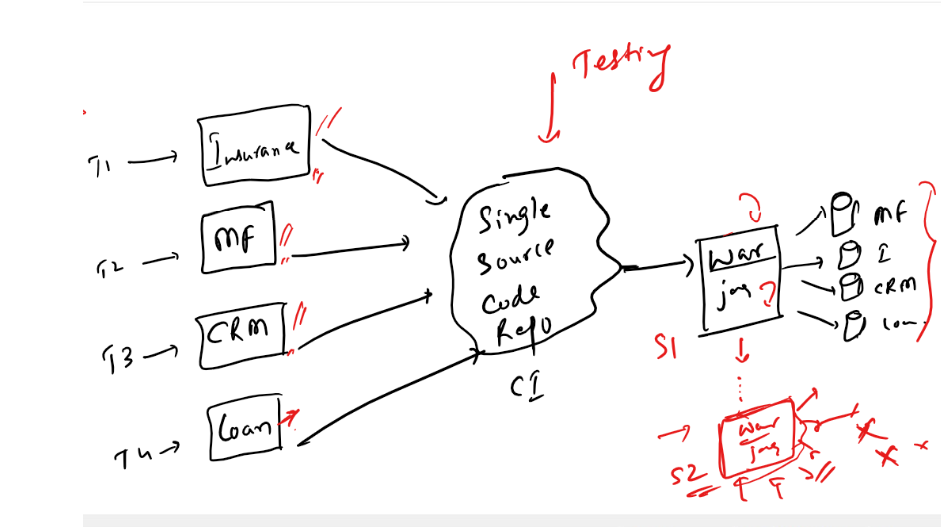
Spring Microservices

Microservices are loosely coupled services that are independent from other services of either same or different applications.

Monolithic architecture

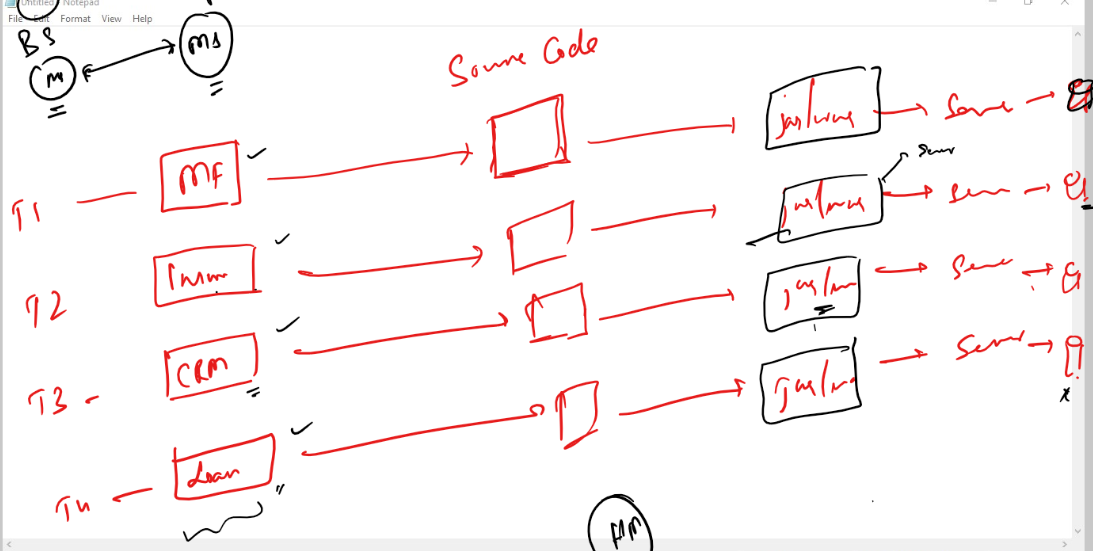


In Monolithic architecture, entire application is divided into multiple modules & each team would work on separate modules & once they integrate all the modules, the testing team had to test all the modules, if any changes happen in any one of the module, the testing team had to retest the entire application even the modules that are not modified should be tested, because to ensure there wouldn’t be problem in the production, once its tested it will be built and deployed as single artifact as jar or war, the application may connect to multiple databases.

Limitations:

1. Testing all the modules even if they are not changed when the requirement need some changes in the application
2. Scaling a particular service is not possible, you have to scale entire application, this would probably cost the company as the server would costly when the resources increase
3. Handling the failures would make the application less reliable, because if any one of the service goes down there could be possible that entire application might go down, it means all the services would stop working.
4. All the modules must be implemented in the same technology as they are at the end built into single artifact.

These issues are very well address in Microservice architecture



Advantages

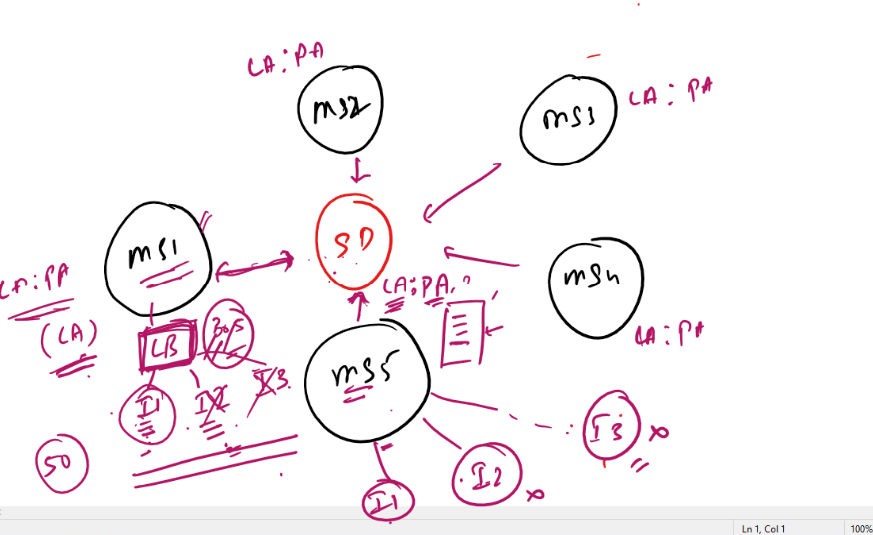
* Here every team can create the services & build separate artifact for each service and deploy in a separate server
* You don’t have to test other services when the changes happen in a particular service as they are deployed in separate servers
* Scaling a particular service instead of entire application is possible
* If any service fail, then issue would only to that service and other service would still work
* You can implement services in different technologies if required.

In Spring you have two projects that allows you to create microservices

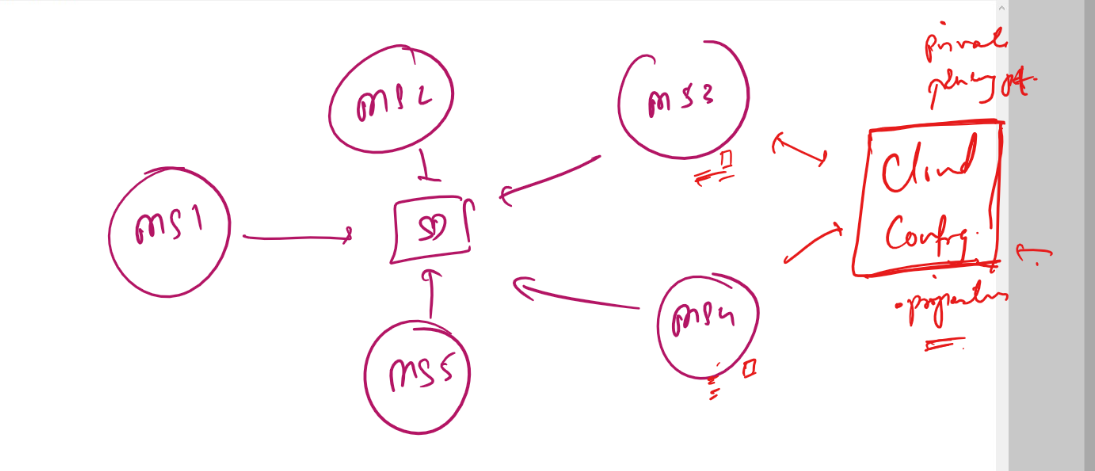
* Spring Boot: To easily create spring applications
* Spring Cloud: It gives all the best practices & design patterns required to create Microserivces easily.

Microservice Components

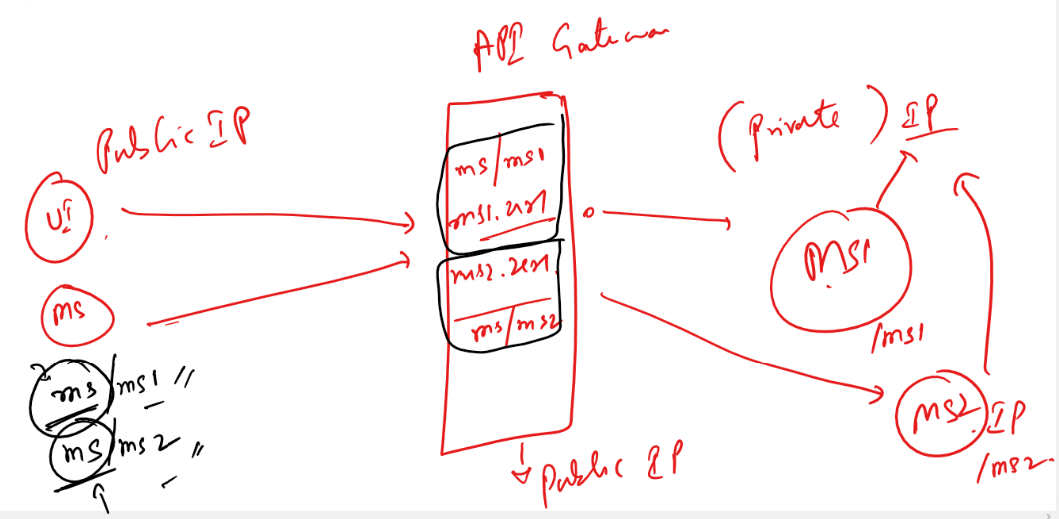
* Service Discovery: It is a registry where the microserivces are registered so that other microservices can locate them
* Discovery Client: These are the microservices which are registered in the service discovery and also they would communicate with other microserivces that are part of same application or different application
* Load Balancer: Takes care of distributing the load across multiple instances of the microserivces



* External Cloud Configuration / Distributed Configuration: Common configuration environment that can be used by multiple microservices, so that at one place the configuration will be maintained



* Circuit Breaker: It avoids cascade of failures when any one of the microservice is down the circuit would break so that the microservice communicating doesn’t send requests to the services that are down, this avoids other microserivces going down.
* Distributed Log Tracing: It is used to log the multiple microservice execution at one place
* API Gateway: Common entry point for the microservice, it is to perform reverse proxy.

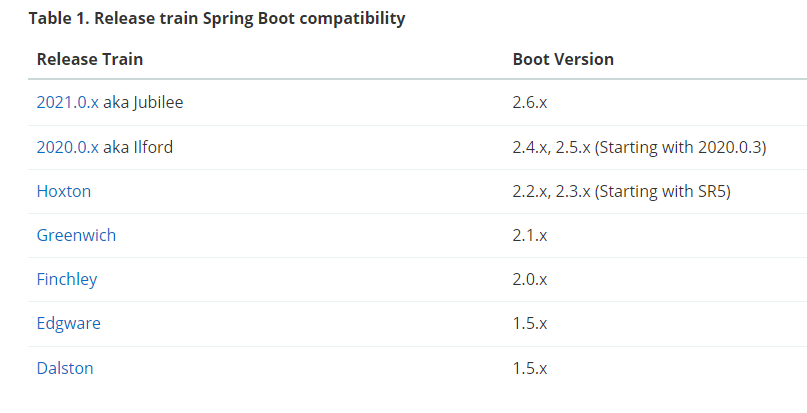


All the above components must be there to create a microservice.

Spring has released two projects to create microserivces

* Spring Boot: To easily develop spring application to use the spring features
* Spring Cloud: It provides all the design patterns required to create microservices, i.e., generic set up for microservices like: Service Discovery, Discovery Client, Load Balancer, Circuit Breaker, Distributed Log Tracing, External Configuration, Security and etc., so that you can easily create microserivces with simple annotations.

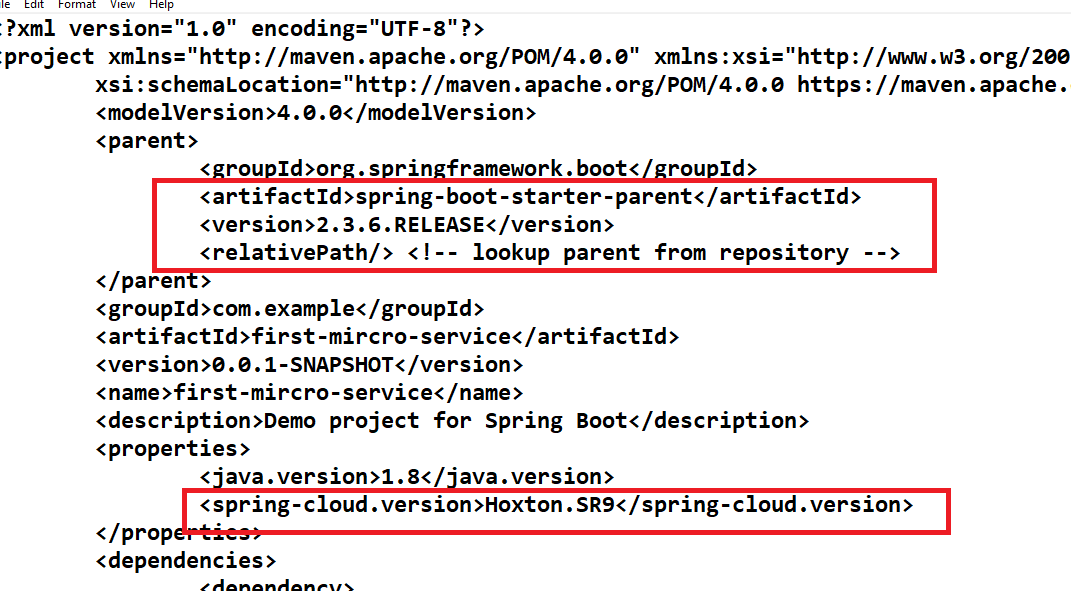
Note: Spring Boot & Spring Cloud are separate projects hence there could be version conflicts which is why spring has provided a release train in their official document, that we need to use to find the compatible version of spring cloud for spring boot.



Here if we want to use Hoxton then the compatible spring boot version must be 2.2.x or 2.3.x starting with SR5, means

**<spring.cloud> Hoxton.SR9</spring.cloud>**

is compatible with   
<parent>   
 <version>2.2.x</version>   
</parent>



The above spring boot parent version is 2.3.6 which means it matches to Hoxton.SR9, we need to manually change this in the pom.xml, because the spring initializr is not showing the earlier spring boot releases.

Creating the service discovery in spring

We have a registry called Eureka Server that acts a service discovery in spring, that registers all the microservice instances and the microserivces can search other microserivces here.

@EnableEurekaServer is the annotation that creates a service discovery, this also acts like a microservice by default it tries to register itself in it, this is the default behaviour of every microservice to register in a service discovery.

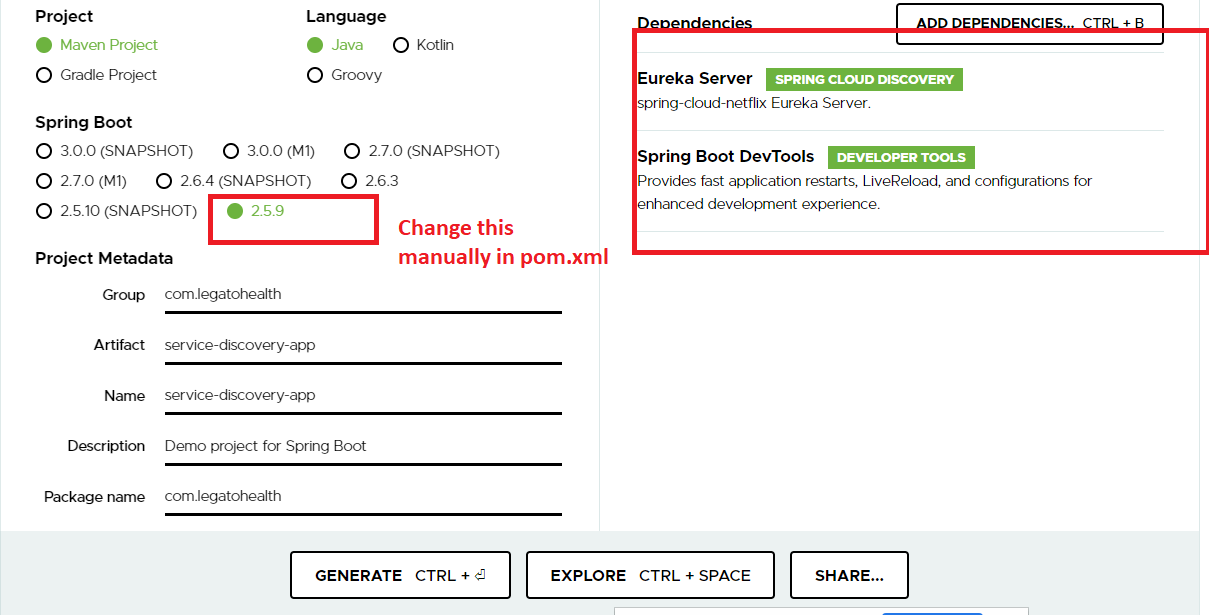
Creating the microservice

You need to create a microservice using @EnableEurekaClient, this by default registers into the eureka server, it must have a logical name as well.

Note: You need to create separate project for service discovery & for microservice

Service Discovery

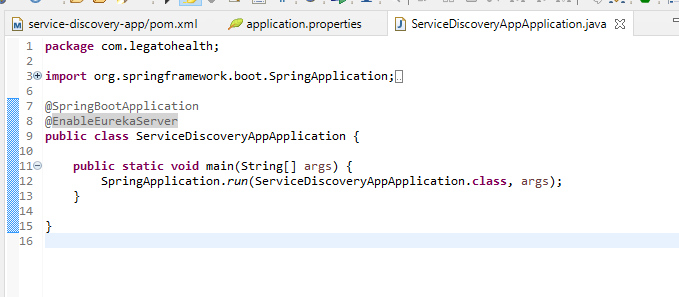
Note: You need to change the spring boot version & spring cloud version manually



Modifying pom.xml to use right spring cloud & spring boot version



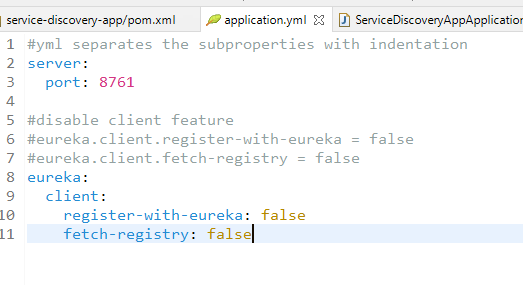
ServiceDiscoveryAppApplication.java



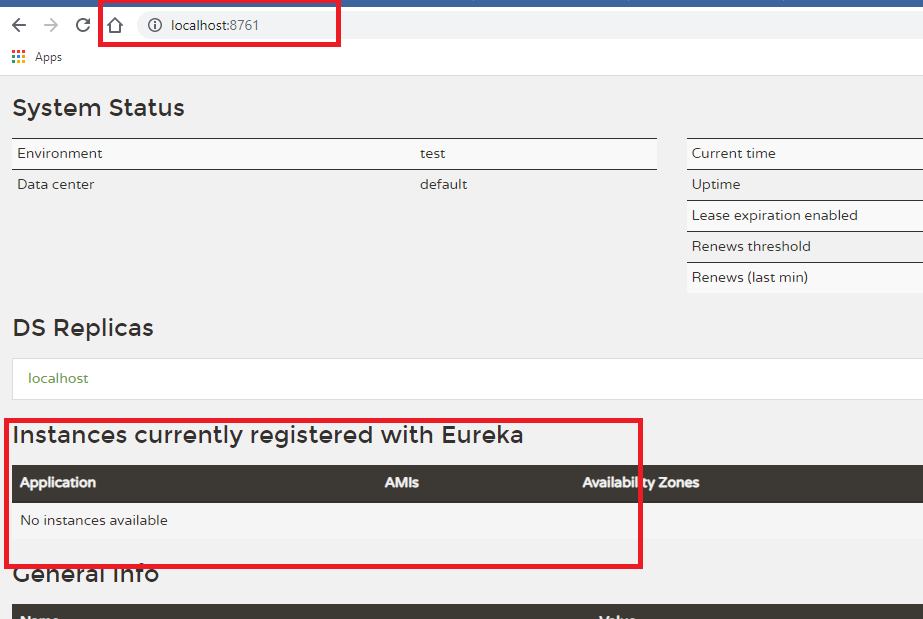
You need to change the port to 8761, because by default all the microservices tries to register in the service discovery by looking service discovery in 8761 port.

You also need to disable the service discovery to register itself as a microservice

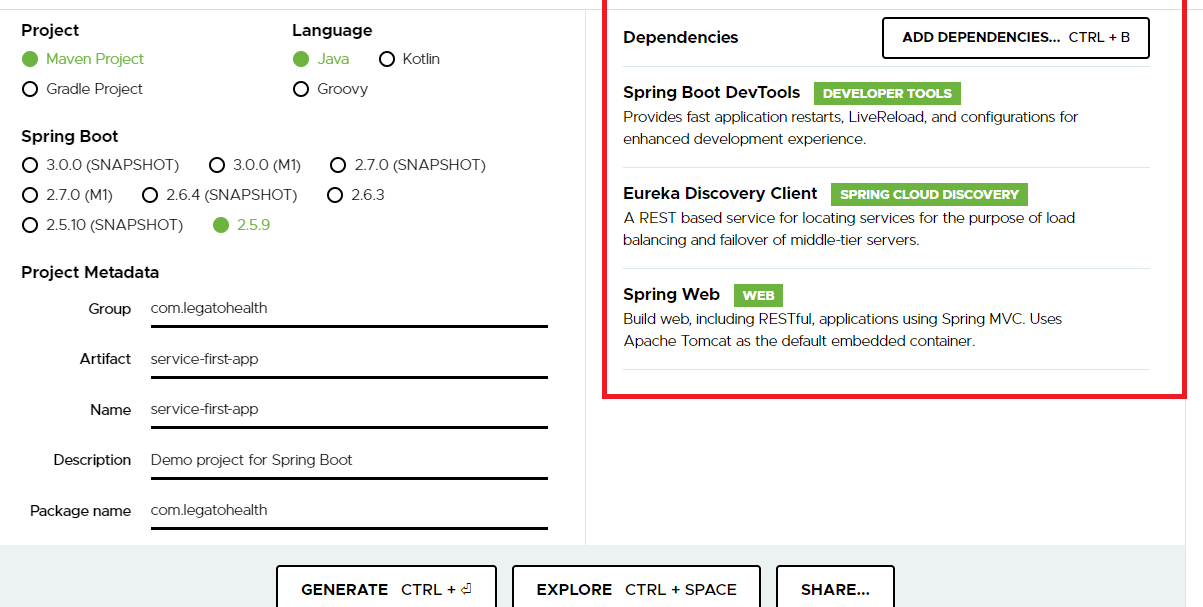
application.yml



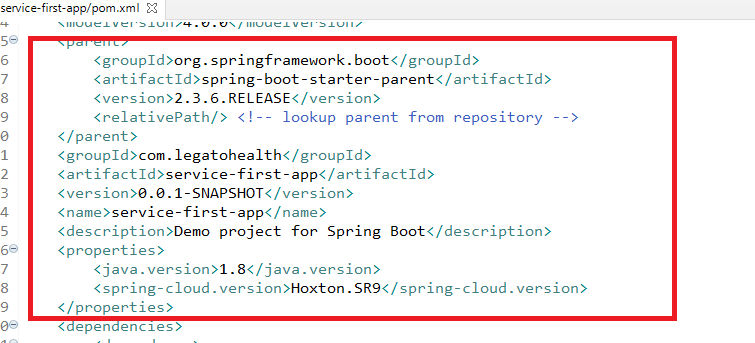
Now you run the application



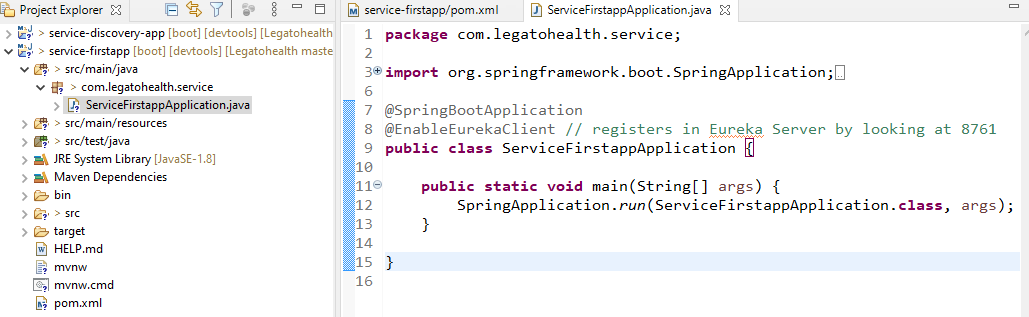
Now we can create a webservice i.e., microservice that can be registered automatically if you use @EnableEurekaClient



Modify pom.xml to use right spring boot & cloud versions

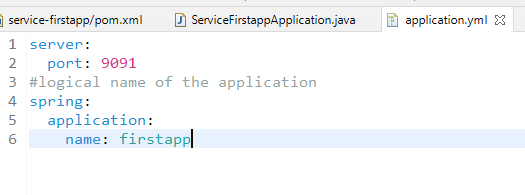


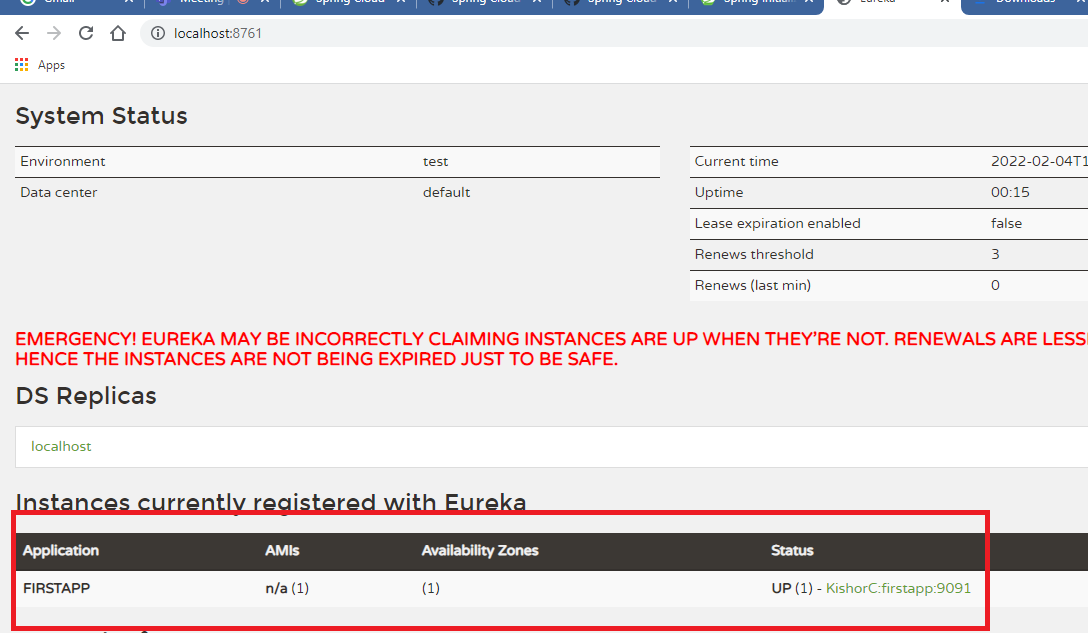
Adding the @EnableEurekaClient to the application



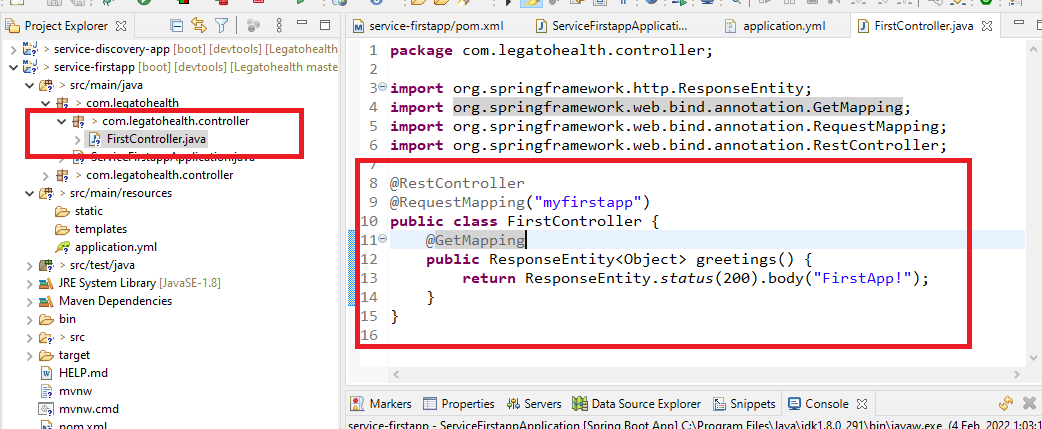
Now this has become microservice, it can have @RestController, @Service, @Component, Database Repository so on.

application.yml



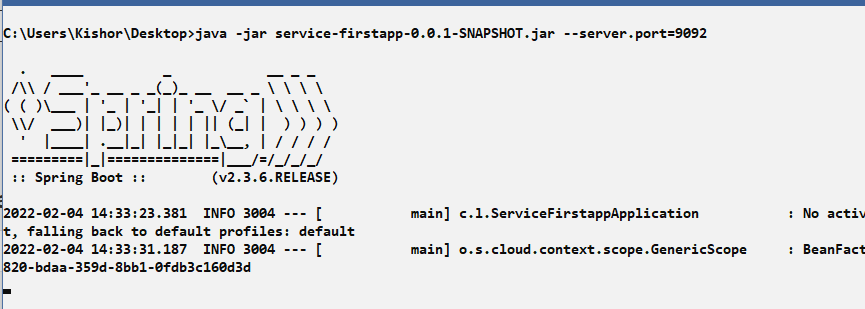


Here the microservice is registered with a logical name FIRSTAPP, but you must have some REST webservices endpoints in this microservice which can be accessed by another microservice.

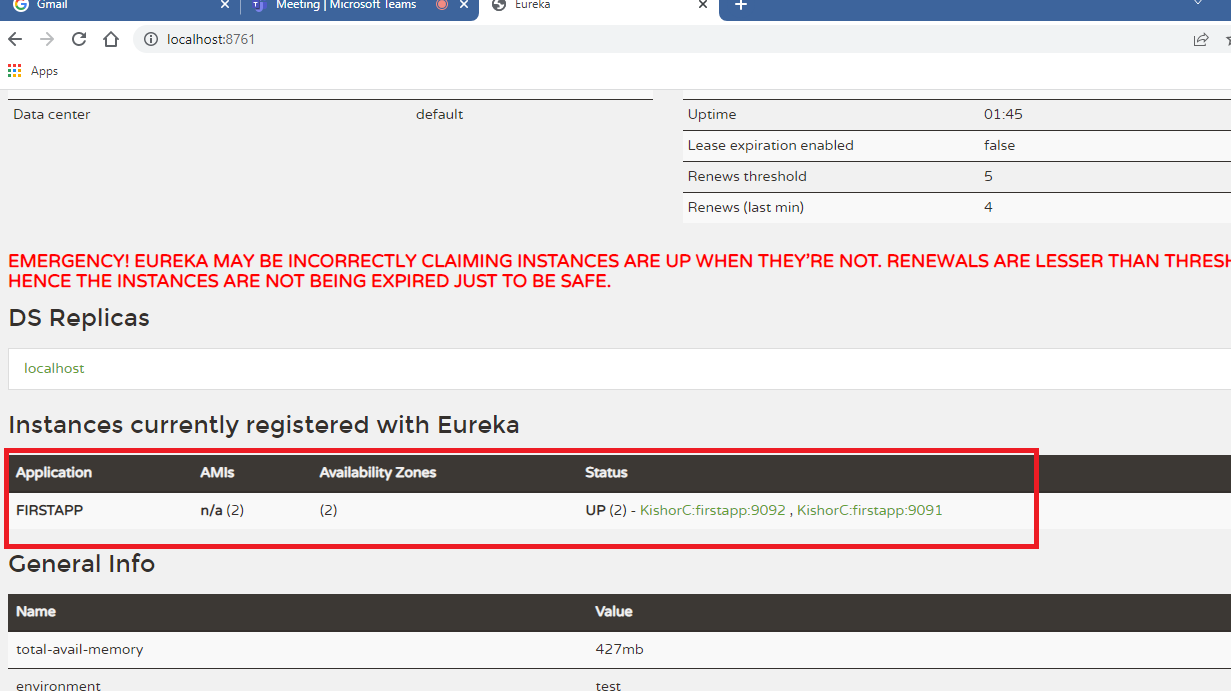


This is a webservice endpoint that can be accessed via path/myfirstapp

The above microservice is having only one instance, but, you can create another instance in the same machine and run in different port, for that you can create an executable jar and run the jar.



You must see 2 instances of same microservice, here the logical name would be same but physical address would be different.



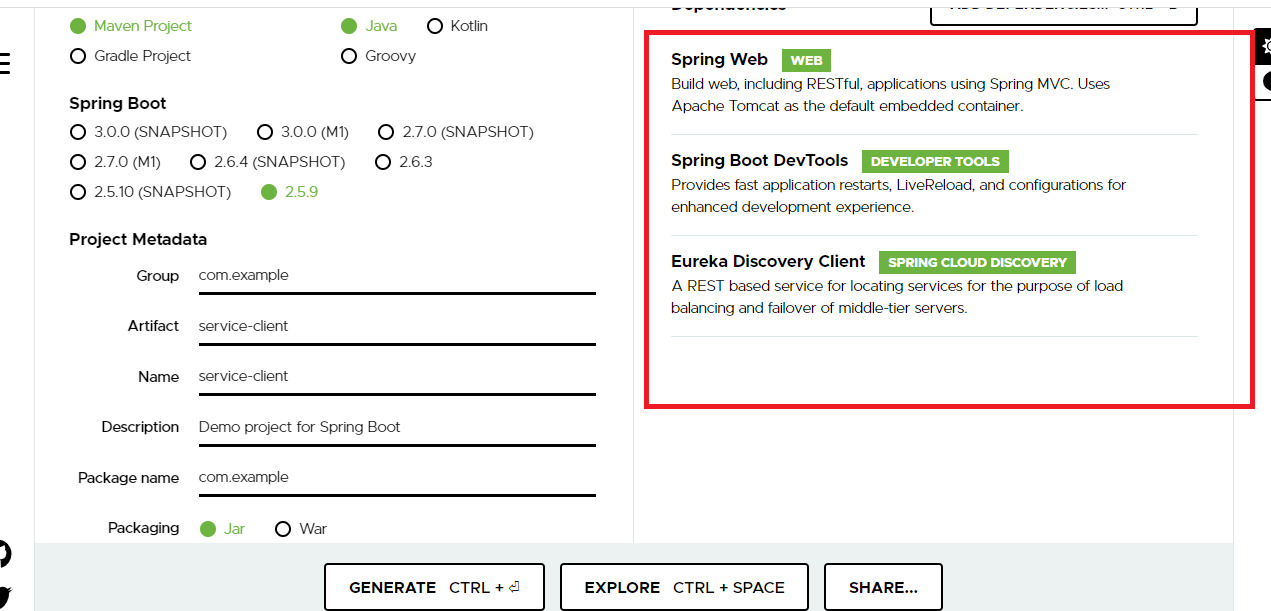
The above microservice has logical name FIRSTAPP which runs in two different address, when the microservices communicate they would use logical name i.e., <http://FIRSTAPP>, the microservice should be registered in the service discovery and access another microservice.

How to communicate with a microservice from another microservice

There are two ways

1. RestTemplate: It is not reusable
2. FeignClient: It is reusable

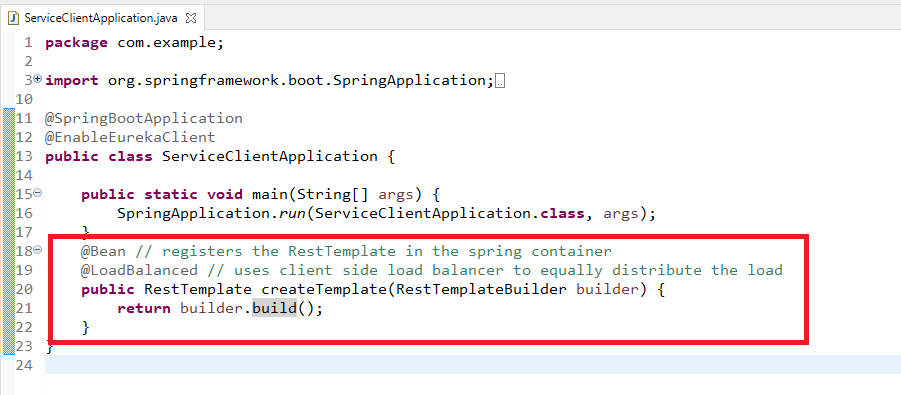
Project to call firstapp from another microservice



Note: Change the spring boot & spring cloud version to be compatible

We must use LoadBalancer at the client side when you are communicating, it must be used using @LoadBalanced on RestTemplate

RestTemplate: It is used to call the webservice/microserivce using spring framework, like HttpClient in angular, axios in React

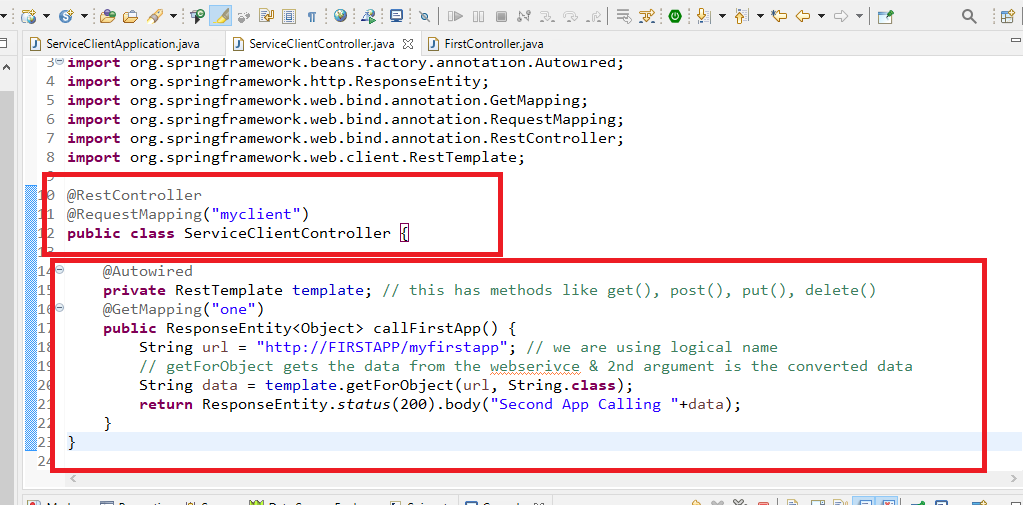


@Bean: Registers the RestTemplate in a spring container so that it can be autowired in different instances which wants to call the webservice/microservice

RestTemplate has methods to call the different http methods in the webservice like getForObject, postForObject, putForObject and so on.

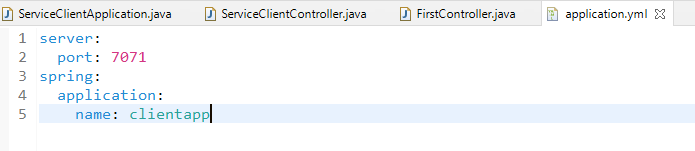
ex: restTemplate.getForObject(url);

Create a webservice that calls the microservice using the logical name

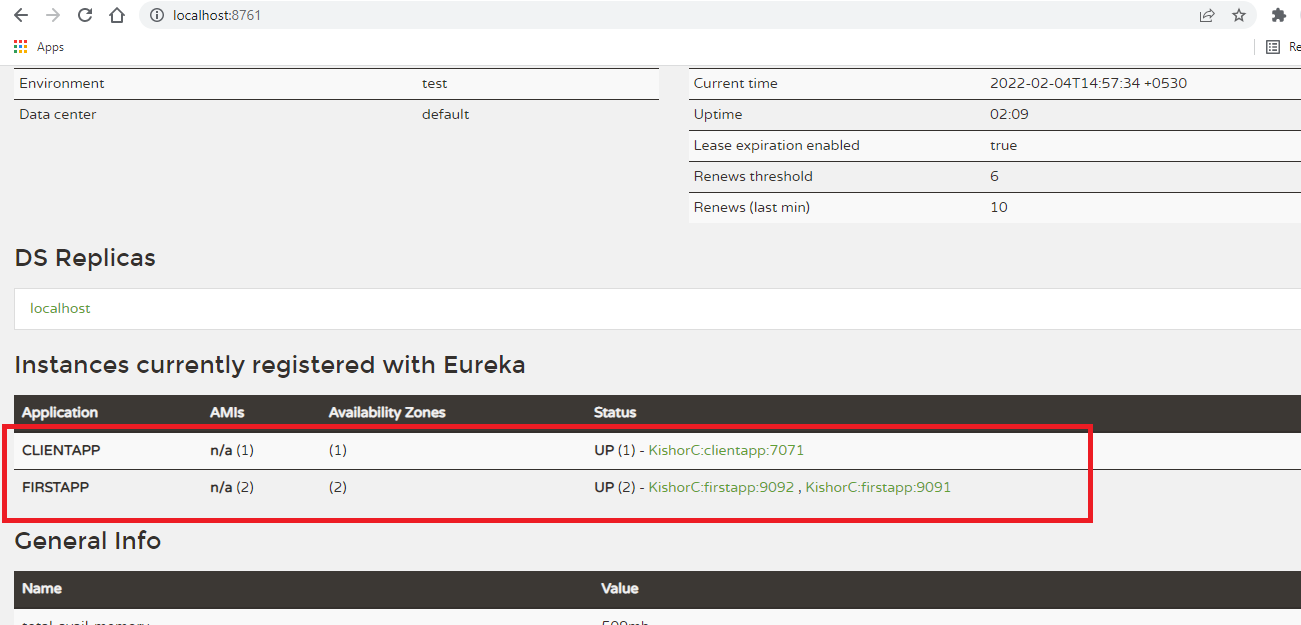


Since RestTemplate is having @LoadBalanced, the load balancer would query the service discovery and finds how many instances of the microservice is there in that logical name

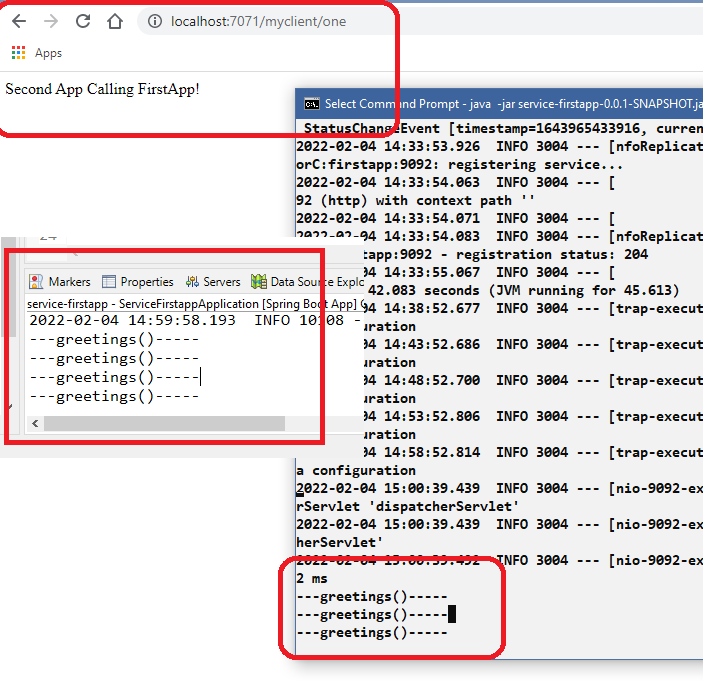
application.yml



Run this application and observe that it is registered in the service discovery



Now send request to the client app, so that it sends request to firstapp i.e., microservice



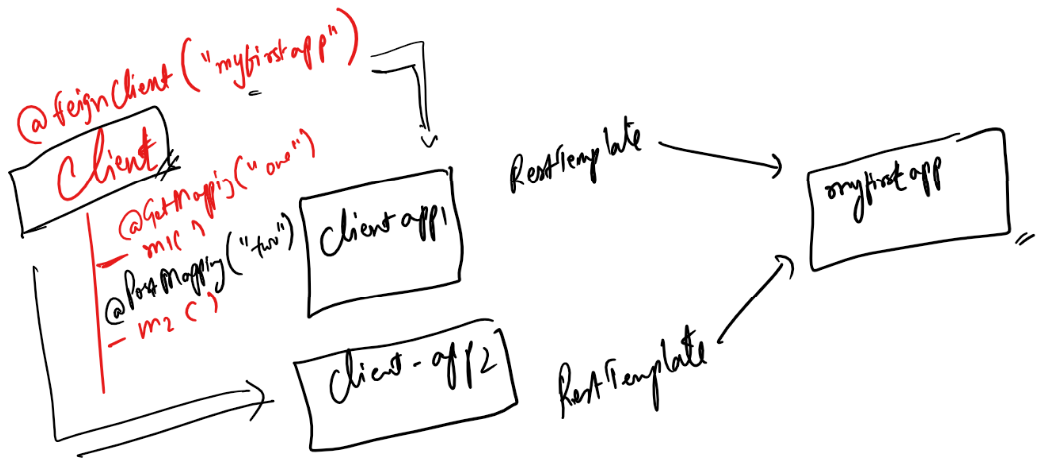
We could see the requests getting distributed in both instances equally, as the logs are printing in eclipse and command prompt.

RestTemplate always uses the URL, so when you want to send requests to multiple endpoints then you need to write these URL’s in multiple places, it is not reusable, because if another client wants to communicate then it must also re-create RestTemplate

i.e., Client1, Client2,… ClientN they all must create RestTempalte using @LoadBalanced

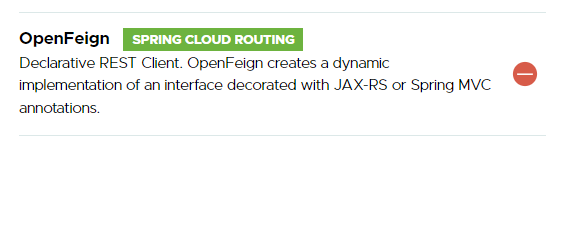
To address this issue we have a reusable methods i.e., FeignClient

FeignClient: It is a client class to invoke a webservice through interfaces, it internally uses loadbalancer, you can create this interface with methods to invoke the webservice and reuse these interfaces in multiple client program, you don’t have to implement this interface, OpenFiegn library does that.



Client is just an interface with @FeginClient annotation, this interface we don’t have to implement, this will be implemented by OpenFeign library, we need to just call the methods of this interface which takes care communicating with appropriate microservice.

Library

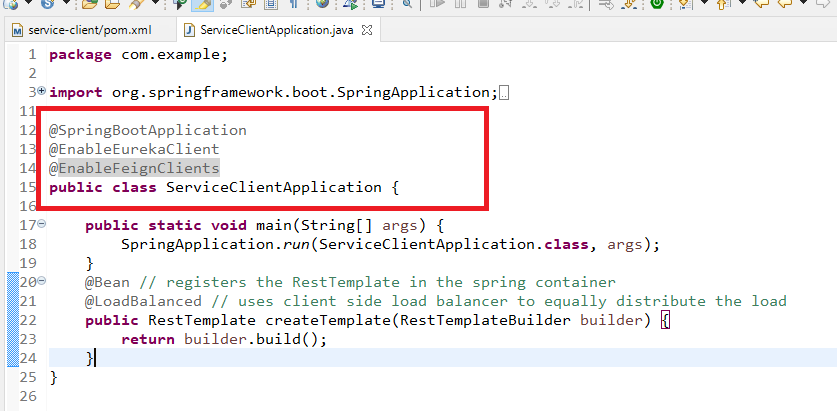


Update the pom.xml of client program that wants to call the microserivce



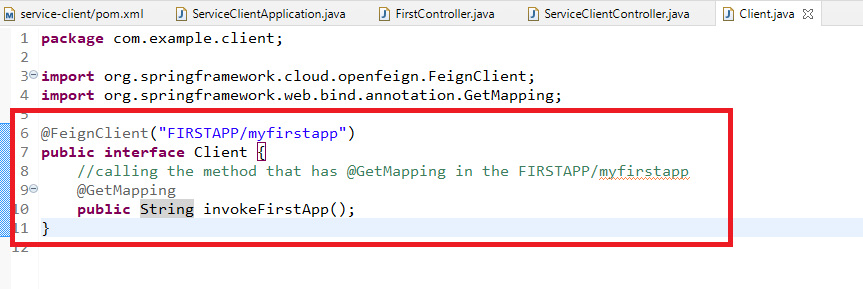
@EnableFeignClients: This implements the interface that has @FeignClient, so that you can inject the instance to the classes that wants to call the microservice

ServiceClientApplication.java



Create the interface that wants to call the <http://FIRSTAPP/myfirstapp>

Client.java

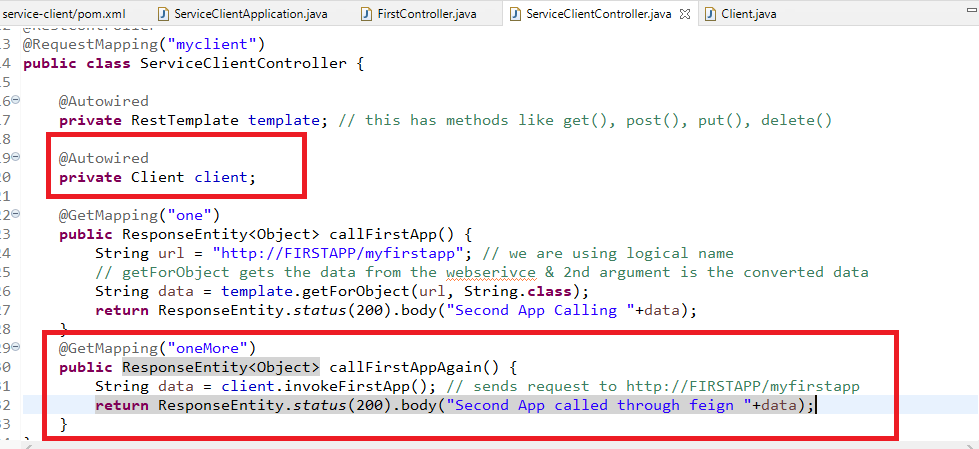


Here you can call invokeFirstApp() that sends request to the FIRSTAPP/myfirstapp, which returns the String content, that content will be the response data the caller would get, here no need of any type conversion like restTemplate.getForObject(url, String.class), is similar to invokeFirstApp() that is mapped to the same URL & returns string.

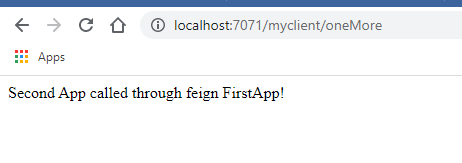
Injecting the client instance,

Note: You don’t have to implement this interface, its taken care by @EnableFiegnClients

ServiceClientController.java



Now if you send request to myclient/oneMore the response from first app should be returned

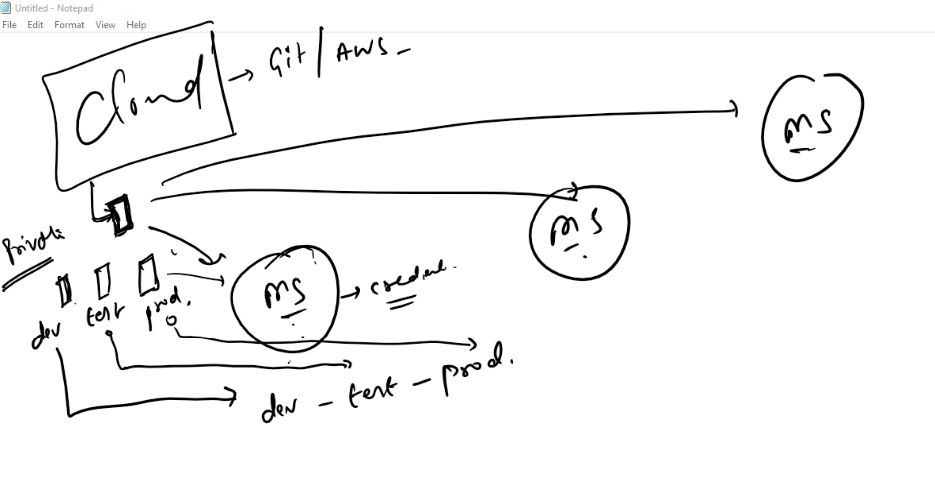


You can also see the output in both the instances

Note: FeignClient is reusable but RestTemplate is not and also FeignClient does the type conversion internally while creating the interface methods.

Note: The default configuration file the spring boot always loads is “application.yml/properties”, but there’s a file ‘bootstrap.yml/properties’ which is loaded before application.yml/properties

Distributed Configuration or Spring Cloud config server



* This allows you to maintain common configurations for multiple microservices
* You can have multiple configuration files for different deployment pipeline like development, testing & production, so that at runtime the microservice can pull the specific configuration file related to the environment its running

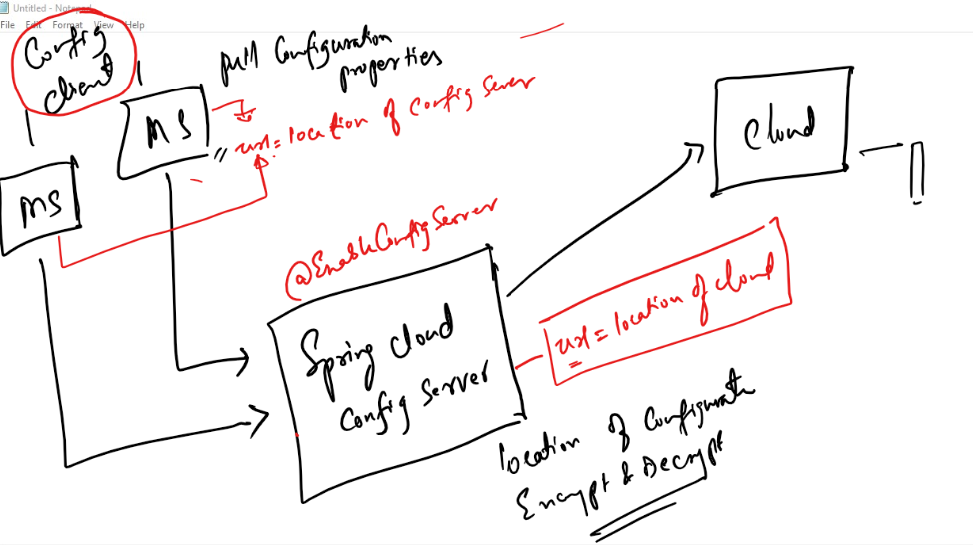
i.e., java -jar file.jar --active.profile=file-name

The above command is usually used to full the configuration file by providing its name at runtime

* You can also privatize the configuration files so that only the microserivces with proper credentials can connect to the cloud to pull the configuration files
* You can also encrypt the configuration file data and make microservice decrypt the configuration file data

All the above features are provided in spring cloud config server, it is a library which allows you to configure the application properties in cloud (GIT/AWS/SVN).

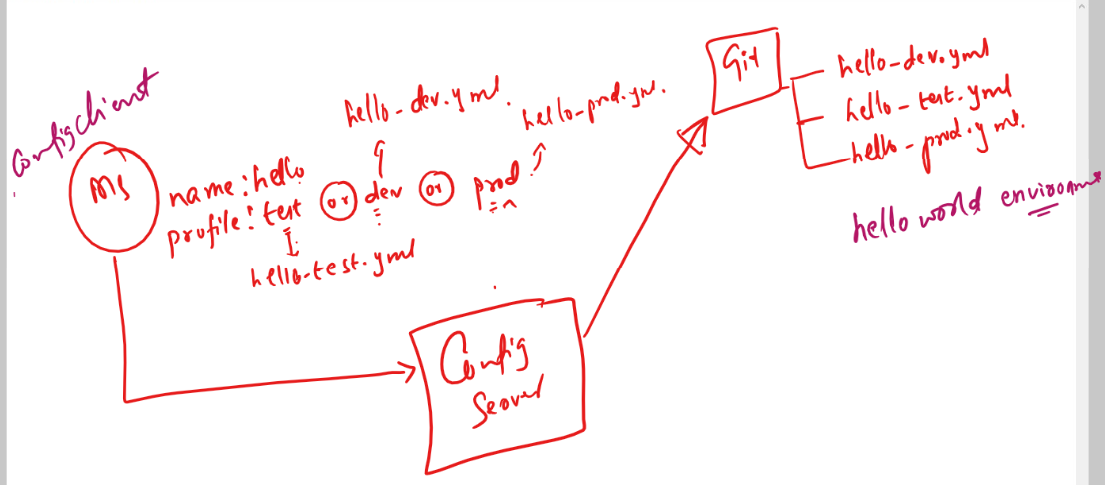
How does this spring cloud configuration work



Spring Cloud Config Server is a program that can connect to any cloud environment like GIT, SVN, AWS, Azure and so on.

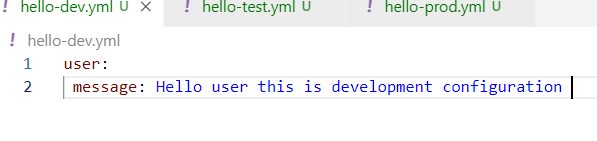
@EnableConfigServer is the annotation spring cloud configuration server program must use to connect to the cloud, here the program must have the url of the cloud

Config Client: These are the client programs could be the microservices who connects to config server not the actual cloud location, they will have the url of the config server, once they connect to the configuration server they pull the configuration files via spring cloud config server, client specify the configuration file in their property file.

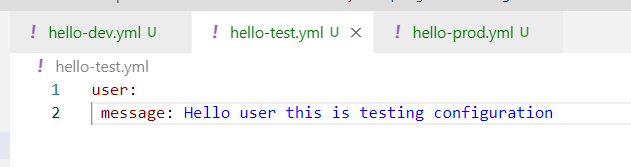


In Git there are 3 files named hello-dev.yml, hello-test.yml, hello-prod.yml that has a simple property with hello world message:

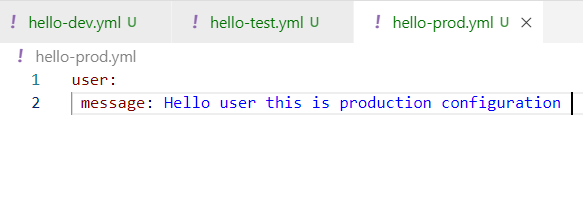
hello-dev.yml



hello-test.yml

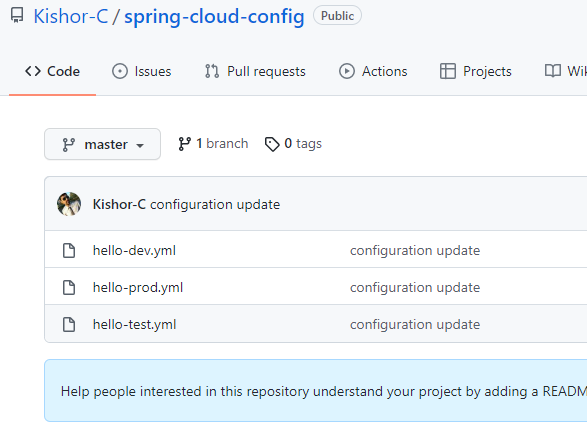


hello-prod.yml

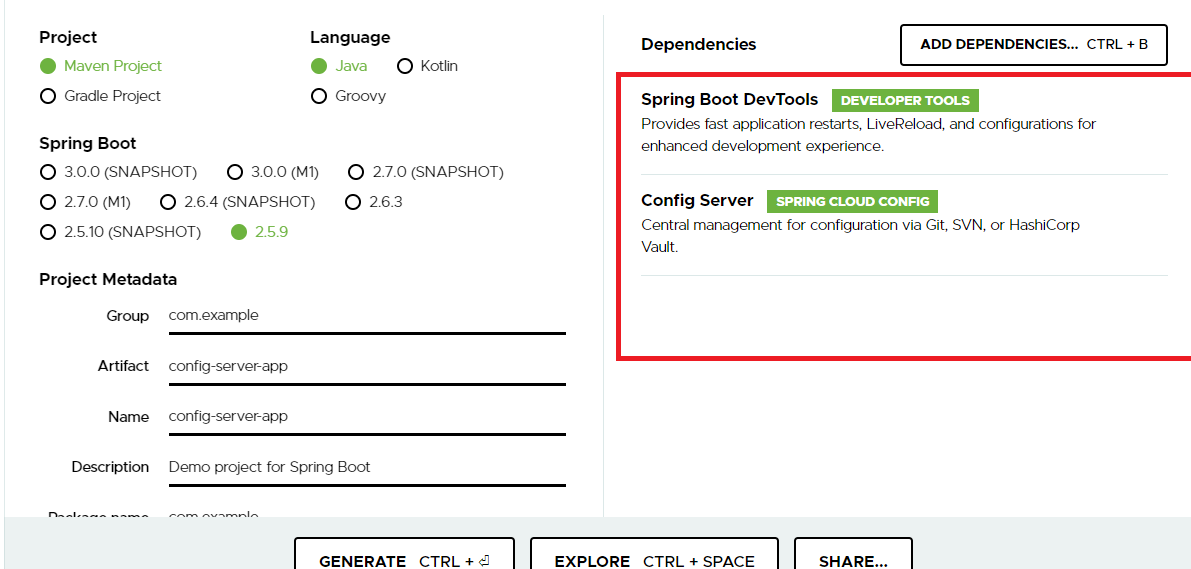


Now these 3 files are available in the GIT the url of this GIT is: <https://github.com/Kishor-C/spring-cloud-config.git>

The above url must be used config server

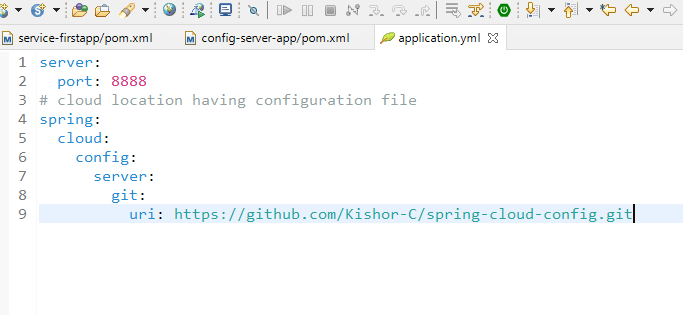


Creating a project for configuration server that connects to the GIT



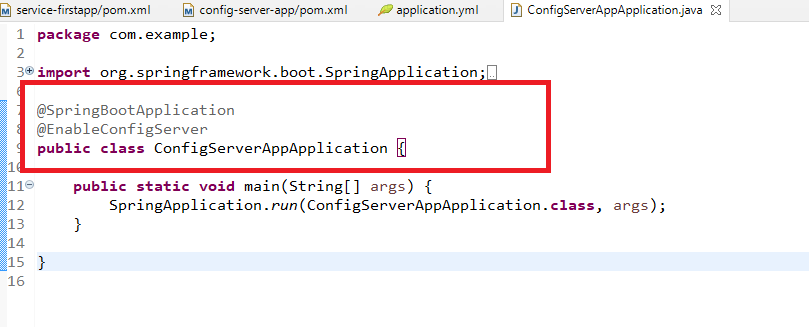
Note: Modify the spring cloud & spring boot versions

application.yml



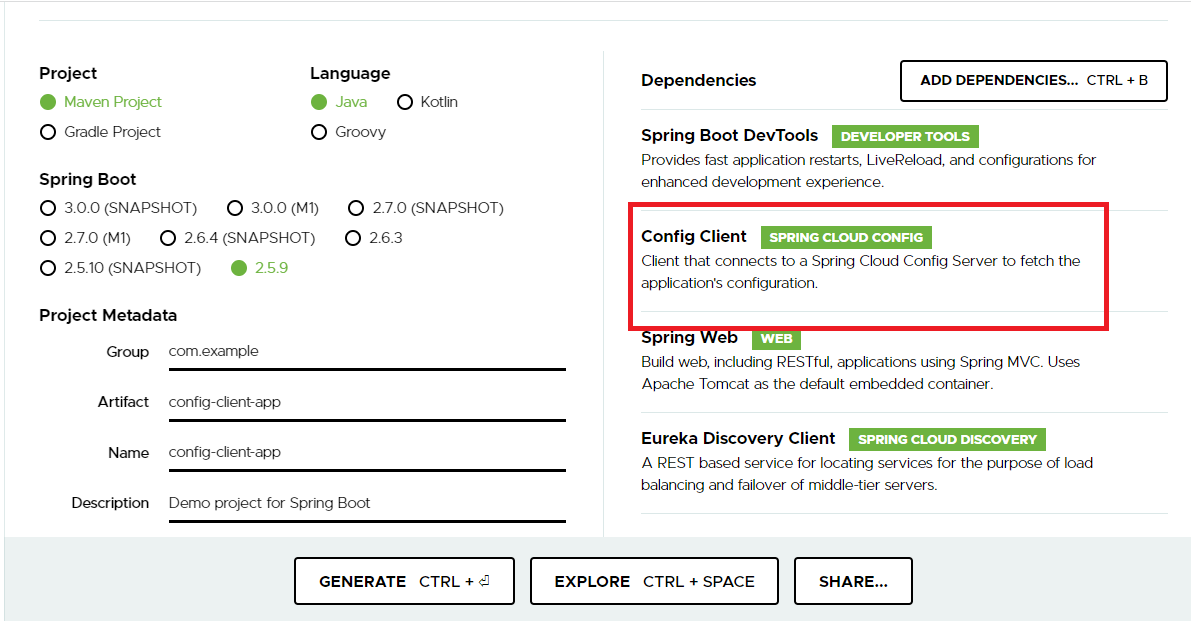
Here the spring boot application doesn’t still know that it must connect to this cloud location, hence we must use @EnableConfigServer

ConfigServerAppApplication.java



Launch the application so that it can connect to git/cloud, then create a microservice that connects to this application, to connect you must use the config server uri in the microserivce: <http://localhsot:8888/>

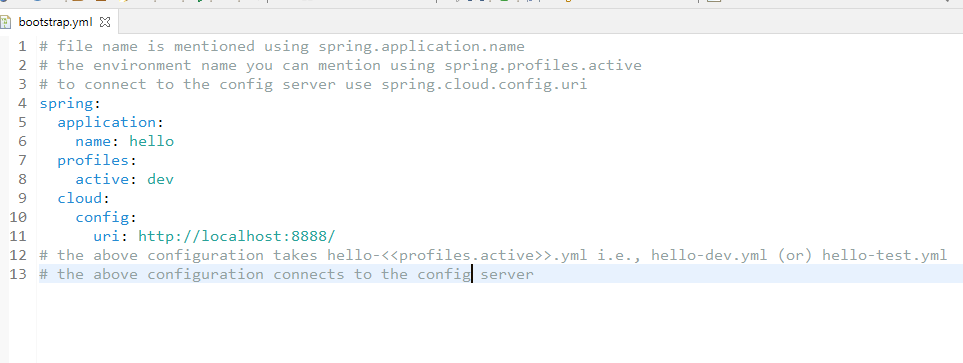
Config Client Project



Note: Modify the spring cloud & boot versions.

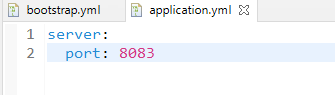
Since the configuration file must be loaded in this microservice from the cloud config server, we must mention the configuration file name before application.yml/properties loads, hence we mention that name in the bootstrap.yml/properties.

bootstrap.yml

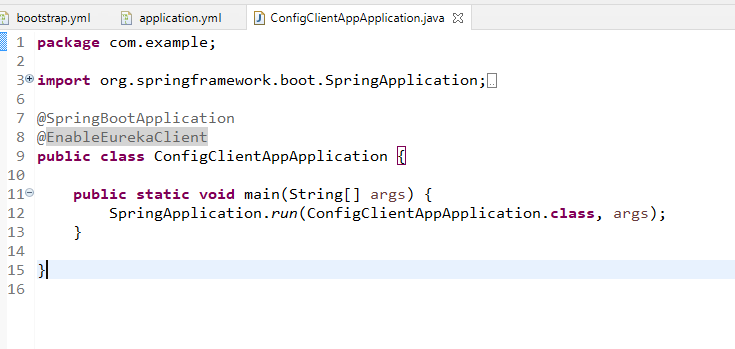


Now you can pull the configuration file that has user.message from 3 different files by just changing profiles.active

application.yml

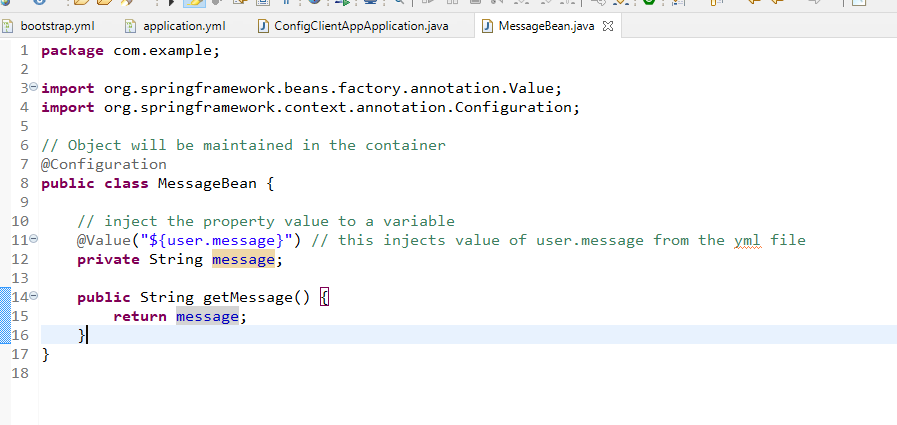


ConfigClientAppApplication.java



Now you can pull the property files and also you can read the property and inject to the variable using @Value(${propertyName})

MessageBean.java

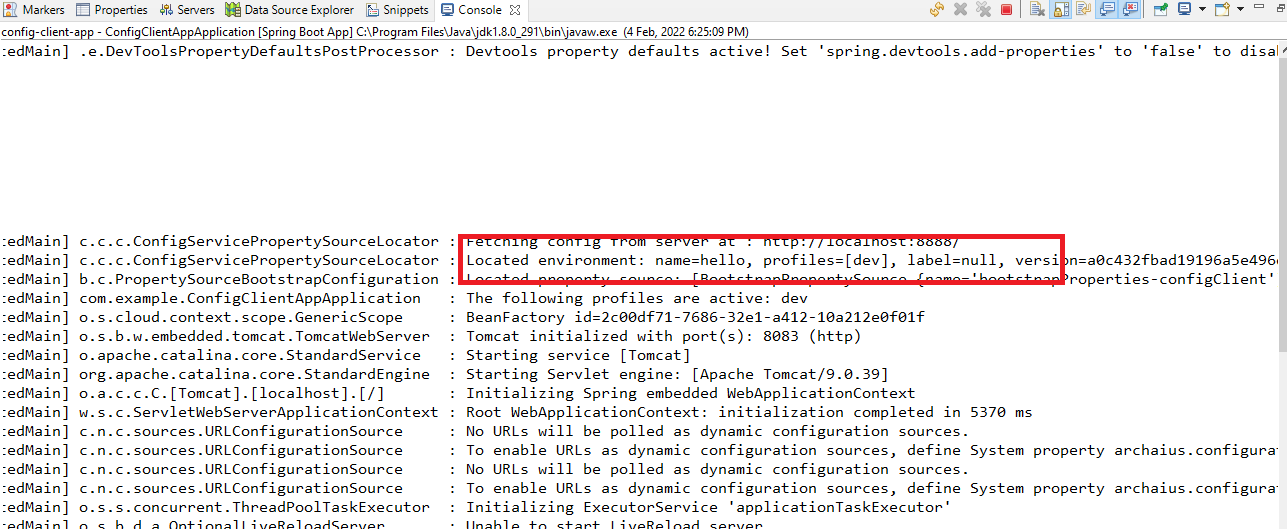


Now you can create a rest controller to see the message

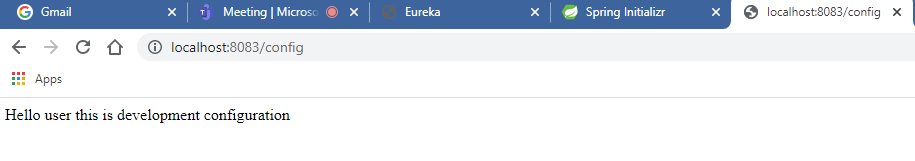
ConfigController.java



Now you can run this application which picks the configuration file from the cloud, which you can see in the eclipse console.

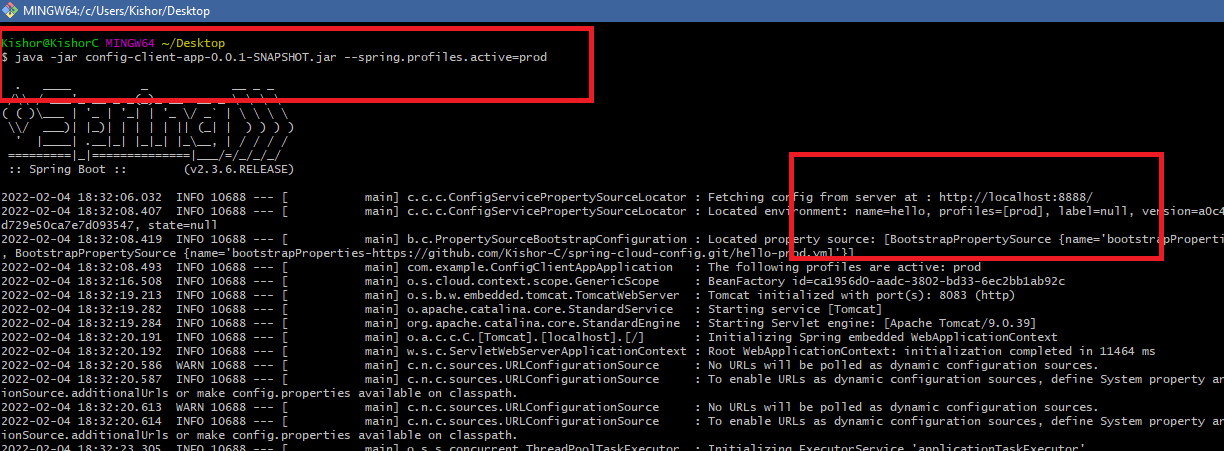


Now call the webservice using the url to see the development configuration file message

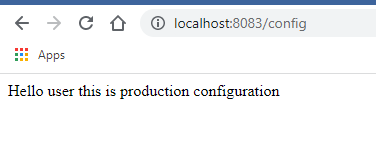


If you want to connect to database you can have those information in the GIT and your microservice can pull your configuration, but the microservice must have spring data jpa, mysql connector libraries.

Note: You can run this application using jar file so that you can pull the configuration at runtime while launching



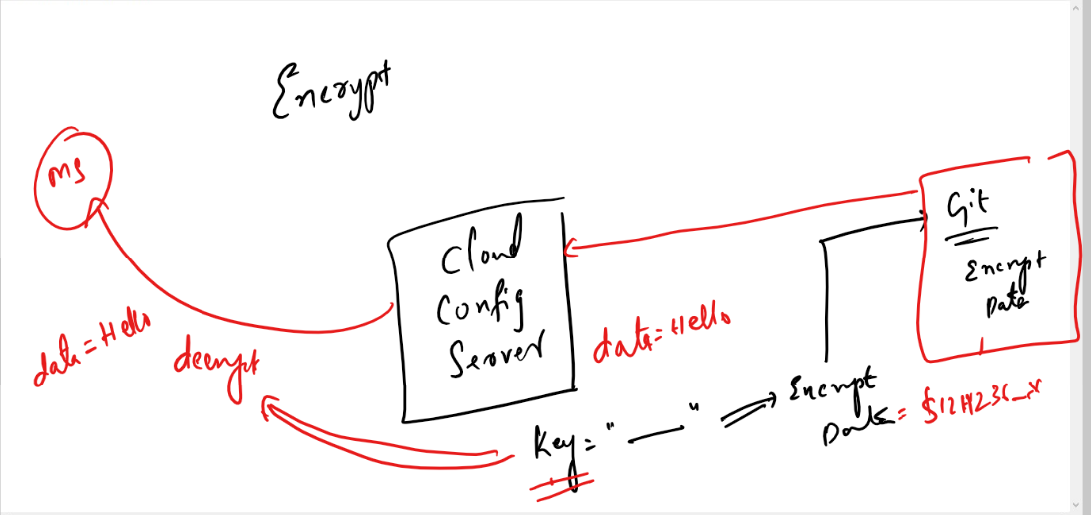
Output:



Activity:

* Configuration the microservice to pull the datasource informations from the cloud config server
* Ensure there are spring data jpa & mysql connector library.
* Test some crud operations

Encryption & Decryption

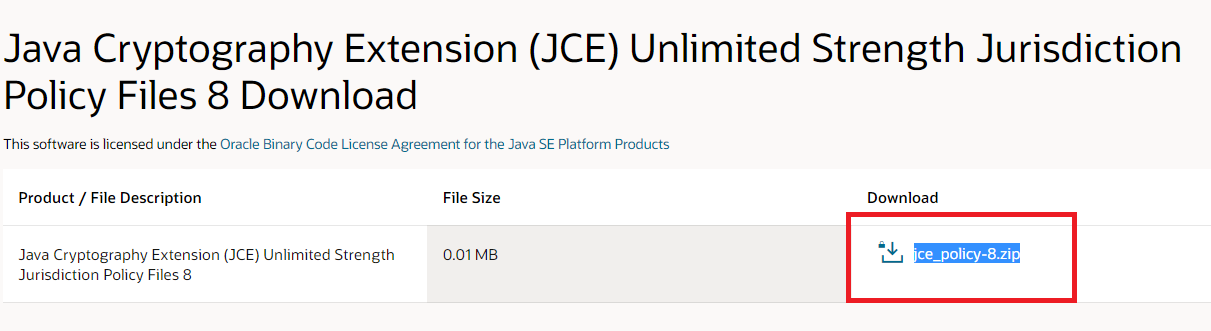


Instead of storing the plain text in the cloud environment you can encrypt the data and keep that encrypted data in the cloud.

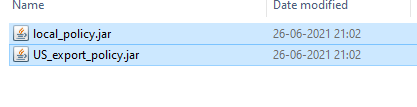
How to encrypt:

Java gives some security library using which you can encrypt & decrypt the data, if in case that library is not available you can download from the internet.

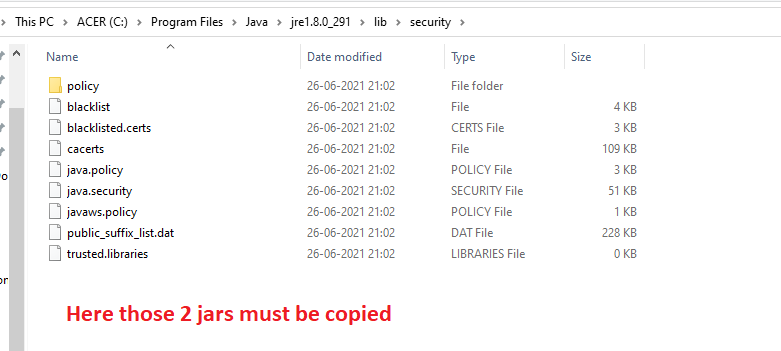
<https://www.oracle.com/in/java/technologies/javase-jce8-downloads.html>

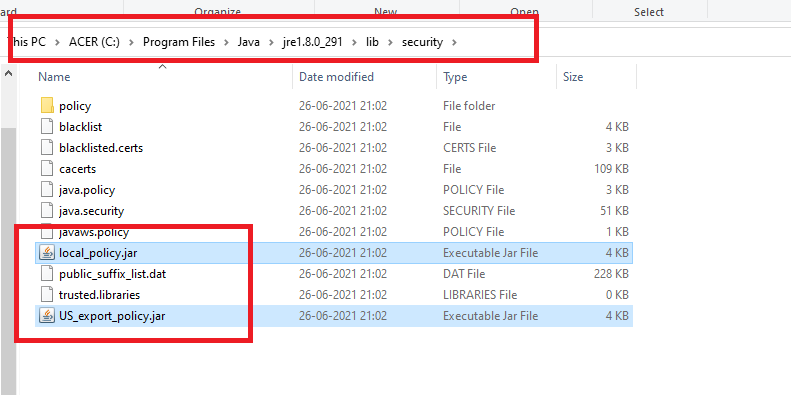


You need to download this zip & unzip it, which will have 2 jars as below



You need to copy these two jars to your JRE\_PATH/lib/security folder





Since your application is using JRE as a execution environment, encryption & decryption works and also in production, these 2 jars must be copied to this location

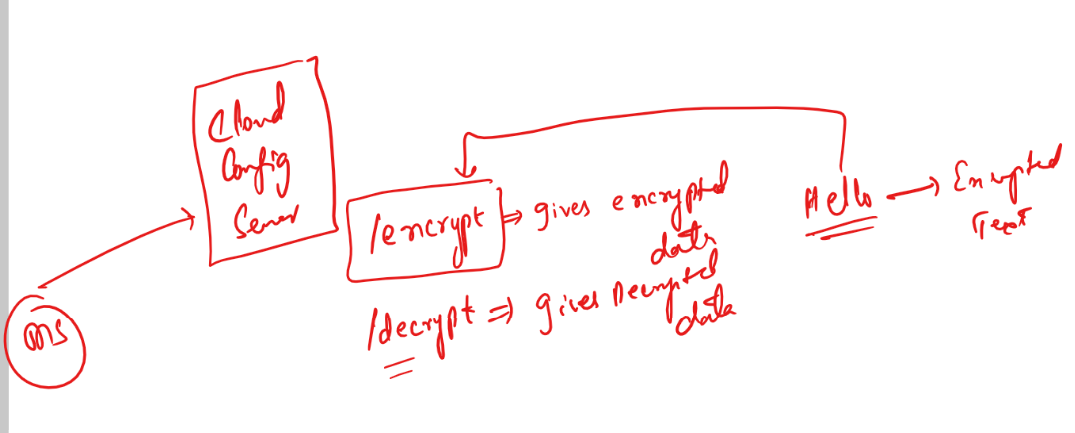
Who takes care of encryption & decryption

Config Server program takes care of encrypting & decrypting the data by using some key present in it, when it encrypts any data it uses the key & encrypts, if the data need to be decrypted the same key must be used, this data can be encrypted using an endpoint config-server-url/encrypt and config server uses another url that will have decrypt to decrypt

Encrypt URI: config-server-uri/encrypt

Decrypt URI: config-server-uri/decrypt

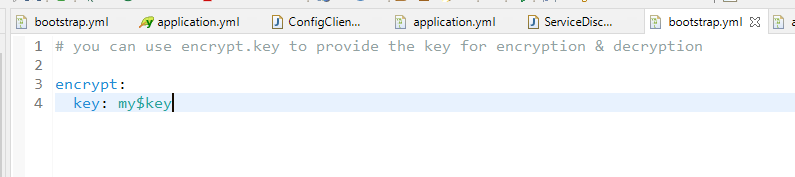
To encrypt we must use the encrypt url, however decrypt url is used by configuration server automatically when it needs to pull the configuration to the microservices.



Key Management:

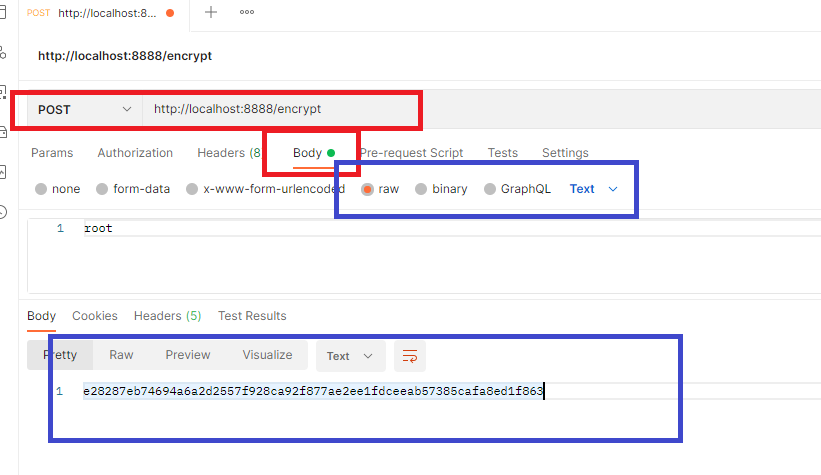
This is a key that must be used to perform encryption & decryption, it is a property for the application which can be mention either in the property/yml file or through command line also while running the jar

bootstrap.yml of config server



Now you can use the /encrypt end point & HTTP post method from the config server to encrypt the data you want, and config server can decrypt the same encrypted data, in both the cases it uses encrypt.key property

Assume you have a password in the hello-dev.yml, then you can encrypt it as below



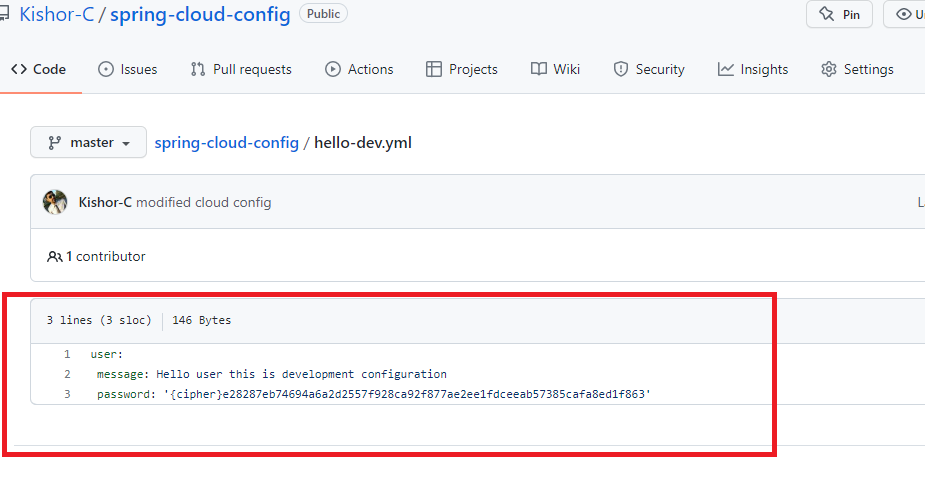
If root is the password then the encrypted data you can keep in the yml file, but it must be kept in a format like {cipher}encyptedText, this is just like a keyword that says the text comes after {cipher} is encrypted so that the config server can decrypt.

hello-dev.yml



Note: keep the encrypted {cipher} in the quotes

Push this to the git and read the password in the microservice it must get the correct data i.e., ‘root’

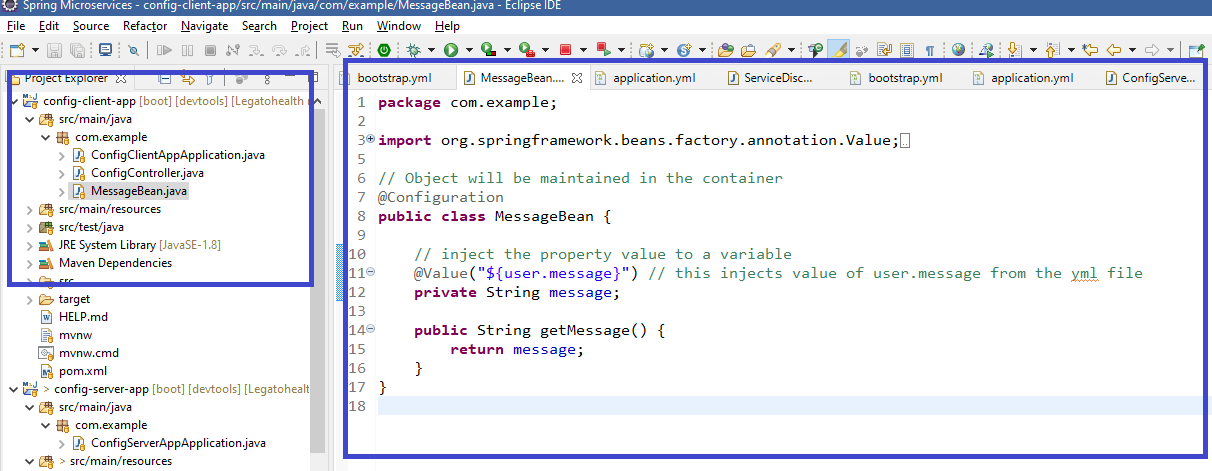


Note: Currently we have password only in hello-dev.yml, but we can also keep different encrypted passwords in other files like hello-test.yml & hello-prod.yml.

So now we need to pull hello-dev.yml and must able get the decrypted form of the password that would be ‘root’

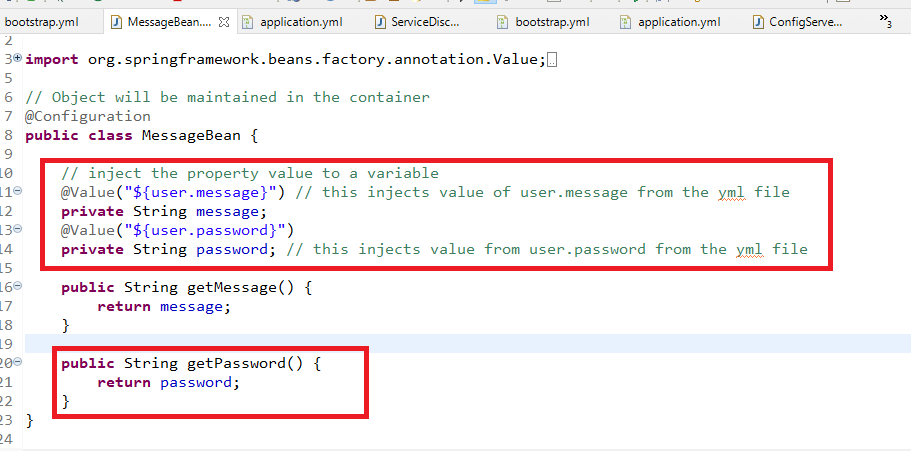
Note: Microservice doesn’t need to encrypt, because decryption is taken care by config server because of secret.key

The Microservice which connects to the configuration server must read password, hence modify the MessageBean class to have the password



MessageBean is in a config client program hence modify the above class

MessageBean.java



Modify the same project controller to confirm the password is decrypted, test in the postman

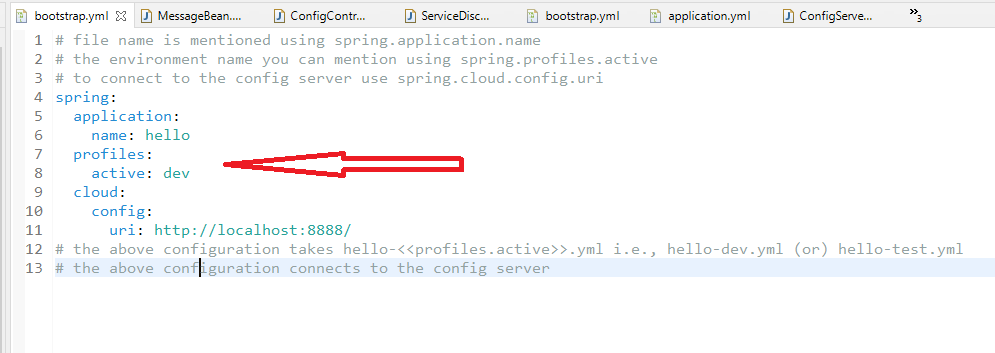
ConfigController.java



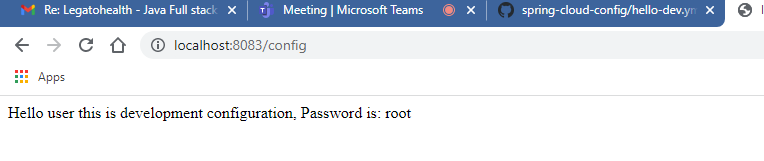
Note: bean.getPassword() must show the decrypted password i.e., root.

Note: Ensure you are pulling hello-dev.yml, because other properties doesn’t have password

bootstrap.yml

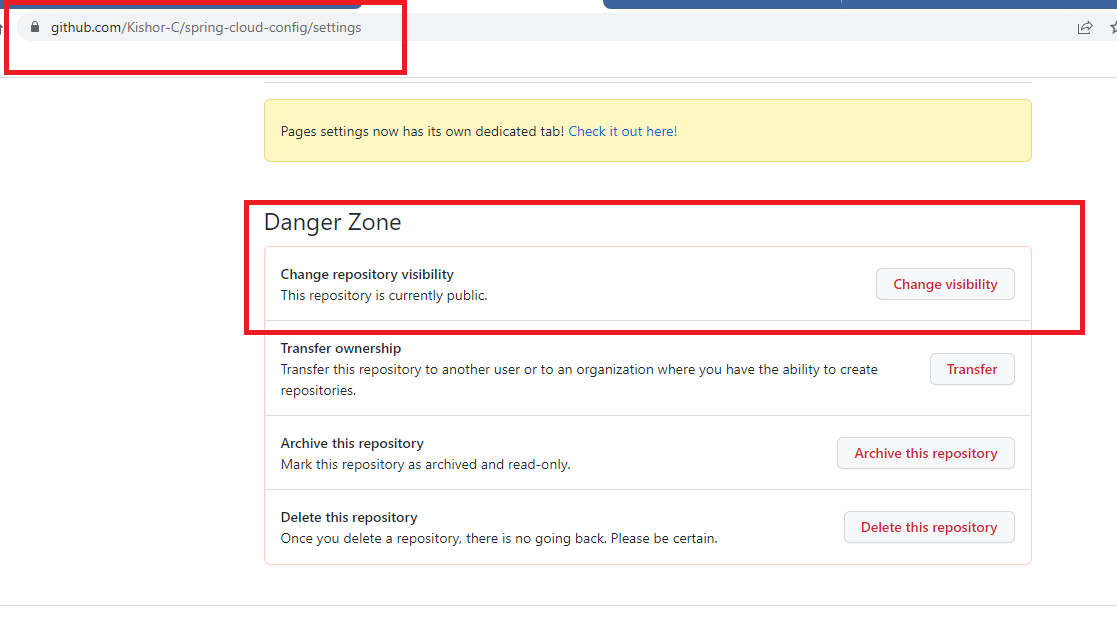


Output: Access the webservice of the client project



Privatizing the configuration

You need to keep your external configuration in the private repository.



Here you can select private so that nobody could able to access your external configuration file, your spring cloud config server can access using username & password of the git.

How to give the username & password of the git

That you need to mention in your application.yml through the properties

