**Hands On Lab 1**

**Create a Spring Cloud Eureka Server and Client**

[Overview 1](#_Toc22072)

[What is Netflix Eureka Server and Clients? 3](#_Toc15109)

[Eureka Service Registry Server 4](#_Toc26419)

[Eureka Client – Student Service 6](#_Toc29719)

[Eureka Client – School Service 11](#_Toc30101)

[Test of Service Discovery and Clients 14](#_Toc9034)

[Things to check if facing any error 15](#_Toc17292)

[Summary 15](#_Toc26320)

In this lab, we will learn to create microservices, based on Spring cloud, with Netflix Eureka registry server and how other microservices (Eureka clients) use it to register and discover services to call their APIs.

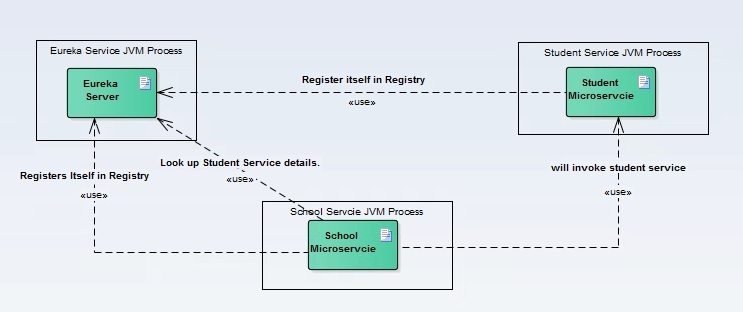
We will be using Spring Boot based Spring Cloud API. We will use Netflix Eureka server for building the service registry server and Eureka clients which will register themselves and discover other services to call REST APIs.

## **Overview**

We will create three microservices for this ****Netflix Eureka example****.

1. ****Eureka Service Registry Server**** – This microservice will provide the service registry and discovery server.
2. ****Student Microservice**** – Which will give some functionality based on Student entity. It will be a rest based service and most importantly it will be a eureka client service, which will talk with eureka service to register itself in the service registry.
3. ****School Microservice**** – Same type as of Student service – only added feature is that it will invoke Student service with *service look up* mechanism. We will not use absolute URL of student service to interact with that service.

Here is the interaction diagram between above listed three services.

[](https://howtodoinjava.com/wp-content/uploads/2017/07/Discovery_interratction-Diagram.jpg)**Component Interactions**

#### **Tech Stack and Runtime**

* Java 11
* Spring Tool Suite
* Spring Cloud
* Spring Boot
* Maven

## **What is Netflix Eureka Server and Clients?**

As we know these days, there is a lot of hype around Microservices. The transition from Monolithic to Microservice based architecture gives many benefits for the future in terms of maintainability, scalability, high availability etc. However at the same time, there are many challenges also while doing this migration. One of them is to maintain individual Microservices addresses. This task can be hugely complex – depending on number of services and their dynamic nature. If the whole infrastructure is distributed and there is some replication as well, then maintaining these service addresses becomes harder.

To solve this, in distributed computing there is a concept called ‘Service registration and discovery’ where one dedicated server is responsible to maintain the registry of all the Microservices that have been deployed and/or removed. This will act like a phone book for all other applications/microservices.

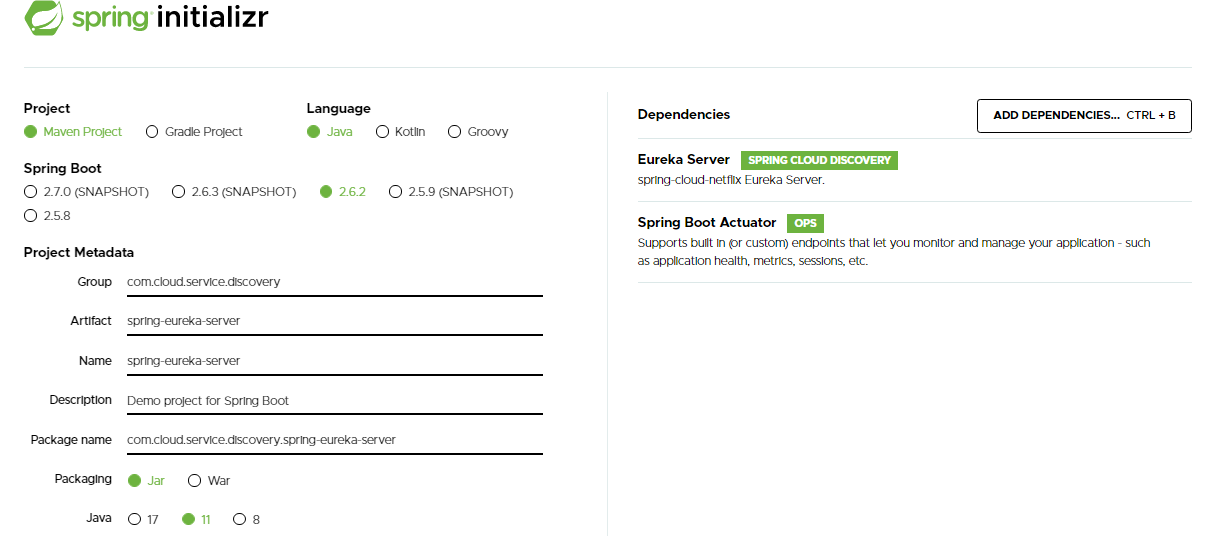
Think of it as a lookup service where microservices (clients) can register themselves and discover other registered microservices. When a client microservice registers with Eureka it provides metadata such as host, port, and health indicators thus allowing for other microservices to discover it. The discovery server expects a regular heartbeat message from each microservice instance. If an instance begins to consistently fail to send a heartbeat, the discovery server will remove the instance from his registry. This way we will have a very stable ecosystem of Microservices collaborating among each other, and on top of it we don’t have to manually maintain addresses of other Microservices, which is a next to impossible task if the scale up/down is very frequent, on demand and we use virtual host to host the services specially in the cloud environment.

## **Eureka Service Registry Server**

Follow these steps to create and run Eureka server.

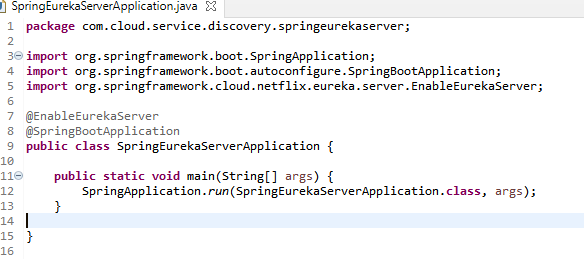
#### **Create Eureka Server**

Create a Spring boot project from [Spring Boot initializer portal](https://start.spring.io/" \t "https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/_blank) with two dependencies i.e. Eureka server and Actuator. Give other maven coordinates and download the project.



Unzip and import the project into Eclipse as existing maven project. In this step, all necessary dependencies will be downloaded from maven repository.

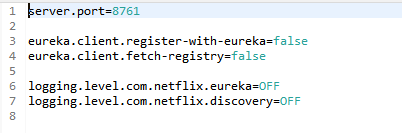
Now open SpringEurekaServerApplication class that spring already has generated in the downloaded project and add the @EnableEurekaServerannotation on the class.



Build the project once again. With this annotation, this artifact will act like microservice registry and discovery server.

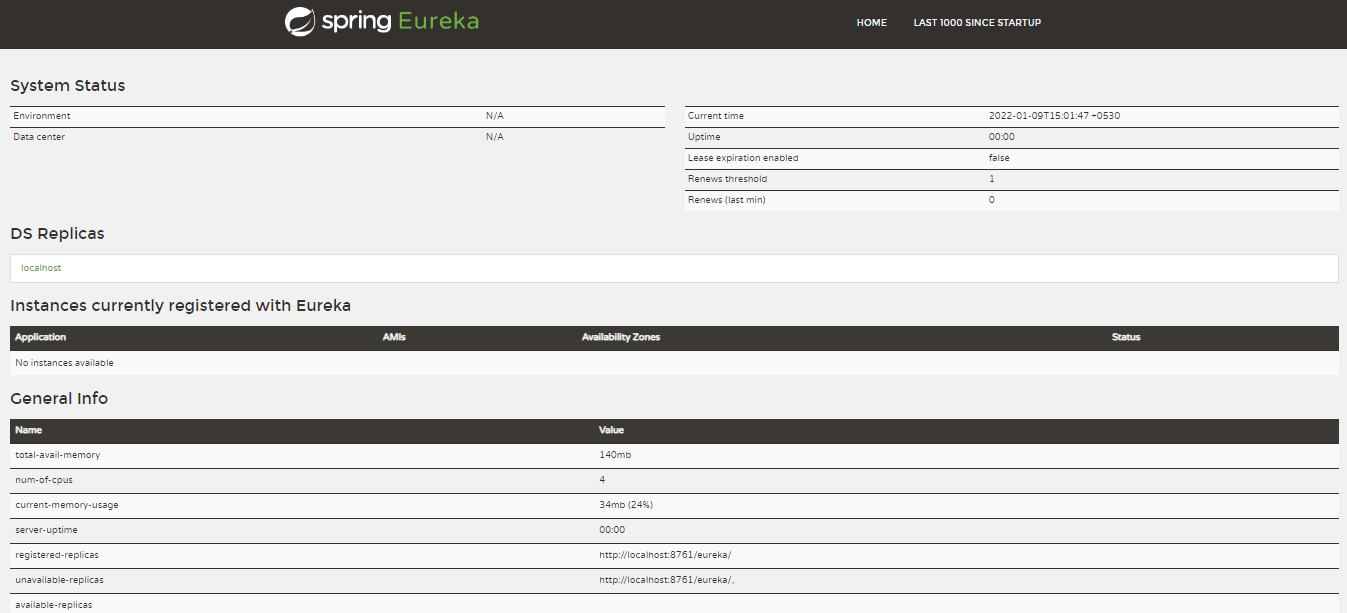
#### **Server Configuration**

Edit or create one file called application.properties in the src\main\resources directory. Add these properties –



#### **Test Eureka Server**

Start the application as spring boot application. Open browser and go to http://localhost:8761/, you should see the eureka server home page which looks like below.



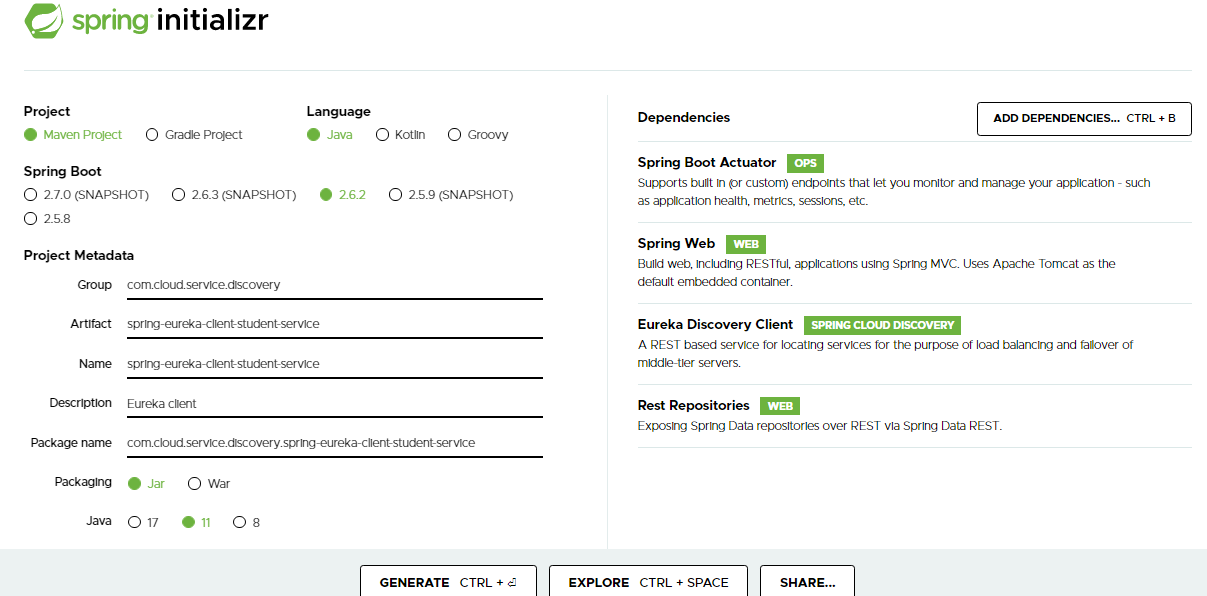
Please note that at this point no service is registered here which is expected and once we will spin up the client services, this server will automatically updated with the details of the client services.

## **Eureka Client – Student Service**

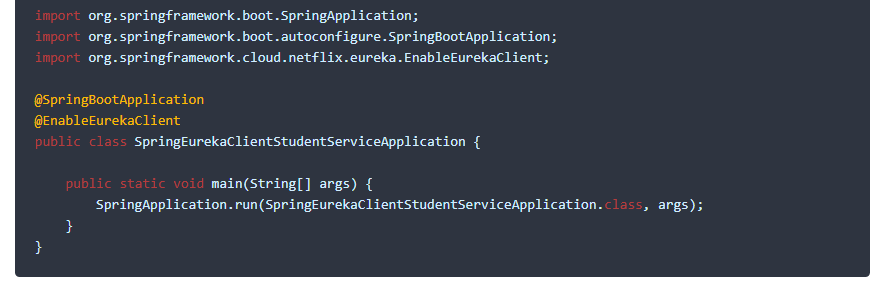
Follow these steps to create and run Eureka client running student service.

#### **Create Eureka Client Project**

Create a Spring boot project from initializer portal with four dependencies i.e. Actuator, Web, Rest Repositories, Eureka Discovery. Give other maven coordinates and download the project.

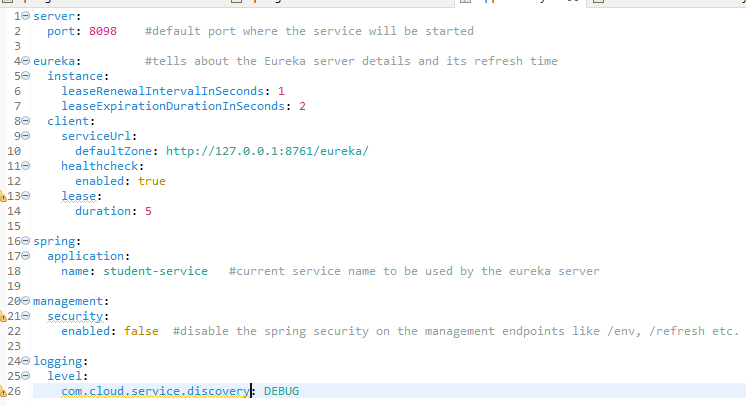
Unzip and import the project into Eclipse as existing maven project.

Now add the [@EnableEurekaClient](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-eureka-client/src/main/java/org/springframework/cloud/netflix/eureka/EnableEurekaClient.java" \t "https://howtodoinjava.com/spring-cloud/spring-cloud-service-discovery-netflix-eureka/_blank) annotation on Spring boot application class present in src folder. With this annotation, this artifact will act like a spring discovery client and will register itself in the eureka server attached to this service.



#### **Client Configuration**

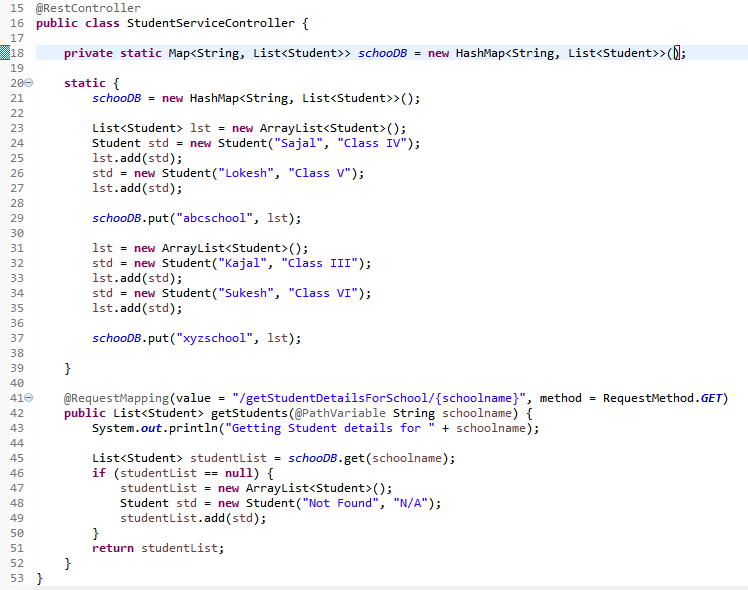
Create one file called application.yml in the src\main\resources directory and add below lines.



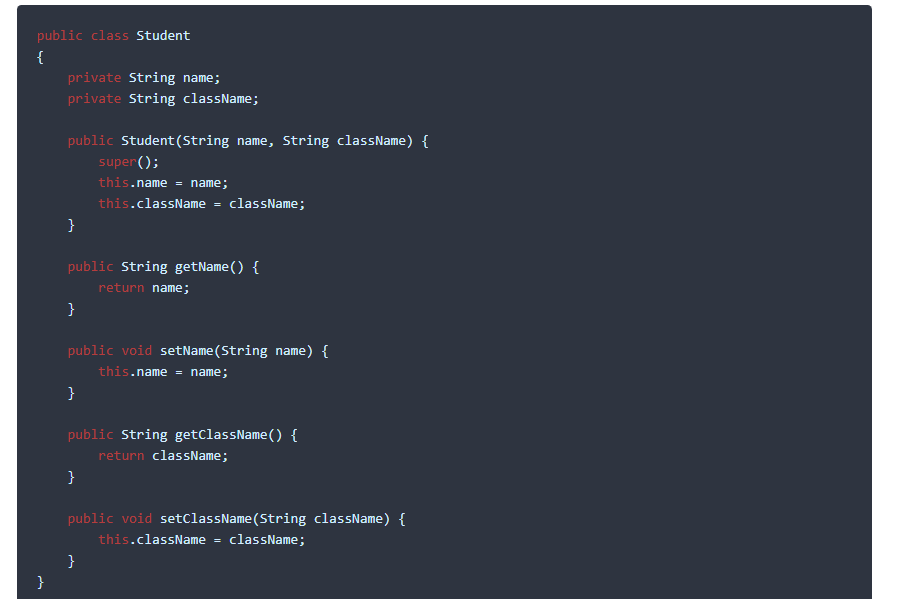
#### **Add REST API**

Now add one RestController and expose one rest endpoint for getting all the student details for a particular school. Here we are exposing /getStudentDetailsForSchool/{schoolname} endpoint to serve the business purpose. For simplicity, we are hard coding the student details.



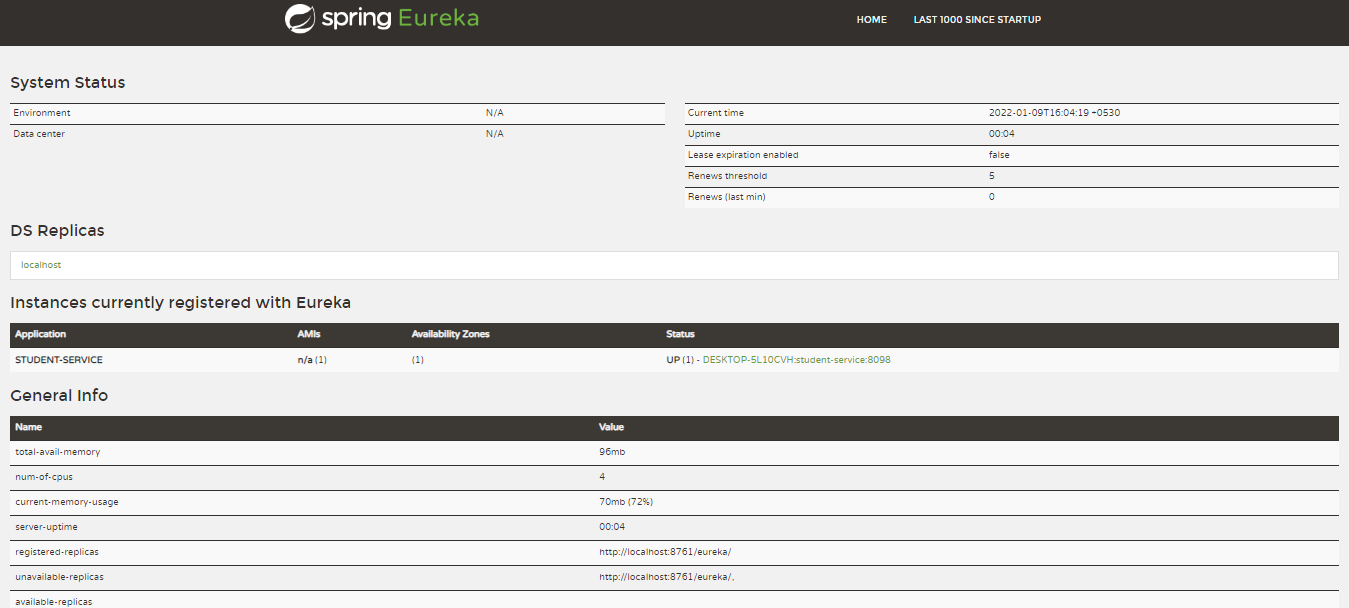


Student class is a simple POJO.



#### **Test Eureka Client**

Start this project as spring boot application. Now verify that this service has been registered in Eureka server automatically. Go to Eureka service console and refresh the page. Now if everything goes well, we will see one entry for ****student-service**** in the eureka service console. This indicates that both Eureka server and client are aware each other.



We will now verify that the /getStudentDetailsForSchool/{schoolname} endpoint is up and running. Go to browser and go to http://localhost:8098/getStudentDetailsForSchool/abcschool, it will give the Student details for a particular school abcschool.





## **Eureka Client – School Service**

Now we will create school service which will register itself with eureka server – and it will discover and invoke student-service without hardcoded URL path.

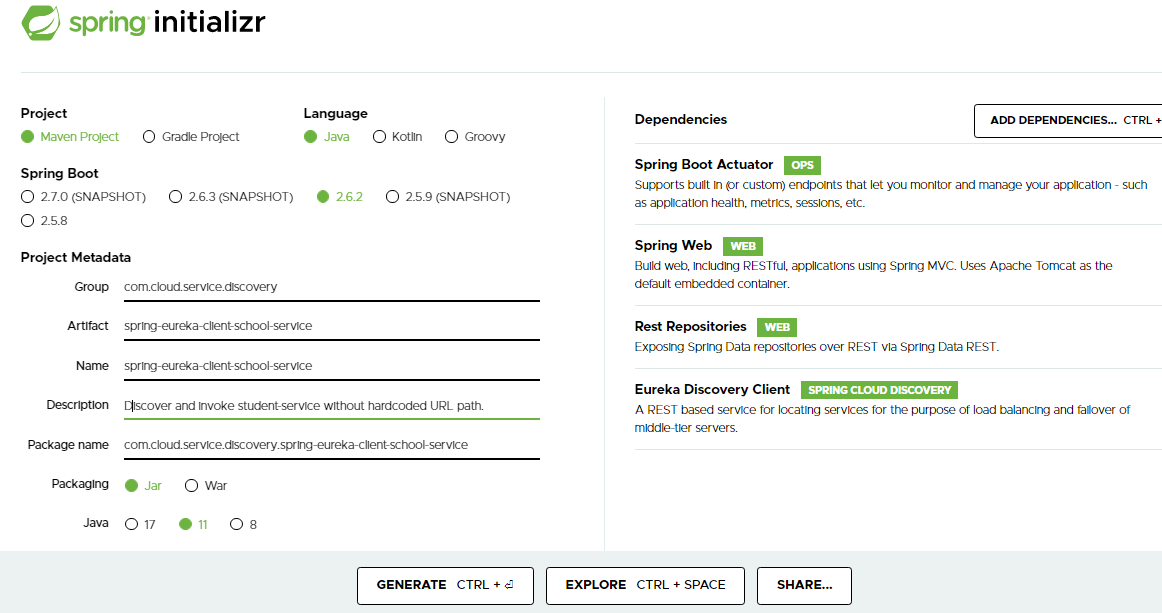
Follow exact steps for creating student service, to create and run Eureka client running school service as well.

#### **Create Eureka Client Project**

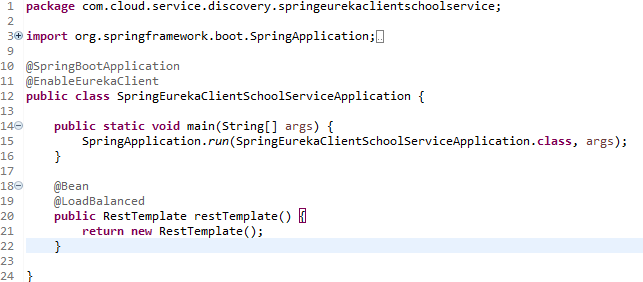
Create a Spring boot project from initializer portal with four dependencies i.e. Actuator, Web, Rest Repositories, Eureka Discovery. Give other maven coordinates and download the project.

Unzip and import the project into Eclipse as existing maven project.

Now add the @EnableEurekaClient annotation on Spring boot application class present in src folder. With this annotation, this artifact will act like a spring discovery client and will register itself in the eureka server attached to this service.



Add a Bean to return the RestTemplate int the main apllication class.



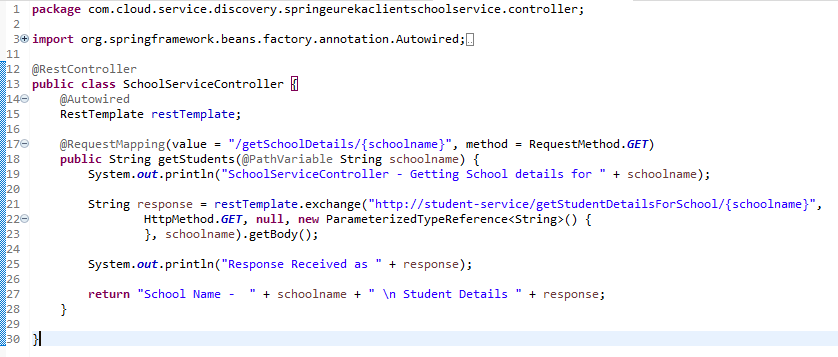
#### **Client Configuration**

Create one file called application.yml in the src\main\resources directory and add below lines. These configurations are very similar to student service *except port number and service name*.



#### **Add REST API which consume student service’s REST API**

Now add one RestController and expose one rest endpoint for getting school details. This endpoint will use the service discovery style URL using the application name, instead full URL with host:port.

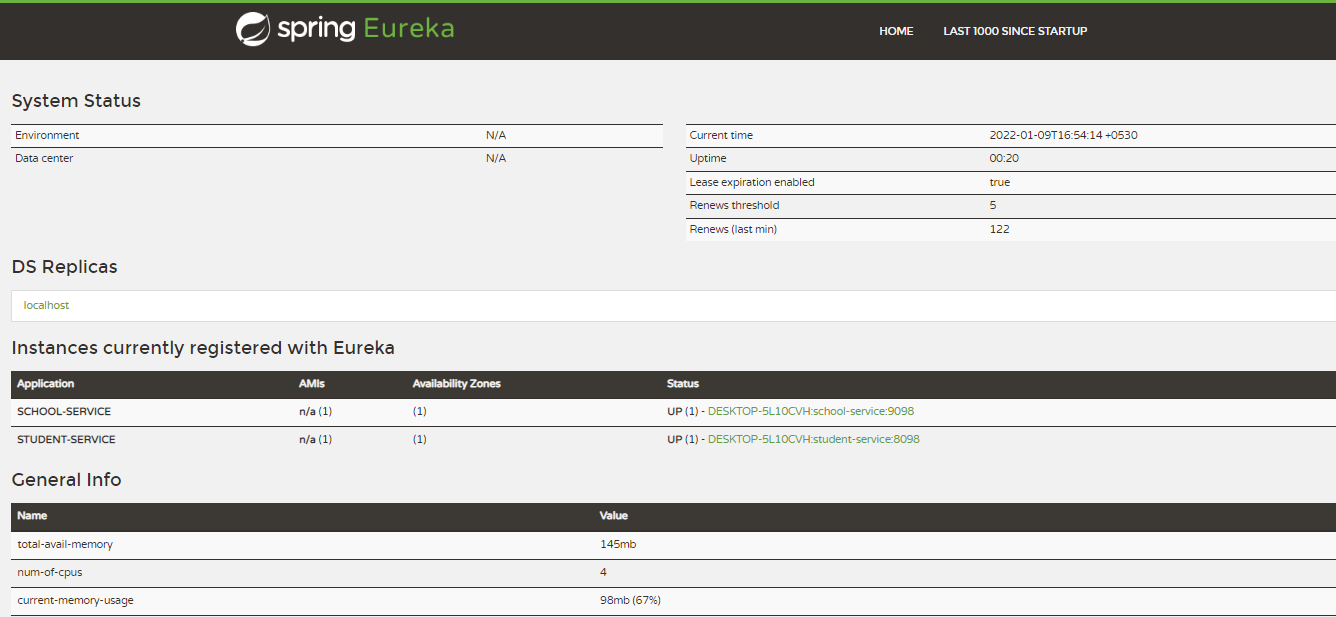


This way we can get rid of specific service configuration and we can give the service look up responsibility to eureka server and rest template provided here. We can also apply load balancing (see @LoadBalanced annotation) here if the multiple instances are running for the same service.

The URL we have used is http://student-service/getStudentDetailsForSchool/{schoolname}. Clearly we are using only service name student-service in the place of host:port. This will be handled internally by spring framework, eureka server and rest template together.

## **Test of Service Discovery and Clients**

Now start the school service as well. All three services are started. Check the eureka server console. Both the student and school services must be registered there.



Go to browser and go to http://localhost:9098/getSchoolDetails/abcschool, it will give the school details for a particular school *abcschool* details. We have ****invoked student service internally****. The response will look like this in the browser:



## **Things to check if facing any error**

1. Annotations @EnableEurekaServer and @EnableEurekaClient are the heart of the application ecosystem. Without those two, things will not work at all.
2. Make sure at the time of starting the config client service, eureka server service is running already, otherwise it might take some time to register, which might create confusion while testing.

## **Summary**

We saw how easily one can deploy service registry and discovery server as well as clients efficiently. Spring framework is maintaining lots of things internally. Here we are just using couple of annotations and very minimal configuration to achieve the whole thing quickly.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*