Course TOC

1. Spring Microservices
2. React.js

Microservices: they are small independent service which can be developed, test & deployed independently

Service Discovery: It is a program that registers the microservices, in Spring there’s a Eureka Server that acts like service discovery

Discovery Client / Microservices: These are the programs which registers in the service discovery, in Spring there’s Eureka Client that takes care of all the common jobs of the microservice like:-

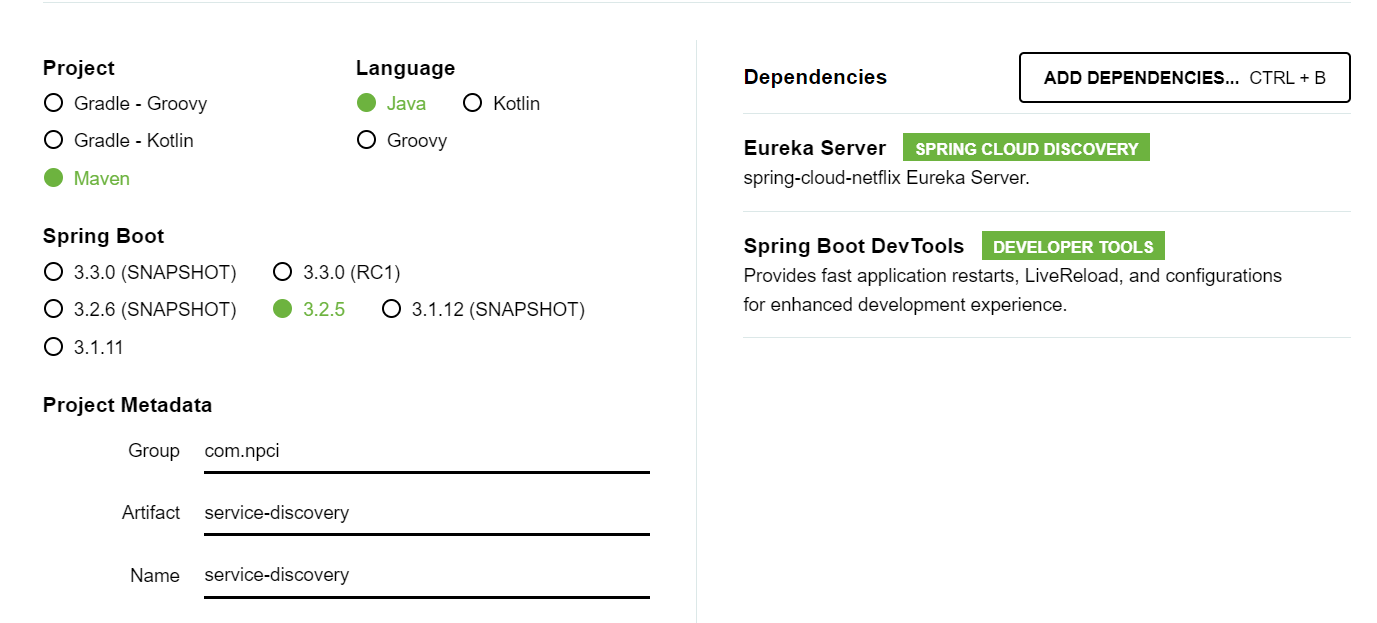
1. Automatically registering in the Eureka Server
2. Constantly pinging to the Eureka Server about its health status every 30s
3. Registers to the Eureka Server by default searching in 8761

Pre-requisites

1. Spring Boot 3.x
2. JDK 17 or later

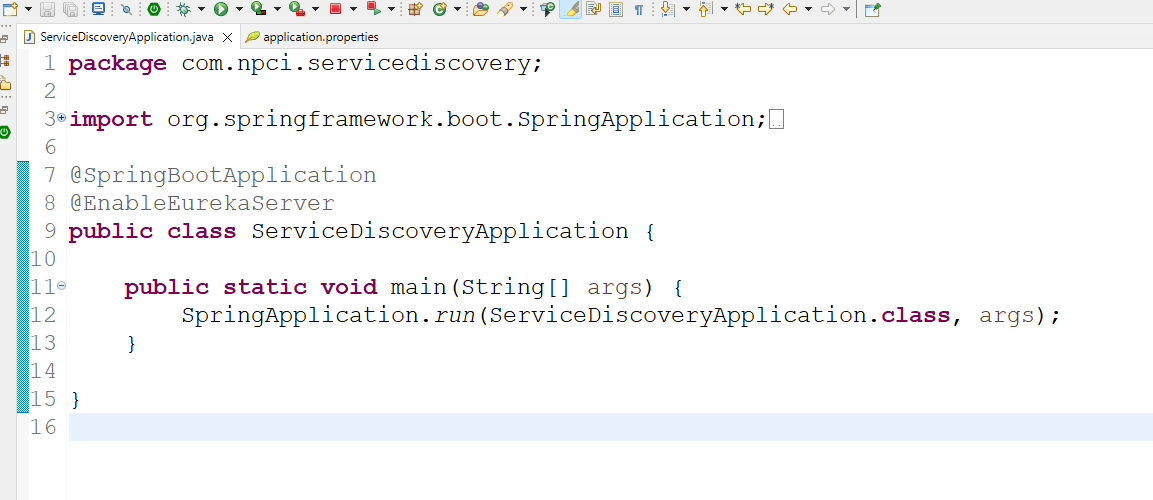
Creating service discovery

1. Eureka Server

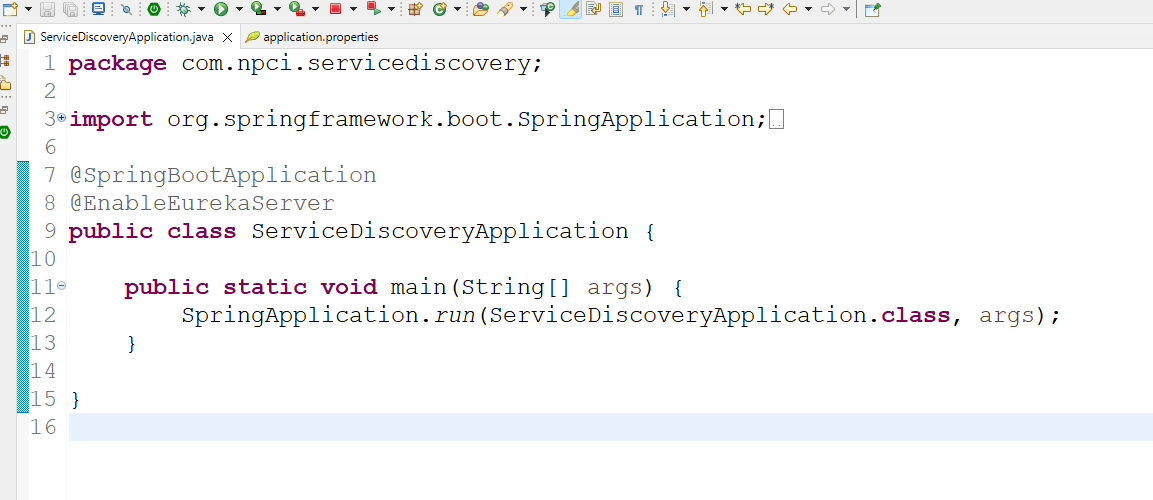


@EnableEurekaServer: This annotation enables the program to behave like service discovery & provides all the service discovery features, like registering & deregistering the microservices, recording the instance-id & physical address of the microservices, providing the UI dashboard to easily watch the registered microservices from any location

Note: Eureka Server downloads Eureka Client library also, we need to disable some client features that are automatically enabled by the spring boot like register with eureka & fetch from registry



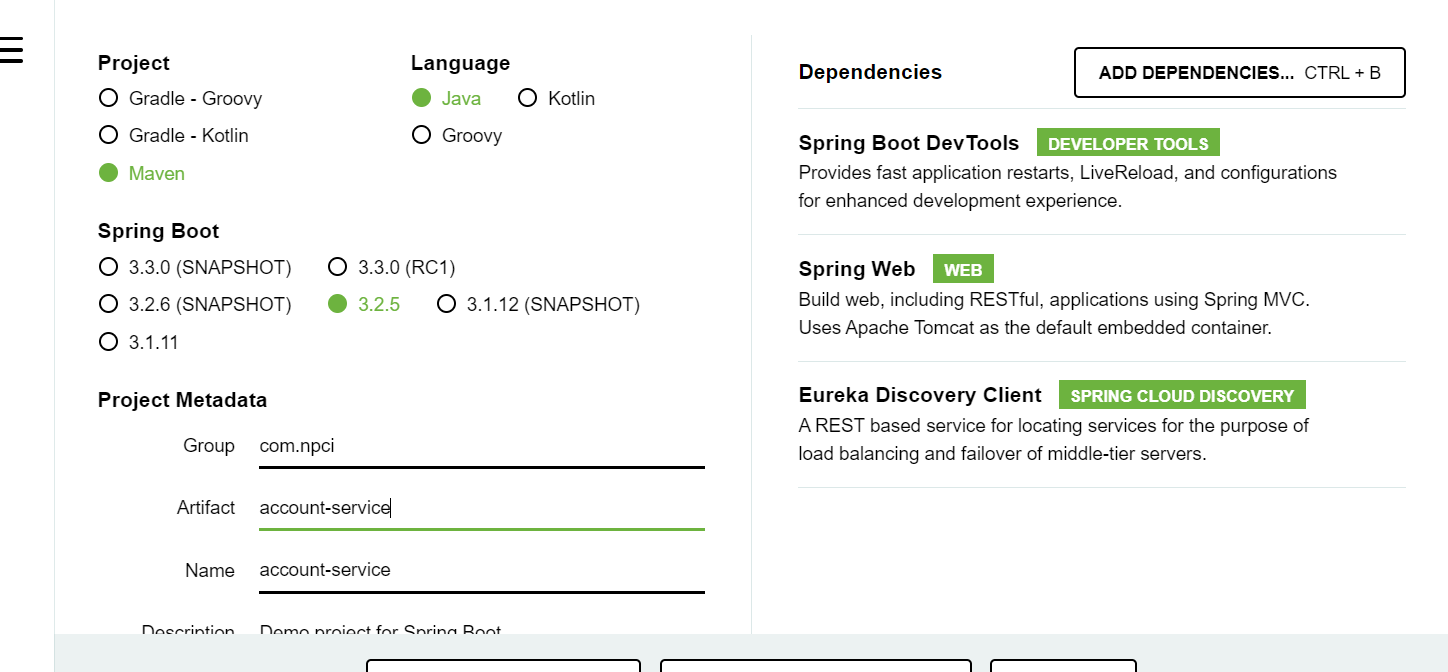
application.properties



Creating Microservices

Library:

1. Eureka Client
2. Web
3. Devtools

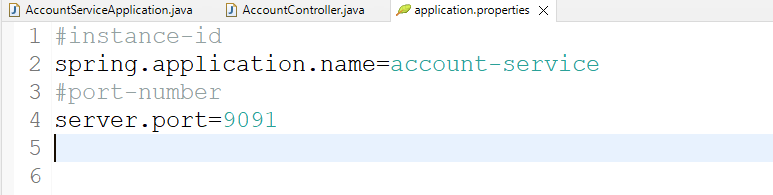


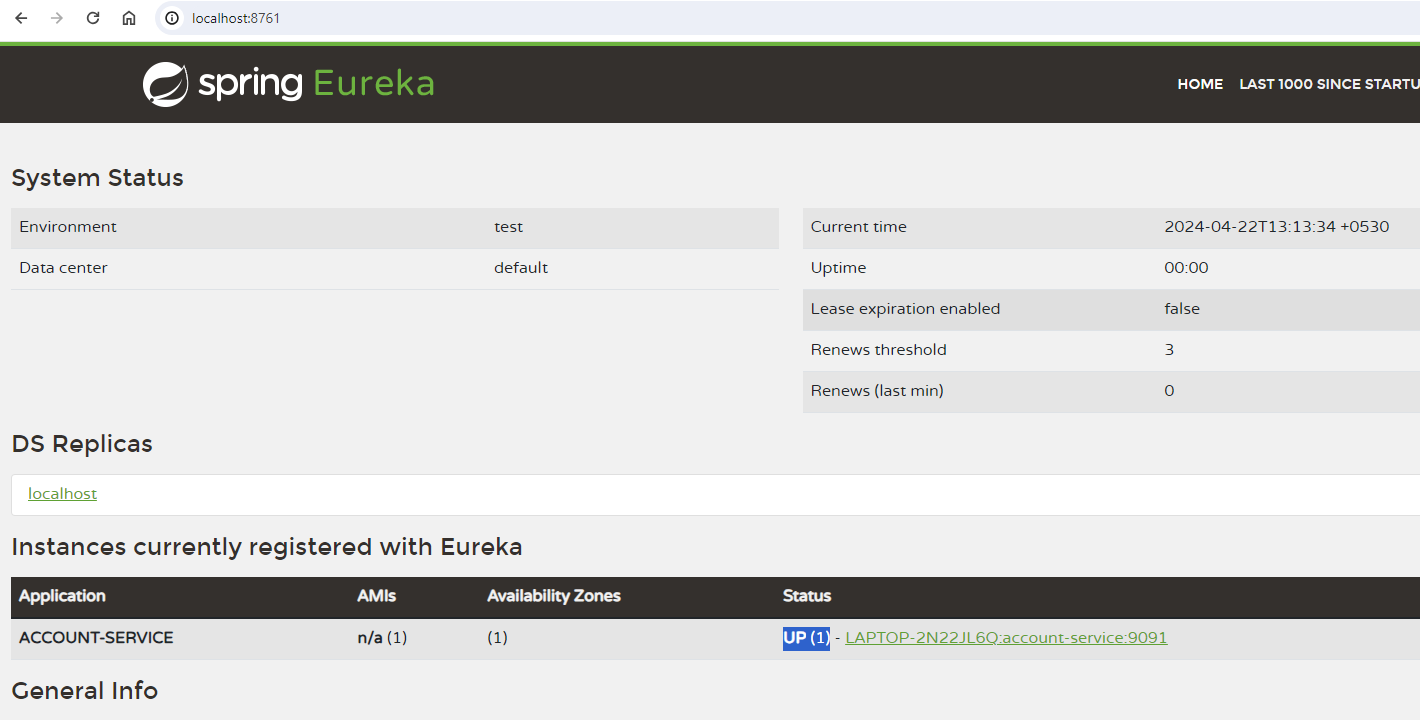
Note: This program automatically registers in the Eureka Server

AccountController.java



application.properties





Client Side Load Balancer:

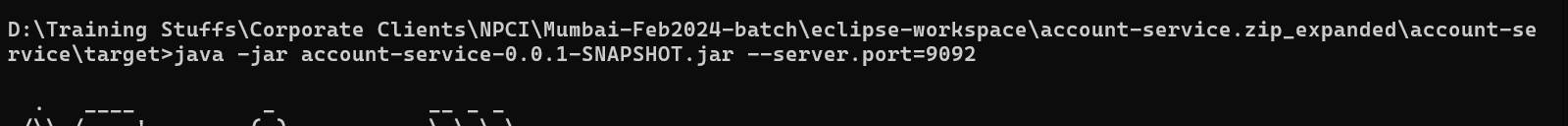
It is a program runs in the microservice to

1. Resolve the actual location of the microservice using the instance-id
2. Distribute the load across the multiple instances of the same instance-id if there are more than one instance.

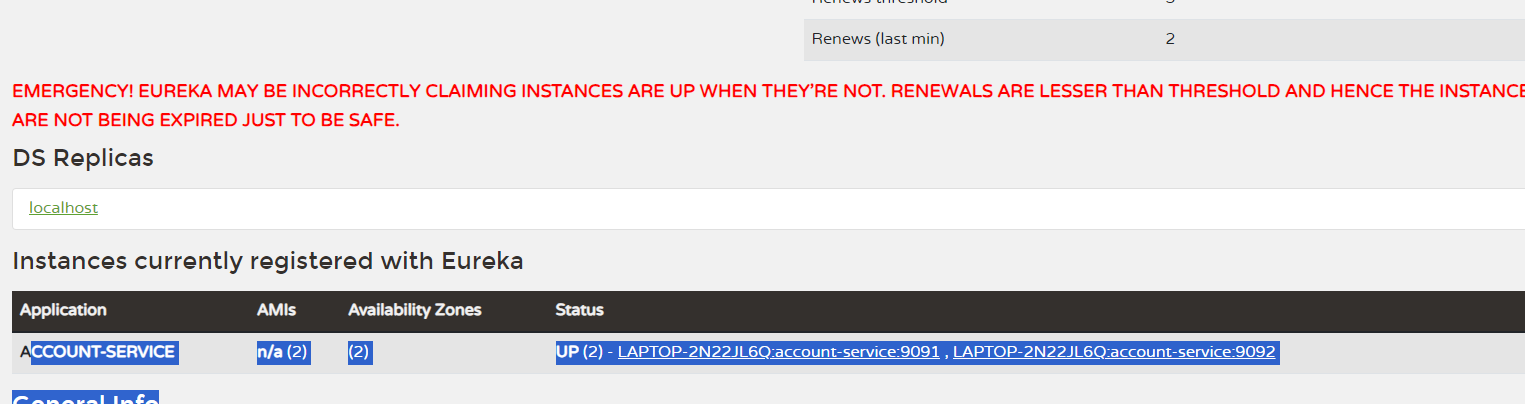
@LoadBalanced: This annotation creates the client side load balancer

How to create multiple instances:

we must have jar file to run the same program in different port number using terminal, but from eclipse you can run the same program only once



You can see 2 instances of account-service in the eureka dashboard



How to communicate from one microservice to another

a client microservice must use the instance-id of the remote microservice, to communicate we can use either

1. RestTemplate [or]
2. Feign Client: It is simplified approach than RestTemplate

RestTemplate: It is used in the client program to access the remote service, it provides inbuilt methods to make HTTP calls via GET, POST, PUT & DELETE, these methods would convert the JSON response to the java objects automatically based on the parameter

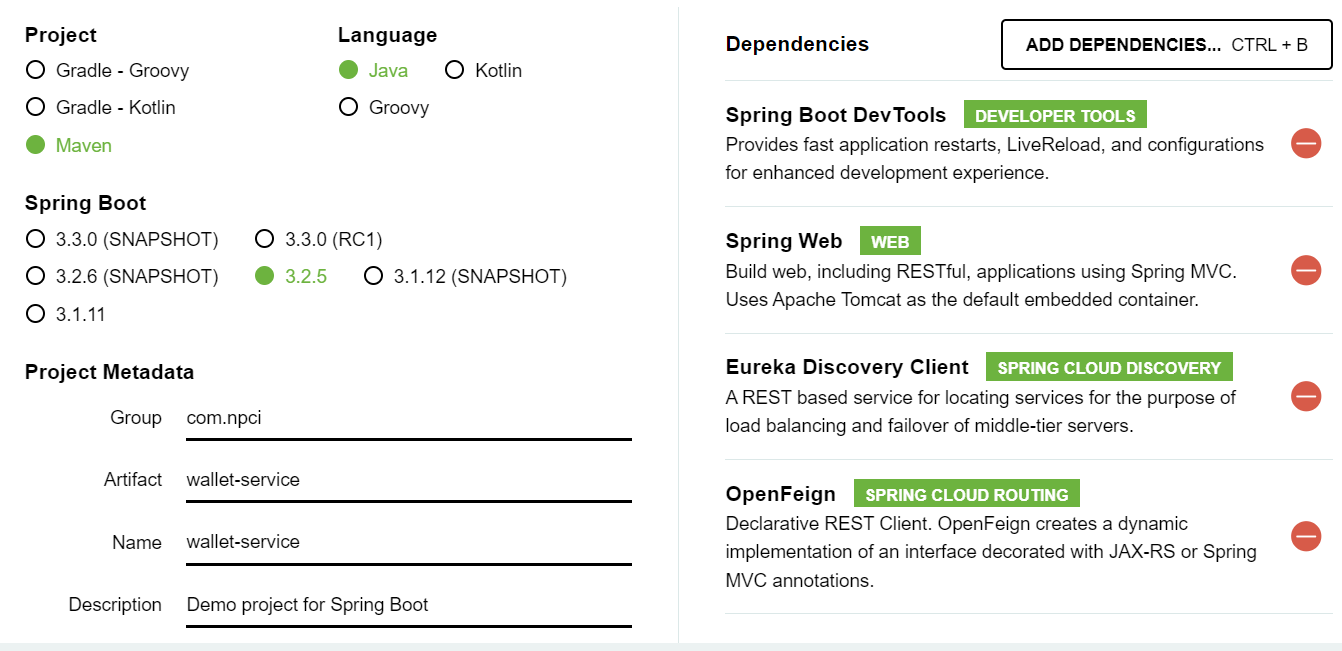
We must create RestTemplate with LoadBalanced attached to it as below:

@LoadBalanced  
@Bean  
public RestTemplate template() {   
 return new RestTemplate();  
}

@LoadBalanced enables rest template to send request that is distributed across the multiple instances

We need to autowire the RestTemplate wherever we want to send request to the remote service  
@Autowired  
RestTemplate rest;  
  
rest.getForObject(URL, String.class): This sends GET request to the remote service & converts the response in string format  
rest.getForObject(URL, Account.class): This sends GET request to the remote service & coverts the response in the form of Account object

Wallet Microservice: This will access Account Microservice

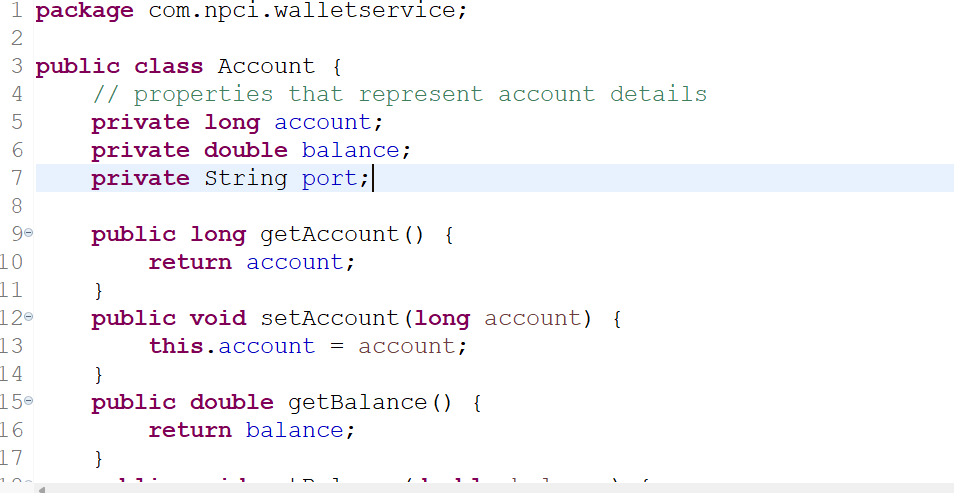


Note: OpenFeign is the library which can also make HTTP calls, but it has inbuilt load balancer, this is provided by spring cloud to simplify the HTTP calls, better than RestTemplate

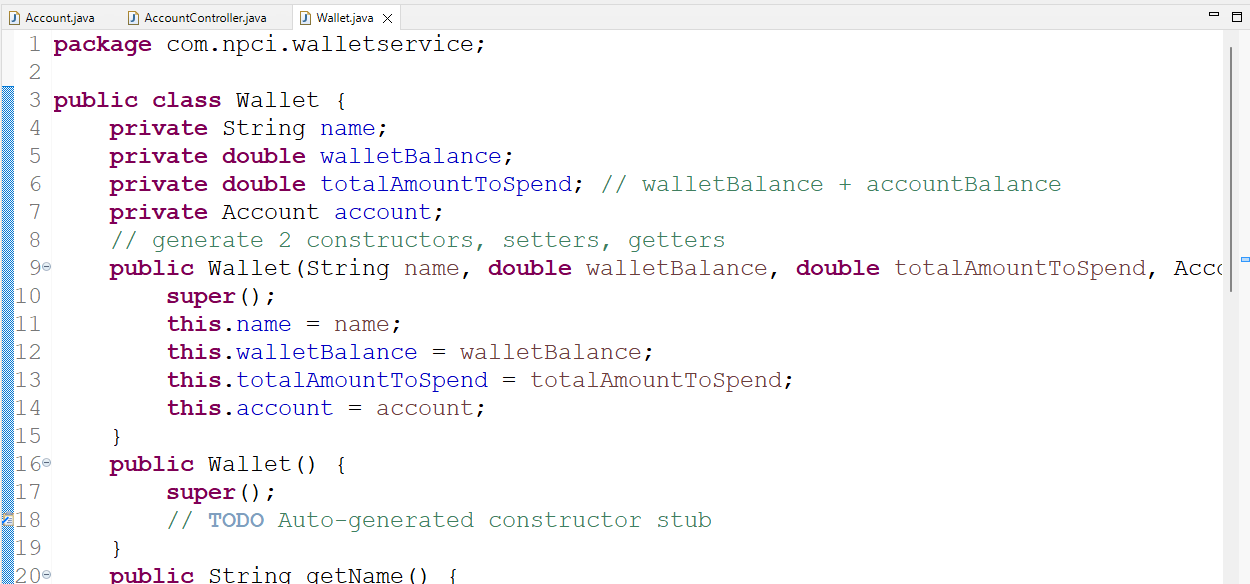
We need to create following classes

1. Account: To represent account details
2. Wallet: To represent account & wallet details
3. WalletService: To communicate with the remote service
4. WalletController: To handle the request from the user & call the WalletService

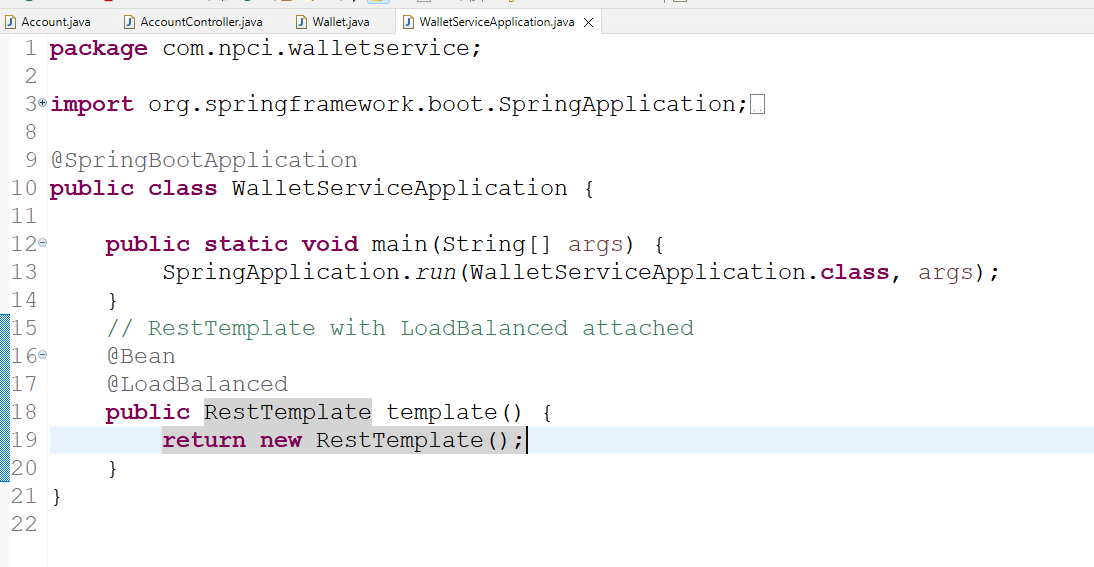
Account.java



Wallet.java

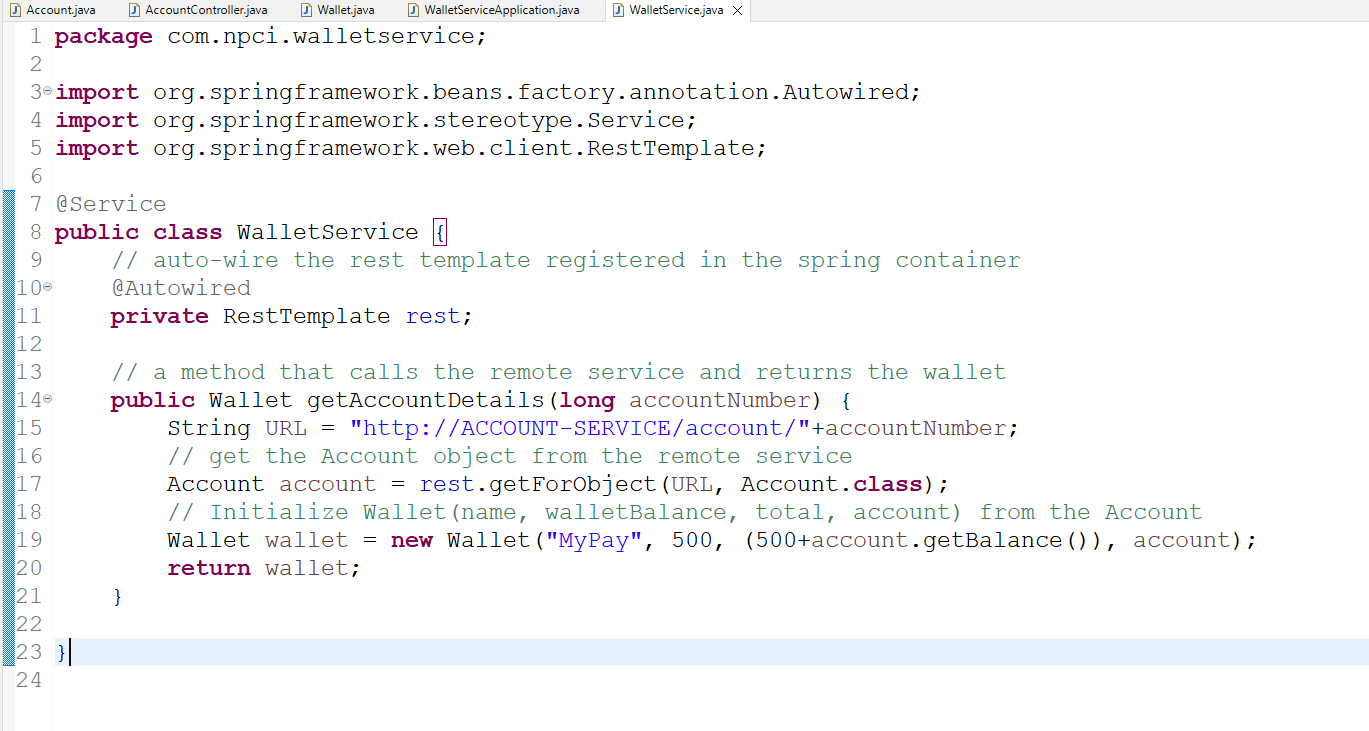


Create RestTemplate & register in the spring container



Autowire the RestTemplate in the Service layer and return the wallet details by extracting the Account details

WalletService.java

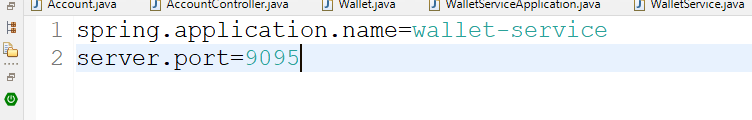


Controller to handle the user request and call the WalletService

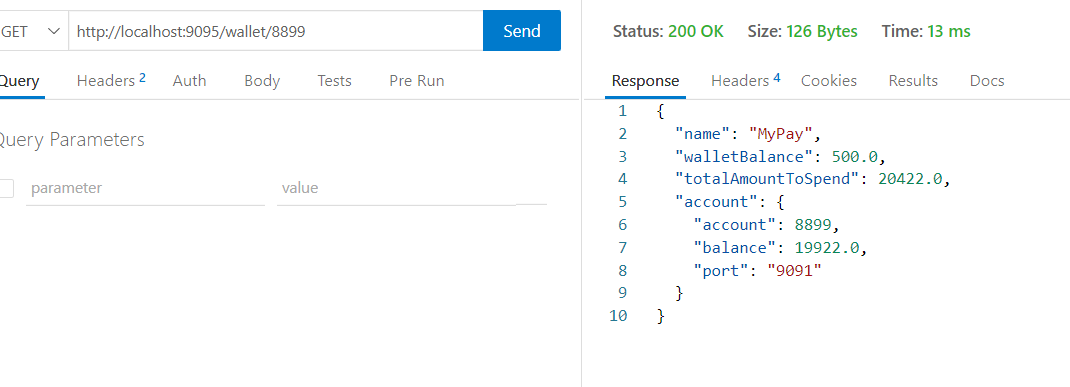
WalletController.java



application.properties



Output:



Drawbacks of RestTemplate

1. It is an older API
2. It can’t be reused like URL need to be repeated

Feign Client

It is used to make remote calls, it is released when spring microservices was released

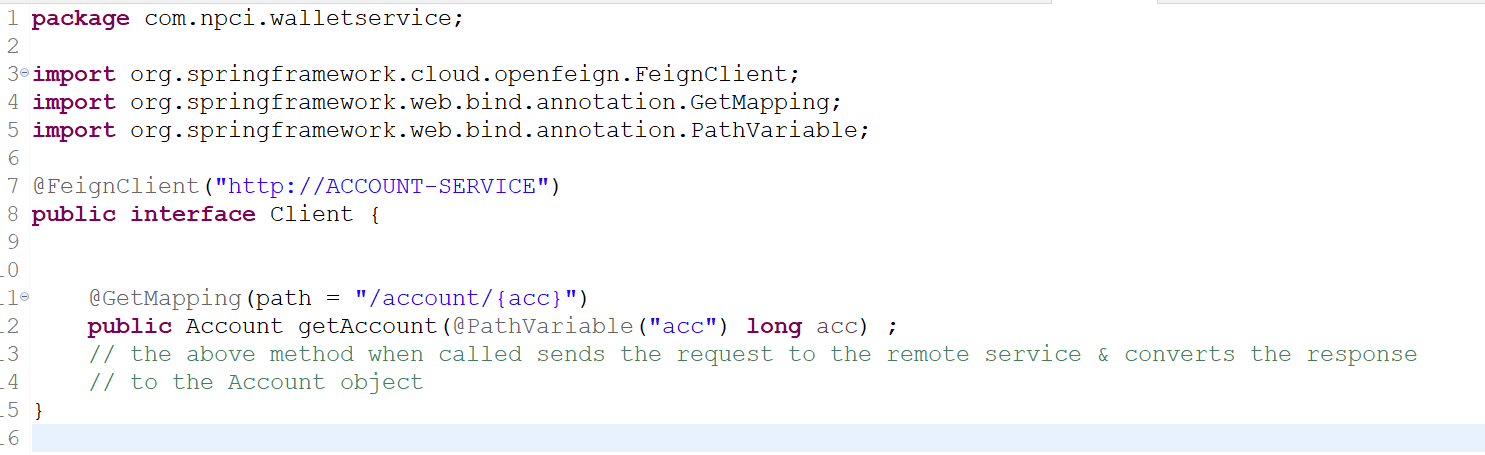
* It helps you to reuse the remote calls
* It internally uses the client side load balancer - you don’t need to use @LoadBalanced

You need to create a reusable interface which helps to create a reusable remote calls

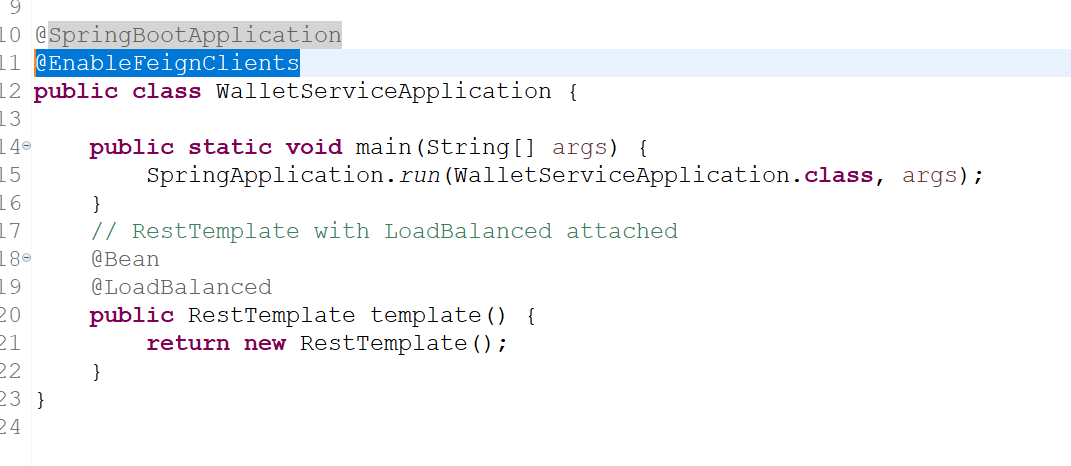
@FeignClient(“http://ACCOUNT-SERVICE”)  
interface Client {   
 @GetMapping(“/account/{accountNumber}”)  
 public Account getAccount(@PathVariable(“accountNumber”) long accountNumber);  
}

Calling getAccount sends a GET request to the remote microservice, the return type tells the response must be converted to which object ex: JSON to Account object  
Note: You don’t have to implement this interface, spring boot automatically implements the interface we only need to use @EnableFeignClients to let spring boot implement the interface  
@EnableFeignClients: This scans all the @FeignClient interface & lets spring boot to implement that interface & register the implemented object in the spring container

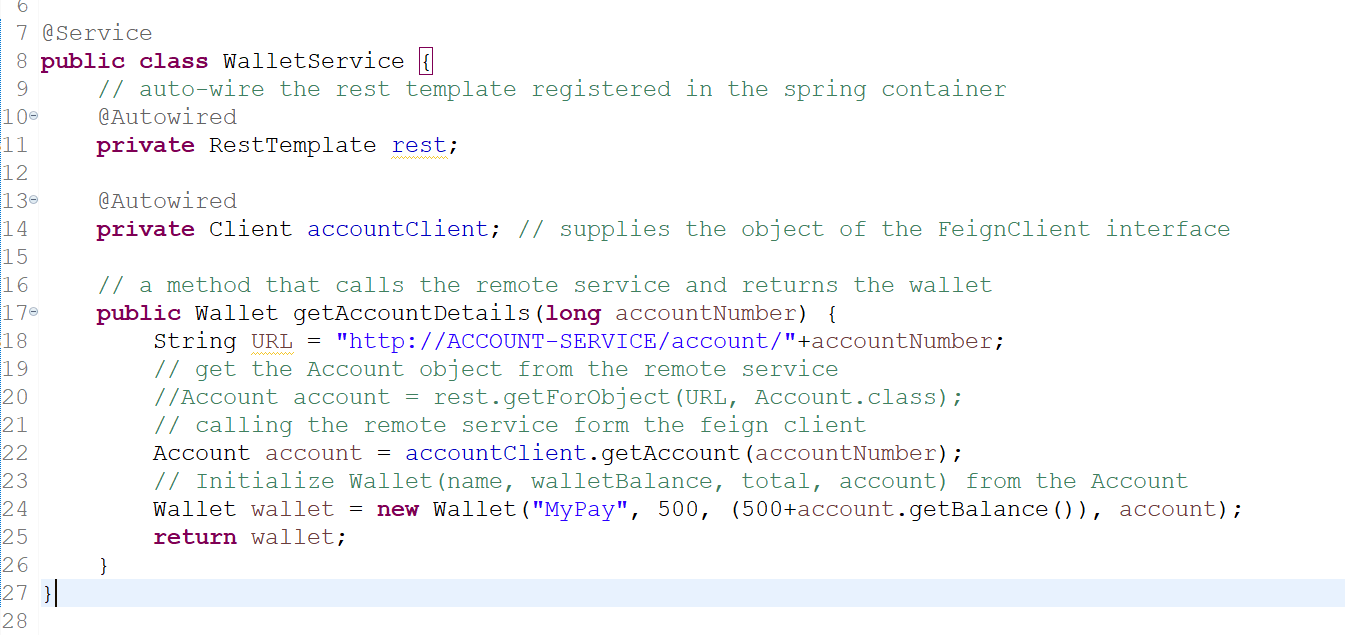
Client.java



We need to mention @EnableFeignClients in the main class so that this interface will be implemented using LoadBalancer behind the scene.



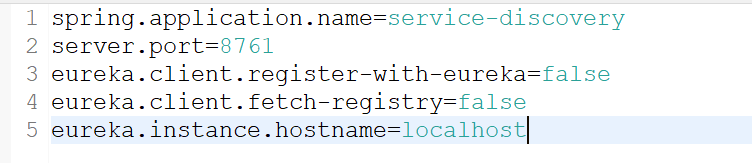
Change the service layer to use the FeignClient interface instead of RestTemplate



How to deploy the microservice in different machine

1. We need to create an EC2 machine
2. Service discovery application.properties must have some changes

application.properties

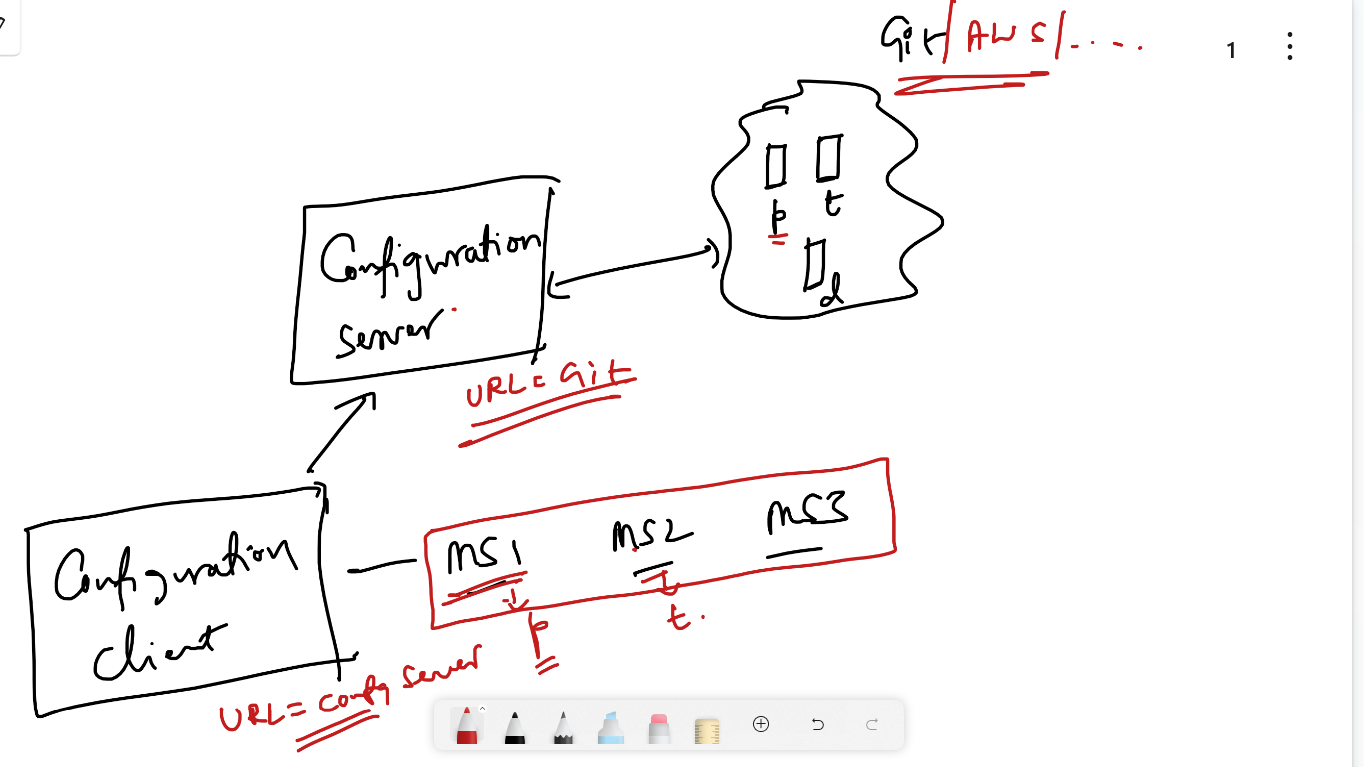


Distributed Cloud Configuration

It is mainly used when you want to share common configurations to the multiple microservices without changing the configurations locally and also when same microservice wants to load different configuration file for different environment (development, production, testing)

Spring helps you to use this type of configurations using 2 programs

1. Configuration Server
2. Configuration Client



Configuration Server: It is a program that connects to the centralized repository like GIT, SVN, AWS and etc.

@EnableConfigServer: This annotation creates a config server which connects to the centralized repository based on the URL provided in the property file & fetches the configurations for the client

Library you must use for server

Config Server

Configuration Client: It is a program that connects to the configuration server so that the config server will pull the configurations for the client, in the property file you will be mentioning the properties that connects to the Config Server & also which configuration file it needs

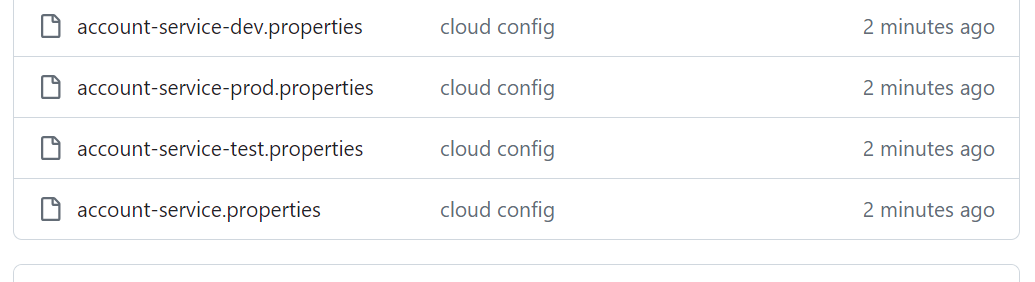
Library you must use for client is

Config Client

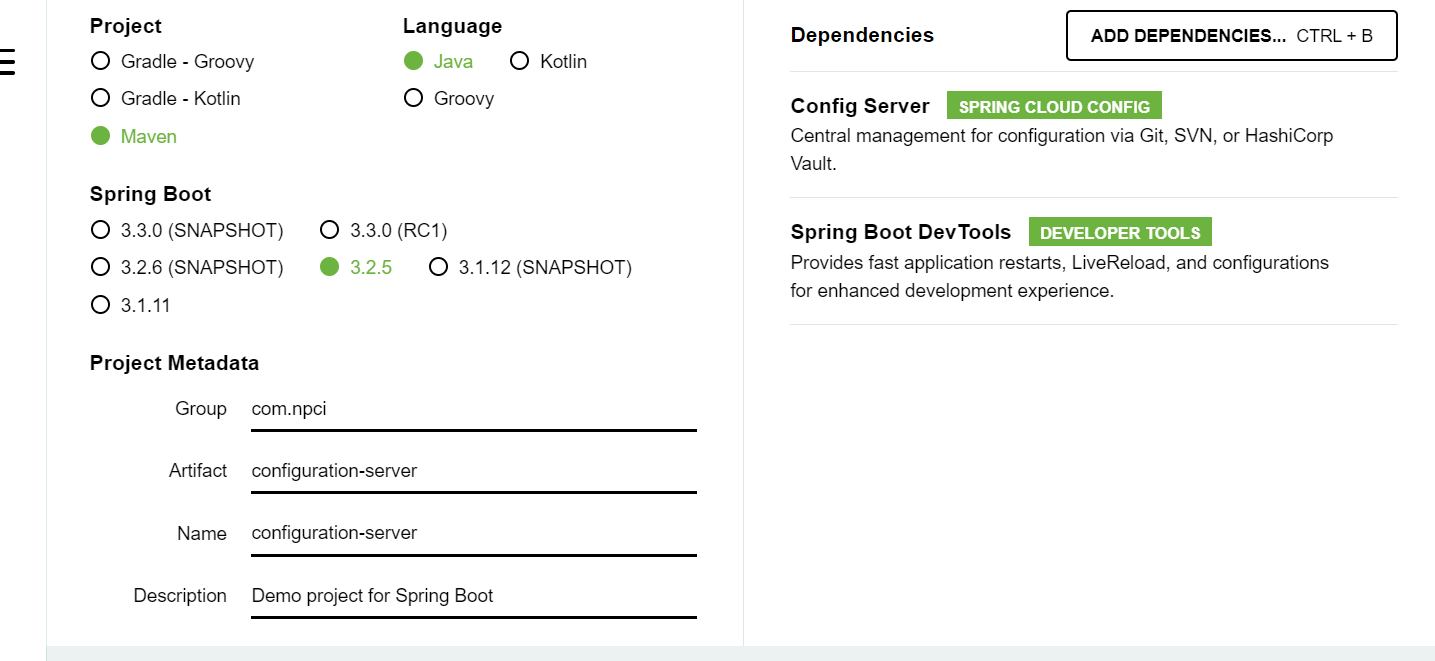
Steps to use the distributed cloud configuration

1. Push some configuration files to the GIT, separate the file names with -dev, -test, -prod i.e., hello-dev.properties, hello-test.properties, hello-prod.properties, hello.properties
2. Create configuration server to connect to the GIT
3. Create configuration client to connect to the Configuration Server & mention in which profiles(environment) you are running the program

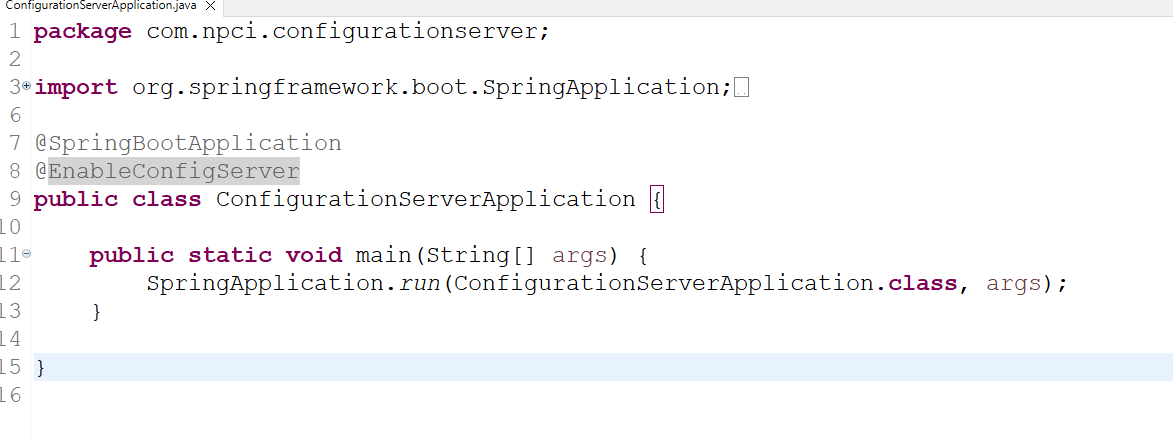
Creating configuration files in the GIT



Creating the configuration server program



Add @EnableConfigServer to the main class

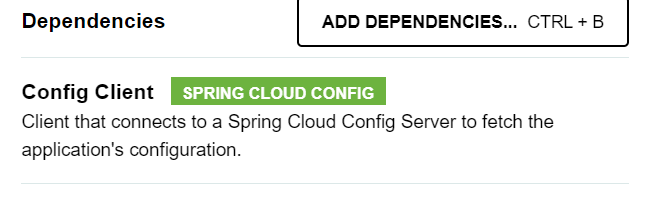


application.properties

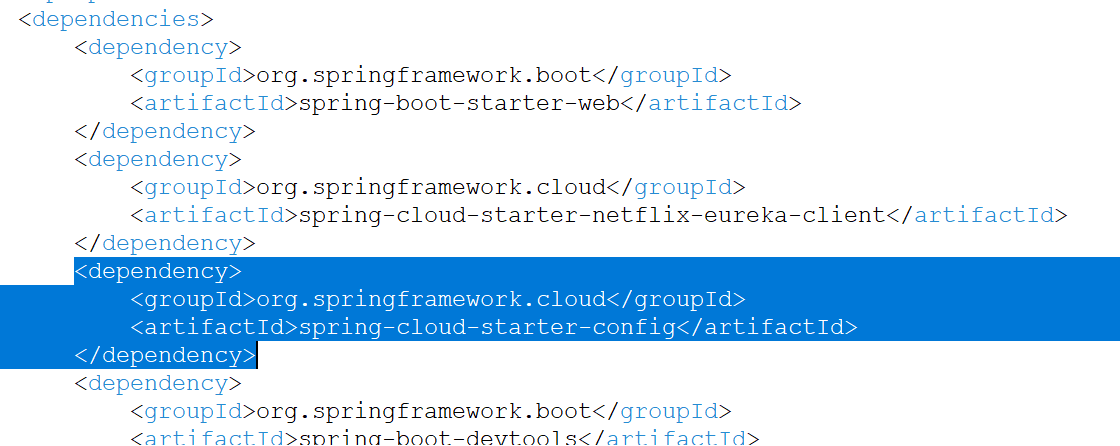


Now the client program should get the configurations from the configuration server by mentioning the configuration server URL & also which configuration file it needs

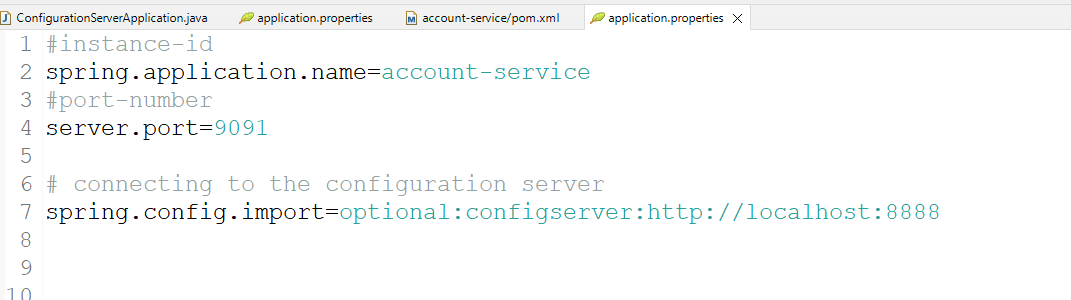
Firstly you need to add config client library



Update the account-service/pom.xml



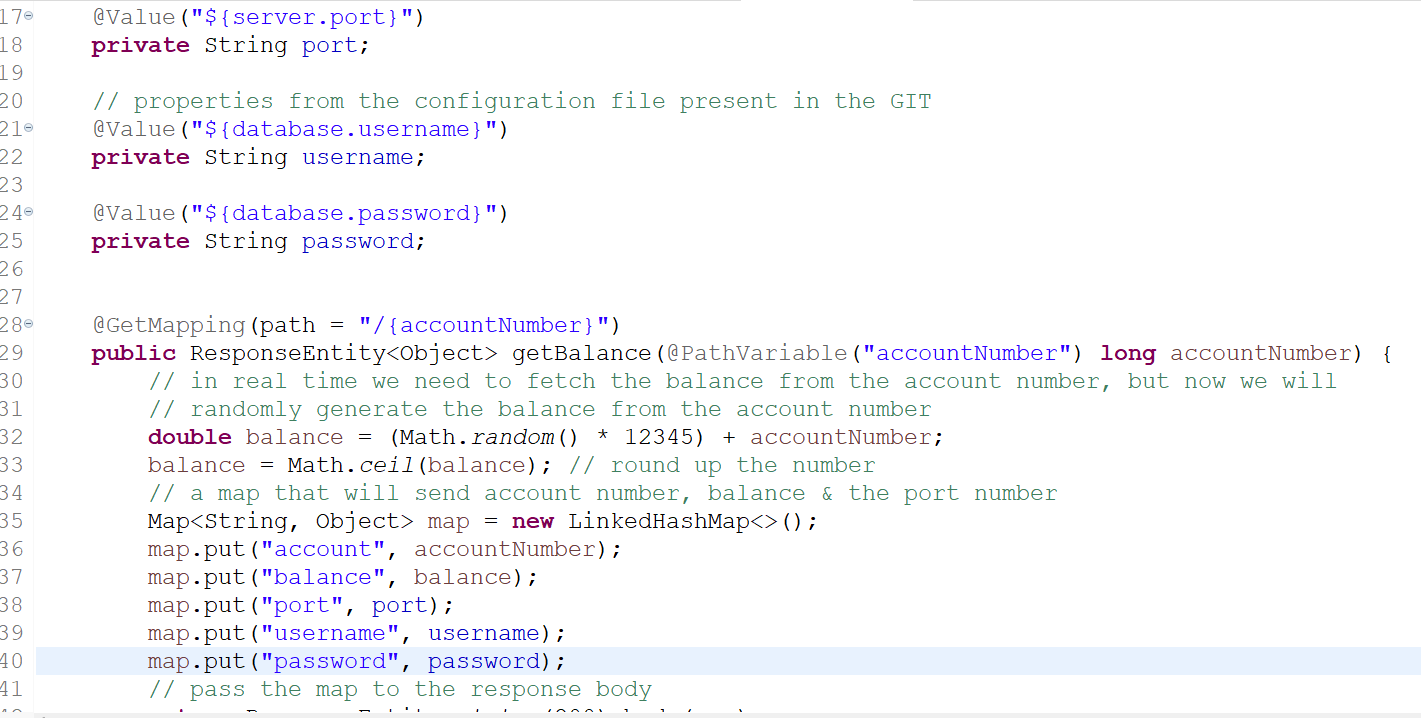
account-service/application.properties



By default it gets account-service.properites from the config server, however you can use a property called

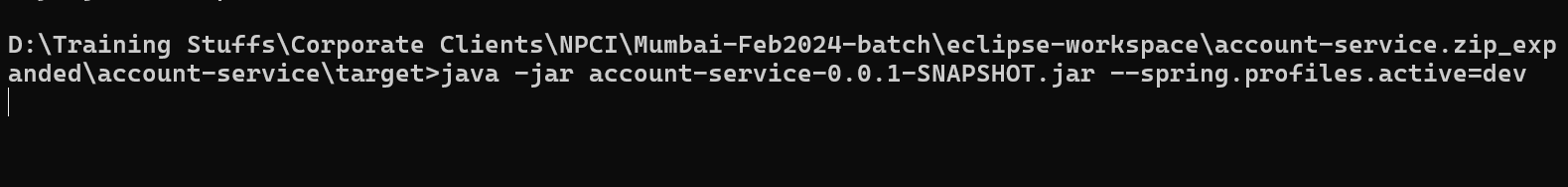
spring.profiles.active=dev # this gets account-service-dev.properties  
spring.profiles.active=prod # this gets account-service-prod.properties

All the property files have common properties like database.username & database.password hence you can have a webservice to read these data so that you will know which property file config client got



Now run these programs in this order

1. Service discovery
2. Config Server
3. Account Microservice : build the jar and run this program by providing spring.profiles.active



Day 2 Agenda

1. Encrypting & Decrypting sensitive data
2. Circuit breaker pattern
3. Reactive Programming using Web Flux
4. Spring Security

Encrypting & Decrypting sensitive data

Sensitive information’s must be encrypted so that end users must not able to understand, there will be a program which will decrypt the data

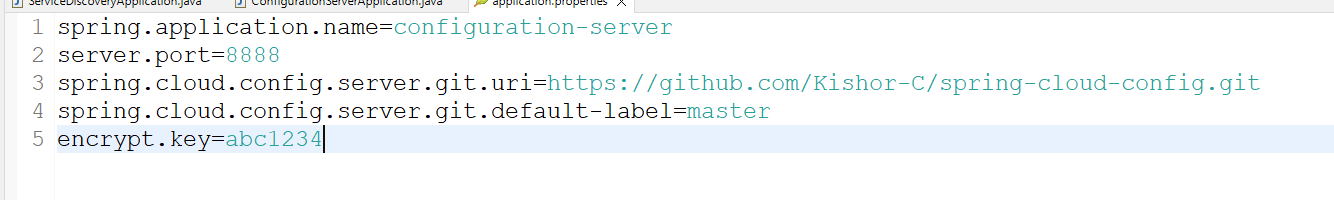
Configuration Server provides two url’s to encrypt & decrypt the url is

1. config-server-ip:port/encrypt
2. config-server-ip:port/decrypt

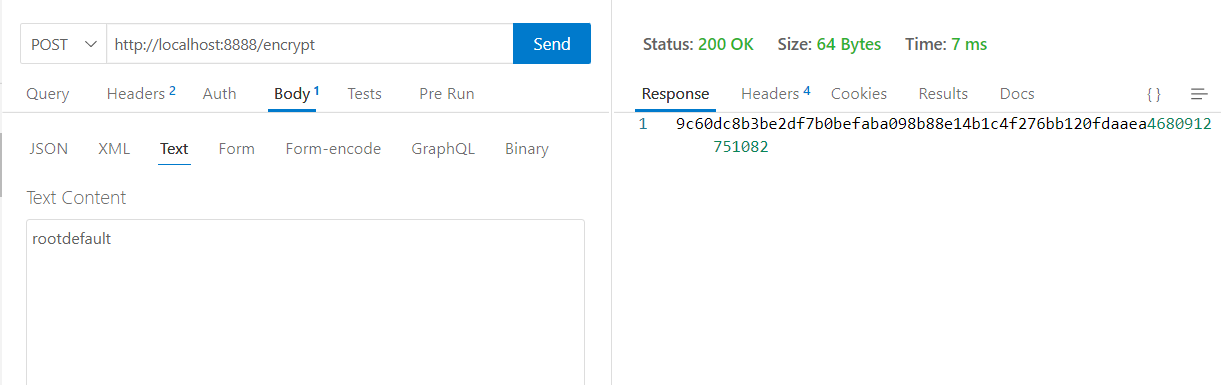
Configuration server needs a key to encrypt or decrypt

encrypt.key = abc1234

configuration-server/application.properties

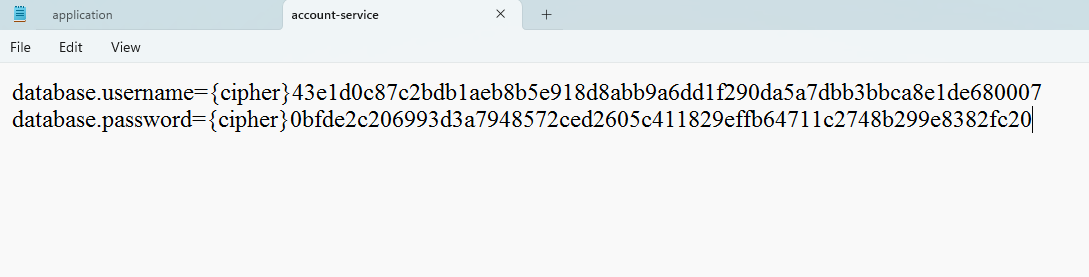


Now you can use /encrypt & /decrypt url to encrypt the data.

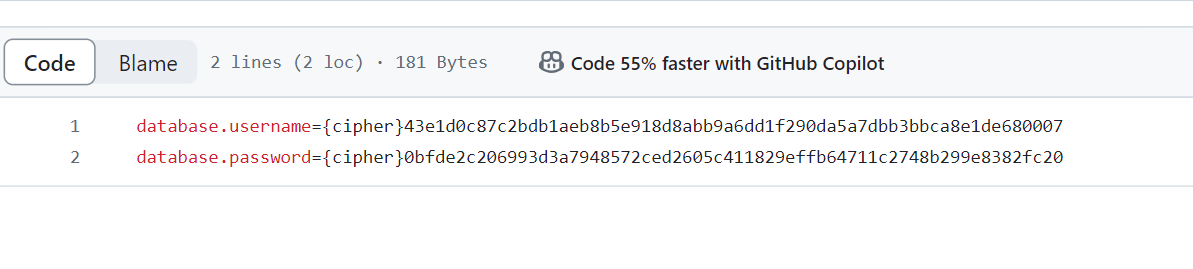


In the properties you need to use {cipher} beside the encrypted data so that the configuration server will decrypt the data and give them to the microservices

account-service.properties



Update all the properties in GIT



Circuit Breaker

It is a design pattern that is used to stop the cascade of failures across the microservices when any remote service is slow/down

It uses three states while communicating with the remote service

1. OPEN: It doesn’t send the request to the remote service
2. HALF\_OPEN: It is to decide whether the circuit should go to open or close state based on the failure rates
3. CLOSE: It sends the request to the remote service

Note: Automatically circuit breaker in open state goes to half open state after 60 seconds

Library we need to use for circuit breaker are:

1. Resilience4J: Gives the circuit breaker configurations
2. Actuator: gives the end points to see the circuit breaker status, failure rates, threshold
3. AOP: Takes care of providing alternate response when the circuit is open without invoking the code i.e., invoking a fallback method

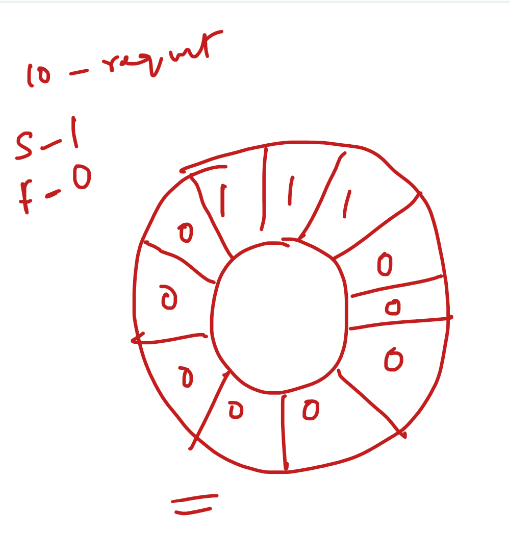
@CircuitBreaker: This annotation is used on the method which makes a remote call, this annotation will have a fallback method configuration so that fallback is called if remote service is unavailable or if circuit breaker state is open

@CircuitBreaker(name = “getAccount”, fallback = “getAccount2”)  
public Wallet getAccountDetails(long acc) { … } // actual method that calls remote service  
  
// fallback method must have the same signature with a Throwable parameter  
public Wallet getAccount2(long acc, Throwable t) { } // fallback method gets called automatically if remote service is down or circuit breaker state is open

In application.properties you will configure the failure rates, threshold, for the getAccount (name of the CircuitBreaker)

ex: resilence4j.circuitbreaker.instance.getAccount.threshold=50 #50% failure is the threshold

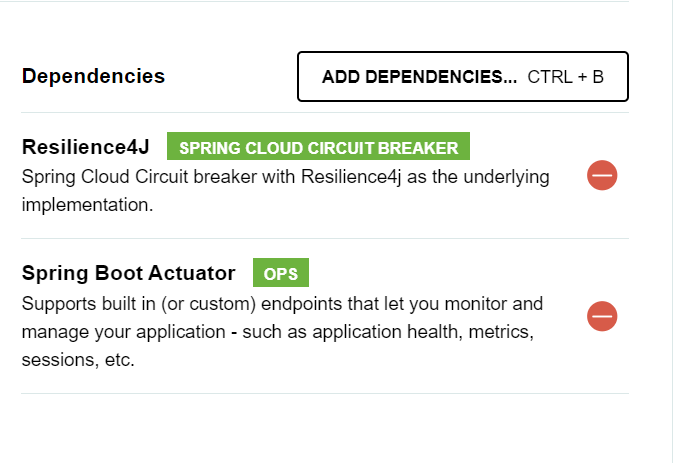
Reslience4j uses RingBit buffer to track the failures



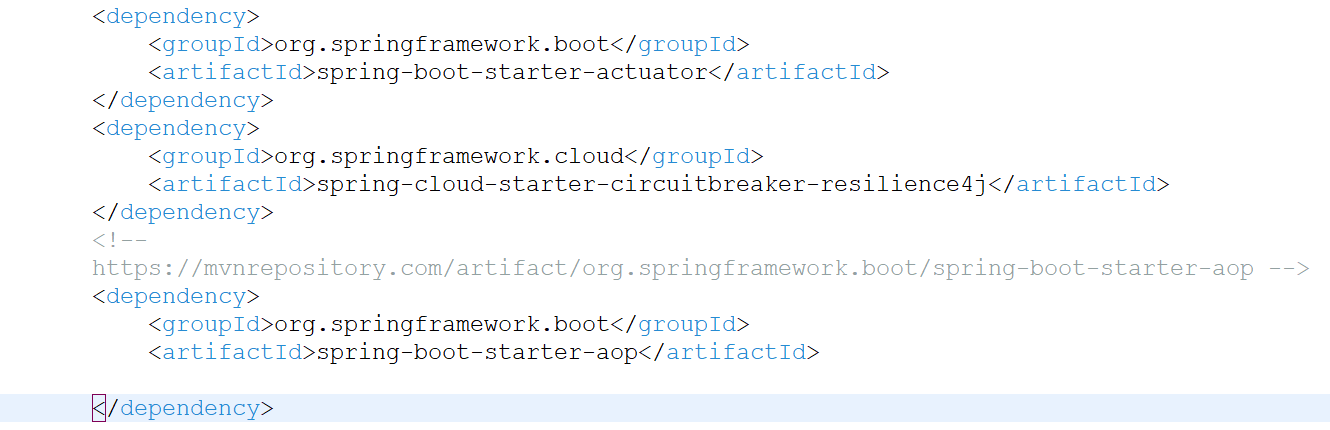
Ring bit buffer is a memory to track the success & failures, success will be entered with 0 & failures will be entered 1, based on these numbers it identifies the failure %.

Note: AOP may not be available in spring initializr, you need to get it from the Maven

Note: We need to configure the circuit breaker in the wallet service

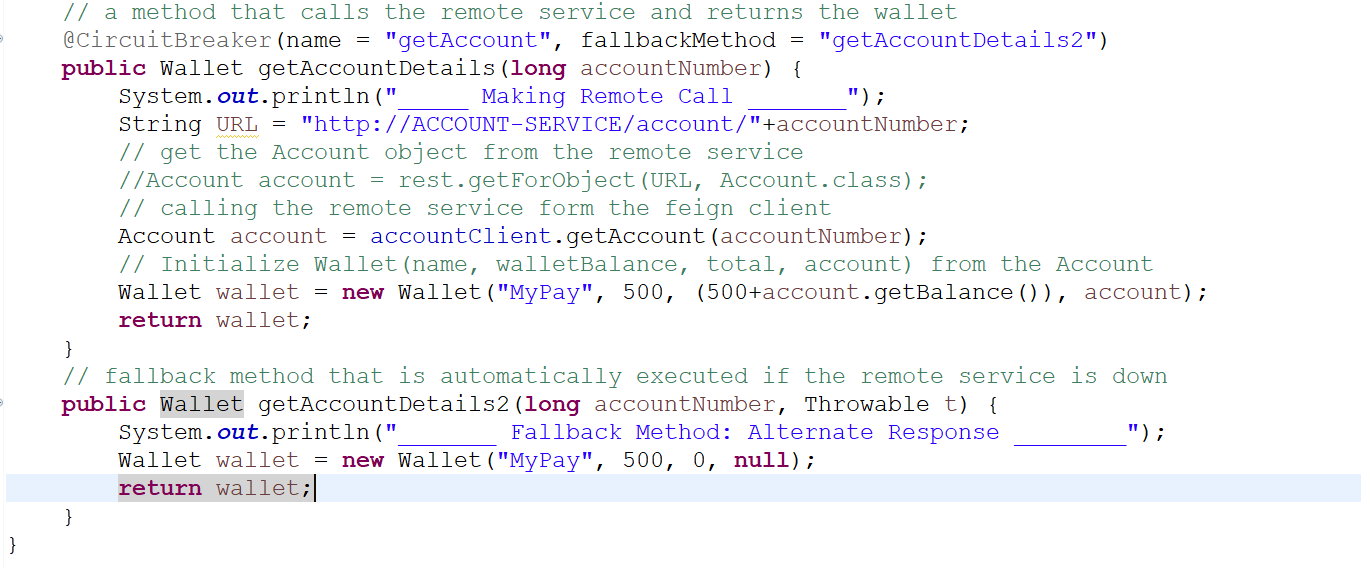


wallet-service/pom.xml

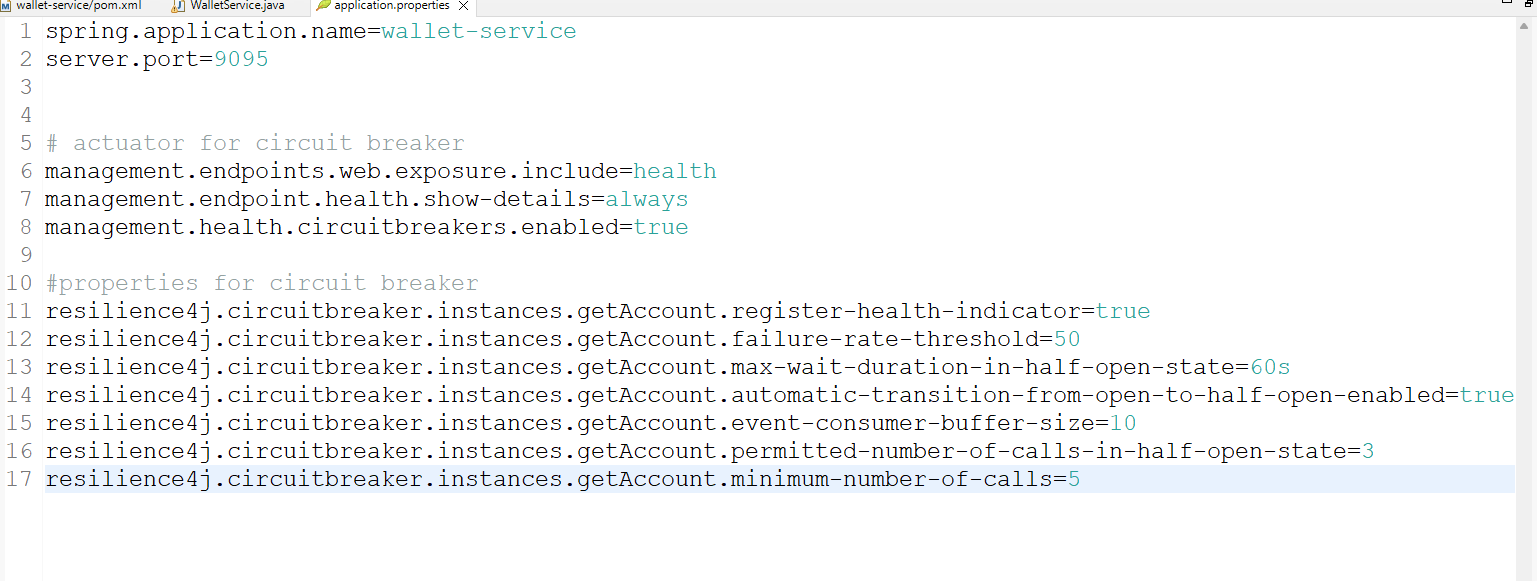


Add @CircuitBreaker in the WalletService method which is making a remote call

WalletService.java



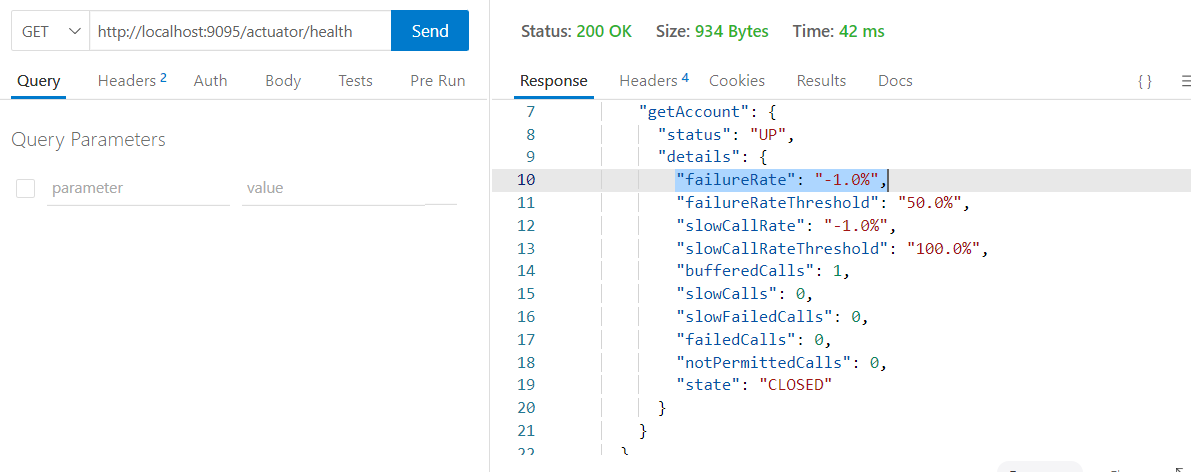
application.properties



Programs to launch

1. Service Discovery
2. Configuration Server
3. Account Microservice
4. Wallet Microservice

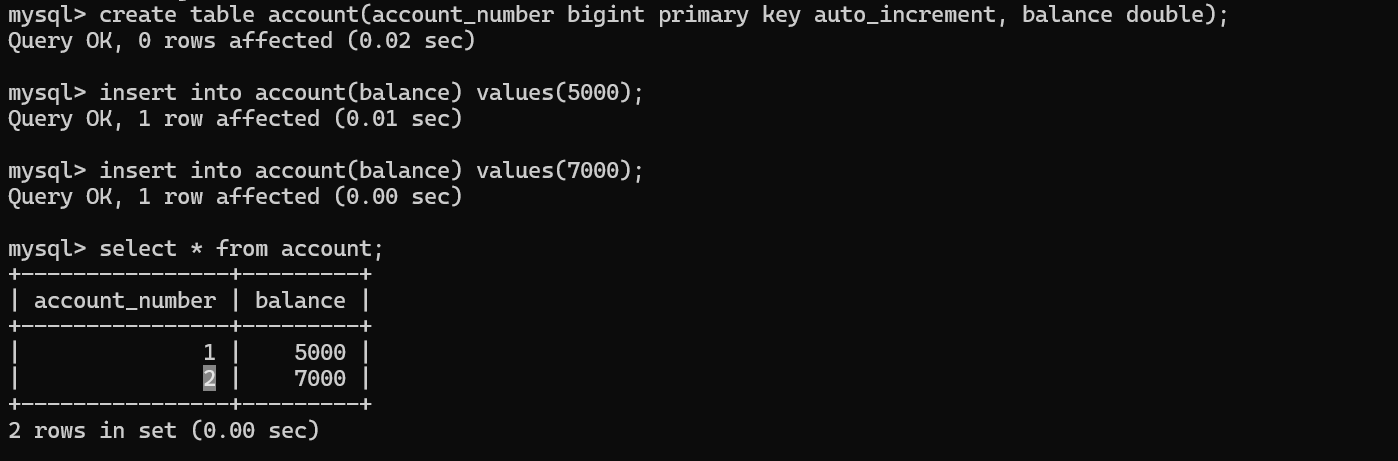
You can see the circuit breaker status in /actuator/health



Stop the Account Service and send few more requests minimum 5, then you can see the OPEN state

1. After 1 min you can see HALF\_OPEN
2. HALF\_OPEN to CLOSE state occurs only if the service is up

Create database table account in mysql & store some records



Activity:

Account Microservice should connect to the MySQL database using Spring Data JPA, however the datasource configurations must be pulled from the GIT and also the username & password must be in encrypted format, when wallet sends the account number it has to send only those numbers present in the database

Steps:

1. Using configuration server create the encrypted data for username & password & store them in your git repository (ensure you are able push the configuration file whose name matches to the spring.application.name of account service)
2. Add Spring Data JPA & MySQL library to the account microservice
3. Create an entity for account and map the account number & balance
4. Create a Repository that extends JpaRepository<Account, Long>
5. From the service layer you must get the Account balance using the account number
6. From the controller call the service layer method and return account number & balance in JSON format