Spring Cloud

* Overview
* Externalized Configuration
* Microservices
* Service Discovery
* Discovery Client
* Security
* Load Balancing
* Circuit Breaker: Resilience4j
* Zuul

Spring Cloud:

It helps in providing the tools to quickly build the distributed applications with common design patterns like:-

Service Discovery  
Discovery Client  
Circuit Breaker  
Load Balacing

Using this coordination between the distributed applications can be simplified.

With spring cloud you can simplify the development of distributed applications with simple annotations & configurations.

Spring Cloud uses the Spring Boot project to quickly develop the applications, we must use the compatible spring boot projects to work spring cloud.

Spring cloud uses many spring boot features:

* Starter Projects
* Compatible version of Spring Boot for Spring Cloud
* Creating an executable jar/war
* Property configuration
* Passing the configurations from command line at runtime
* Deploying the war on external server
* Actuators endpoints: health, metrics, beans
* MVC implementation: Service, DAO
* JPA Repository
* REST based services
* Curl commands
* Working on GIT

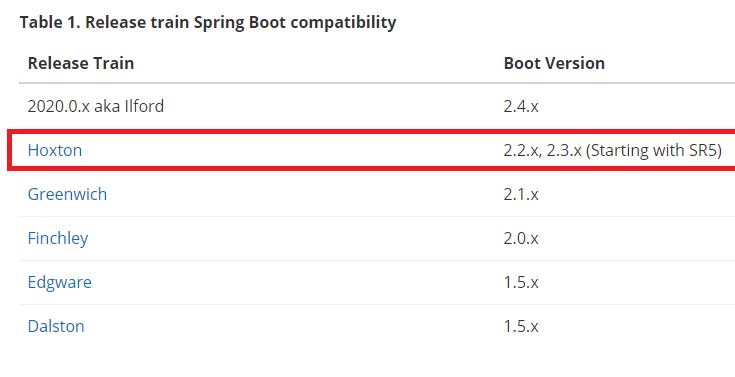
In spring cloud we are going to deal with:

* Externalized Configuration: This makes a configuration which is remotely available for multiple applications, you can apply security like Encryption by using JCE (Java Cryptographic Extension), You can also force applications to pass the credentials while accessing the remote location
* Microservices: Service Discovery, Discovery Client, Load-Balancing, we need to know some design patterns, like Development patterns, resilience pattern, routing pattern.
* Circuit Breaker: Hystrix (Deprecated), Resilience4j
* Zuul: Common door for all the incoming request to your service discovery, dynamic routing
* Secuirty

Getting Started

1. New Spring Boot project
2. Add spring cloud version to the existing project

Spring Boot compatible release train for spring cloud

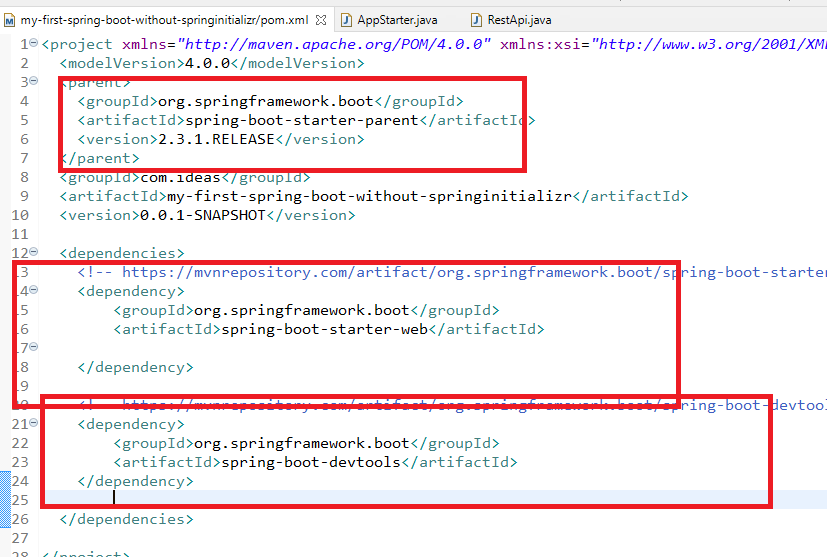


Spring Boot essentials:

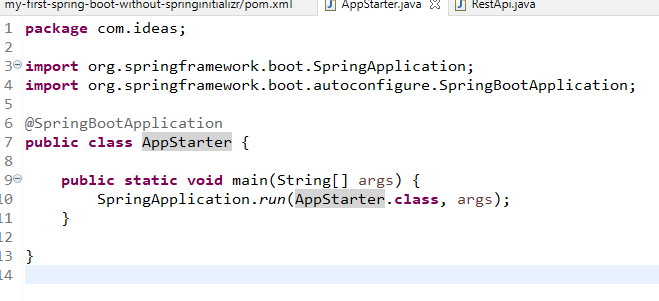
Starting from creating project to Creating rest services with JPA (in-memory database h2)

Spring Boot with Maven without spring initializr

pom.xml



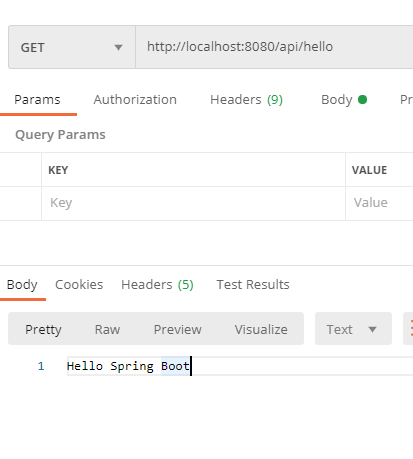
AppStarter



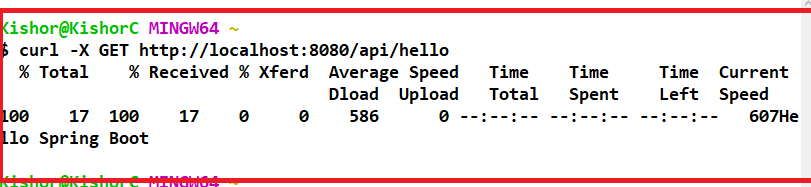
RestApi



Output:



Through cURL you can pass the GET request

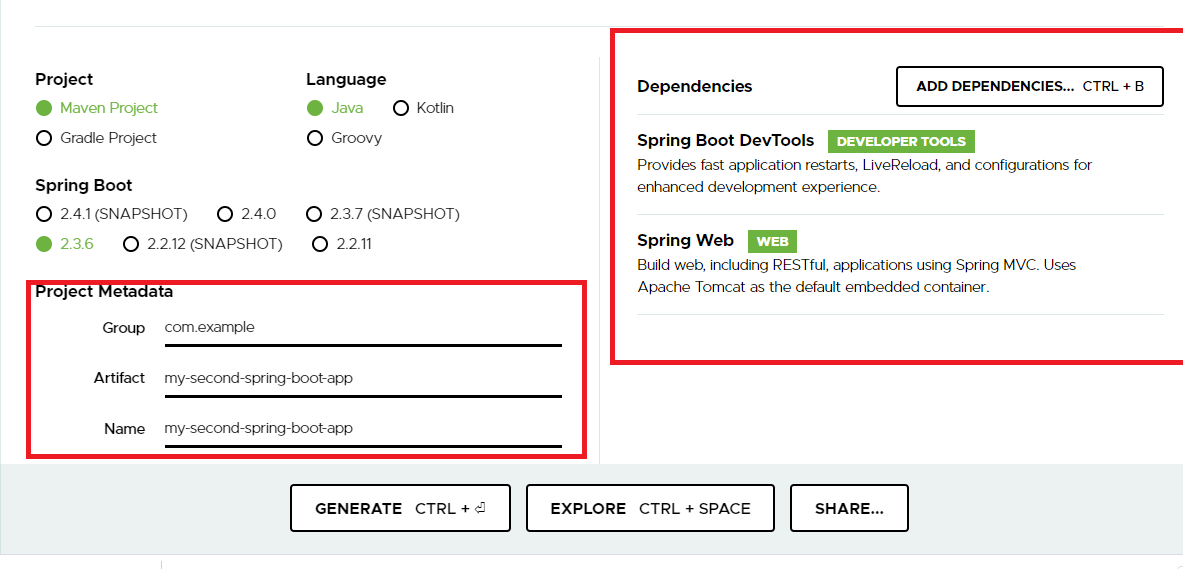


Some of the useful curl commands:

POST request with JSON data:   
curl -X POST url -H “Content-Type:application/json” -d “{...}”

PUT request:  
curl -X PUT url

Spring boot project with Spring Initializr



RestApi

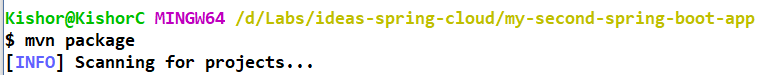


Benefits

* You will get UI based dependency configuration
* You will get pre-defined code to run/launch spring application
* You will get a plugin to create executable Jar/War
* You will get application.properties
* You will get compatible version of spring boot

How to create an executable jar file

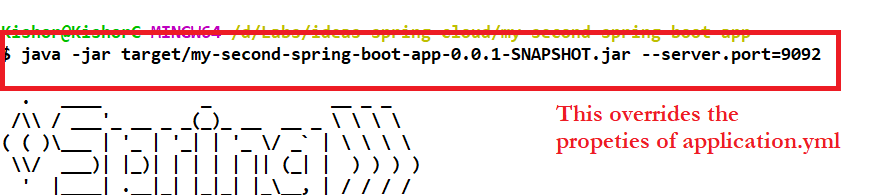
You should use *mvn package* command



You will get a jar file inside target folder

How to execute the jar file

You should use *java -jar <<file.jar>>*



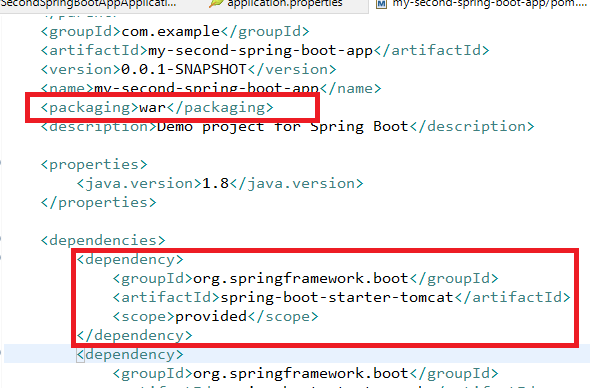
Deploying the war in an external server

* You need to perform some changes
* Main method doesn’t work hence you must use a filter called SpringBootServletInitializer which has a configure() method
* Once any class extends SpringBootServletInitializer configure() method will be executed you can write the launching of spring boot application inside the configure
* You must create WAR file instead of JAR file

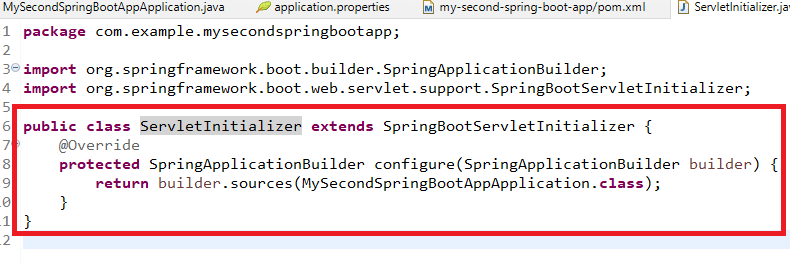
Steps:

* Change the packaging in pom.xml to war
* Embedded server must not be considered when war is built, you must use <scope>provided in pom.xml
* You must create a class that extends SpringBootServletInitializer

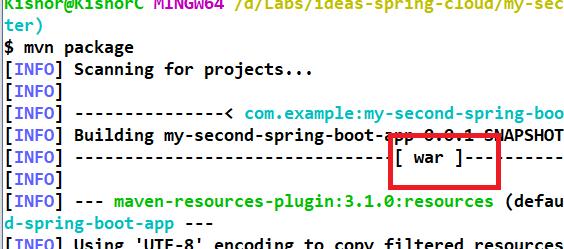
pom.xml



ServletInitializer

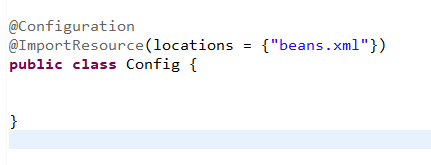


Now you can use mvn package



How to load the existing XML configuration in spring boot

You must use @Configuration class with @ImportResource annotation



Spring Boot best practices

1. Build systems - Maven, Gradle
2. Structuring your code - keep @SpringBootApplication in the root package
3. Avoid default package
4. Try to use @Configuration class instead of XML files
5. Don’t enable devtools in the production
6. Refer the migration guides when you want to migrate from old version to new version

Spring Actuator

You can monitor and manage applications in the production

you can see beans, health, metrics

health:

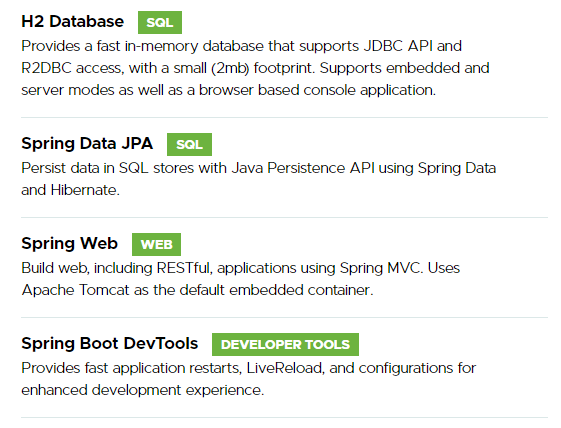
beans:

env:

metrics:

Spring Boot Starter JPA

You can use JpaRepository and CrudRepository which has inbuilt implementation as per the generics of your entity.



JpaRepository gives you lot of methods like

save() : it is used store & update both

deleteById():

findById()

findAll()

application.properties

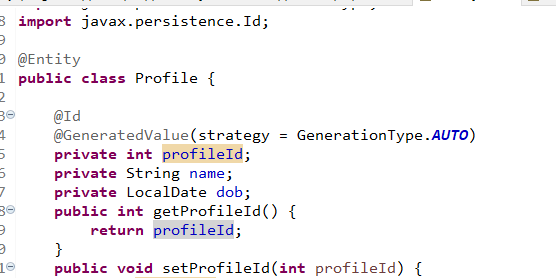
spring.datasource.driver-class-name=org.h2.Driver

spring.datasource.url=jdbc:h2:mem:myDB

spring.datasource.username=username

spring.datasource.password=password

Profile.java



ProfileDaoRepository.java

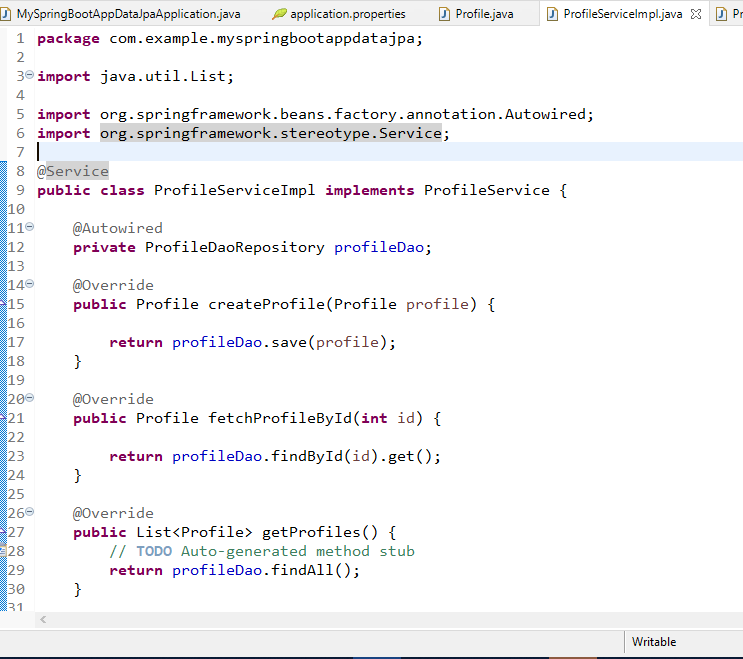
**package** com.example.myspringbootappdatajpa;

**import** org.springframework.data.jpa.repository.JpaRepository;

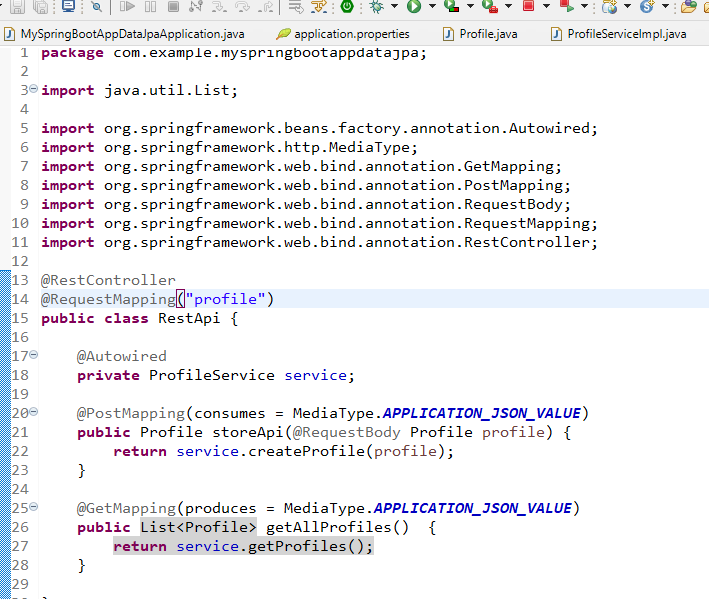
**public** **interface** ProfileDaoRepository **extends** JpaRepository<Profile, Integer> {

}

Service class



Rest



Spring Cloud main projects

Spring Cloud Config

Spring Cloud Netflix

Spring Cloud Security

Spring Cloud Configuration Server / Externalized Configuration:

Here the client & server can access the externalized configuration in a distributed system.

The externalized configuration location can be a filesystem / GIT

Features:

There are two things

1. Spring Cloud Server Features
   1. This takes care of locating the original location of the centralized repository/system
   2. Encrypt & Decrypt the sensitive information
2. Config Client Features
   1. This binds to the config server & spring environment will load the remote resource properties

You need to create 2 projects

1. Configuration Server
2. Configuration Client

Dependencies required for Configuration server:

Config-Server

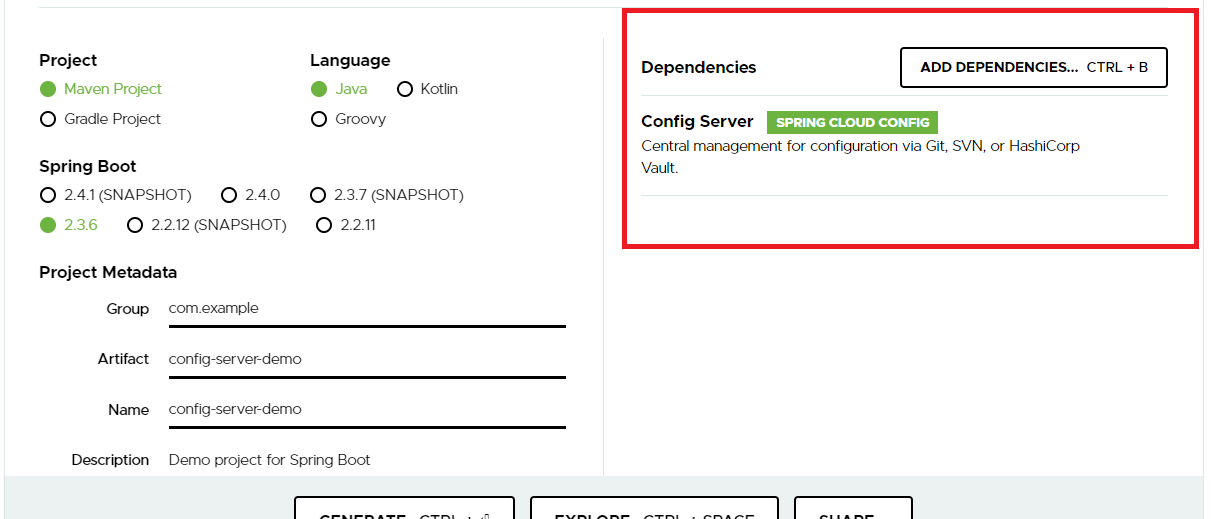
Dependencies required for Configuration client:

Web

Config-Client

(Optionals: Jpa, Devtools, Actuator)

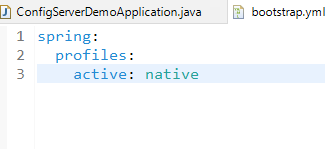
Config Server



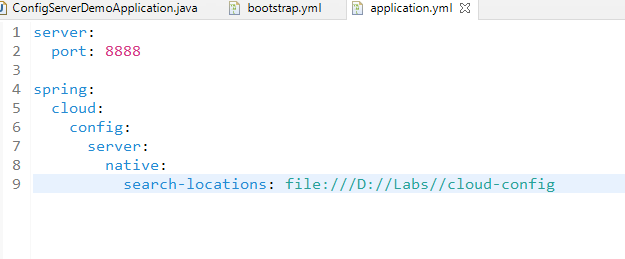
bootstrap.properties or bootstrap.yml: it is loaded before any property files, it is loaded at the time boostraping the application.

Create 2 yml files inside the configuration server project

bootstrap.yml

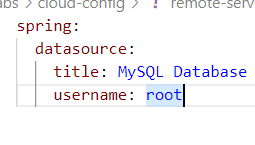


application.yml

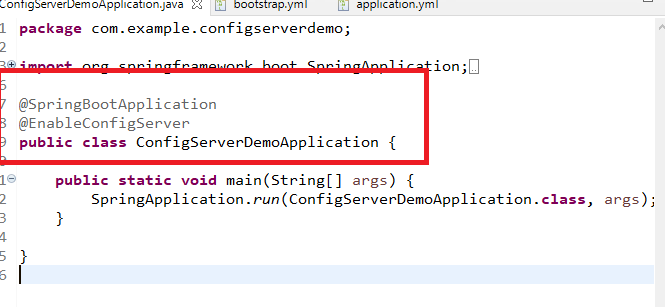


You can create one externalized configuration file in some directory.

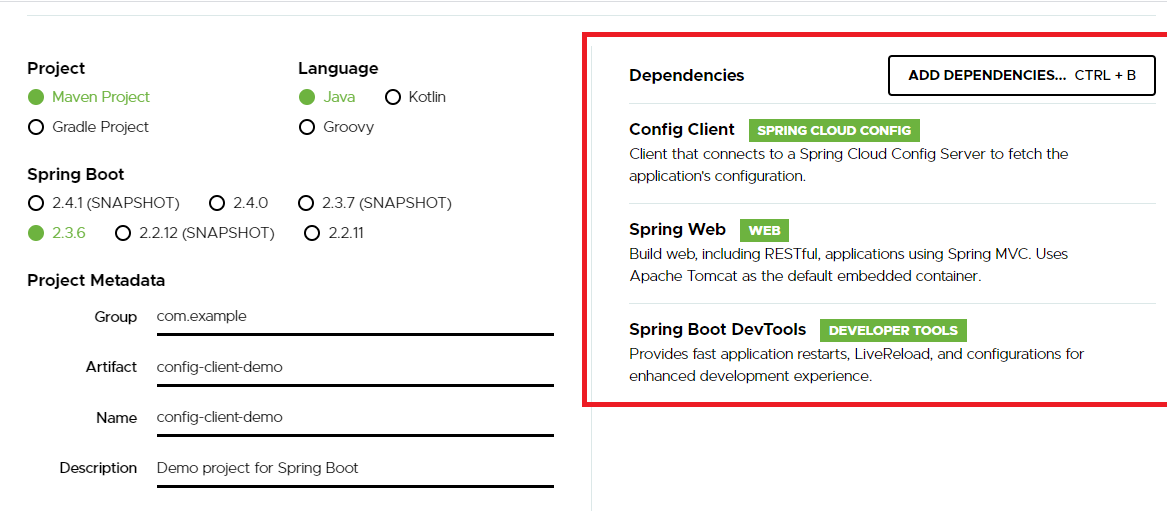
remote-server.yml



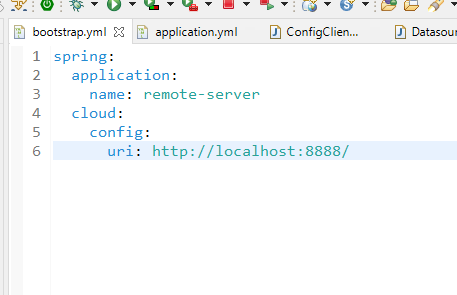
Now you can run the configuration server



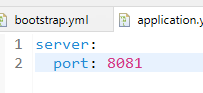
Client Program / Configuration Client



bootstrap.yml

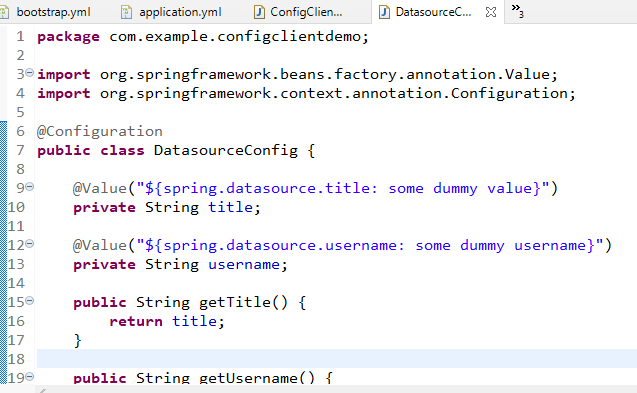


application.yml

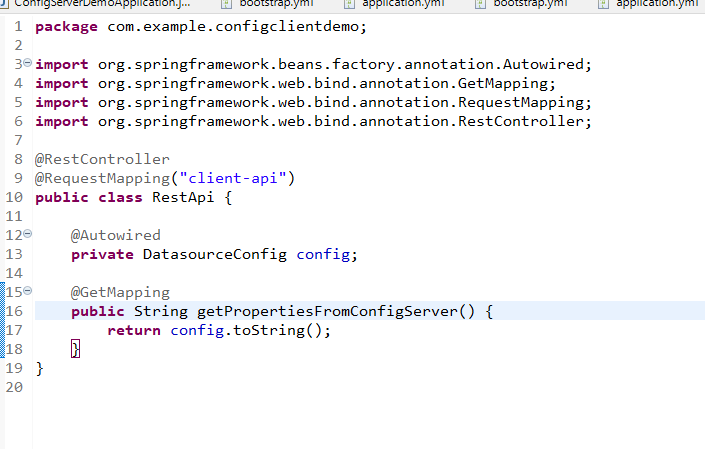


DatasourceConfig.java

Note: Here we are binding the external properties of configuration server to the DatasourceConfig variables

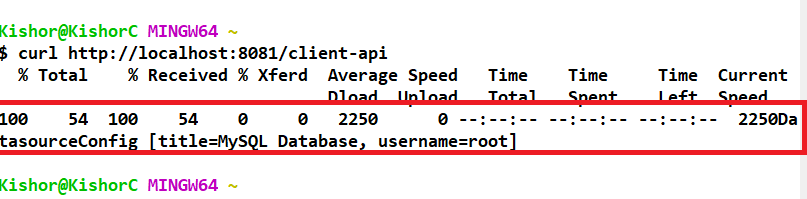


RestApi.java



Now you can launch the client application

Output:

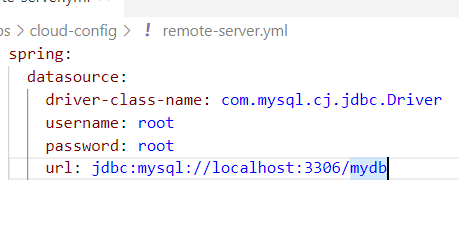


Note: If configuration server properties are modified it wouldn’t be affected to the client, because the client loads the configuration at the time of startup, hence you must use some kind of endpoints given by actuator where you can get the modified configurations or you can use pub/sub mechanism through the events automatically changes will be affected

How to use the datasource information from the configuration server

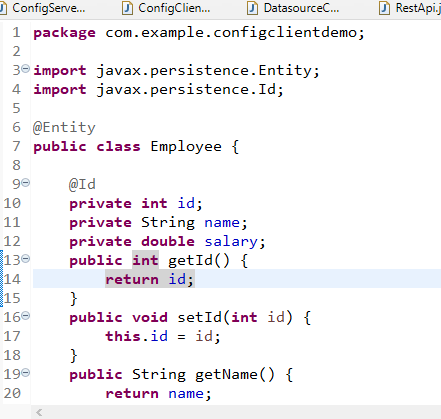
1. You must add spring-boot-starter-jpa & database drviver in the client pom.xml file
2. you must add datasource properties in the remote configuration server

remote-server.yml

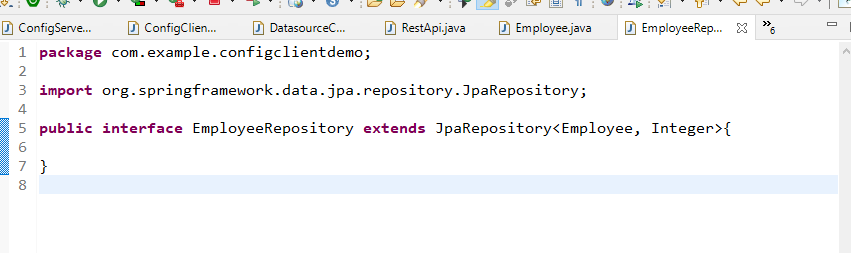


You can have Employee entity and JpaRepository interface type in the client application

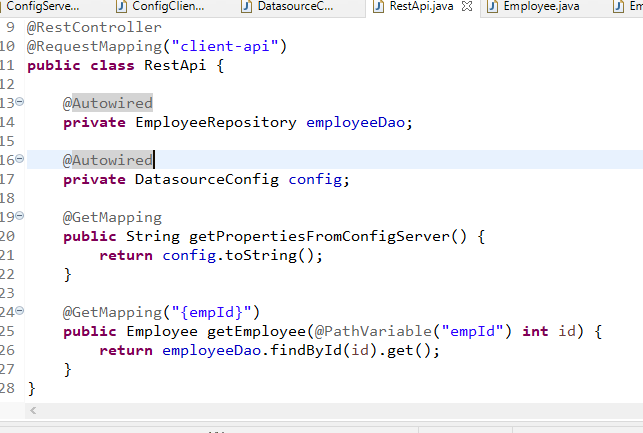
Employee.java



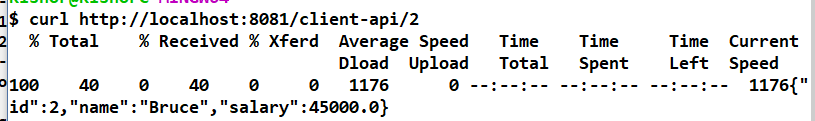
EmployeeRepository.java



RestApi.java



Output:



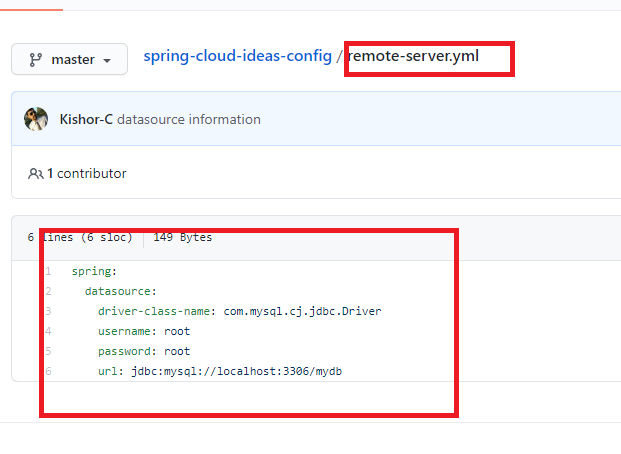
Todays Agenda

* Configuration server to pointing to GIT
* Search configurations in other locations
* Reloading the configurations without restarting the client application
* Encryption and Decryption of configurations (sensitive informations) - JCE
* Active Profiles: load the configurations based on the provide (developer, production, testing, default)
* Accessing the private repositories

Configurations using Git backend

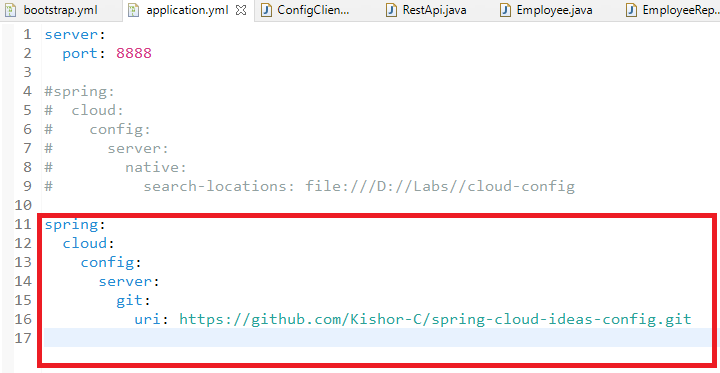
You must use config.server.uri which comes as a part of Configuration Server library

Create remote-server.yml and push to git



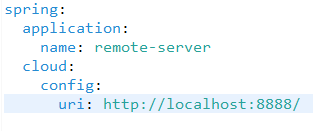
Configuration Server

No content in bootstrap.yml

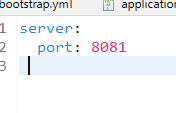


Configuration Client

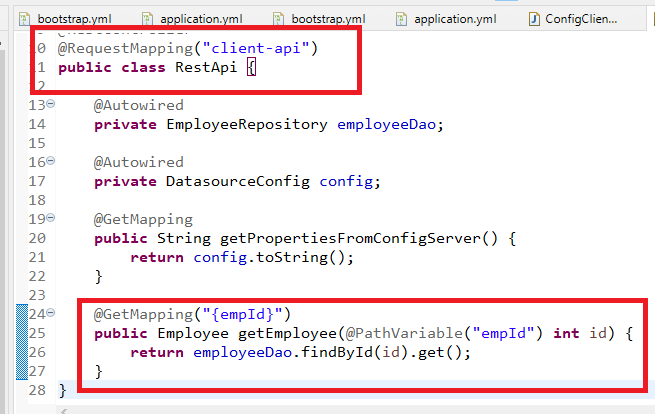
bootstrap.yml



application.yml



RestApi.java



Output:



The configurations are not updated to the client so we can use actuator @RefreshScope.

You must add actuator library

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

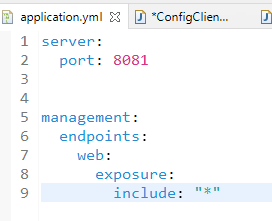
<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

Once you update the configuration file / configuration server, the client application can use the below end point

client-app-url/actuator/referesh

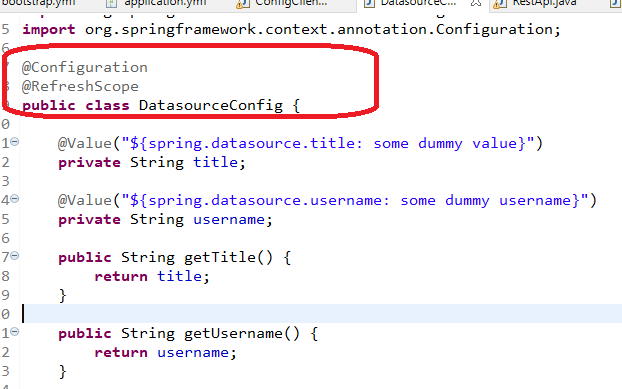
The Client application must also have exposed these above endpoints using below configuration in the application.yml



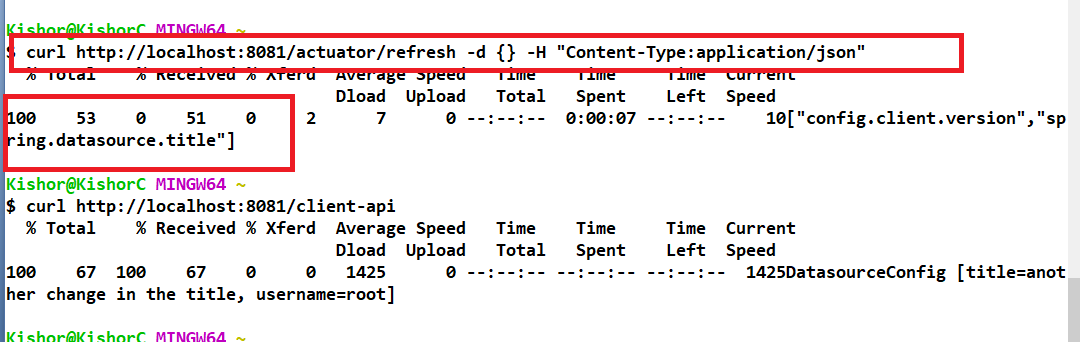
Enter refresh with POST method and empty JSON value as below:

curl -X POST -d {} -H “Content-Type: application/json” URL

Mention @RefreshScope on top the configuration class



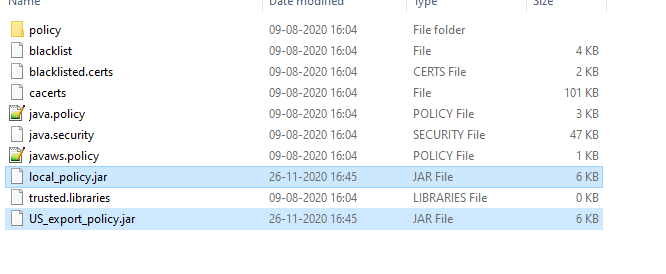
CURL command output after actuator/refresh



Encrypting and Decryption the Sensitive information

We must download JCE library from the Oracle web site & store inside jre/lib/security/ folder

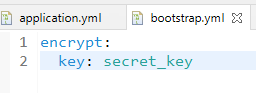
Spring Cloud can perform encryption & decryption using some key, this key must be combination of special characters and better more than 12 characters.

local\_policy.java & US\_export\_policy.jar are the two files which gives us the cryptographic algorithm

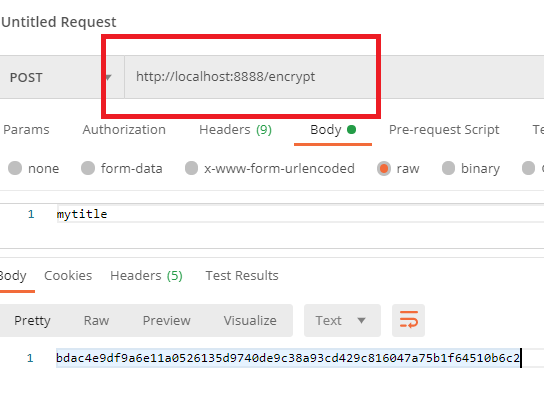
The only thing spring cloud must do is mention the encryption key and apply encrypt & decrypt.

Note: Configuration Server needs to specify the encrypt.key

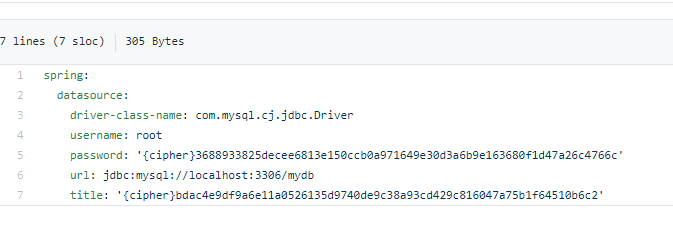
bootstrap.yml



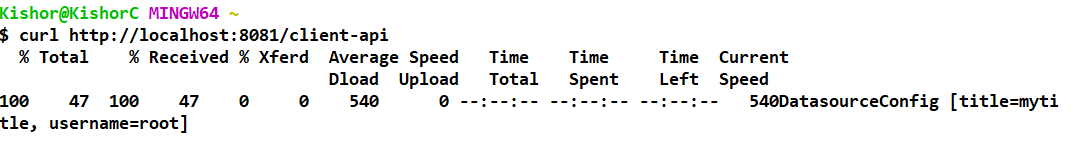
Perform encryption and add to the configuration file



Add these code to the configuration file & push to the GIT

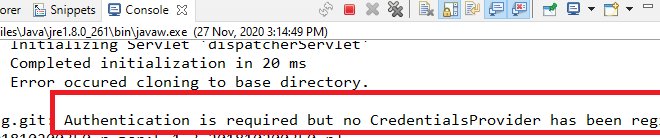


Now check the client endpoints to see the decrypted output

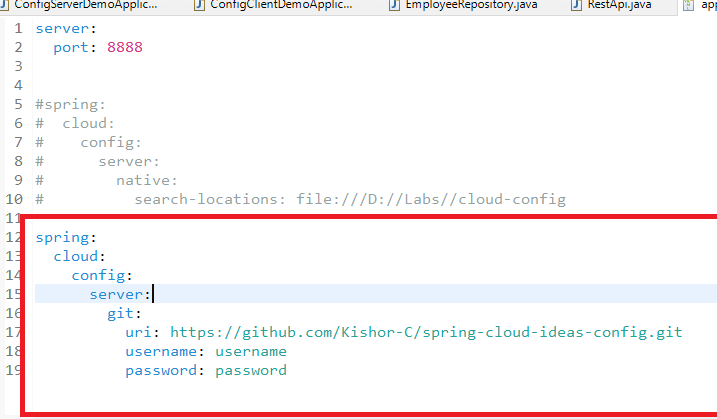


Private repository:

When you have private repository the configuration server must use username & password in the application.yml, else you will get below error where client application tries to fetch the configuration file



application.yml



Loading the configurations based on the user profiles like

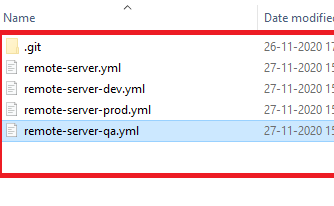
* development
* production
* qa

Here the configuration files will be stored in the naming pattern

configname.yml  
configname-development.yml  
configname-production.yml  
configname-qa.yml

You can mention these profile names using spring.active.profiles property.

You will have multiple configuration files



You can change any one property ex: title

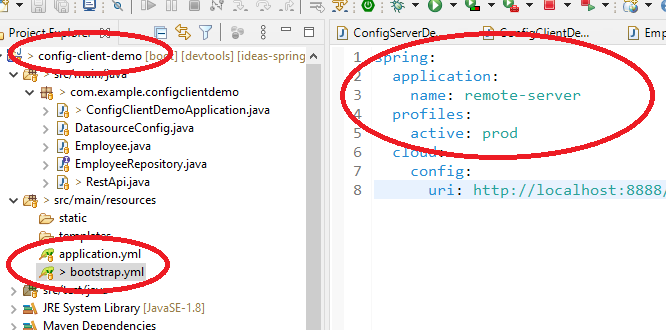
remote-server-qa.yml  
 title = simple testing title

remote-server-prod.yml  
 title = simple production title

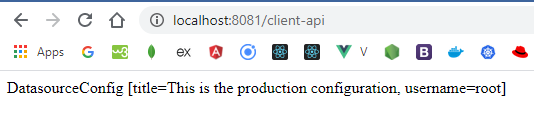
You can mention the profiles.active properties in the client configuration file or you can mention through terminal --spring.profiles.active

Note: bootstrap.yml of client application will have the profile configurations

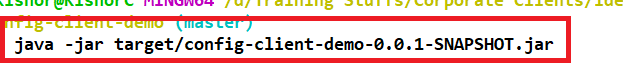
bootstrap.yml (client-configuration -application)



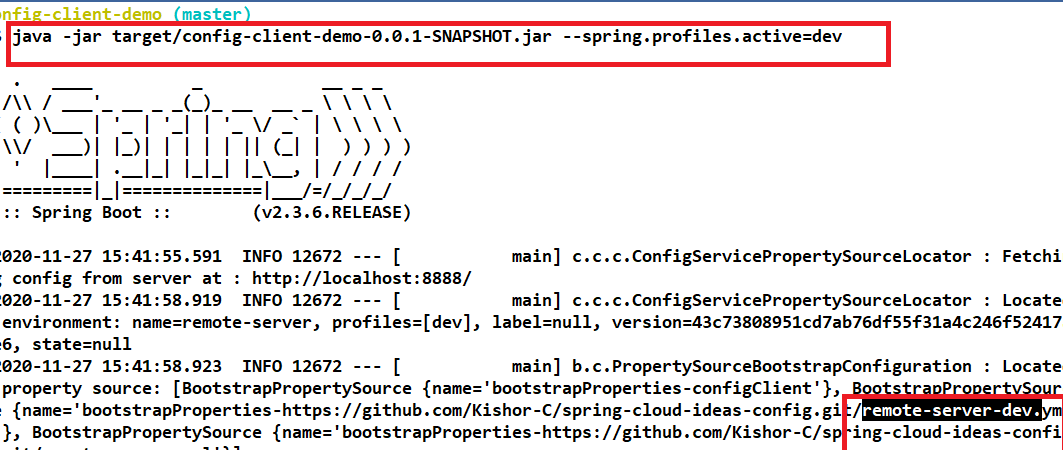
Output:



Note: You can give the profiles from the terminal without mentioning in the bootstrap.yml



Loads the default configuration file remote-server.yml



Output:



Suppose your configurations are present in different folder you can use search-locations property



You can also load configurations from multiple repositories for that you can user

spring.cloud.config.server.git.uri = <<default-uri>>

spring.cloud.config.server.git.repos.uri = <<particular-repos-uri>>

Spring Microservices

Microservices are independent services which can be deployed on servers they are independent from other services of same project or different project.

Monolithic architecture

The entire application is built as a single deployable unit and they all share single code repository and it goes to the server.

Drawbacks:

1. If any service affects the other services and chances of entire application going down is high, that makes all the services to go down
2. If any changes happens in a module the entire application has to be rested and rebuilt
3. You can’t scale only few services you had to scale the entire application

Microservice architecture

Microservices are independent services which are deployed on servers independently

1. They are easily scalable
2. They don’t affect other services if any service goes down
3. Each service will have separate code base repository
4. Services don’t directly talk to other service databases
5. If any change happens in a microservice you dont’ have to test other services
6. Microservice communicate with other microservices using REST calls
7. You can develop services in other languages
8. It uses service discovery which is location transparent that makes microservice communications easier without changing the client code.
9. Various design patterns to simplify the work
   1. Service Discovery (Eureka Server)
   2. Load Balancing (Ribbon)
   3. Resilient pattern - Circuit Breaker (Hystrix/Resilience4j)
   4. Security pattern - OAuth2 and JWT
   5. Logging - Zipkin

Spring has made microservice development easier with the help Netflix implementation, through simple annotations you can get the job done.

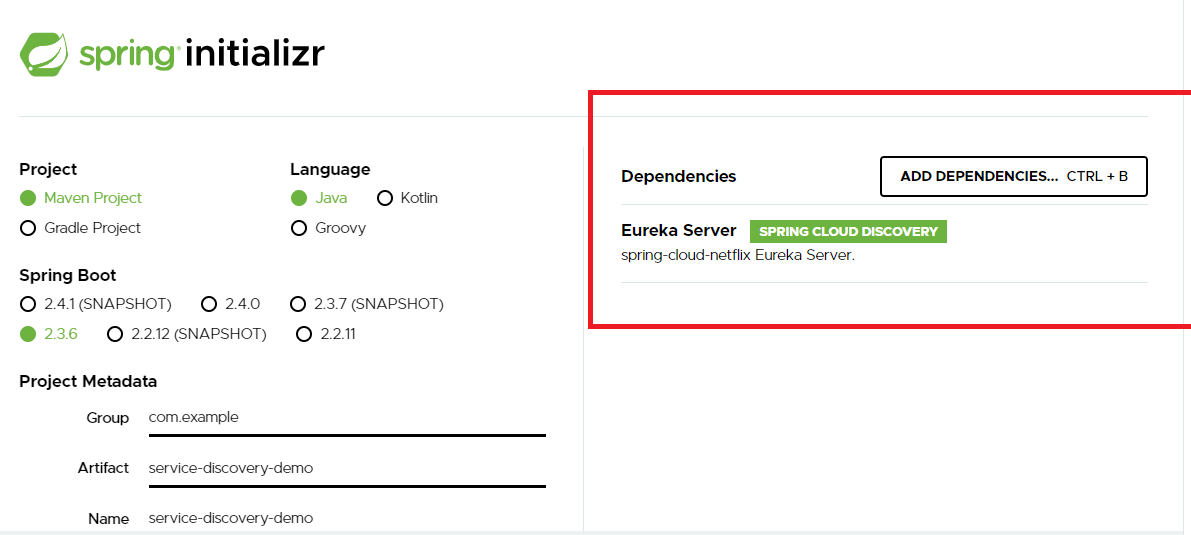
Pre-requisites

* Spring Cloud
* Spring Boot

Service Discovery:

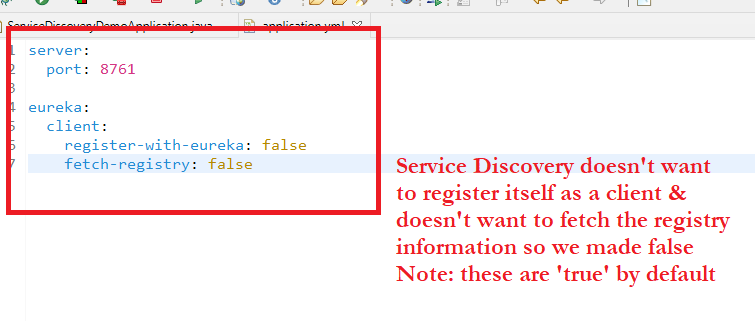
This is a design pattern where all the microservice can register them so that other microservices can locate, this helps to have a logical name that abstracts the physical location of the microservice.

* Multiple instances of same microservices will have same logical name
* Spring Netflix has given an implementation to the service discovery called EurekaServer, this can be enabled using @EnableEurekaServer
* This eureka server gives you the dashboard where you see all the registered services along with their instances
* Dependency we need is Eureka Server



Service Discovery will have all the service registered and by default all the microservice looks for service discovery at 8761 port.

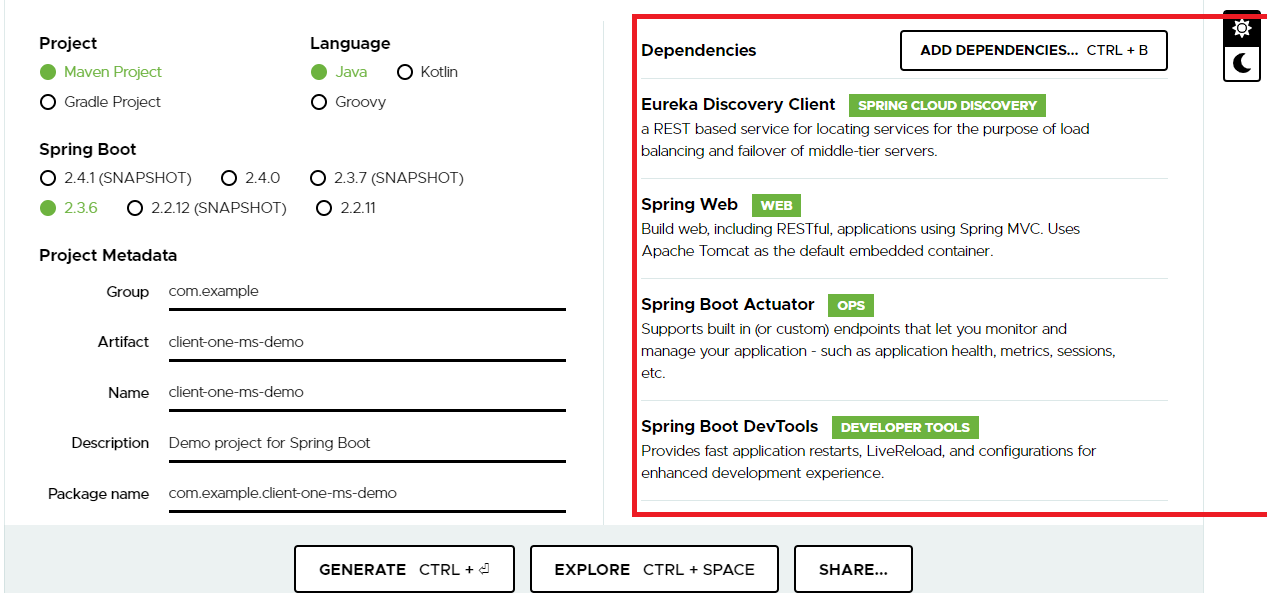
Service Discovery is by default acts like a client so you can disable this property, so that it wouldn’t register as a service by itself



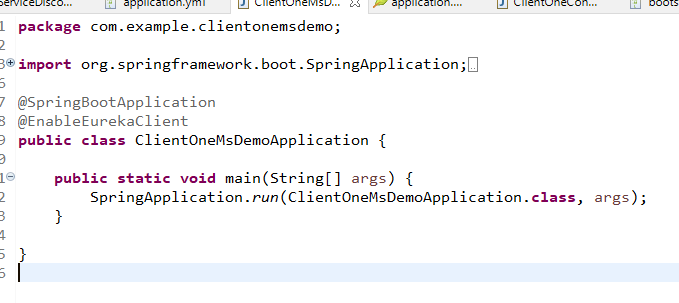
Creating a microservice and registering with the service discovery

Dependencies required:

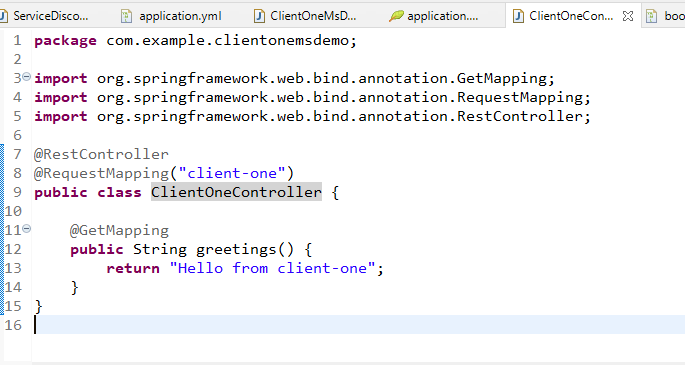
* Eureka Discovery Client
* Web
* Actuator
* Devtools



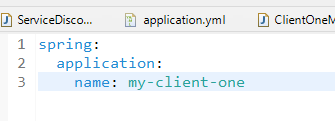
Add @EnableEurekaServer to the application



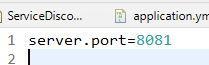
Controller code



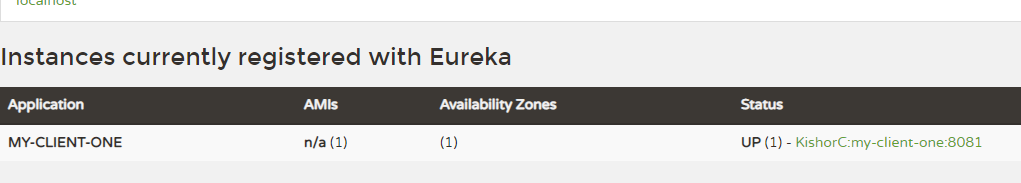
bootstrap.yml



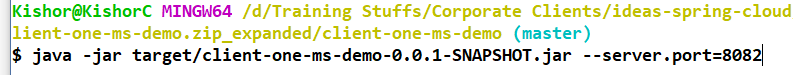
application.properties



After running the application you will see the instances in the eureka-dashboard



You can create another instance of the same service and deploy it on a different server, but what we can do is we can use localmachine and run the application.jar in different port



Output in the Eureka Dashboard



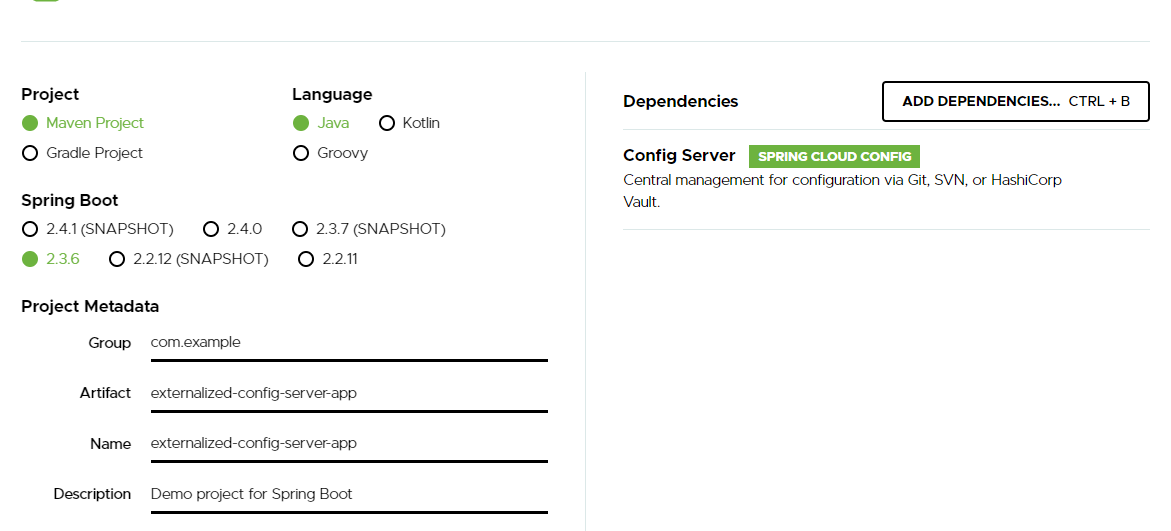
Exercise:

Keep microservice & service discovery configurations in the configuration server and fetch the configurations from the configuration server.

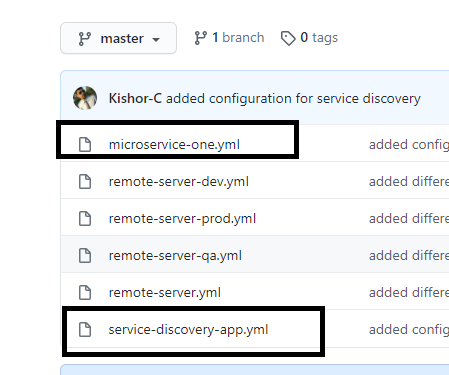
Dependencies required

1. Configuration Server:
   1. Config Server
2. Service Discovery:
   1. Eureka Server
   2. Web
   3. Config Client
3. Microservice:
   1. Eureka Client
   2. Web
   3. Config Client
   4. Actuators
   5. Devtools

Step 1: creating configuration server & also creating some configuration files in the remote repository



Create a service-discovery-app.yml and microservice-one files and push to the GIT repository



service-discovery-app.yml

eureka:

  client:

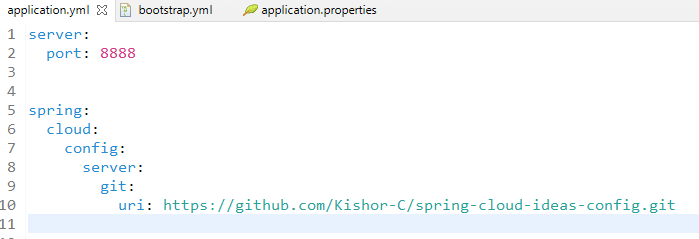
    register-with-eureka: false

    fetch-registry: false

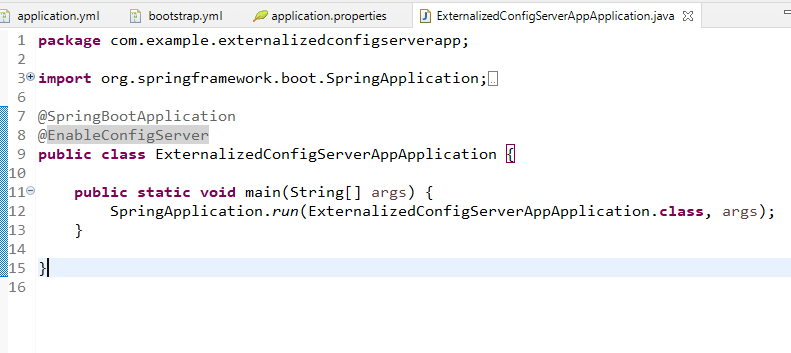
microservice-one.yml

title: This is fetched by microservice-one

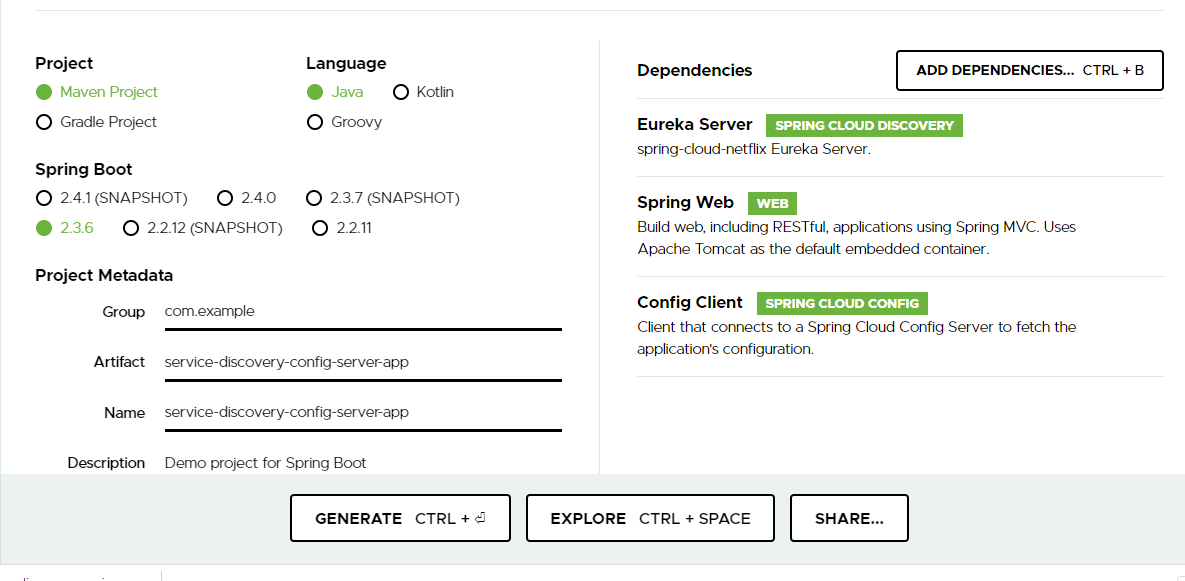
application.yml file in the Configuration Server project



Configuration Server Application

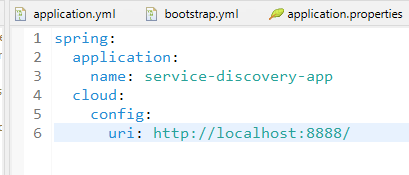


Step 2: Service discovery that connects to the configuration server



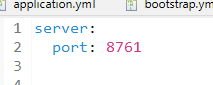
Since this service discovery need to fetch the configuration from the configuration server it has to provide configuration server uri

bootstrap.yml

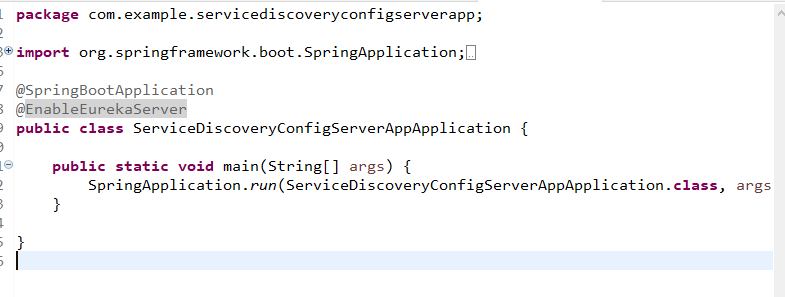


Note: *spring.application.name* will fetch the *service-discovery-app.yml* file from the configuration server running at <http://localhost:8888/>

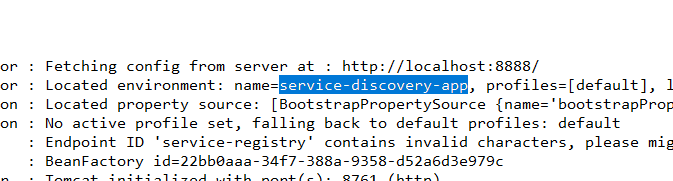
application.yml



Add @EnableEurekaServer in the main method

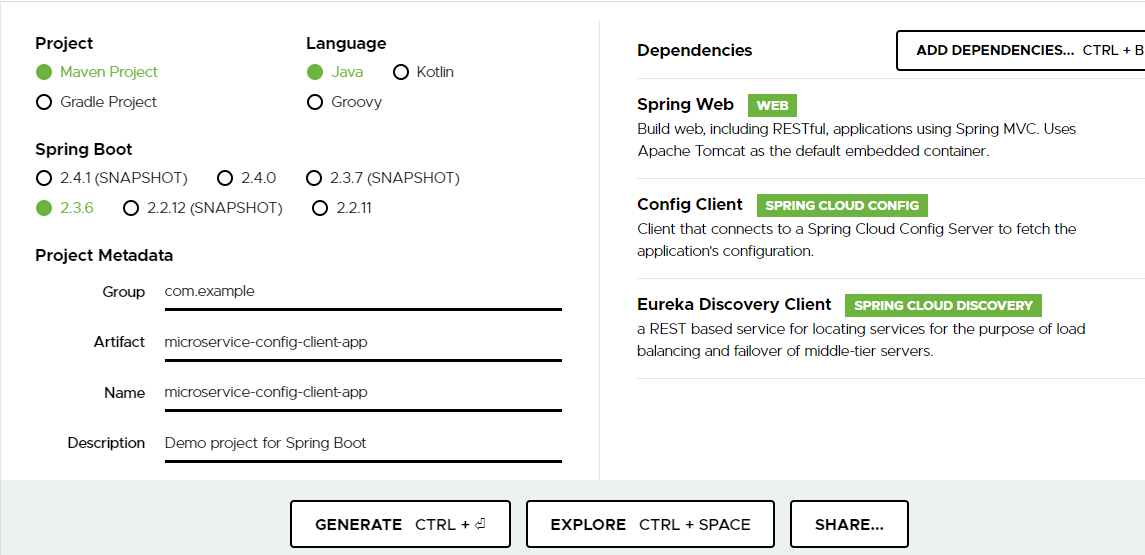


After launch

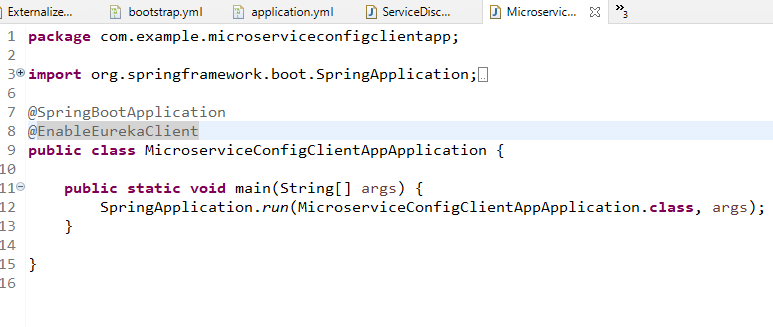


Note: above project name can be used as service-discovery-config-client instead of service-discovery-config-server

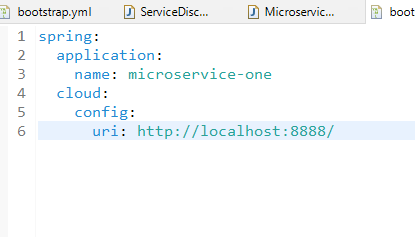
Step3: A microservice that has to register to the service discovery and fetch the configurations from the configuration server



Add @EnableEurekaClient to the application



bootstrap.yml

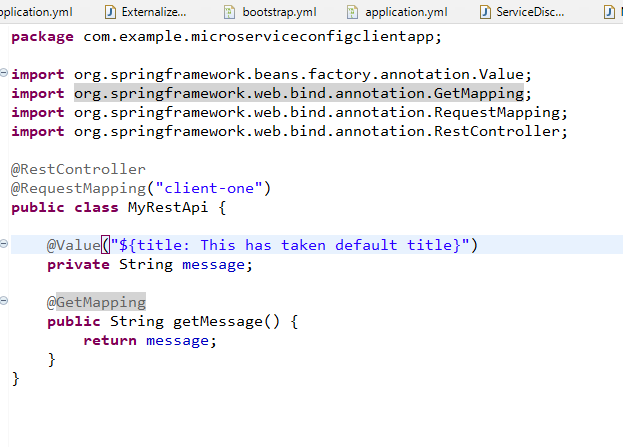


application.yml

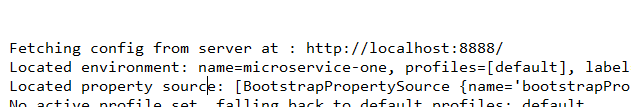
server:

port: 8081

Create a rest endpoint that shows the configuration properties fetched from the configuration server.

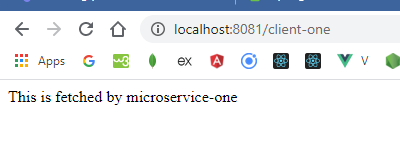


Now you must see the configuration fetched from the configuration server in the logs



Two things you can notice here is

1. This application registers with eureka
2. This application fetching configuration from configuration server



Communicating one microservice with another microservice

We will call Account Microservice with Wallet Microservice using RestTemplate backed by LoadBalancer.

We need two projects with @EnableEurekaClient

1. Account-Microservice
   1. AccountController
   2. AccountService
   3. Transaction
2. Wallet-Microservice
   1. WalletController
   2. WalletService
   3. Transaction
   4. Wallet

Account Microservice

application.yml

server:  
 port = 8081

bootstrap.yml  
spring:   
 application:  
 name: account-service-app

Application

**package** com.example.accountmicroserviceapp;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication

@EnableEurekaClient

**public** **class** AccountMicroserviceAppApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(AccountMicroserviceAppApplication.**class**, args);

}

}

Transaction.java

**package** com.example.accountmicroserviceapp;

**import** java.time.LocalDateTime;

**public** **class** Transaction {

**private** **long** transactionId;

**private** **double** amount;

**private** LocalDateTime time;

**public** **long** getTransactionId() {

**return** transactionId;

}

**public** **void** setTransactionId(**long** transactionId) {

**this**.transactionId = transactionId;

}

**public** **double** getAmount() {

**return** amount;

}

**public** **void** setAmount(**double** amount) {

**this**.amount = amount;

}

**public** LocalDateTime getTime() {

**return** time;

}

**public** **void** setTime(LocalDateTime time) {

**this**.time = time;

}

}

AccountService.java

**package** com.example.accountmicroserviceapp;

**import** java.time.LocalDateTime;

**import** org.springframework.stereotype.Service;

@Service

**public** **class** AccountService {

**public** Transaction debit(**int** accountNumber, **double** amount) {

System.***out***.println("----- debit from AccountService -----");

Transaction transaction = **new** Transaction();

transaction.setTransactionId(accountNumber \* 2);

transaction.setAmount(amount);

transaction.setTime(LocalDateTime.*now*());

**return** transaction;

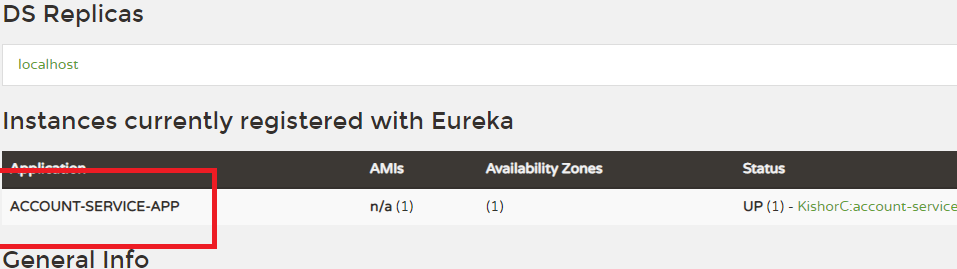
}

}

AccountController.java



Now the wallet microservice must use the below name



Wallet Microservice application

bootstrap.yml

spring:

application:

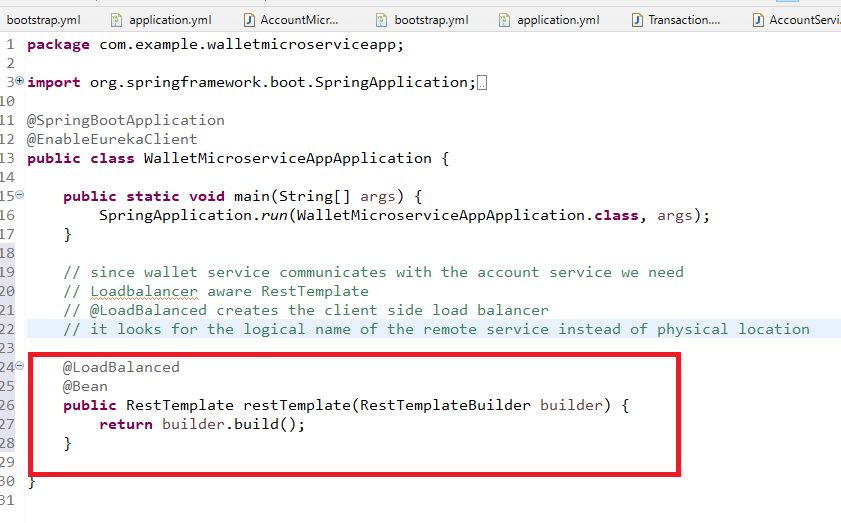
name: wallet-service-app

application.yml

server:

port: 8082

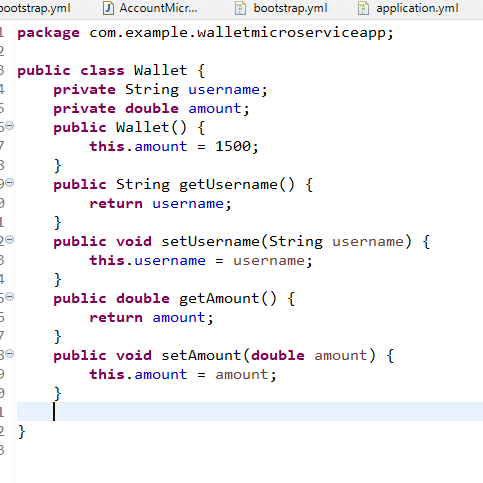
WalletApplication



Transaction.java: this will match to the JSON response structure



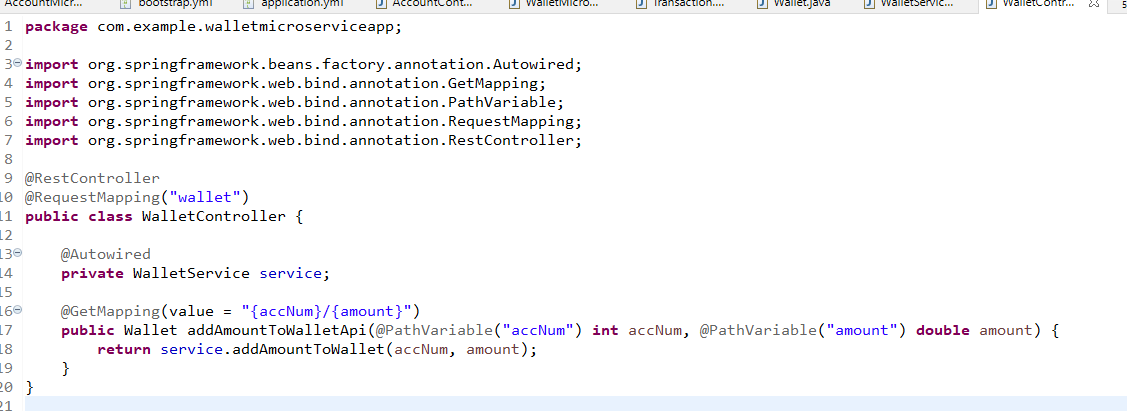
Wallet.java: Model in the wallet service



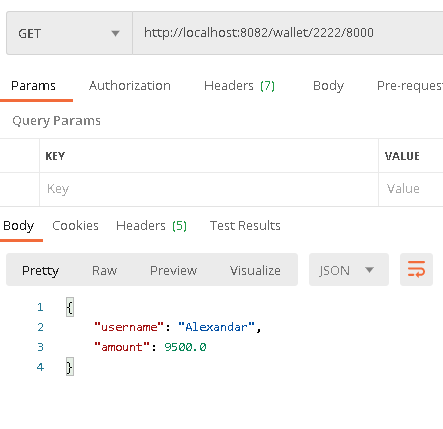
WalletService.java: This calls the account microservice and extracts the amount and adds to wallet



WalletController.java

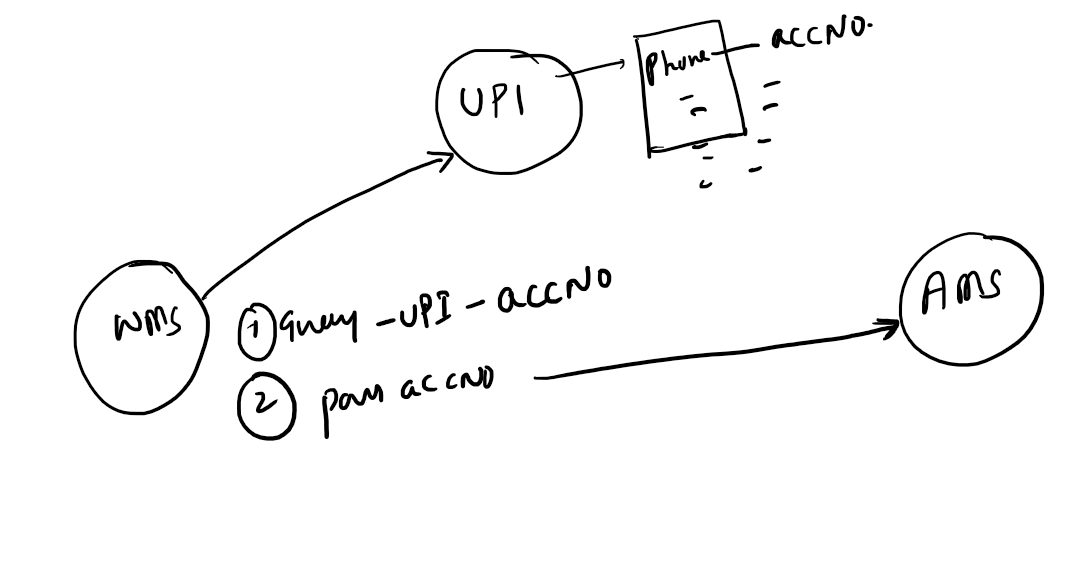


Output:



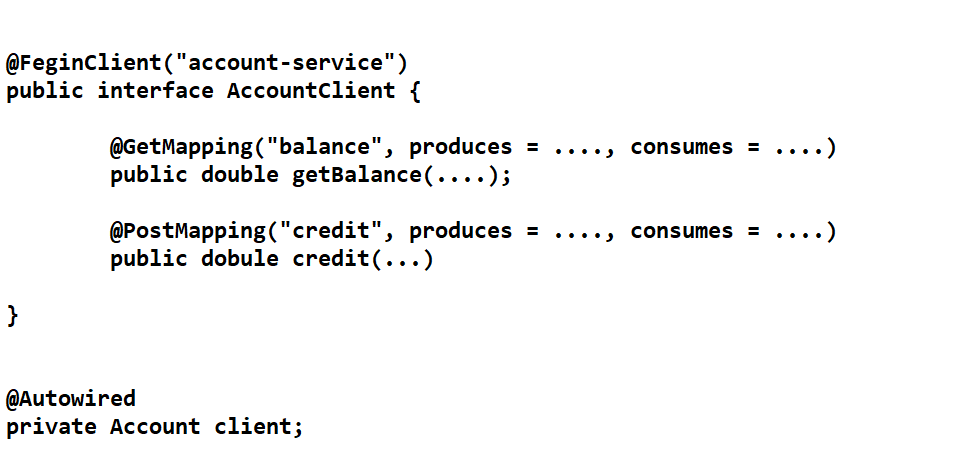
You can create another instance of account and see how the load balance is distributing the traffic across multiple instances

Exercise



Feign Client:

It is used to call the external microservice but is by default backed by client-side load balance & it is more reusable with other clients who wants to call remote service.



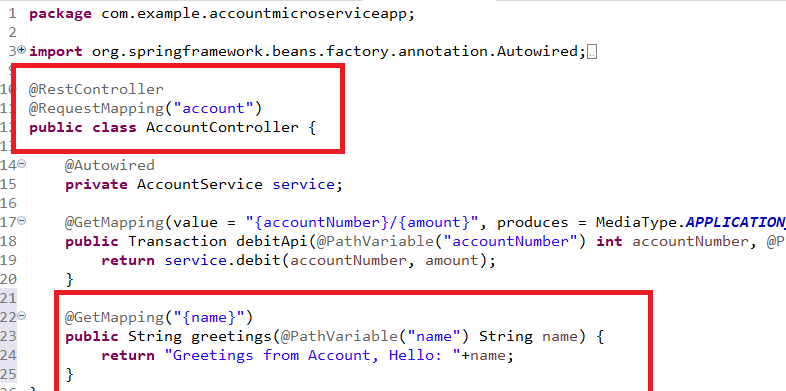
We need a dependency

OpenFeign Client

We need to use

@EnableFeignClients: it scans for all the @FeignClient interfaces to provide a proxy implementations

Add one method in AccountController



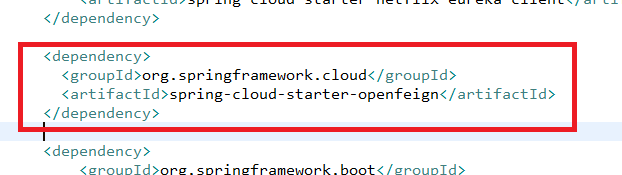
We will call this greetings() from wallet service using FeignClient

Firstly we need to add OpenFeign dependency to wallet service project

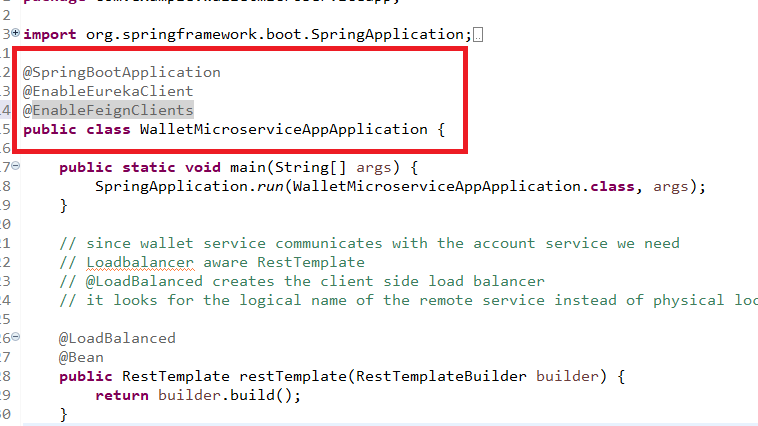


Wallet-Service

pom.xml

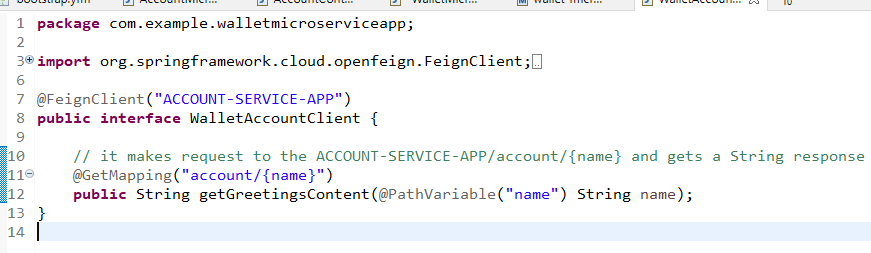


Add @EnableFeignClients: that scans all the @FeignClient interfaces to provide the proxy implementations

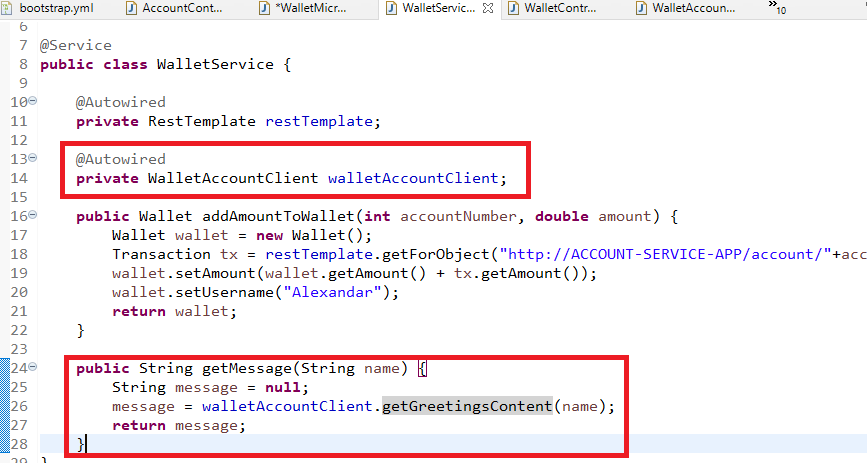


Here RestTemplate is not necessary, we have an interface with a common uri that can access all the apis of a particular microservice

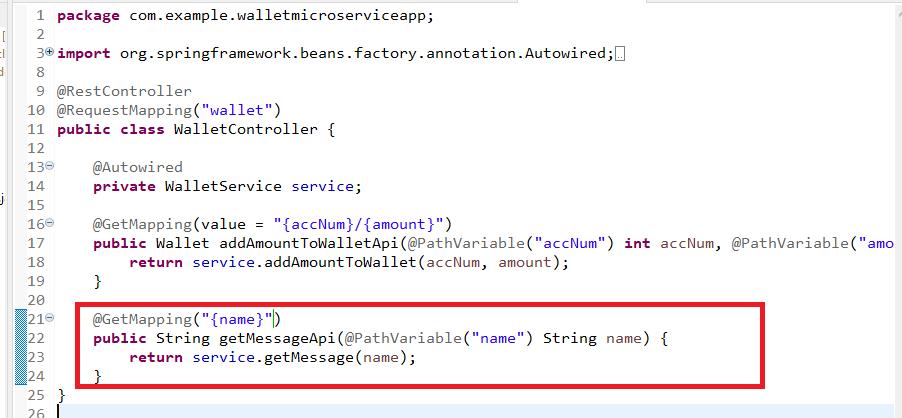
WalletAccountClient.java



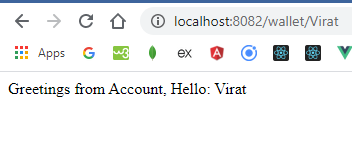
WalletService.java



WalletController.java



Here when you call wallet/name -> it calls   
ACCOUNT-SERVICE-APP/account/name



API Gateway

Zuul: It is used to add reverse proxy features to the clients who are trying to communicate with the microservices

You can make all the requests pass through the API gateway and apply some of the common capabilities like logging, tracking & so on.

Zuul is the library provided by the Netflix to implement API gateway.

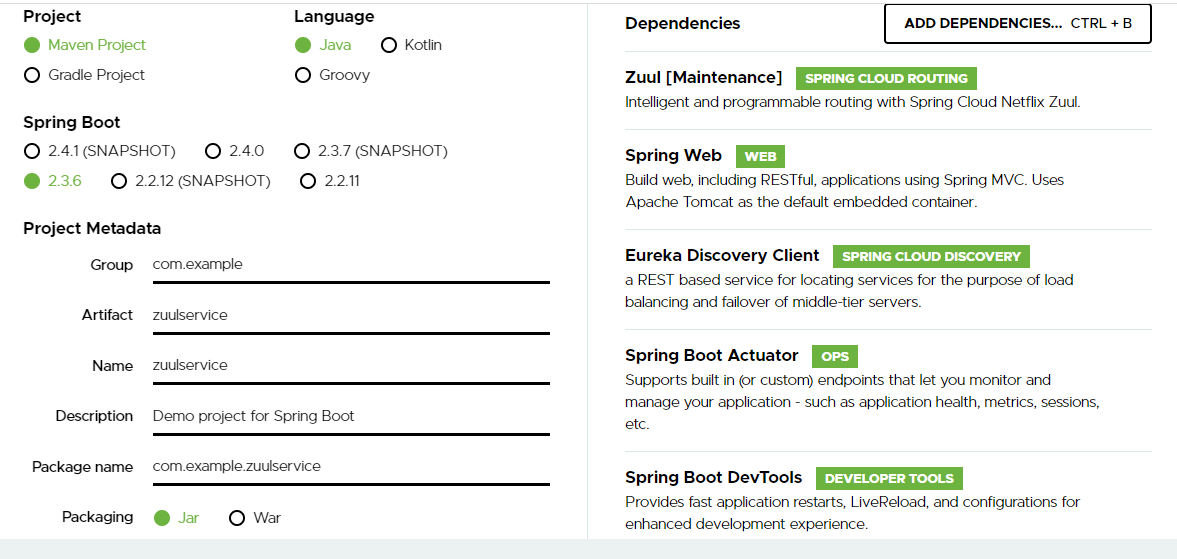
You can register the Zuul with Service Discovery so that it apply the routes to the microservices that are registered with the service discovery.

With the help of Zuul clients don’t directly communicate with other microservices

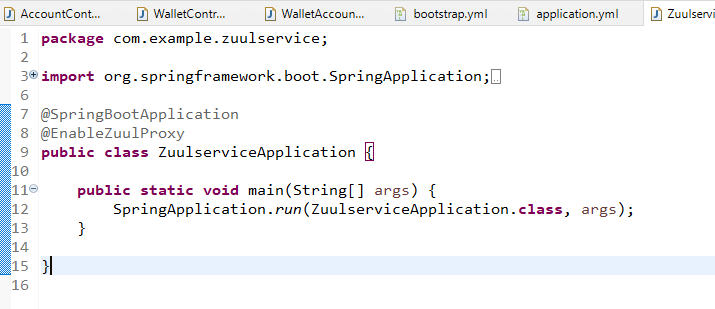
Zuul gives you a single end point which can be used by all the clients.

The dependencies we need is:

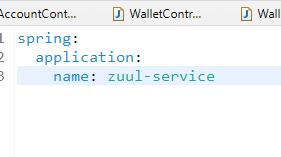
* Zuul
* Eureka Client: register with eureka & fetch registry
* Actuators: routes will be available



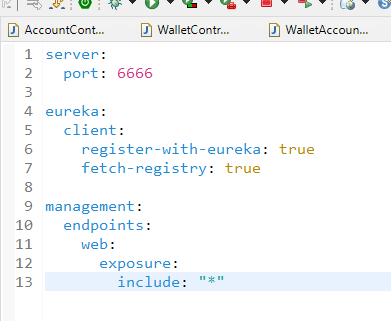
ZuulService



bootstrap.yml

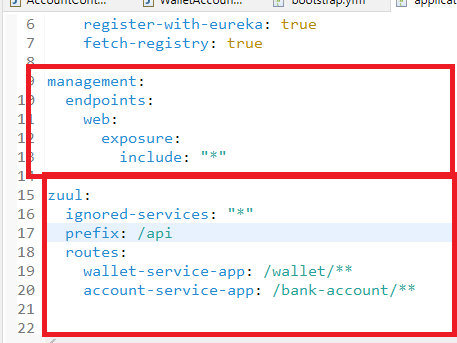


application.yml



Actuator gives routes endpoint to see all the routes.

application.yml



All the end users applications must talk to the microservice through common gate way i.e., zuul api gateway

Ex: If wallet has to talk to account then wallet must use

<http://zuul-service/api/wallet/wallet/name>

Circuit Breaker with Resilience4j:

It allows you to avoid cascade of failures when services are calling nested remote services.

Circuit Breaker use Ring Bit Buffer to do a transition from CLOSE to OPEN or OPEN TO HALF\_OPEN or HALF\_OPEN to CLOSE.

CLOSE: All the requests will go to remote service.

OPEN: No request goes to remote service instead goes to the fallback method

HALF\_OPEN: Request goes to remote service, if the number of failures are more than the threshold then it goes to OPEN state else state changes to CLOSE.

Dependencies we need

Resilience4j - spring boot2

AOP

Actuator

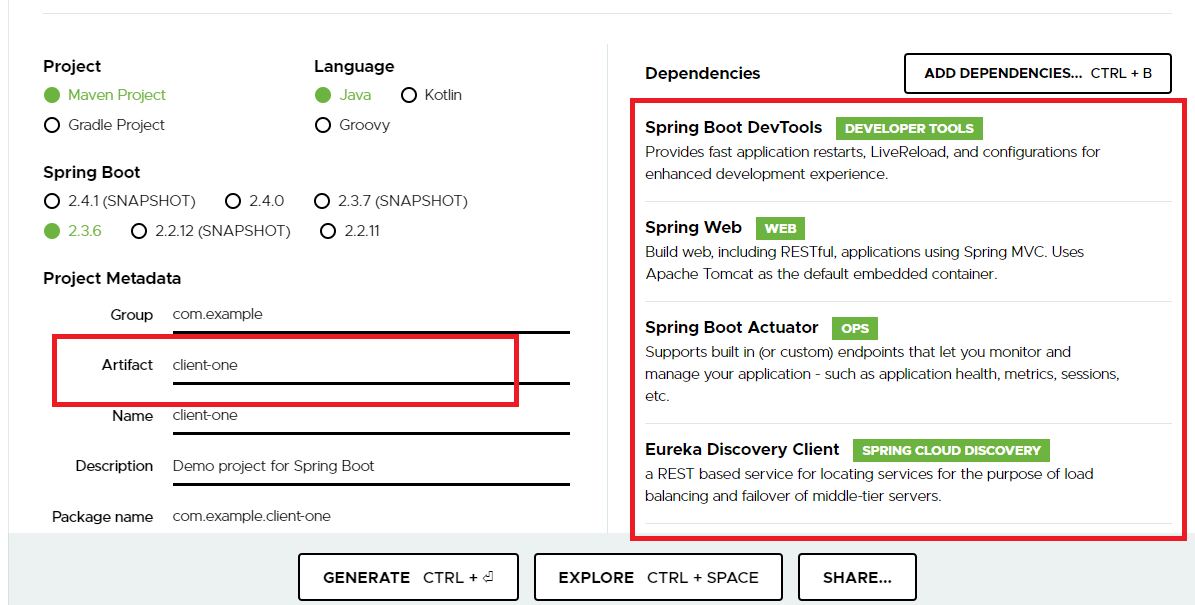
Project Setup

We are create 2 projects

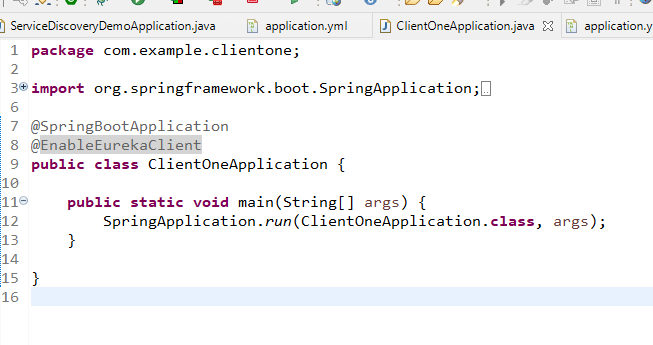
1. client-one
2. client-two

Client-one will have 2 services that returns some string value

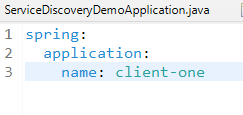
Client-two will use Circuit breaker configurations to trip the circuit for failure services.



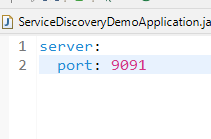
ClientOneApplication



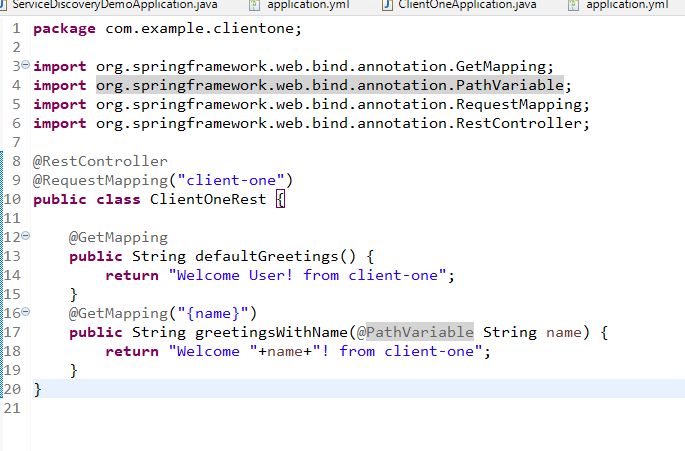
bootstrap.yml



application.yml



ClientOneRest.java



Note: Register client-one with eureka

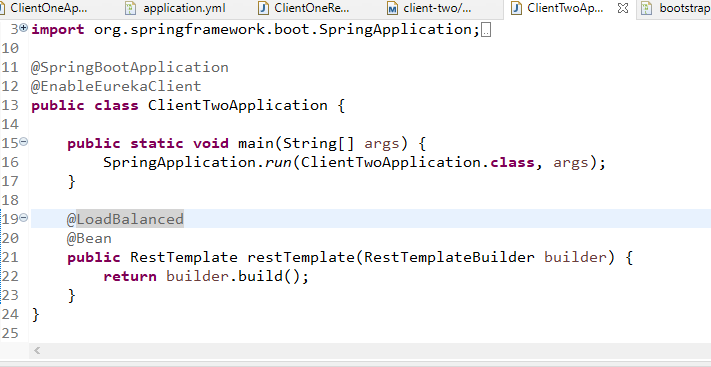
Client two application: This will call client one services but uses circuit breaker pattern where it can open the circuit if number of failures are greater than the threshold.

pom.xml

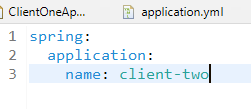


Note: resilience4j and spring aop you can get from Maven repository

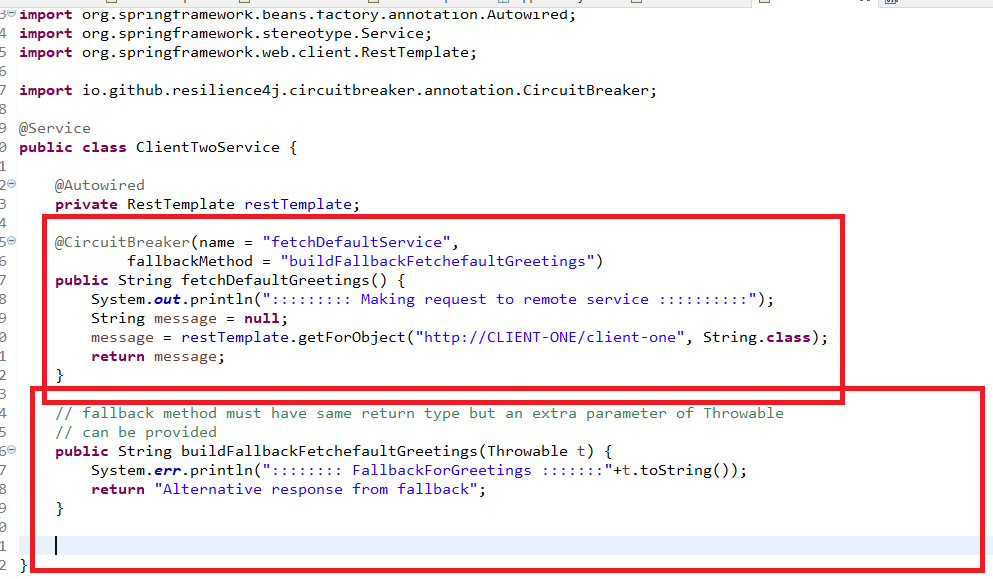
ClientTwoApplication.java



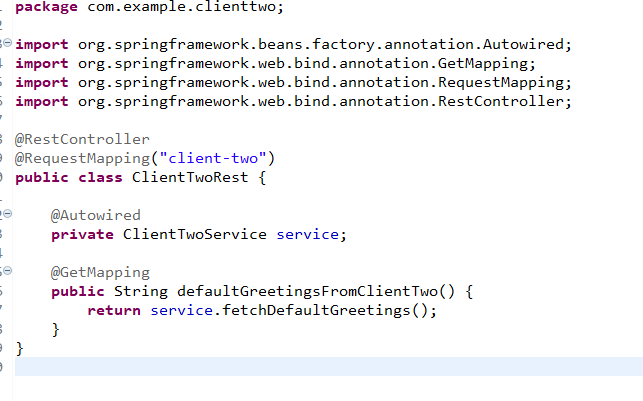
bootstrap.yml



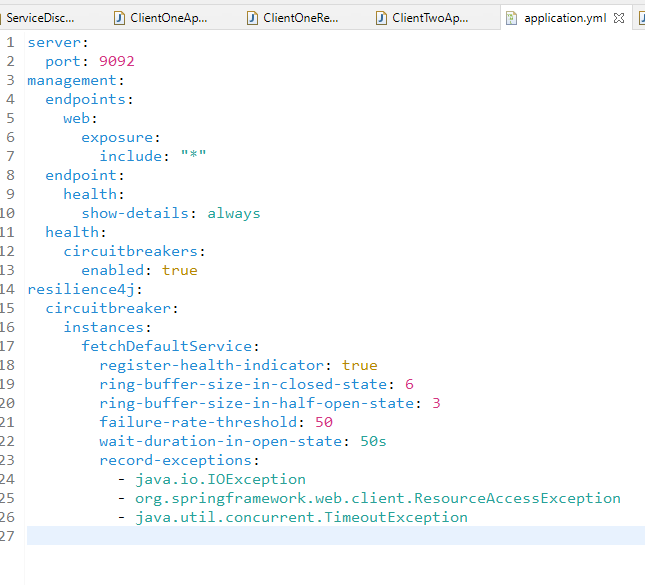
ClientTwoService.java



ClientTwoRest.java



application.yml

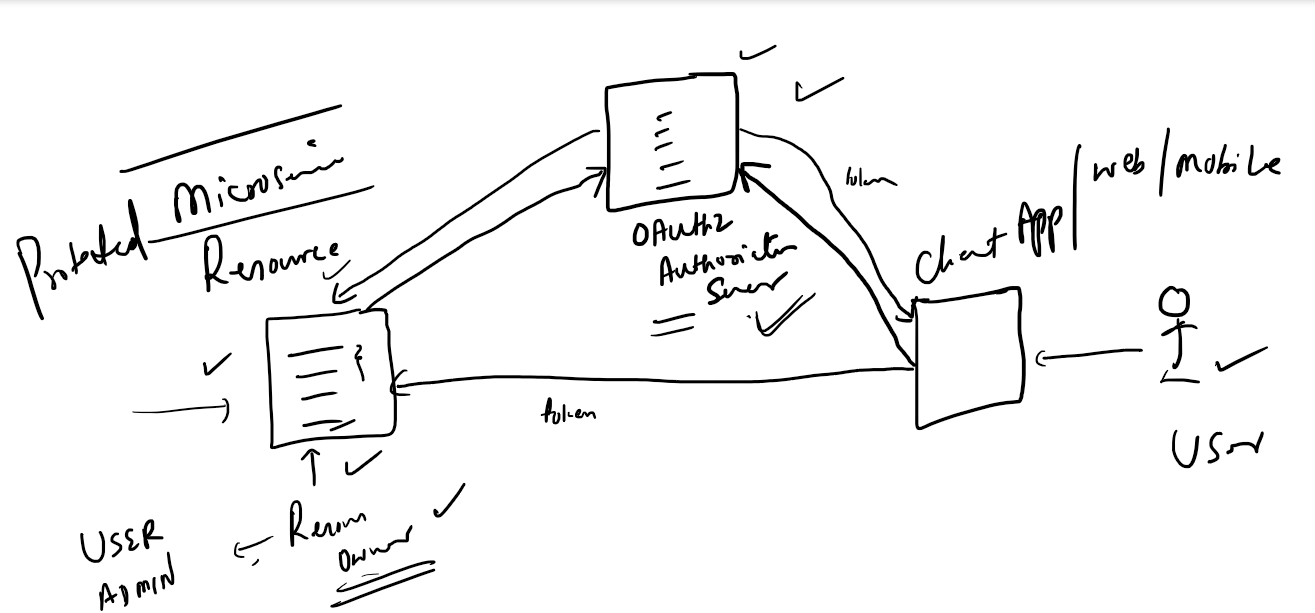


Output: health check



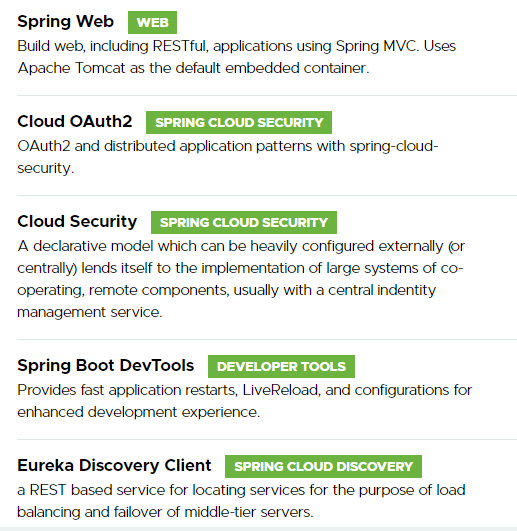
Spring Security:

OAuth2 token based authentication approach to authenticate users & it ensures that each microserives that are receiving the user request doesn’t need to send credentials on each call.

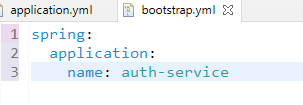


End user will use some client application that is registered to give access to the user, but the tokens will not be having only end user credentials it will also have application credentials also.

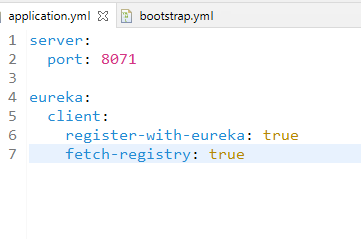
Authorization Server



bootstrap.yml

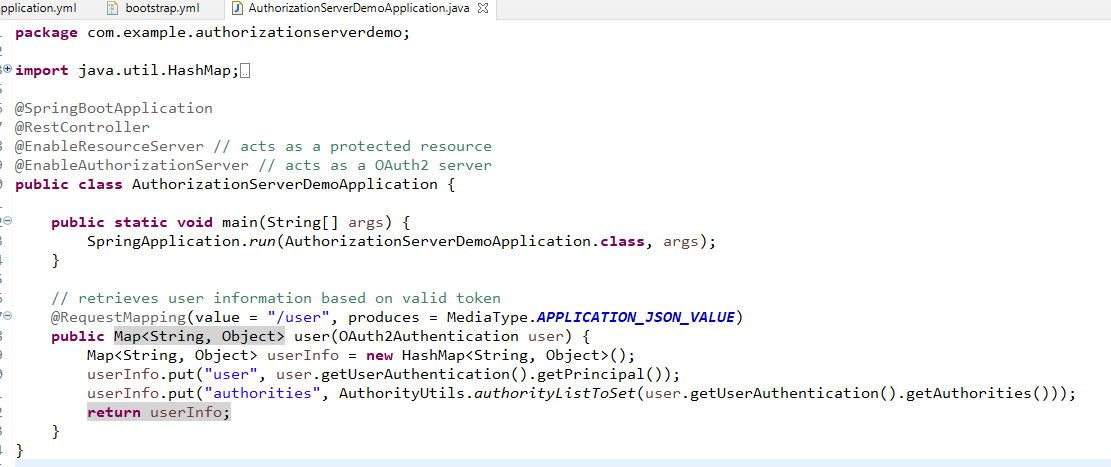


application.yml



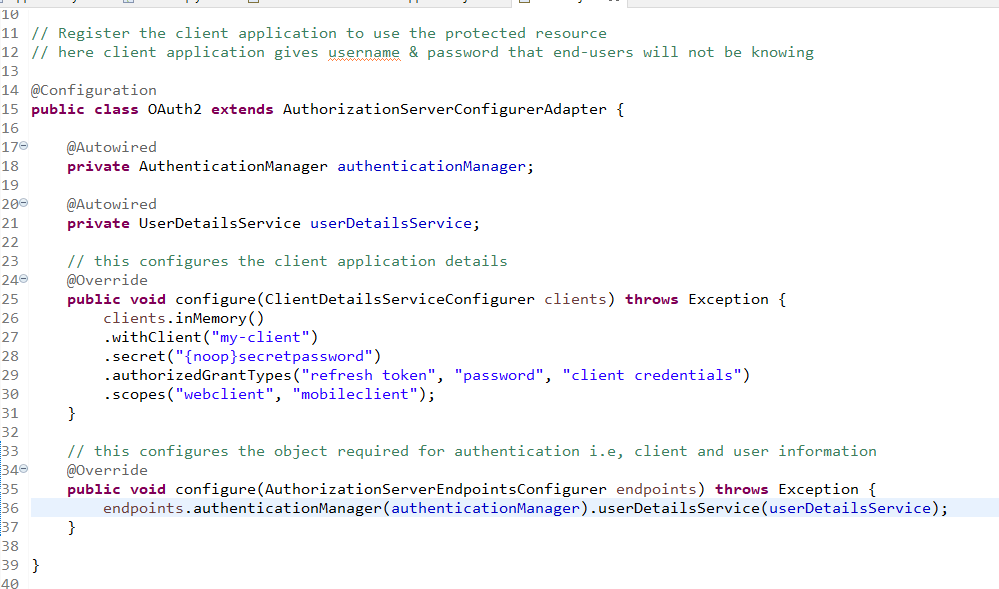
Creating an Authentication Server

Creating an endpoint the is called by protected resources to validate the user & retrieve the user information



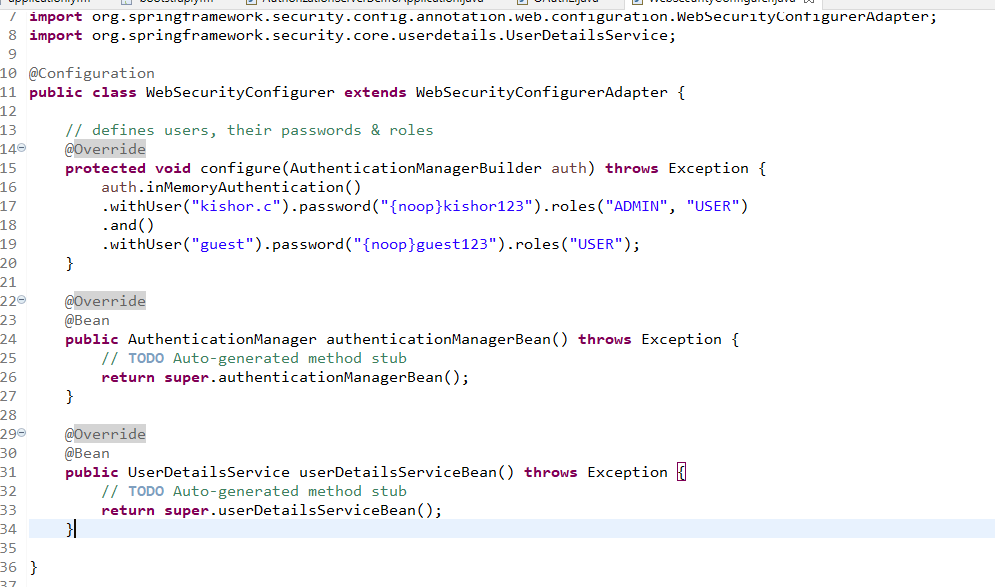
Registering the applications (client-application) with id & password which can use the protected resource

OAuth2.java



withClient & secret: this provides name of the application that we are registering along with the secret that will be used when the application(my-client) calls our OAuth2 server to receive an access token.

Configure the users using my-client application

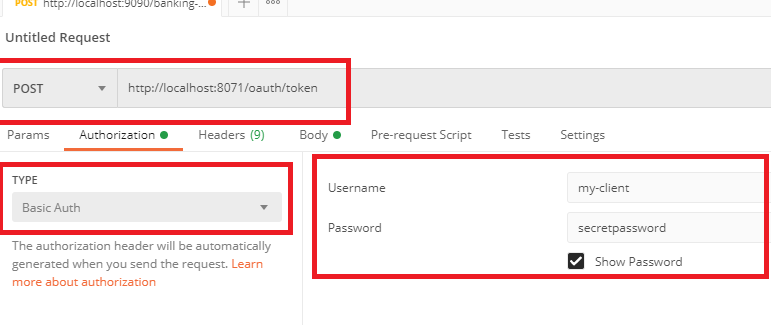


We have client application name: my-client and password secretpassword, this is something which the application client will have and as an end user he provides his credentials, firstly when he enters the credentials the request will be sent /oauth/token end point.

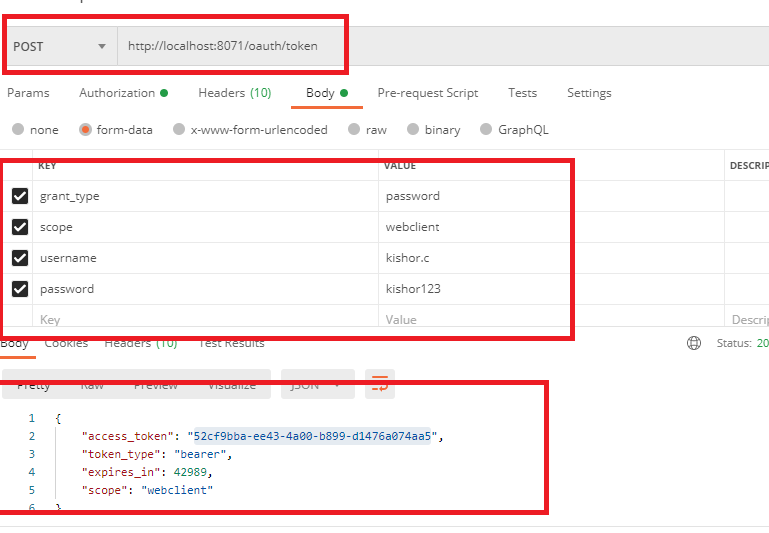
Note: this end point gives the application a token that will be internally sent in the request header in every request when the end user tries to access the protected resource

Postman:

In postman you must enter client application credentials in the Authorization header & then in the body you must send user credentials along with other application information.



The above screenshot only gives application details but not everything like whether it’s a webclient or mobileclient is provided in the form parameters



Here /oauth/token is the url used by the client-application when user needs a token, the token is passed in the HTTP request header in each call to the microservice & microservice will send the token to oauth server to validate the token by using /user end point.

Microservice must use the below end point to send the token to the OAuth server which receives from the client

