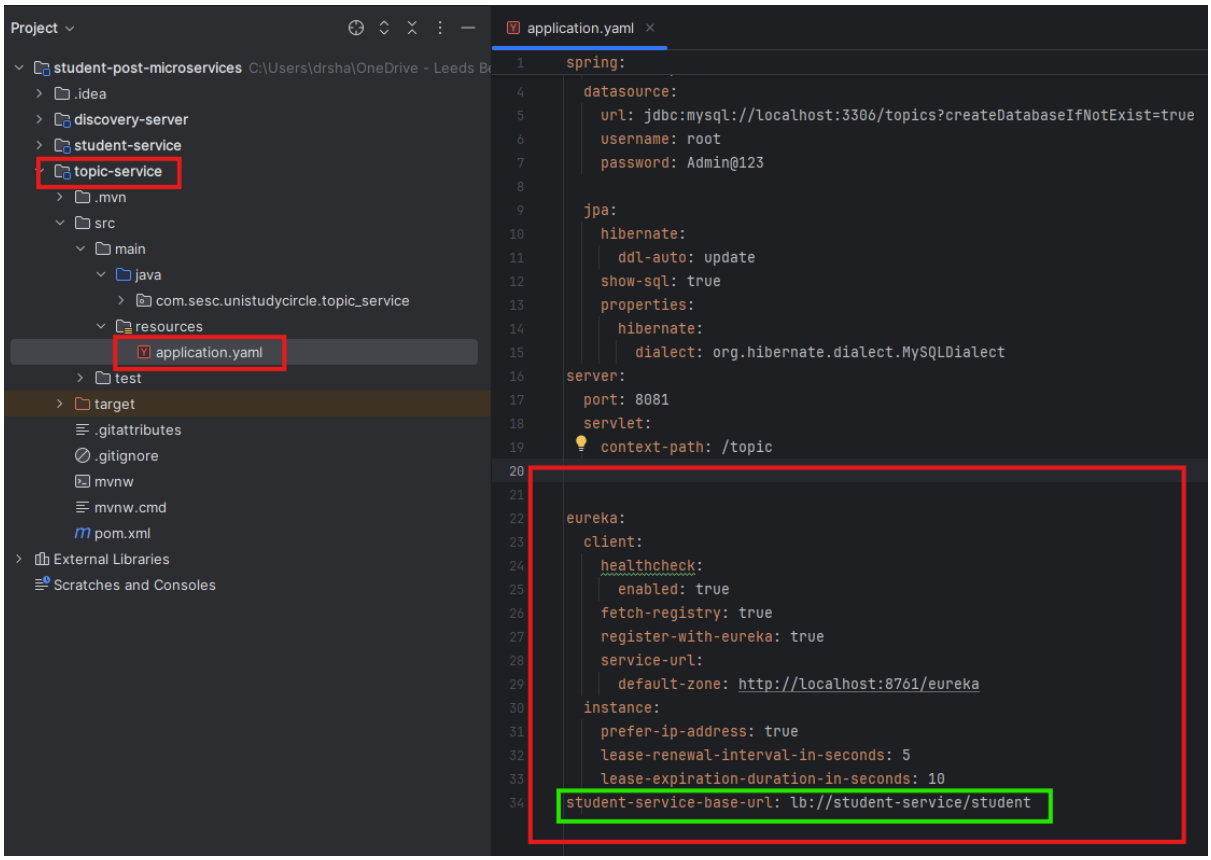
We now annotate the TopicServiceApplication class with @EnableDiscoveryClient that enables our topic-service to register with the service registry and become discoverable by other microservices.

Let's open the TopicServiceApplication located in the src/main/java/com.sesc.unistudycircle.topic_service package and add @EnableDiscoveryClient annotation on the class.



The next step is to provide necessary configuration to topic-service to register with Eureka Server for service discovery.

Let's open the application.yaml file of topic-service and add following configuration



We have applied a similar configuration for the student-service, with the exception of the green box. As shown in the green box, we implemented Service Discovery with Load Balancer for making student-service calls more flexible and dynamic.

Let's understand **student-service-base-url: lb://student-service/student**

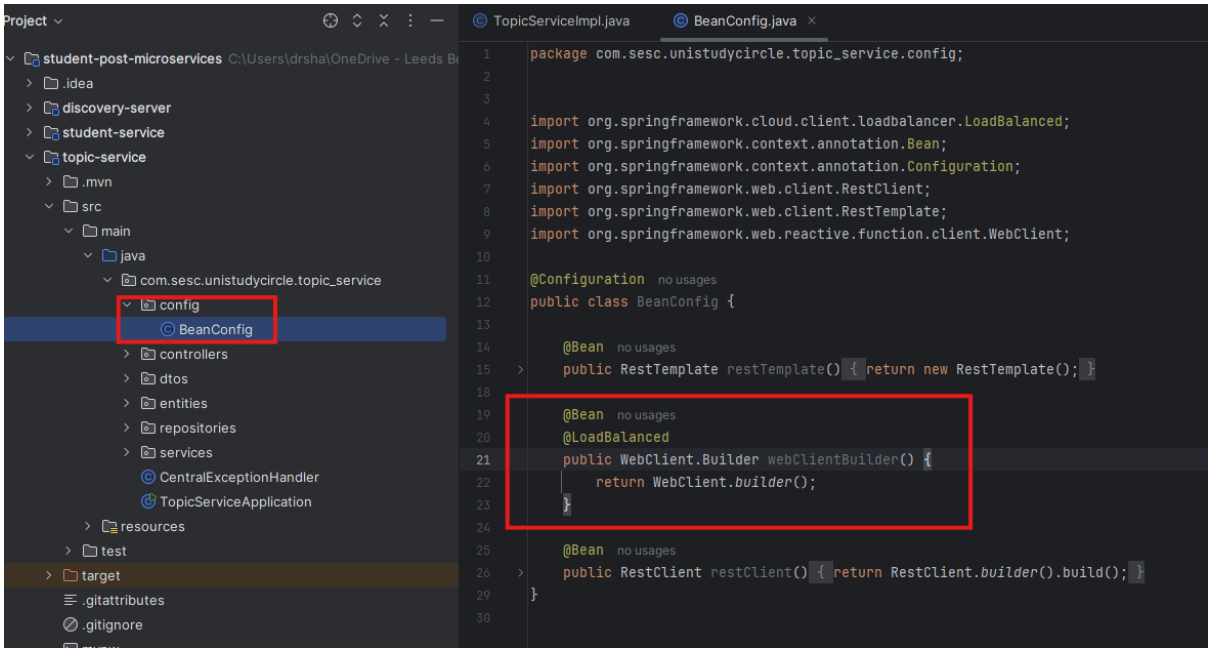
The URL format lb:// indicates that Spring Cloud should use client-side load balancing to discover the student-service from the Eureka service registry.

lb://student-service means that the topic-service will look up the student-service in the Eureka server to find one or more available instances of the student-service.

After discovering available student-service instances, Spring Cloud will use a load balancer (Spring Cloud LoadBalancer) to choose an instance of student-service and append /student to the URL for making an API call to the student-service.

We have implemented client side load balancing on WebClient as part of our service integration. First, the WebClient bean is annotated with @LoadBalanced annotation to enable a client side load balancing when making HTTP requests to student-service.This annotation is part of Spring Cloud's load-balancing mechanism, used alongside Eureka service discovery to dynamically choose between instances of a service when calling it.

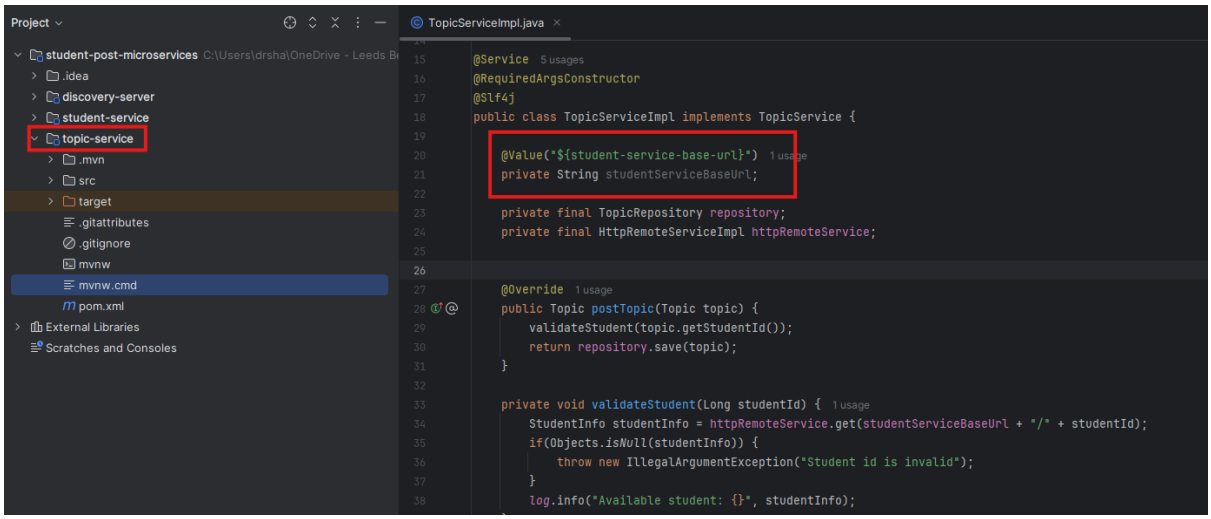
Let's add a @LoadBalanced annotation on the WebClient bean located in the BeanConfig class under config package.



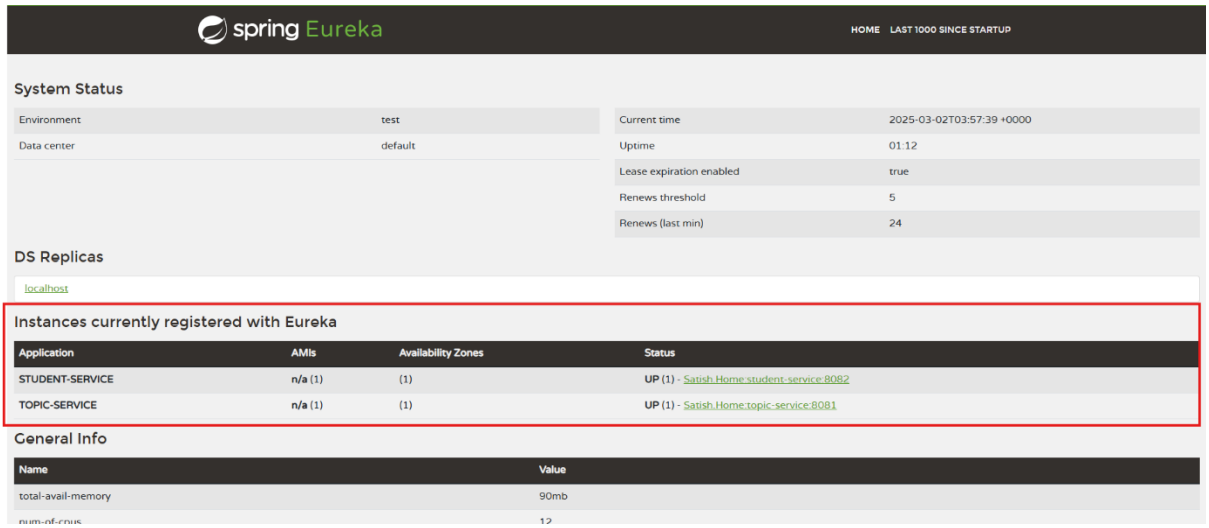
Instead of static URL, the topic-service retrieve the student-service URL dynamically from the configuration available in the eureka service registry (eureka discovery server).

In topic-service's application.yaml: The student-service-base-url property is set to lb://student-service/student. This tells the application where to find the student-service using service discovery

Let's open the TopicServiceImpl class located in the services package and add @Value("${student-service-base-url}") on the studentServiceBaseUrl field. This annotation injects the value of the student-service-base-url property into studentServiceBaseUrl as shown below



let's start the topic-service microservice and verify whether the application successfully registers with the eureka server or not. We now see that the student-service and topic-service successfully registered with eureka server as shown below



API Gateway

In a microservices architecture, a client might interact with more than one microservices. Given this fact, how does a client know what endpoints to call? What happens when new services are introduced, or existing services are refactored? An *API gateway* can help to address these challenges.

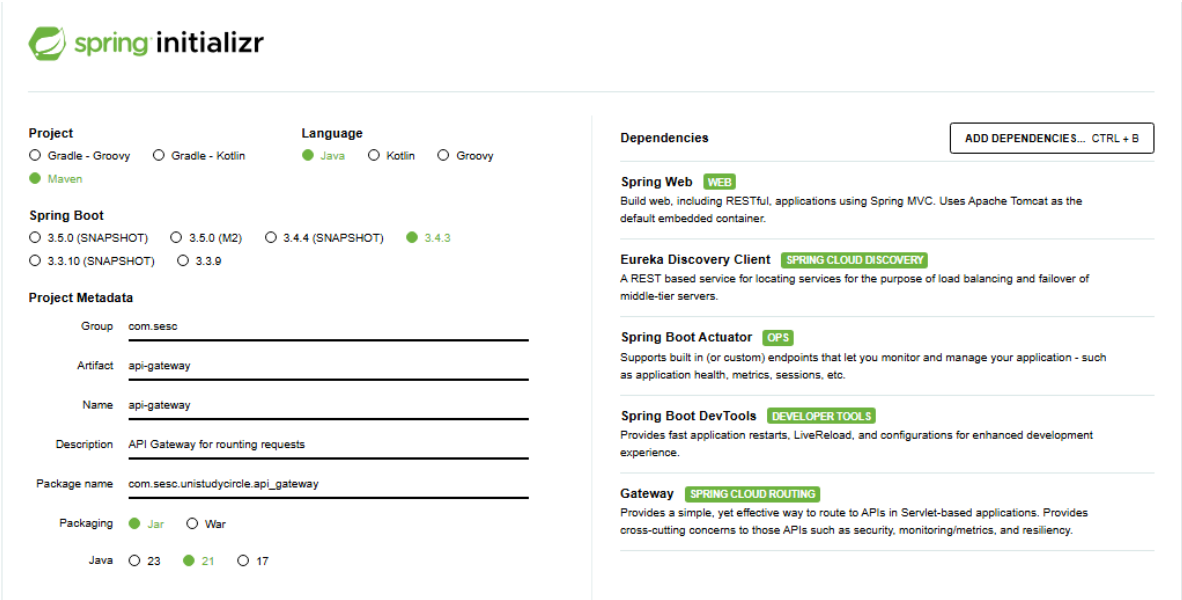
An API gateway provides a centralized entry point for managing interactions between clients and application services. It acts as a reverse proxy and routes clients' requests to the appropriate services.

Implementing API Gateway for routing HTTP requests on student-service and topic-service

First, we need to create a api-gateway microservice project and add the following dependencies and click on the generate button to download this project. You can access spring initializr at https://start.spring.io/

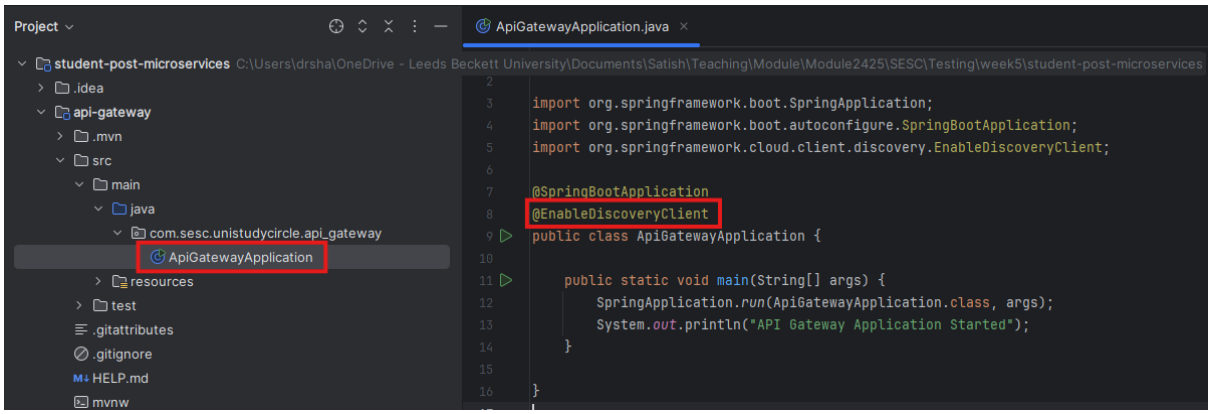
- Spring Web
- Gateway (Spring Cloud Gateway)
- Eureka Discovery Client
- Spring Boot Tools
- Spring Boot Actuator

The next step is to unzip and copy the **api-gateway** microservice and paste it inside the student-post-microservice folder (the current working project opened with IntelliJ IDEA) and import (initialize) **api-gateway** microservice by follow similar steps used in the discovery-server initialization.



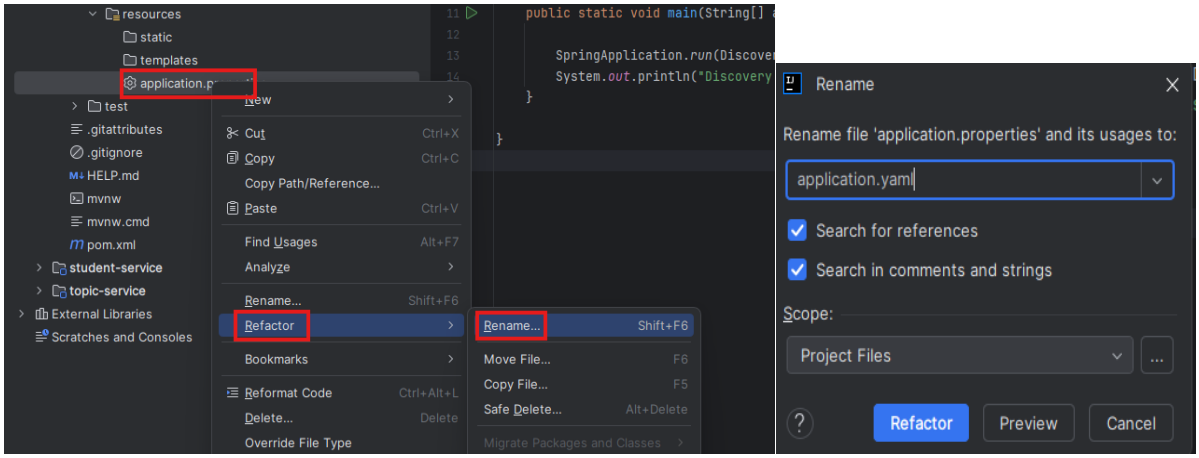
We now annotate the ApiGatewayApplication class with @EnableDiscoveryClient that enables our api-gateway to register with the service registry and become discoverable by other microservices.

Let's open the ApiGatewayApplication located in the src/main/java/com.sesc.unistudycircle.api_gateway package and add @EnableDiscoveryClient annotation on the class.



Let's change application.properties to **application.yaml** file by refactoring file name as shown below and add necessary configuration in the application.yaml file

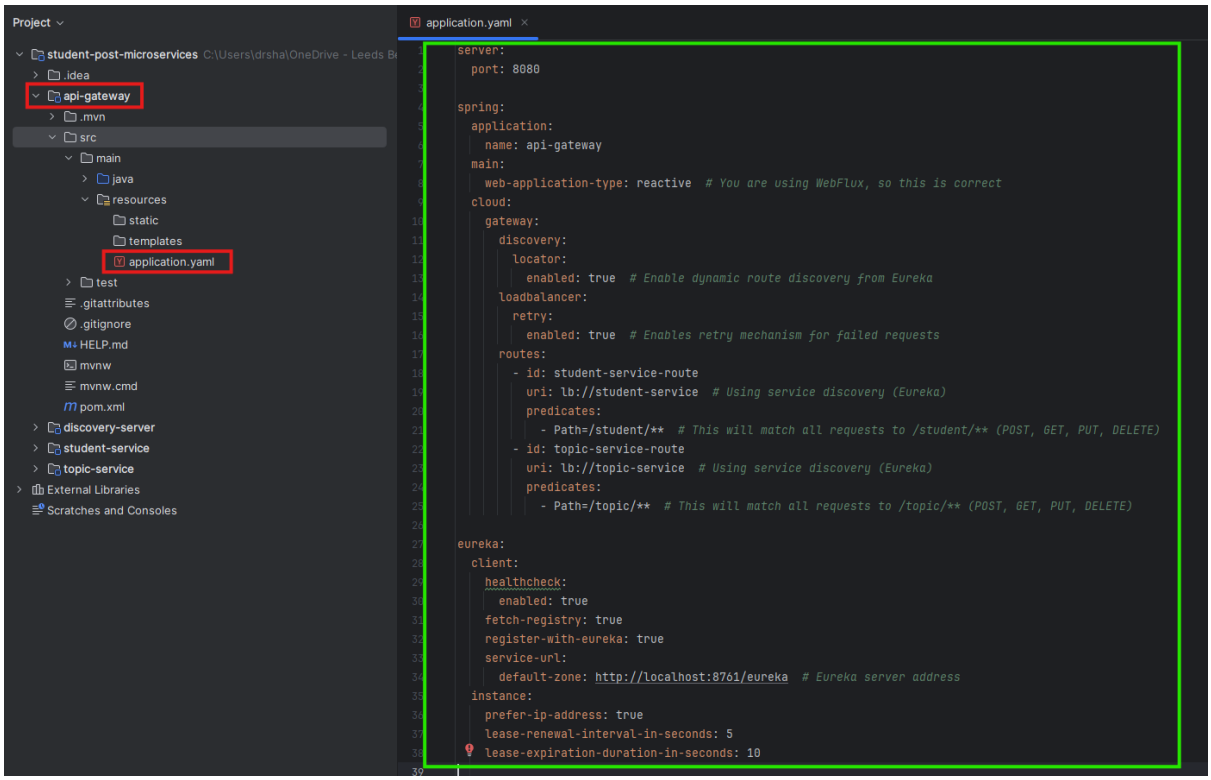
Note: The **application.properties** file is located in the src/main/resources folder.



We now ready to provide configuration to api-gateway microservice to route requests dynamically to the student-service and topic-service registered in Eureka Discovery Server.

To achieve this, first api-gateway microservice needs to register itself in the Eureka service registry and then discover other microservice such as student-service and topic-service in the discovery server based on following configurations

```yaml
server:
  port: 8080

spring:
  application:
    name: api-gateway
  main:
    web-application-type: reactive  # You are using WebFlux, so this is correct
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true  # Enable dynamic route discovery from Eureka
      loadbalancer:
        retry:
          enabled: true  # Enables retry mechanism for failed requests
      routes:
        - id: student-service-route
          uri: lb://student-service  # Using service discovery (Eureka)
          predicates:
            - Path=/student/**  # This will match all requests to /student/** (POST, GET, PUT, DELETE)
        - id: topic-service-route
          uri: lb://topic-service  # Using service discovery (Eureka)
          predicates:
            - Path=/topic/**  # This will match all requests to /topic/** (POST, GET, PUT, DELETE)

eureka:
  client:
    healthcheck:
      enabled: true
    fetch-registry: true
    register-with-eureka: true
    service-url:
      default-zone: http://localhost:8761/eureka  # Eureka server address
  instance:
    prefer-ip-address: true
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 10
```



NOTE: If you get any error related to dependencies, then copy pom.xml file of api-gateway available at https://github.com/LBUSESC/week6 and paste it in your api-gateway's pom.xml

Now, start api-gateway and call student-service and post-service through API-GATEWAY microservice available at http://localhost:8080

Open the postman client and call the student-service at http://localhost:8080/student/create as shown below

POST http://localhost:8080/student/create

Params Authorization Headers (8) Body Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
  "firstName": "Tony",
  "lastName": "Jenkins",
  "email": "t.jenkins@leedsbeckett.ac.uk",
  "qualification": "MSc",
  "university": "Leeds Beckett University"
}
```

Body Cookies Headers (5) Test Results

JSON Preview Visualize

```
{
    "studentId": 1,
    "firstName": "Tony",
    "lastName": "Jenkins",
    "email": "t.jenkins@leedsbeckett.ac.uk",
    "qualification": "MSc",
    "university": "Leeds Beckett University"
}
```

Now let's call the topic-service through api-gate at http://localhost:8080/topic/create as shown below

POST http://localhost:8080/topic/create Send

Params Authorization Headers (8) Body Scripts Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
  "topicName": "Introduction to AI",
  "topicDate": "2025-02-26",
  "topicContent": "Artificial Intelligence (AI) refers to the simulation of human intelligence processes by machines, particularly computer systems. These processes include learning, reasoning, problem-solving, perception, and language understanding. AI can be classified into two types: Narrow AI, which is designed for a specific task (e.g., facial recognition, language translation), and General AI, which aims to mimic human cognitive abilities across a broad range of tasks. Machine learning (ML) is a subset of AI that enables systems to learn from data and improve over time without explicit programming. Deep learning, a branch of ML, utilizes neural networks with many layers to analyze complex data patterns. AI is revolutionizing various industries, including healthcare, finance, transportation, and entertainment, offering increased efficiency, automation, and predictive capabilities. However, it also raises ethical concerns regarding data privacy, job displacement, and decision-making transparency.",
  "studentId": 1,
  "studentName": "Tony Jenkins"
}
```

Body Cookies Headers (5) Test Results 201 Created 1.25 s 1.27 KB Save Response

JSON Preview Visualize

```
{
    "topicId": 1,
    "topicName": "Introduction to AI",
    "topicDate": "2025-02-26",
    "topicContent": "Artificial Intelligence (AI) refers to the simulation of human intelligence processes by machines, particularly computer systems. These processes include learning, reasoning, problem-solving, perception, and language understanding. AI can be classified into two types: Narrow AI, which is designed for a specific task (e.g., facial recognition, language translation), and General AI, which aims to mimic human cognitive abilities across a broad range of tasks. Machine learning (ML) is a subset of AI that enables systems to learn from data and improve over time without explicit programming. Deep learning, a branch of ML, utilizes neural networks with many layers to analyze complex data patterns. AI is revolutionizing various industries, including healthcare, finance, transportation, and entertainment, offering increased efficiency, automation, and predictive capabilities. However, it also raises ethical concerns regarding data privacy, job displacement, and decision-making transparency.",
    "studentId": 1,
    "studentName": "Tony Jenkins"
}
```

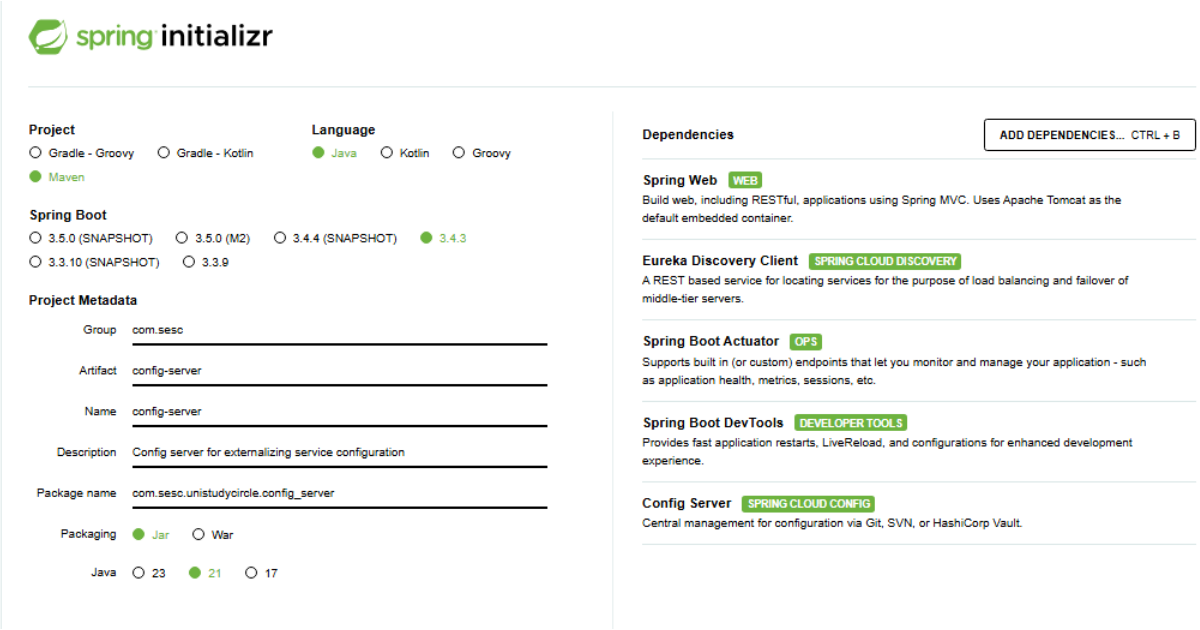
Config Server

> Spring Cloud Config provides server-side and client-side support for externalized configuration in a distributed system. With the Config Server, you have a central place to manage external properties for your microservices across all environments.

First, we need to create a config-server microservice project and add the following dependencies and click on the generate button to download this project. You can access spring initializr at https://start.spring.io/

- Spring Web
- Config Server
- Eureka Discovery Client

- Spring Boot Tools
- Spring Boot Actuator

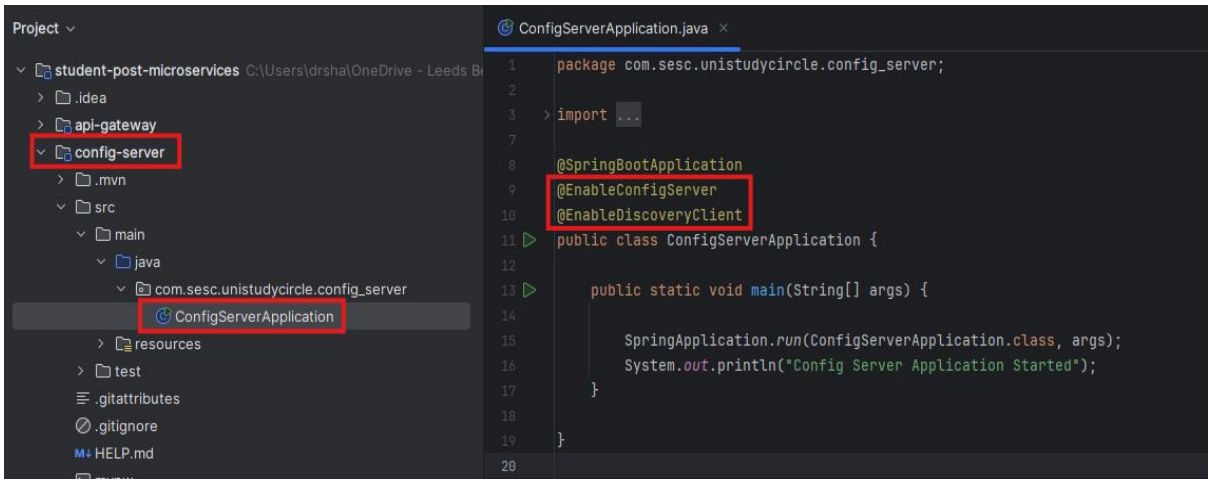


The next step is to unzip and copy the **config-server** microservice and paste it inside the student-post-microservice folder (the current working project opened with IntelliJ IDEA) and then import (initialize) **config-server** microservice by follow similar steps used in the discovery-server and api-gateway initialization.

We now annotate the ConfigServerApplication class with @EnableConfigServer

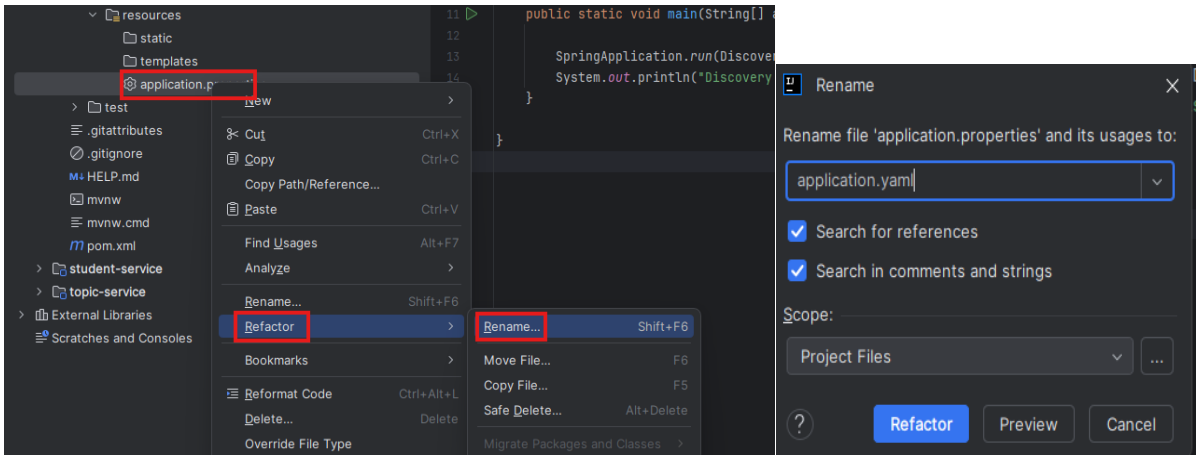
And @EnableDiscoveryClient that enables our config-server microservice to register with the service registry and become discoverable by other microservices.

Let's open the ConfigServerApplication located in the src/main/java/com.sesc.unistudycircle.config_server package and add @EnableConfigServer and @EnableDiscoveryClient annotation on the class.



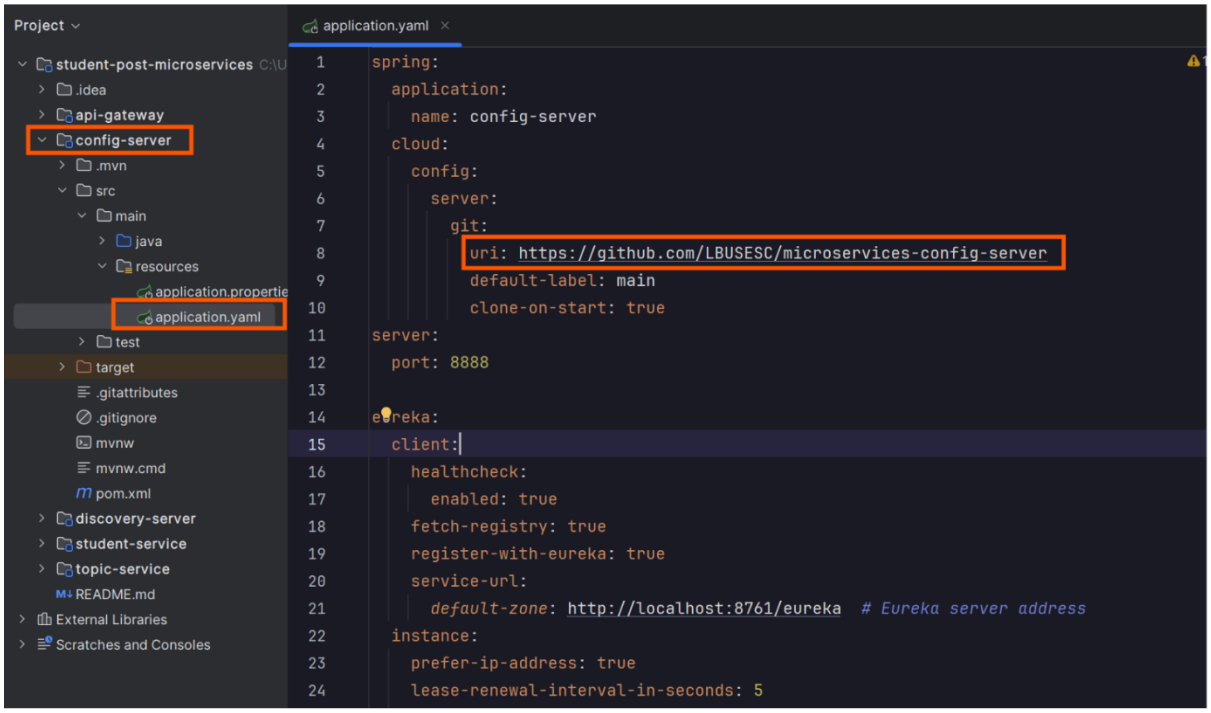
We now ready to provide configuration to config-server microservice to register itself in the Eureka service discovery and externalize other microservices (student-service and post-service) configuration on the GitHub.

Let's change your config-server application.properties to **application.yaml** file by refactoring file name as shown

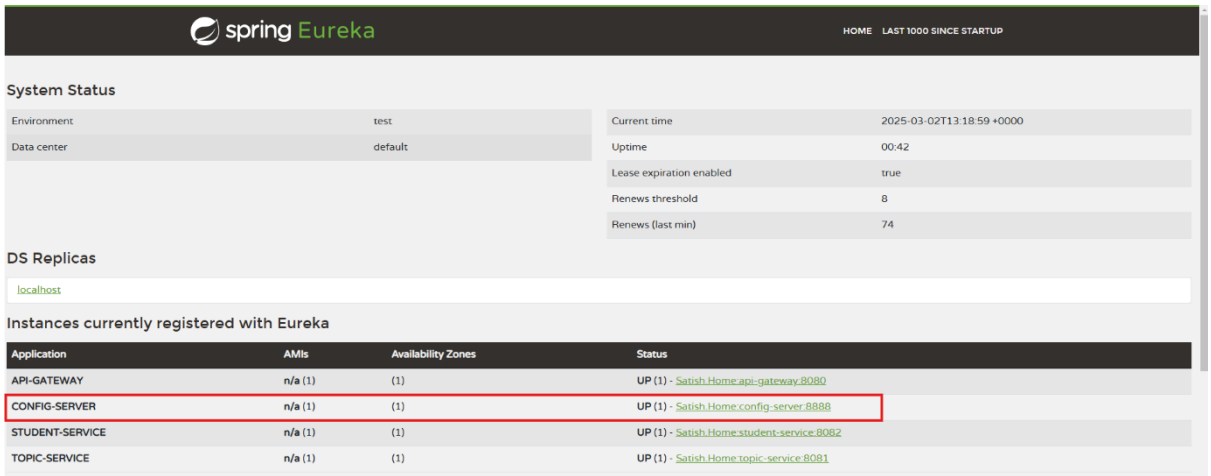


Let's add following configuration in application.yaml file of config-server as shown below

```yaml
spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/LBUSESC/microservices-config-server
          default-label: main
          clone-on-start: true
server:
  port: 8888

eureka:
  client:
    healthcheck:
      enabled: true
    fetch-registry: true
    register-with-eureka: true
    service-url:
      default-zone: http://localhost:8761/eureka  # Eureka server address
  instance:
    prefer-ip-address: true
    lease-renewal-interval-in-seconds: 5
    lease-expiration-duration-in-seconds: 10

management:
  endpoints:
    web:
      exposure:
        include: "*"  # Exposing all management endpoints
  endpoint:
    info:
      enabled: true
```



Let's start config-server microservice and see it status as registered in the Eureka discovery-server as shown below



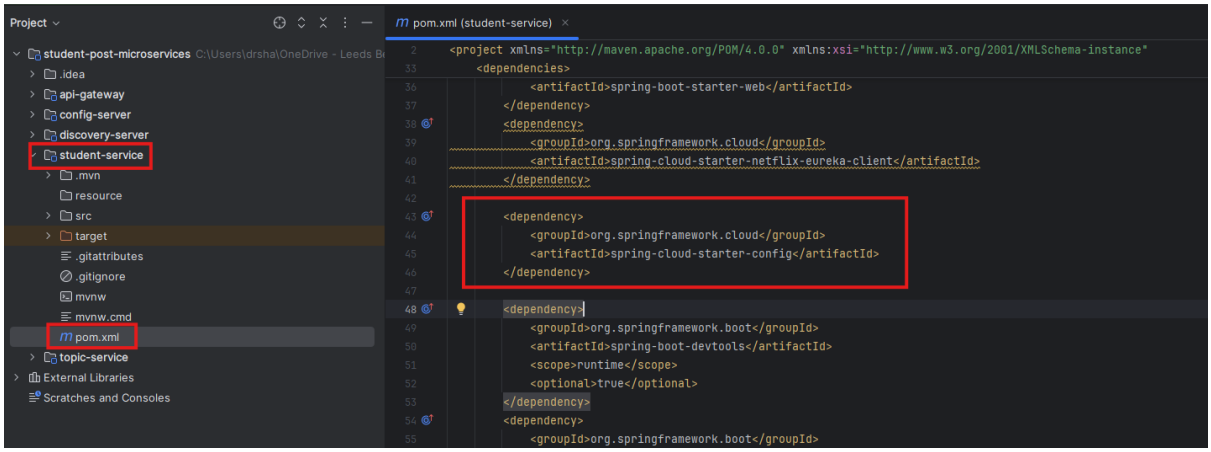
We now ready to externalize our student-service and topic-service configuration on the GitHub.

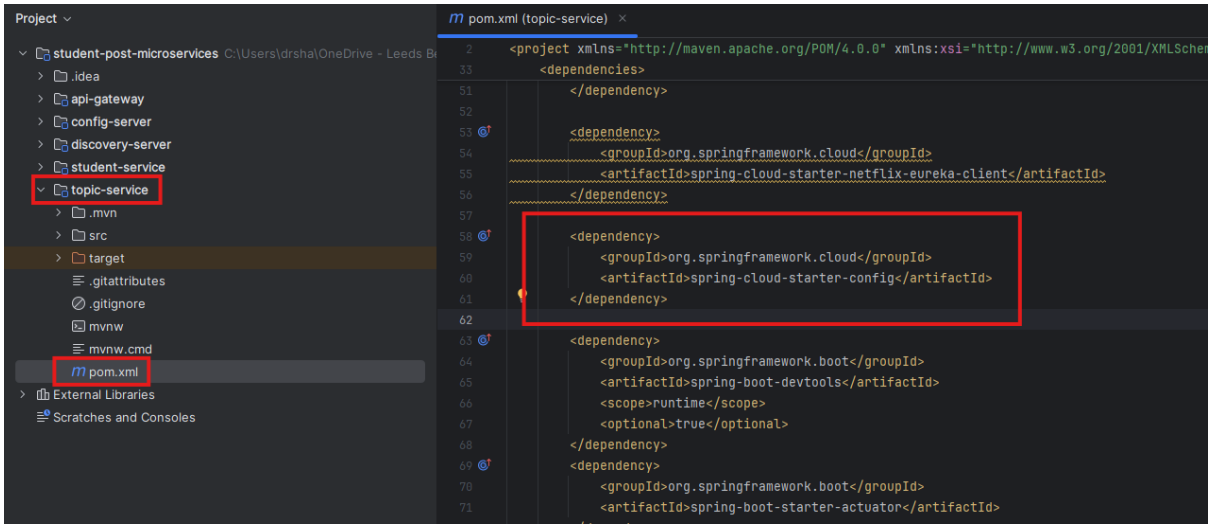
First, we need to add Config Client dependency in the pom.xml file of both student-service and topic-service as follows

```
<dependency>

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

```
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

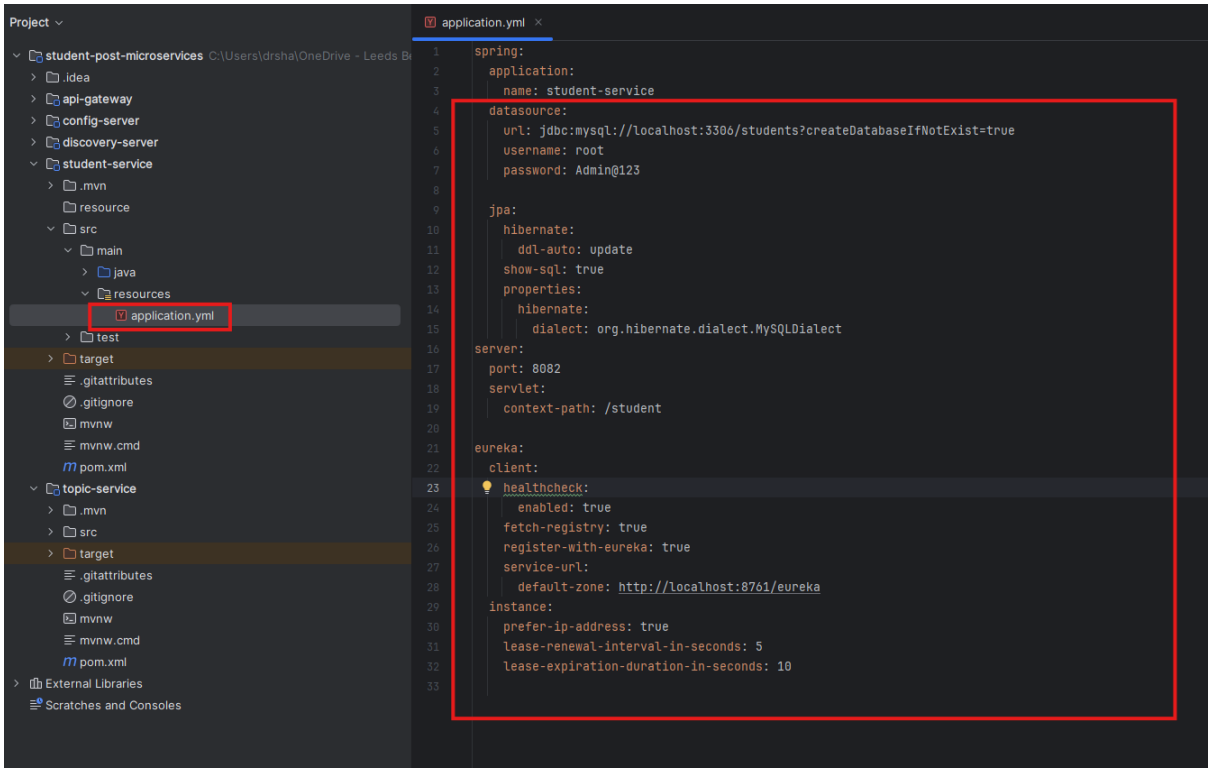
Let’s add above dependency in the pom.xml file of student-service and topic-service





Externalize configuration for student-service

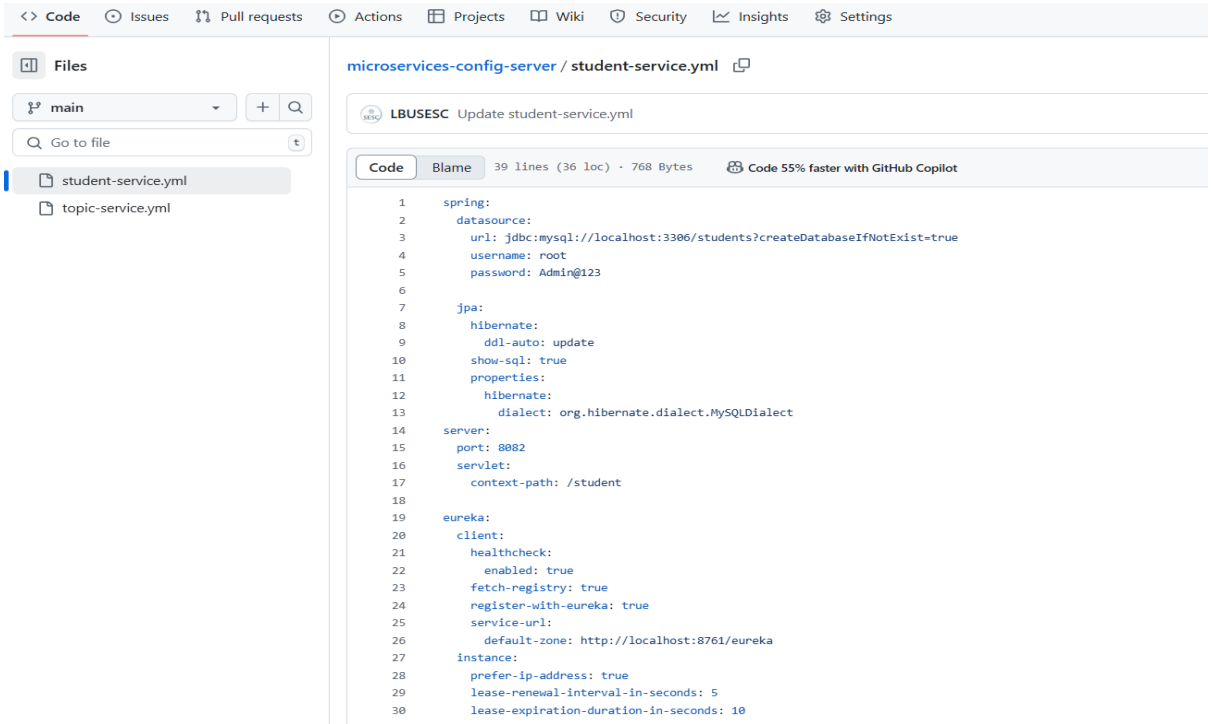
Now, we are in a position to externalize the student-service’s database and service discovery configuration on the GitHub repository as shown below



add above red box configuration to the github repository.

We now ready to externalize student-service red box configuration as shown above. You can find this configuration at https://github.com/LBUSESC/microservices-config-server

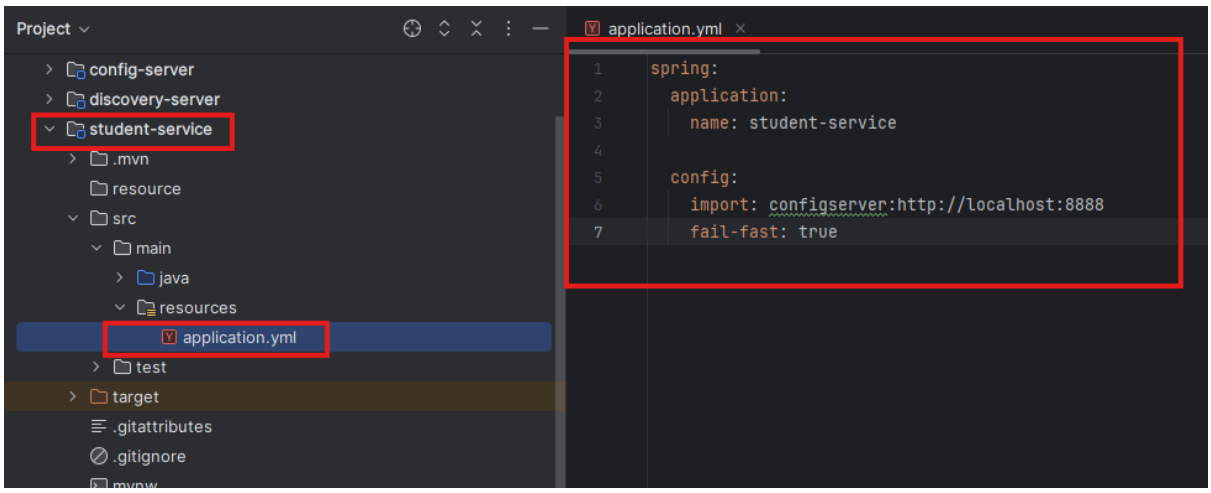
Now you can explore student-service.yml at above URL and check student-service.yml configuration as shown below



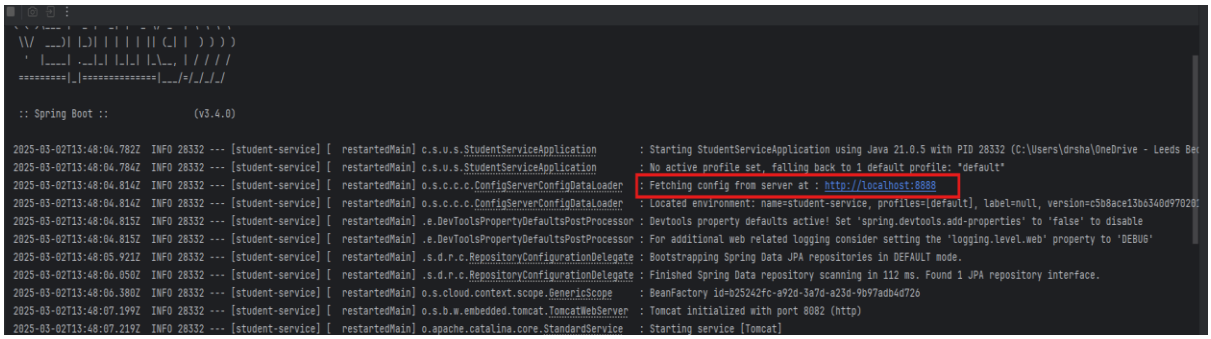
The next step is to update the application.yaml file of student-service to fetch this configuration

Let's update the application.yaml file of student-service

```yaml
spring:
  application:
    name: student-service

  config:
    import: configserver:http://localhost:8888
    fail-fast: true
```

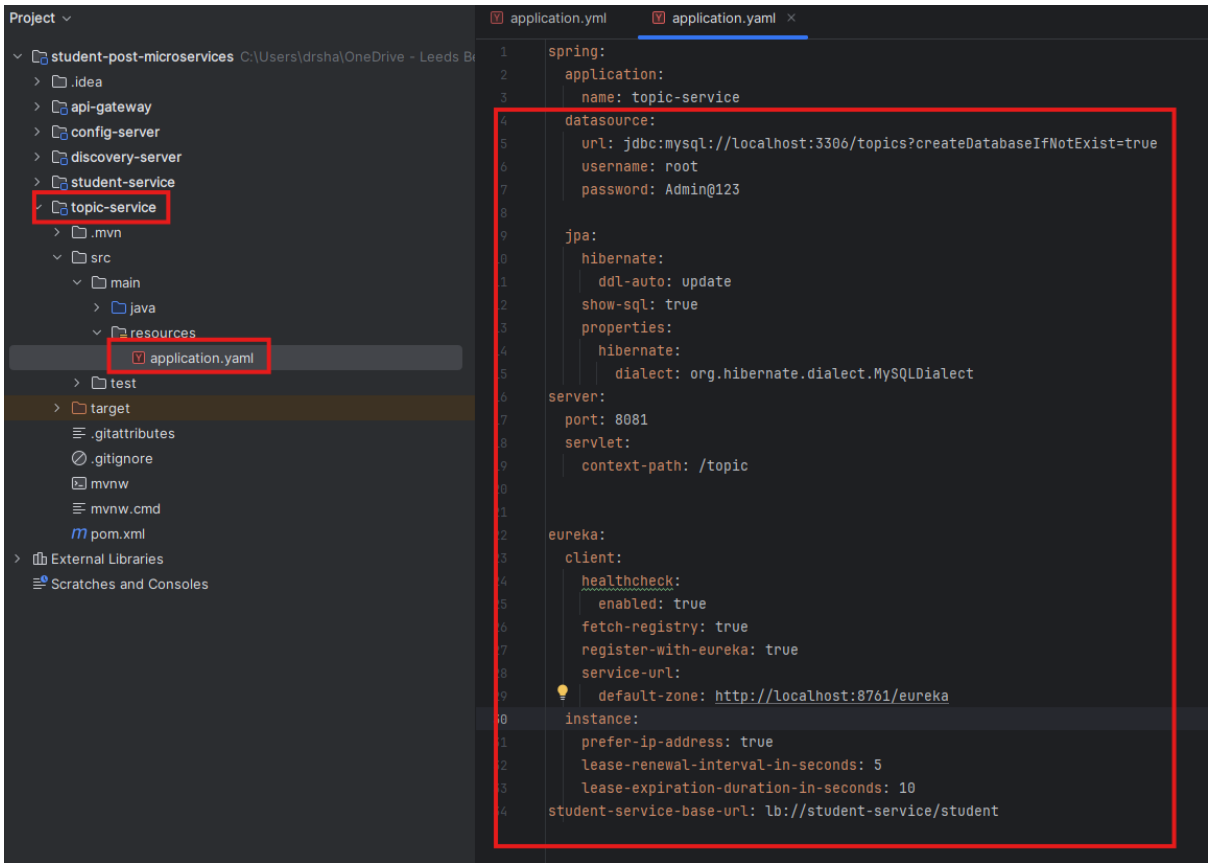


Now start student-service and check the execution log to verify that the student-service fetching configuration from the config server as shown below



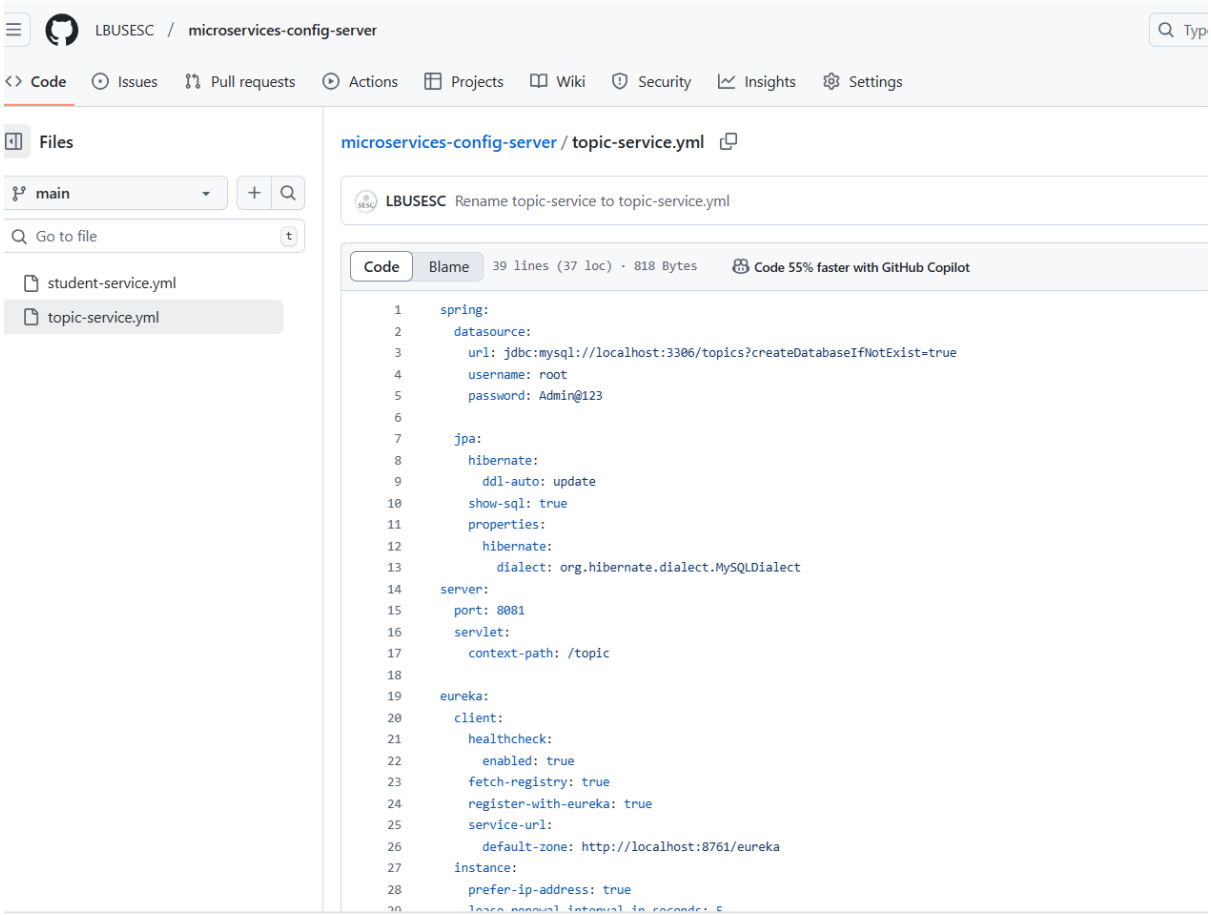
Externalize configuration for topic-service

Let's externalize the topic-service's database and service discovery configuration on the github repository as shown below



We now ready to externalize topic-service red box configuration as shown above. You can find this configuration at https://github.com/LBUSESC/microservices-config-server

Now you can explore topic-service.yml at above URL and check topic-service.yml configuration as shown below



The next step is to update the application.yaml file of topic-service to fetch this configuration

Let's update the application.yaml file of topic-service

```
spring:
  application:
    name: topic-service

  config:
    import: configserver:http://localhost:8888
    fail-fast: true
```

Now start topic-service and check the execution log to verify that the topic-service fetching configuration from the config server.

Well done.

Conclusion

At this point you have:

- gained theoretical and practical knowledge of implementing spring cloud config server, service discovery and api-gateway.