Microservices with Spring Boot

Software Required

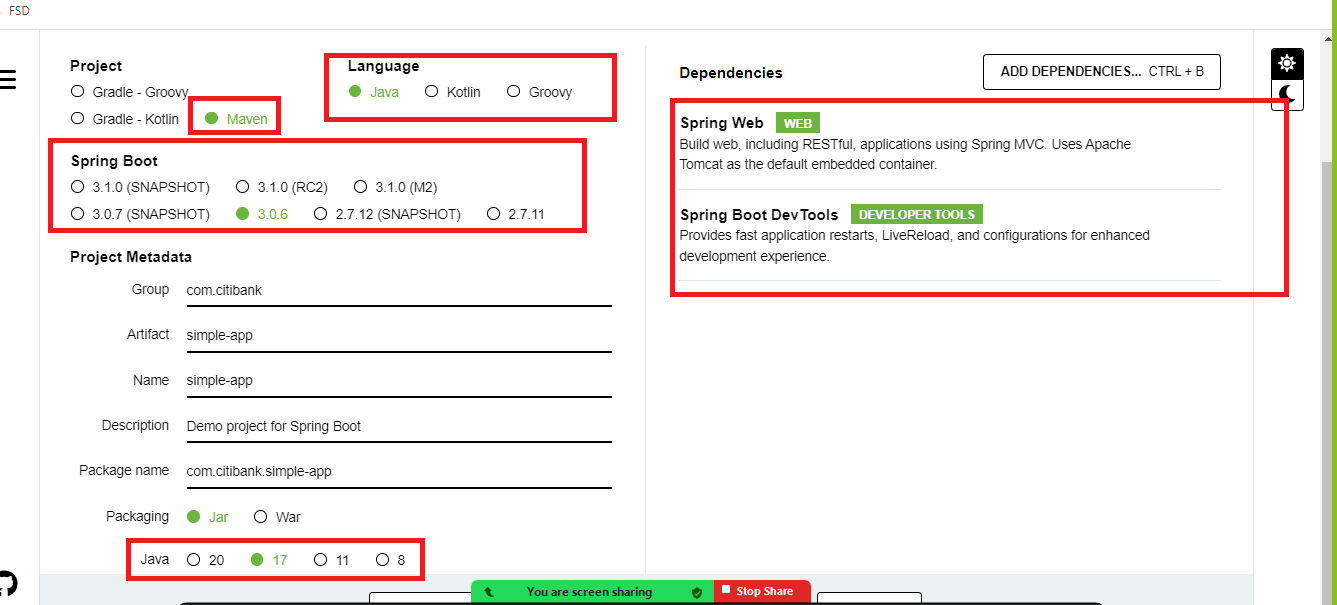
* Java 8 or later
* Eclipse / STS
* Postman
* MySQL

Spring Boot:

It automates all the generic setup for the applications so that you can quickly create spring related applications by concentrating only the application level instead of concentrating on the infrastructure level

Note: Spring Boot can be used only with build tools like Maven or Gradle, they will take care of downloading all the libraries for your project

Project Template



In Spring Boot, Java objects are automatically converted to JSON, we can use any custom objects or predefined APIs in Java like Map, List and so on.

Pre-requisites for microservices in Spring Boot

* Dependency Injection
* Layered architecture like @Service, @RestController, @Repository
* Annotations like @Autowired, @Bean, @Configuration, @Component

Postman app:

It is used to test the webservices

Most of the webservices will have CRUD operations with HTTP methods like post, put, get, delete, we have corresponding annotations from spring boot to create webservices of different methods

* GetMapping
* PostMapping
* PutMapping
* DeleteMapping



Today’s Agenda

* Configuring the project to use base machine JRE
* Creating an executable JAR & running through command prompt
* Microservices

Configuring the project to use base machine JRE

1. Setup the version in pom.xml
2. Change the project build path to use base machine JDK
3. Change the compiler version that matches to base machine JDK
4. Update the project

Building the project into a single artifact

1. You need to use a maven package command from the eclipse or through command prompt, here jar will be created inside target folder
2. You need to run the jar file using “java -jar *filename.jar* <<options>>”

Microservice:

These are loosely coupled services which are independent from other services

Benefits

* Services are loosely coupled
* Testing will be easier as we need to only test a particular service we want
* We can choose any language we want
* We can scale the service which we want
* If a particular service goes down it affects only the services that are down, and other services will be available.

Challenges:

1. Expensive - because we need to many resources
2. Important to identify the right approach
3. Adapting to the changes
4. Global competition

Design principles in microservices

1. Service Discovery: Registers the microservices Instance Id & physical address
2. Discovery Client: These are microservices which locates other microservices using the instance ID, it must send acknowledgement to the service discovery frequently about its health status
3. Client Side Load Balancer: Takes care of distributing the load across multiple instances of microservices
4. Fault Tolerance - Circuit breaker pattern: It takes care of breaking the circuit to stop sending the requests to the already down service
5. Distributed configuration
6. API gateway
7. Distributed log tracing

Spring provides a project called Spring Cloud which has already implemented these design patterns for us

Spring cloud: It uses Netflix OSS library to implement all these design patterns

Microservices with spring uses two projects of spring

* Spring Cloud - gives all the tools & design patterns for microservices
* Spring Boot - to easily create applications using automated features available

Note: We can use simple annotations to add the design patterns we want in our code

ex: @EnableEurekaServer is used to create service discovery

@EnableEurekaClient is used to create discovery client

@LoadBalanced is used to create client side load balancer

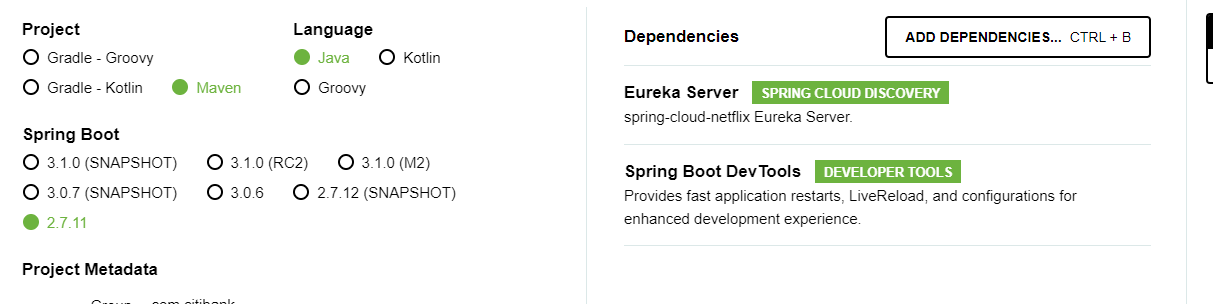
We need to create minimum 2 projects

1. Service Discovery
2. Discovery Client

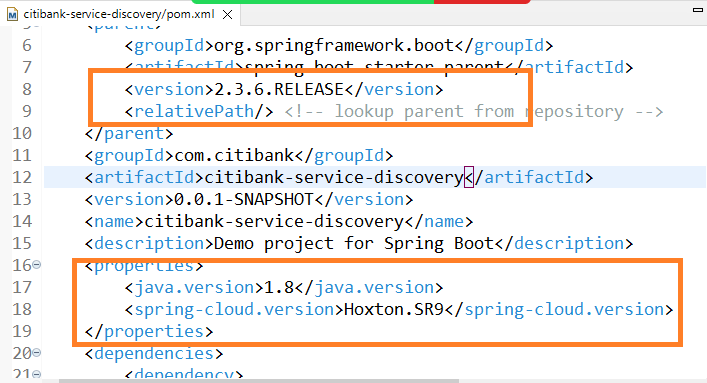
Note: Since we are using spring cloud & spring boot projects, we must refer to the spring website to see the version compatibility

Service discovery project

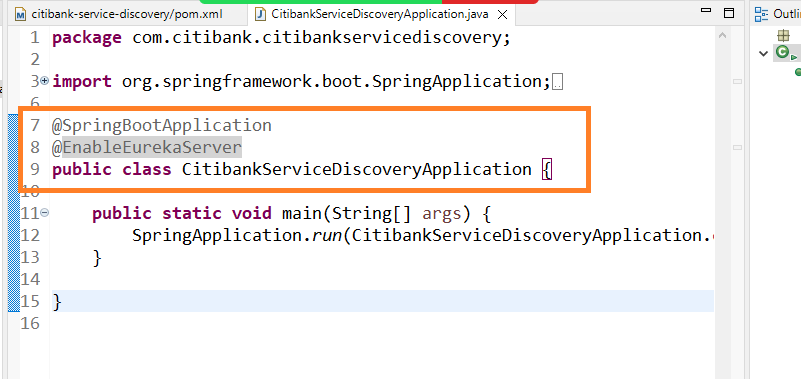
1. Eureka Server
2. Devtools



pom.xml



Service Discovery Enabling

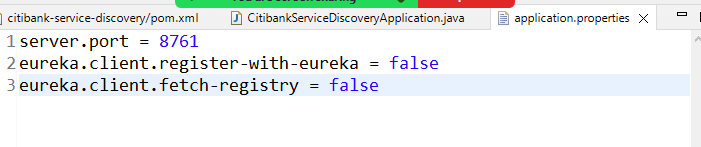


Since Eureka Server downloads Eureka Client library we need to disable client feature in the application

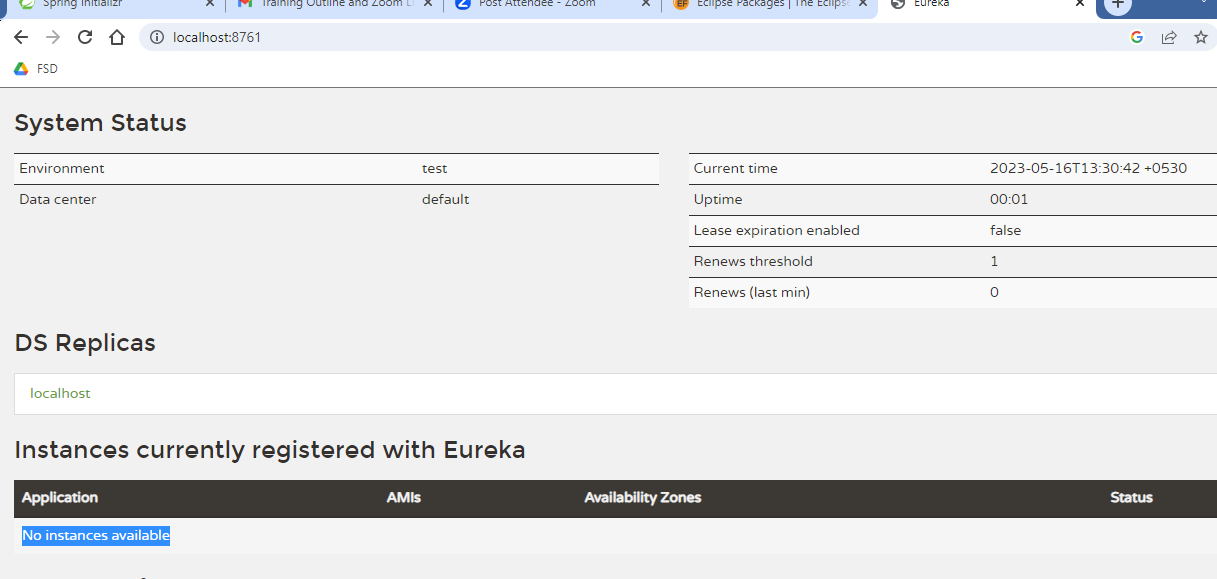
like fetching the registry, registering as a service

Note: By default all the microservice searches eureka server in 8761 port, hence you need to run eureka server in 8761

application.properties



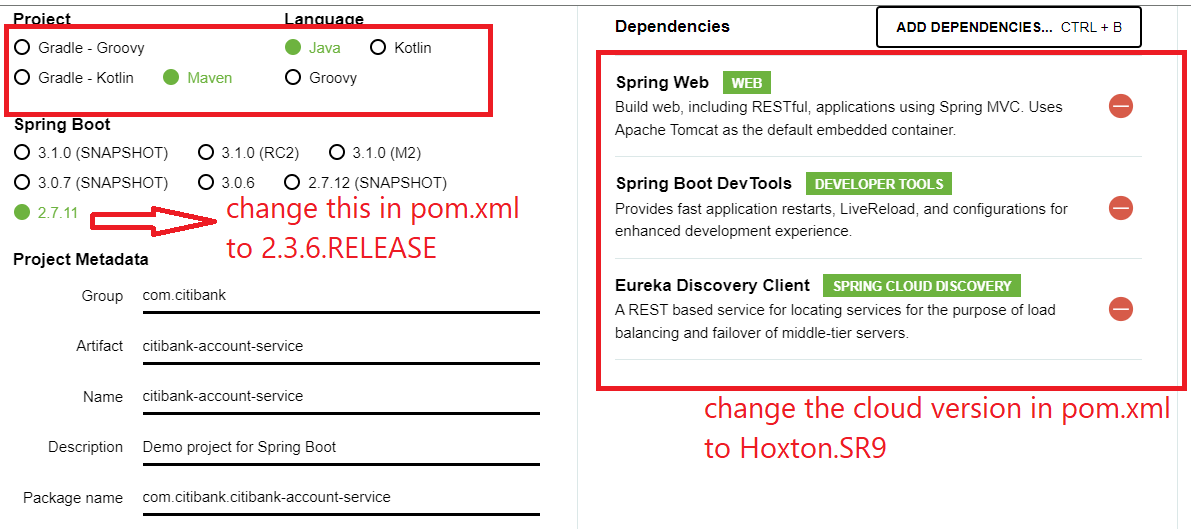
Note: Eureka Server gives you a client dashboard so that you can see all the services



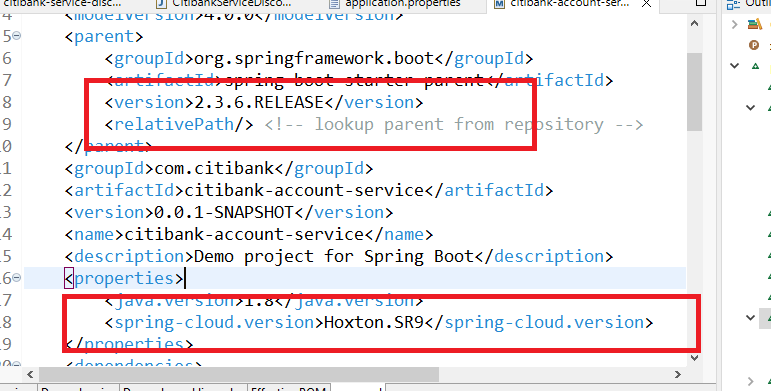
Note: While using microservices avoid running the programs in eclipse, instead use executable jars

Discovery client project

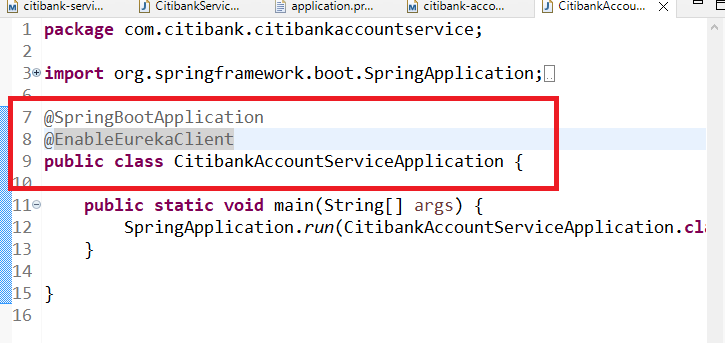
1. Eureka Client
2. Web
3. Devtools



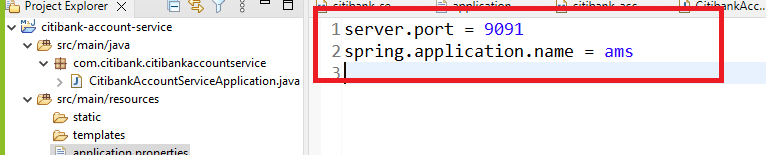
Change the pom.xml as per the below configuration



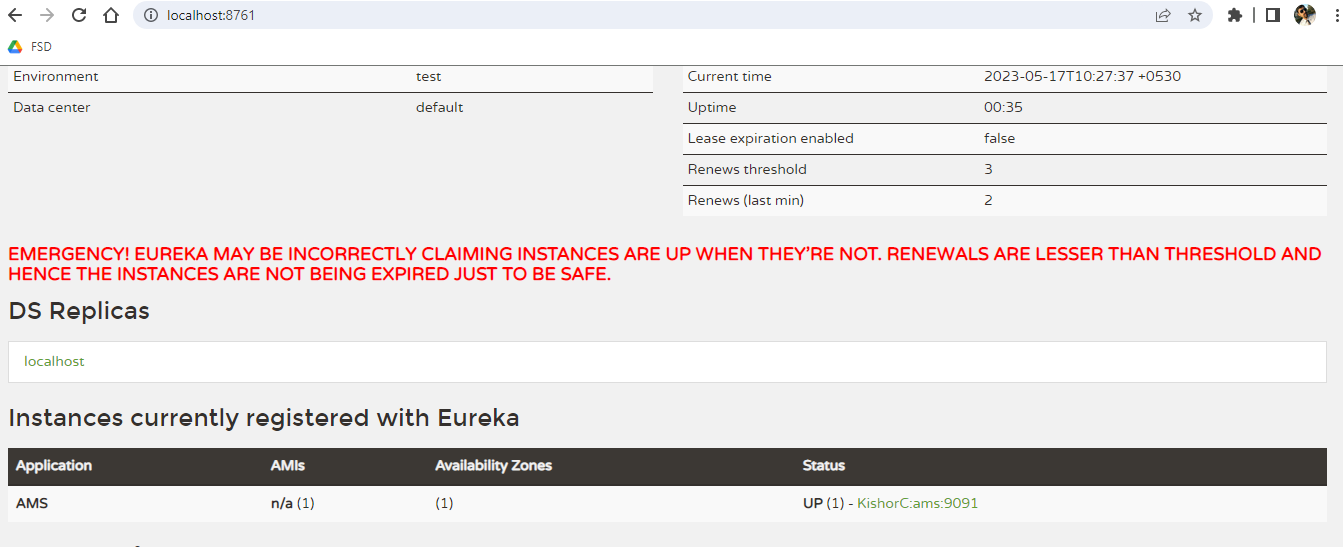
To register this as a microservice we must use @EnableEurekaClient in our application



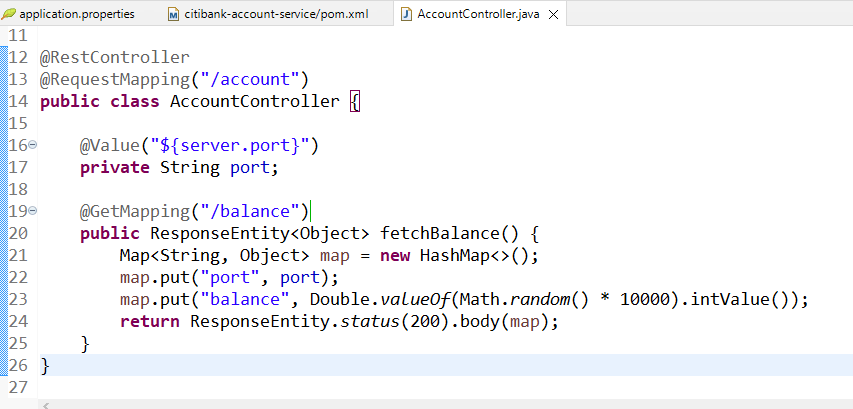
application.properties



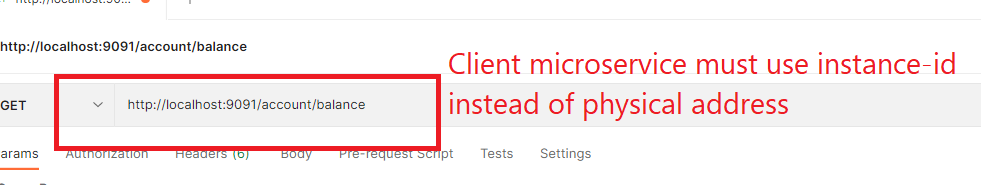
Create an executable jar & run it in command prompt



Create one controller that can return some data to the client



We need to access this microservice from another microservice but to test we can use postman with the physical address.



Another microservice that acts as a client to call a remote microservice (AMS)

RestTemplate: It is an object used to call the remote service

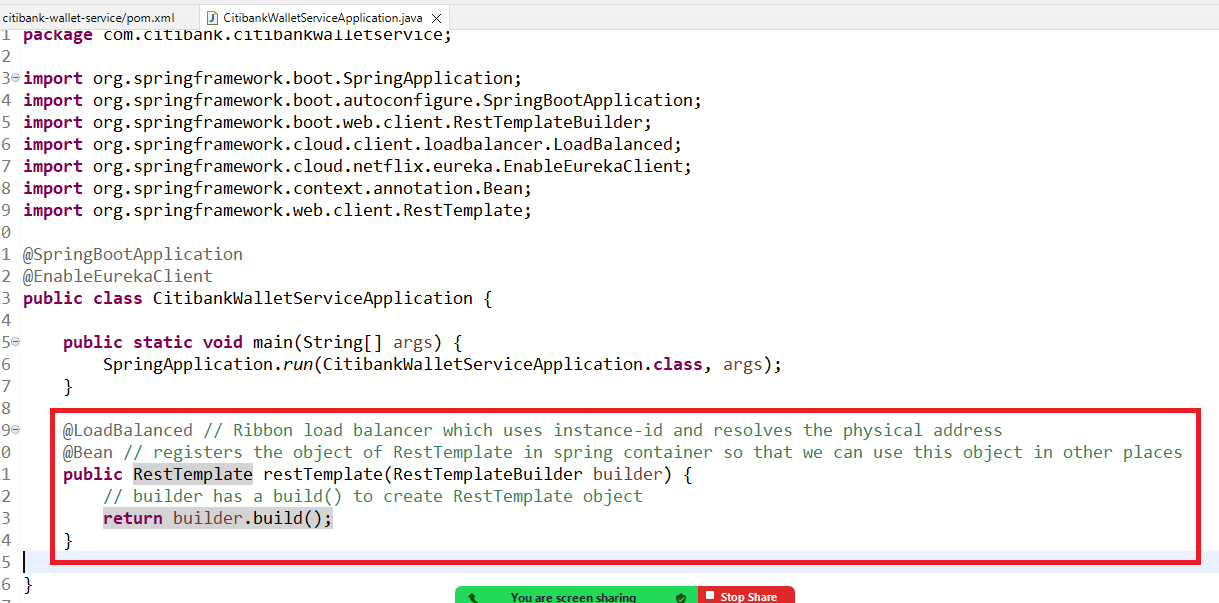
LoadBalancer: Spring cloud gives a library called Ribbon which provides the client side load balancer, we need to use @LoadBalanced annotation to add the load balancer to the RestTemplate

@LoadBalanced  
@Bean  
public RestTemplate restTemplate() {   
 ….  
}

Dependencies

1. Eureka Client
2. Web
3. Dev tools

Enable the microservice and Ribbon load balancer in the application



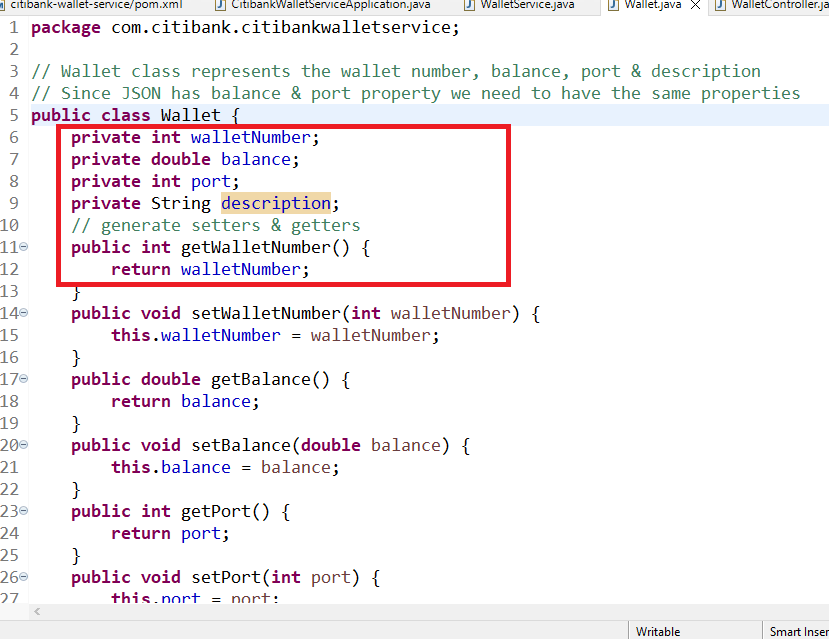
RestTemplate: It is an object used to access remote service using URL and HTTP methods like get, post, put, delete

We need to create a Service layer that would access the remote service & a controller that would access the Service layer

Create 3 classes

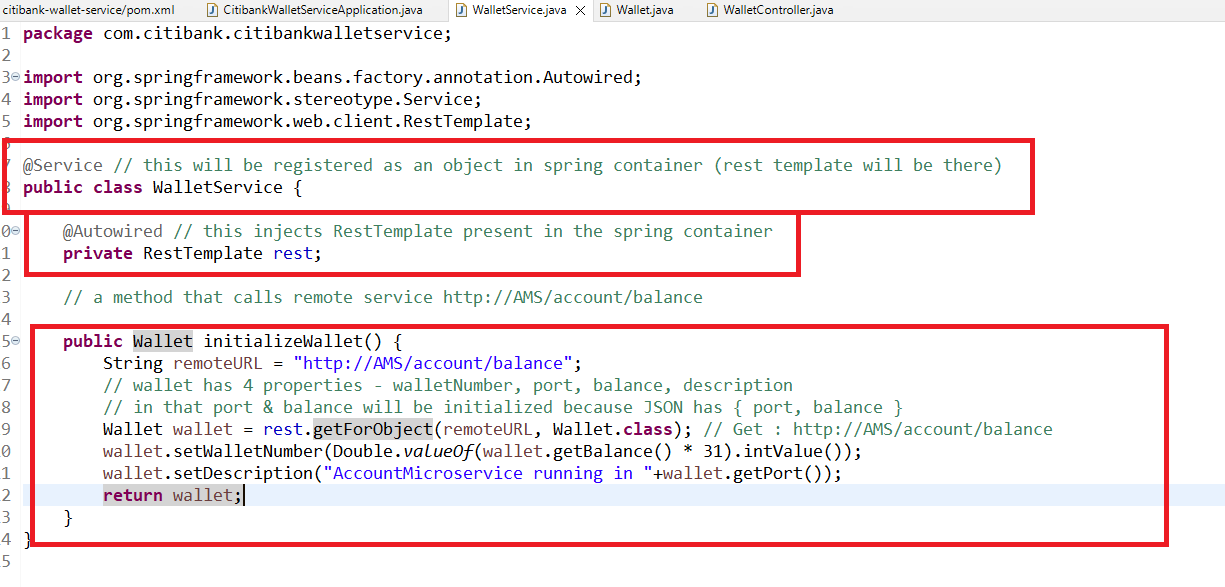
1. WalletService : This will call remote service & gets its data
2. Wallet: a model that represents the remote service & current service data
3. WalletController: this calls the wallet service methods

Wallet.java

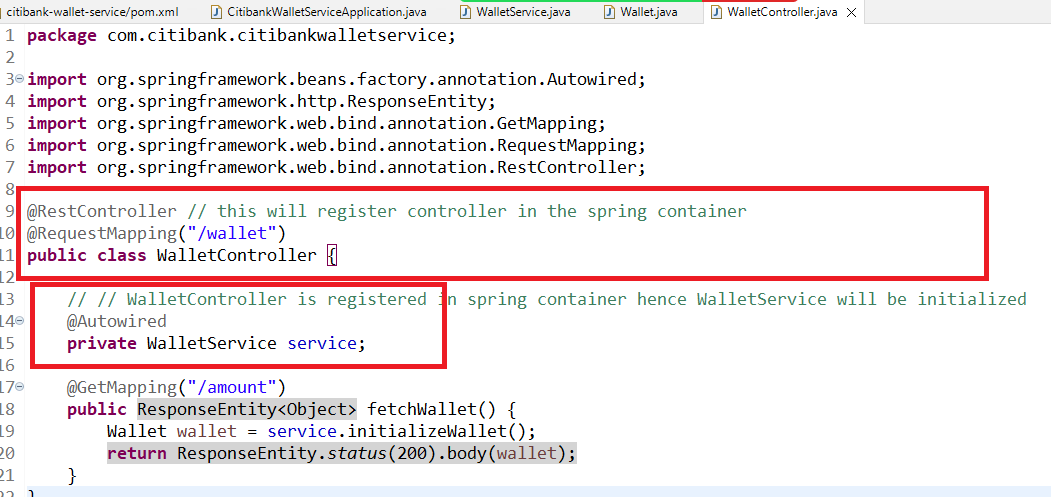


WalletService.java

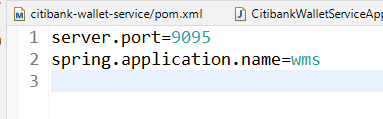
Bind the RestTemplate object here & call Account Microservice using RestTemplate & initialize the Wallet in any one method



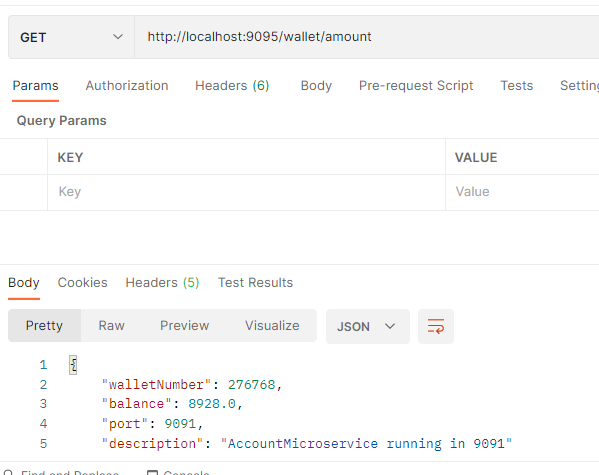
Now you can call this intializeWallet from the controller



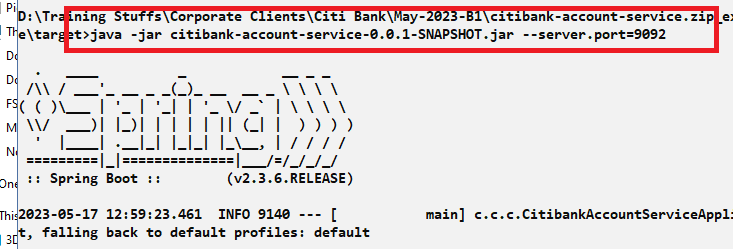
application.properties



Now you can run this project & send request to wallet/balance



Creating another instance of account microservice to verify the load-balancer job



Try to communicate with the microservice without service discovery and check whether load balancer is able to delegate request.

Agenda

1. Fiegn Client to communicate with Microservice
2. Fault tolerance i.e., Circuit breaker pattern - Hystrix & Resilence4j
3. Api gateway - Zuul
4. Security - oauth2

Feign Clients: These are reusable objects that helps to access remote service & internally it uses client side load balancer, you don’t have to use URL to access the remote service instead you can create an interface that will have URL mappings of the remote service so that at any place you want to access the remote service you can use the interface method.

@FeignClient(“http://AMS”)

interface AccountClient {   
 @GetMapping(“/account/balance”)

public Wallet fetchAccountBalance();  
}

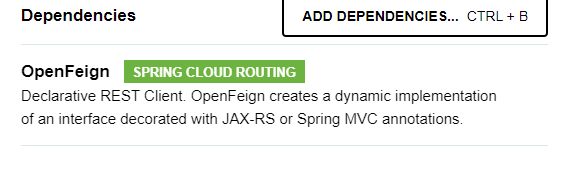
accountClient.fetchAccountBalance() : sends Get request to http://AMS/account/balance

In RestTemplate the same thing is written as below

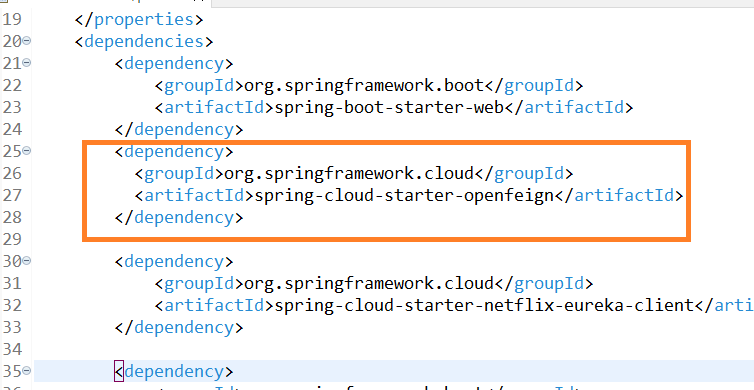
restTemplate.getForObject(“http://AMS/account/balance”, Wallet.class)

Note: RestTemplate must be created with @LoadBalanced in your application

We need a library to use Feign Client

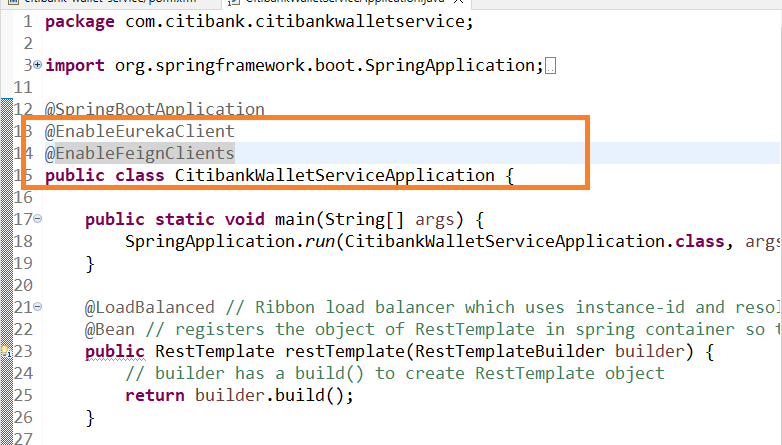


Modify wallet microservice pom.xml

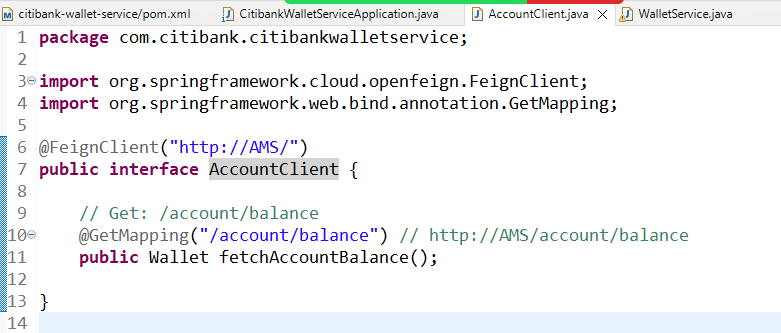


@EnableFeignClients must be used in the application to scan the interfaces having @FeignClient so that these interfaces will be auto-implemented

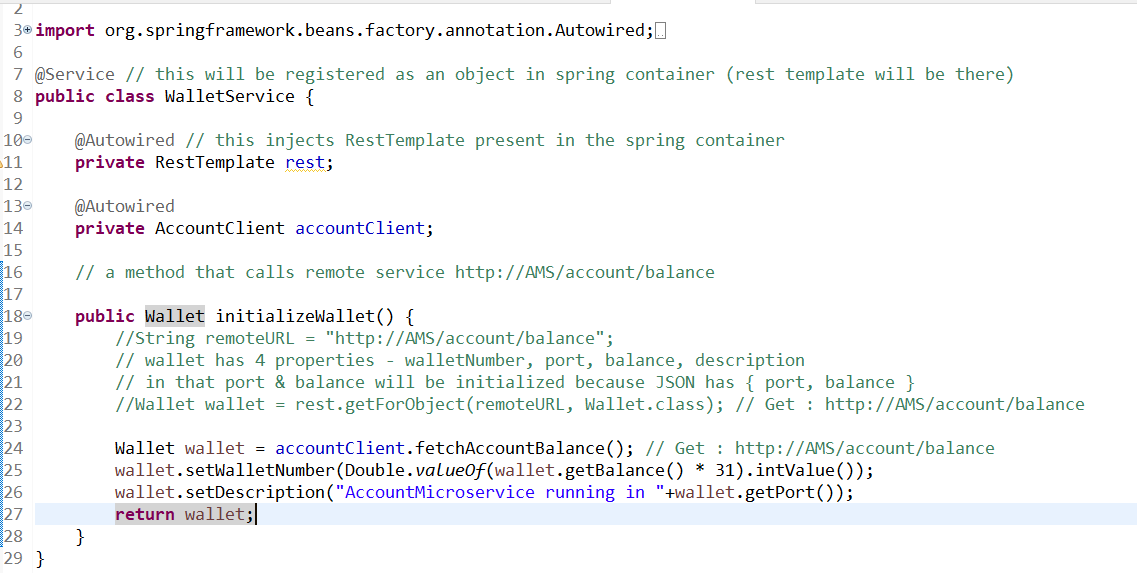
Modify the main class to use @EnableFeignClients



You can write the interface with @FeignClient



Note: You don’t to implement this interface because EnableFeignClients will do that & also it registers the object of the interface implementation in the spring container, so that you can tell spring to inject the object at any place



Circuit Breaker:

Whenever a client microservice is calling remote microservice there could be chance that remote service is down and its problem can be cascaded to the client service, hence we need to apply a circuit breaker so that after some threshold limit the circuit breaker state will become open so that there won’t be any request goes to the remote service & after some period of time the circuit must be closed and client must able to send the request to the remote service, if the service is up the request flow should be there else , it must open the circuit till the remote service is up, but periodically the client must send request to the remote service to verify the status of the remote service.

There are 3 states a circuit breaker will have

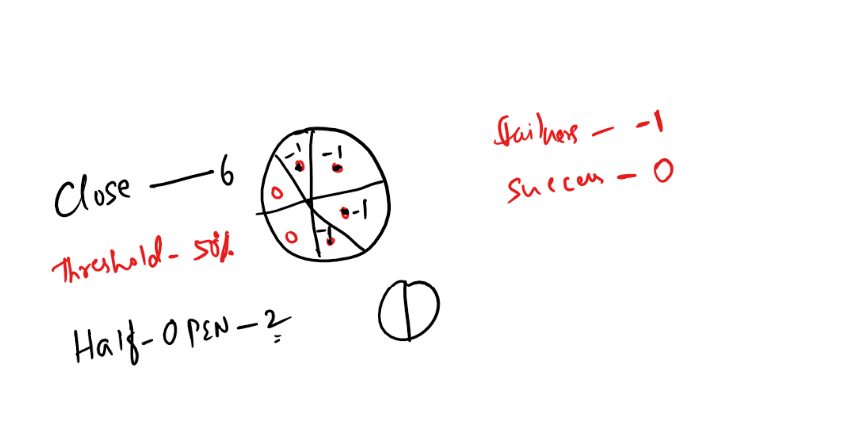
1. OPEN
2. CLOSE
3. HALF\_OPEN

OPEN: Calls go directly to the fallback method present in the client service

CLOSE: Calls go to the remote service

HALF\_OPEN: Calls go the remote service & it is to decide the state to go to OPEN or CLOSE in limited number of requests

Circuit Breaker uses Ring Bit Buffer to store the success/failure request status, based on this ring bit buffer tracking the circuit breaker changes its state.



To apply circuit breaker pattern we have

1. Hystrix: It is now under maintenance, it supports programmatic configuration
2. Resilience4j: It is still under long term support, it is built with hystrix library only, it supports declarative configuration

Library we need are

1. Resilience4j from spring-boot2: io.github.resilience4j
2. Spring Boot Starter AOP: Takes care of invoking the fallback methods

Note: Fallback method must have the same method signature of the method calling remote service with one extra argument that handles Exception,

// If the below method is calling remote service

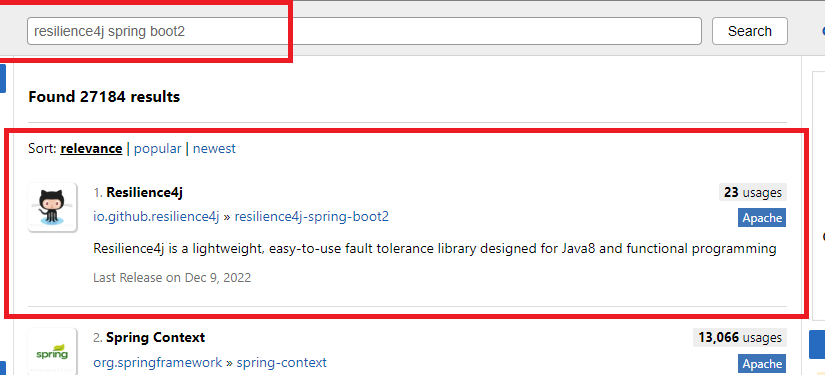
@CircuitBreaker(name = “instanceName”, fallbackMethod = “fallback”)  
Wallet initializeWallet() {  
 // remote calls  
}

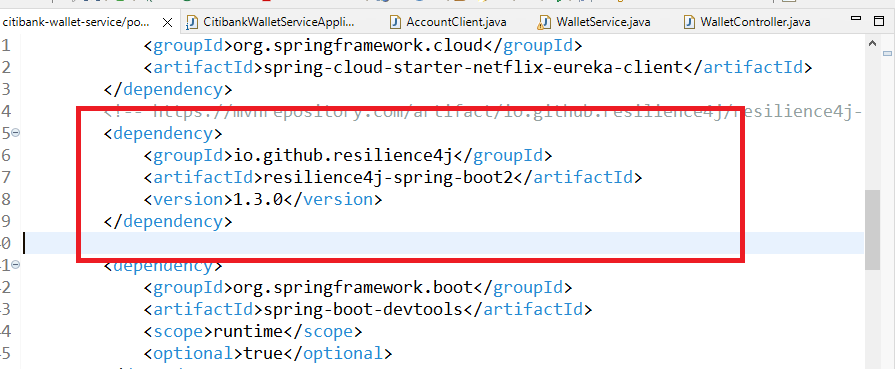
// fallback method must have same signature with extra argument

Wallet fallback(Throwable t) {  
 // alternate response   
}

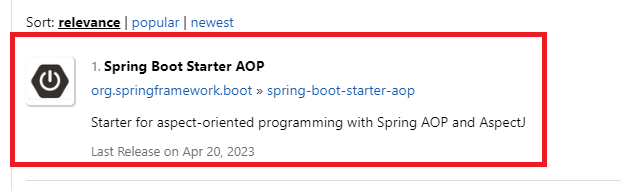
Note: Add all the libraries in the wallet microservice

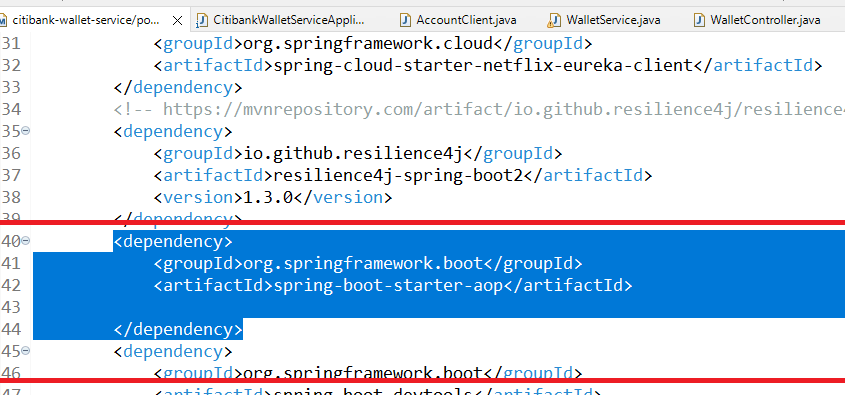
Search for Resilience4j & Spring AOP in Maven





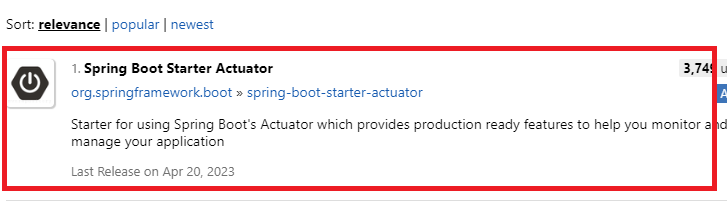
Adding Spring Boot Starter AOP



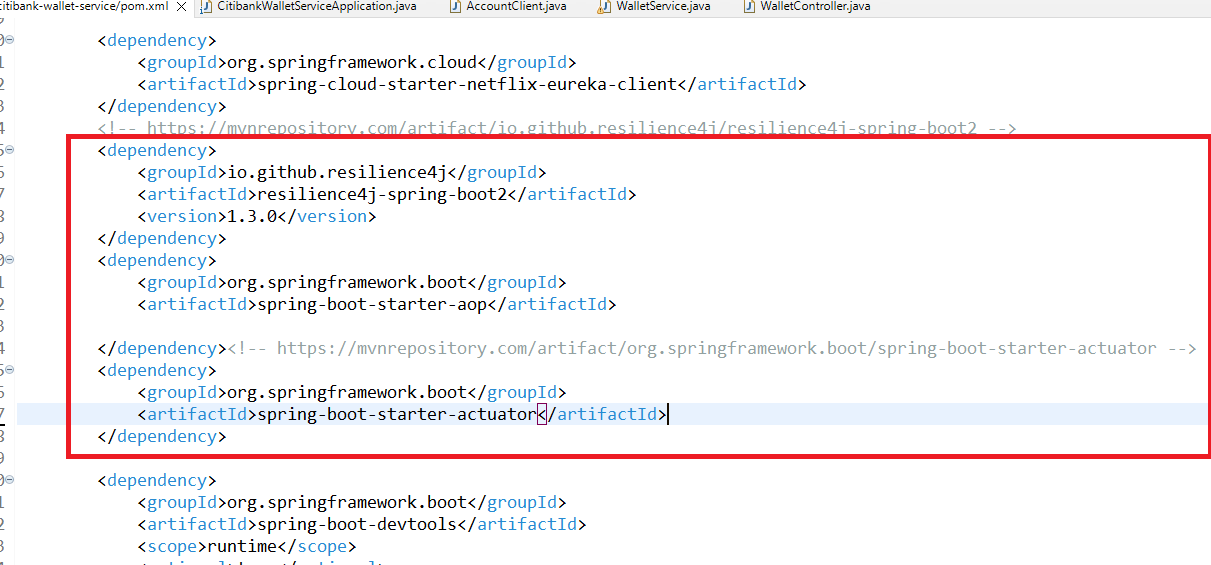


We need to add Spring Boot Actuator

Spring Boot Actuator helps you to give some endpoints to see the application status, like health, routes, circuit breaker states



pom.xml



Steps to implement to apply the circuit breaker to avoid fault tolerance

1. Adding the circuit breaker to a method that is calling the remote service, the circuit breaker must have a name that is used to apply the properties in application.properties
2. Creating a fallback method for the remote service calling method
3. Configurating the circuit breaker properties in the application.properties

Ex:

@CircuitBreaker(name = “walletInstance”, fallbackMethod = “fallback”)  
public Wallet initializeWallet() {   
  
}

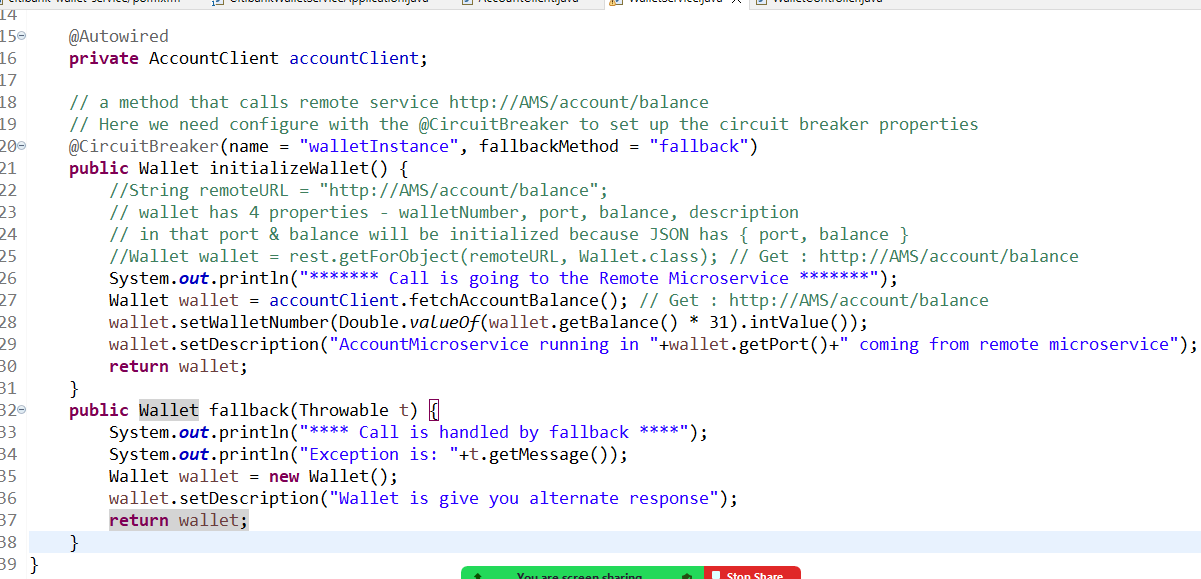
public Wallet fallback(Throwable t) {  
  
}

application.properties

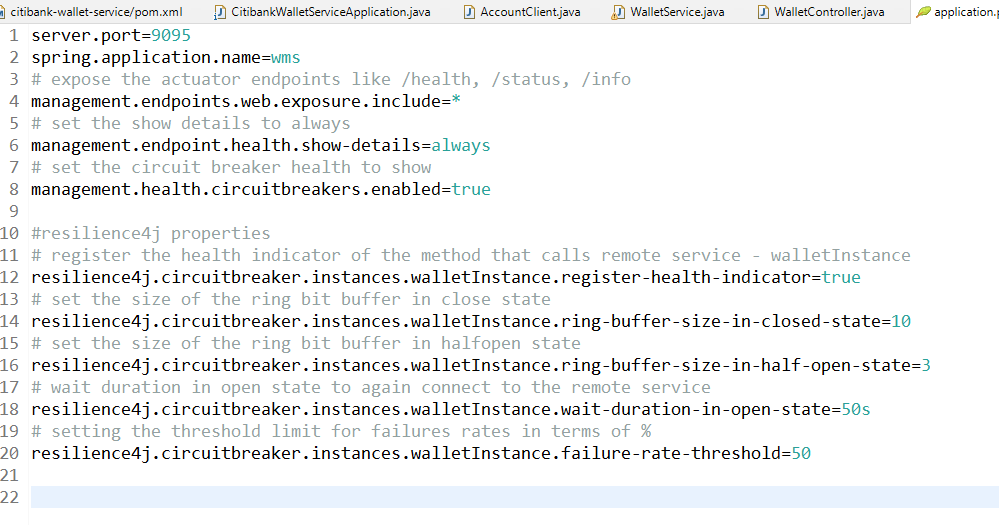
resilience4j.circuitbreaker.instances.walletInstance.ring-buffer-size-in-closed-state=10

resilience4j.circuitbreaker.instances.walletInstance.failure-rate-threshold=50

WalletService.java



application.properties



we need to see the circuit breaker states using actuator endpoints /health

i.e., <http://localhost:9095/actuator/health>

The above URL lists all the circuit breaker properties like

failureRate, failedCalls, state, and etc.

Steps to see all the circuit breaker features

1. Re-run the wallet microservice by re-building
2. Stops the account microservice & send some requests
3. While sending some requests parallelly check the /health to see the circuit breaker properties

Agenda

1. API Gateway - Zuul - pre & post processing filters
2. Security
3. Deploying microservices on cloud
4. Transaction management is done in Microservices using saga pattern
5. Testing

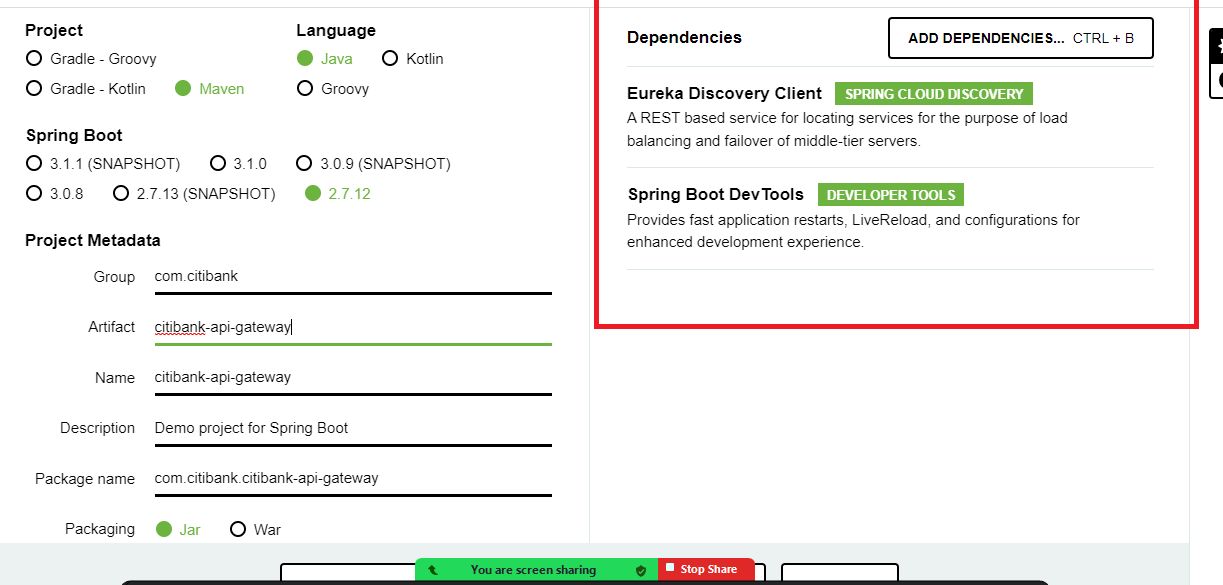
API Gateway:

It is a common entry point for all the microservices including the service discovery, you can do reverse proxy with API gateway i.e., client would use a different URL & api gateway will route the request to different URL/services

API gateway provides filters that can intercept the request to perform pre & post processing

1. Pre Filters: run before the request is routed to the service
2. Route: This routes the request to the service after pre filter
3. Post Filters: run before sending the response to the client

In Spring Cloud we have a library called Zuul which is used to create API gateway



We need another library called spring cloud starter Netflix zuul which you can get it from the maven.

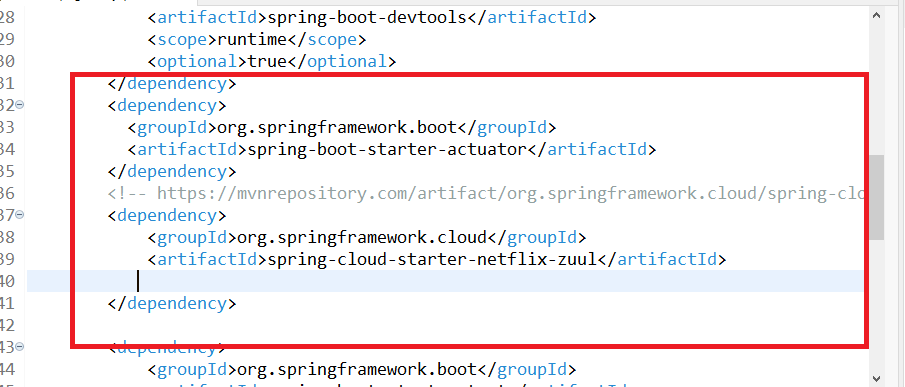
Note: Setup the project to use 2.3.6.RELEASE & Hoxton.SR9

We need to use

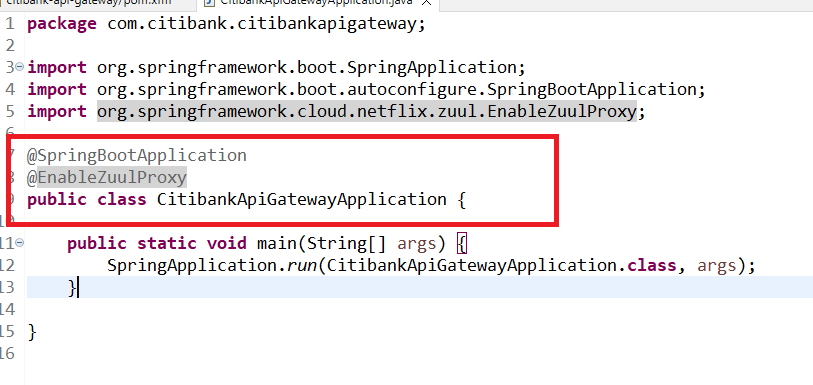
1. Actuator : to get the routes endpoints
2. Zuul: to add reverse proxy server

@EnableZuulProxy: This will create API gateway and allows all the clients to go through the API gateway, you also need to register this in the service discovery so that you can access all the microservices using a single end point

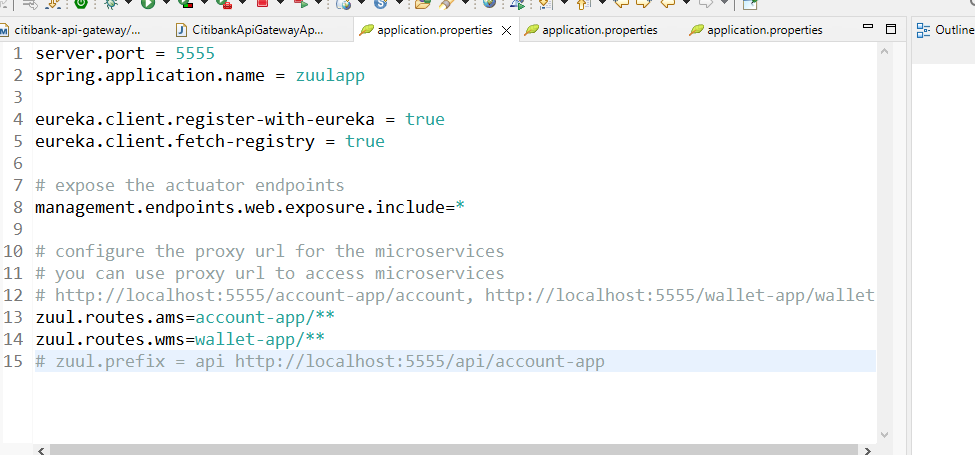
API gateway has an endpoint called /actuator/routes that shows all the URL mappings for your microservices

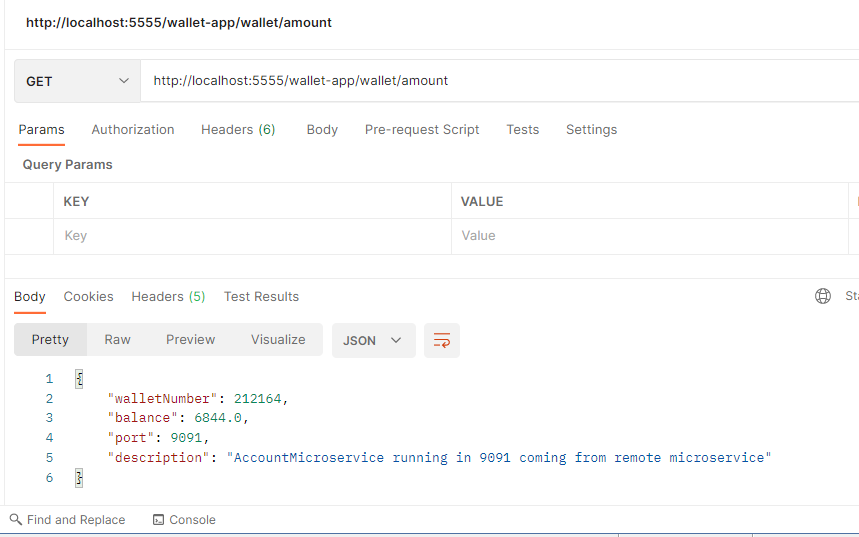


Add zuul proxy to the project using @EnableZuulProxy



Configure application.properties to register API gateway in the service discovery & routes configuration





http://localhost:5555/account-app/account/balance : gives response from account microservice

http://localhost:5555/wallet-app/wallet/amount : gives response from wallet microservice

Securing microservices

