Microservices with Spring Boot

Software’s requirement

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

Pre-requisites

* Spring Boot
* Spring REST

Spring Boot: It is one of the spring module, which helps to create various types of applications quickly by providing all the configurations automatically

* Server configuration (Embedded tomcat server is used)
* Front Controller (Takes care of mapping the request to the right controller)
* Initializing the Spring container (Takes care of Dependency Injection)

Note: Spring Boot projects are created with build tools (Gradle, Maven)

Restful webservice

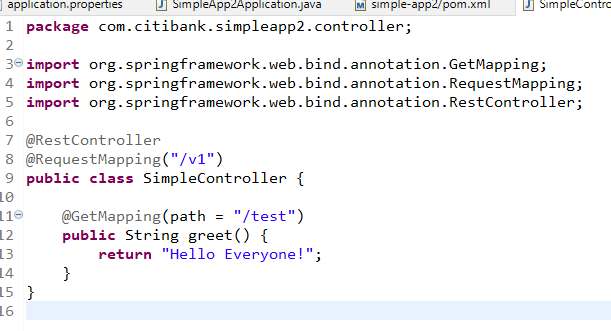
REST stands for Representational State Transfer which helps two applications written in different technologies to exchange the data in a common format like JSON, XML, CSV & etc.

Rules while creating webservice

1. Locating the webservices using the URL
2. Marking the operations of the webservice using HTTP methods like GET, POST, PUT, DELETE

In Spring Boot we can create webservices using some annotations

1. @RestController: Must be written on top of the class that will have webservice
2. @RequestMapping: Must be written on top of the class or a method to mention the URL
3. @GetMapping, @PostMapping, @PutMapping, @DeleteMapping: Written on top of the method to specify the http methods



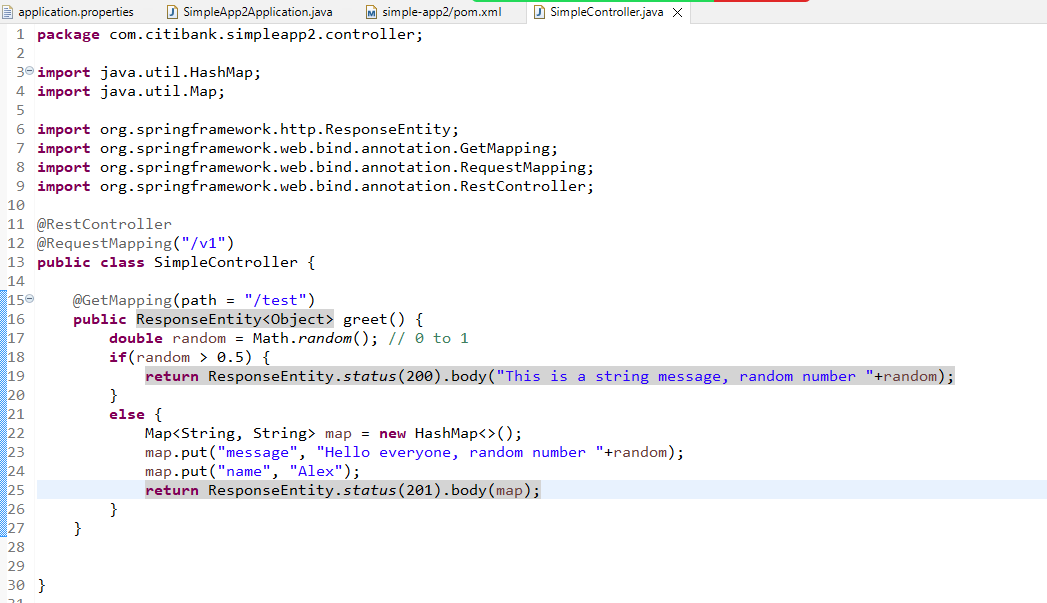
The above code only returns text data & by default the status code will be 200, but we can customize the response status so that we can send different status codes like 200, 201, 204, 401, 404 and so on

ResponseEntity is the object we need to use to customize the status code & the content

We can use ResponseEntity.status(200).body(content); to return the response with 200 status code & some content

ResponseEntity.status(404).body( obj1 );

ResponseEntity.status(200).body( obj2 );



Creating an executable jar file and running it

We must build the project using Maven, so that it creates a deployable artifact like jar or war, we need to run this file on the server machine through command prompt.

Microservices

These are loosely coupled services which are independent from other services of same or different applications

Monolithic Architecture

Here all the services will be part of the same application & it will tightly coupled

* You can’t scale a particular service, you had to scale the entire application
* You need to test all the services if any changes happen or a new service is added
* Releasing the new feature will take more time based on the complexity of the application
* If any one service goes down, then the entire application services will be unavailable
* You can’t use a different language for a particular service, all the services might use the same language

Microservice Architecture

Nextflix OSS used this architecture for the first time, where they created services independently that can be deployed in separate machines

* You can scale a particular service you want
* If any one service goes down it doesn’t affect other services
* Testing is easy, we need to test only the service which is modified
* Creating & Releasing new feature is quick
* You can use different languages for different services

When to go with Microservice & when not to go with microservice architecture

Whenever you have global customers for your application then you can go with microservice architecture.

When your application is used within the organization you can use monolithic architecture ex: Payroll system, Leave management system

Design patterns while developing microservices

1. Service Discovery
2. Discovery Client
3. Client Side Load Balancer
4. External Configuration
5. Circuit Breaker
6. API gateway
7. Security
8. Distributed Log tracing

Service Discovery: It registers the microservices in the registry

Discovery Client: It is a microservice that registers in the service discovery & sends acknowledgement for every 30s, automatically registers its instance id & physical id

Client Side Load Balancer: It takes care of equally distributing the load across multiple instances of the same microservice, it is the one which resolves the actual address of the microservice it has to communicate with.

Circuit Breaker: It breaks the flow of microservice communication based on certain failure limits so that other microservices will not slow down.

Spring provides 2 projects for microservices

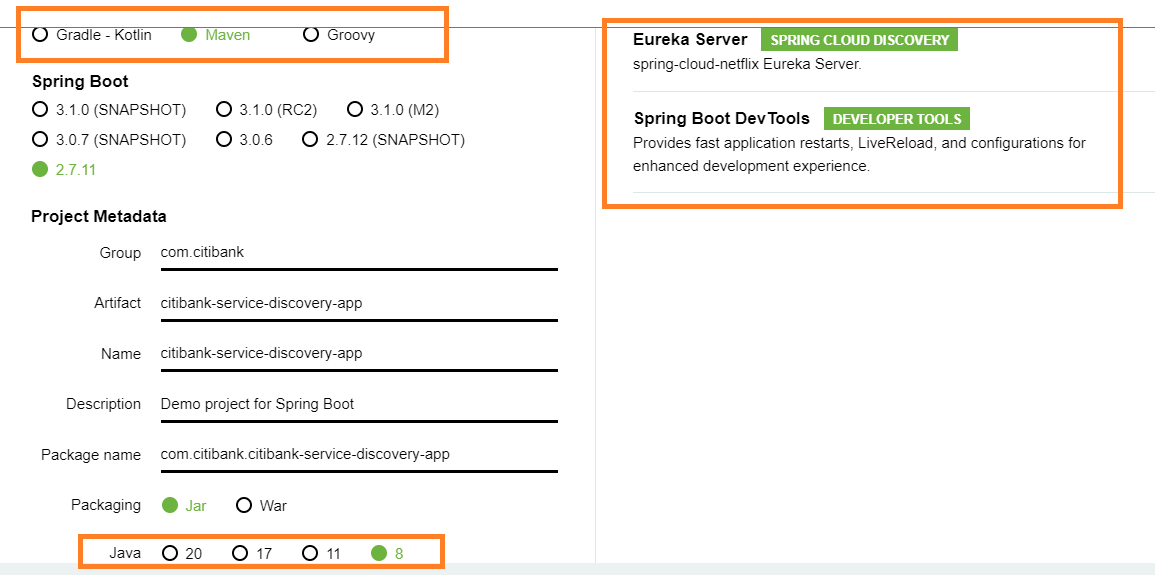
1. Spring Cloud: It provides all the necessary tools & design patterns to implement microservices like Service discovery (Eureka Server), Discovery Client (Eureka Client), Client Side Load Balancer, Circuit Breaker Pattern and etc.
2. Spring Boot: It takes care of auto-configurations based on the library you add.

Note: Spring Cloud & Spring boot versions compatibility is mandatory, we need to check the spring cloud release train to see which cloud is mapped which version of spring boot

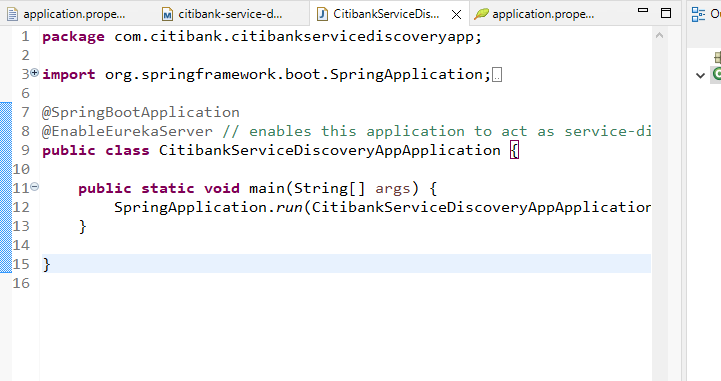
Service Discovery

1. Eureka Server
2. Dev tools

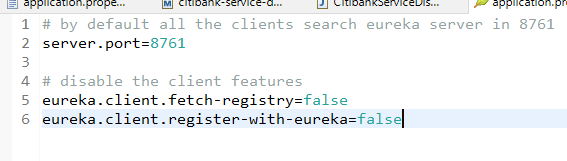
Note: Eureka Server depends on Eureka Client which means service discovery will try to automatically register itself as a microservice because of spring boot, we need to disable the client feature in service discovery in application.properties



Enabling Eureka Server



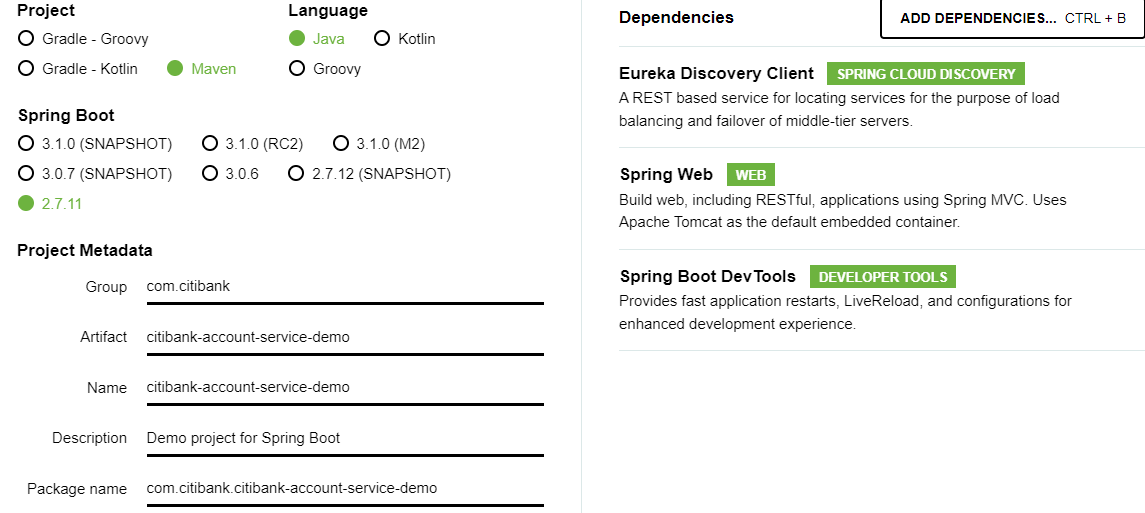
application.properties

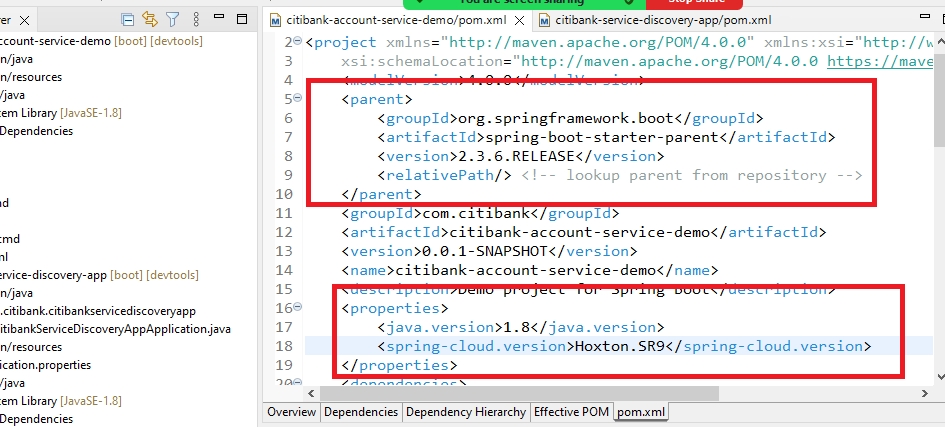


Creating clients for service discovery i.e., microservices which needs to have instance-id

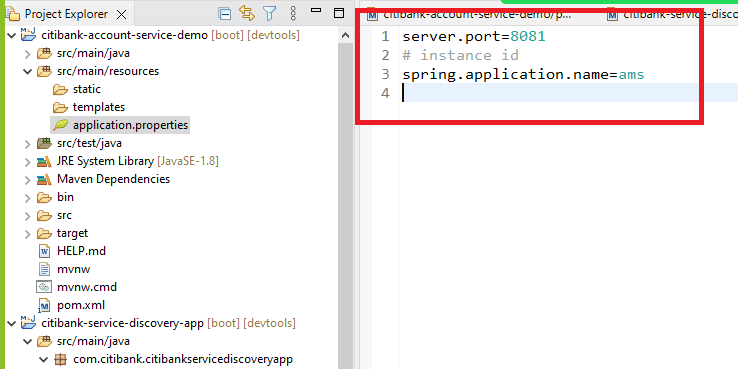
Dependencies

* Eureka Client
* Web
* Devtools

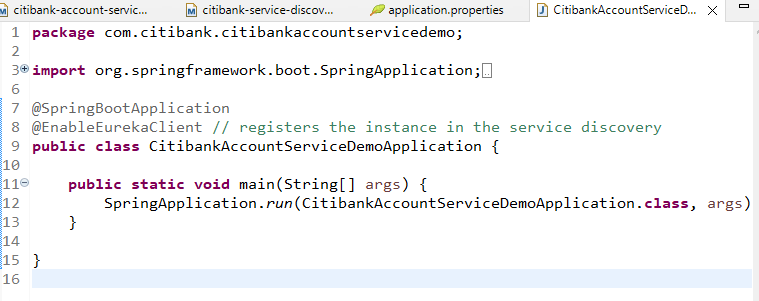




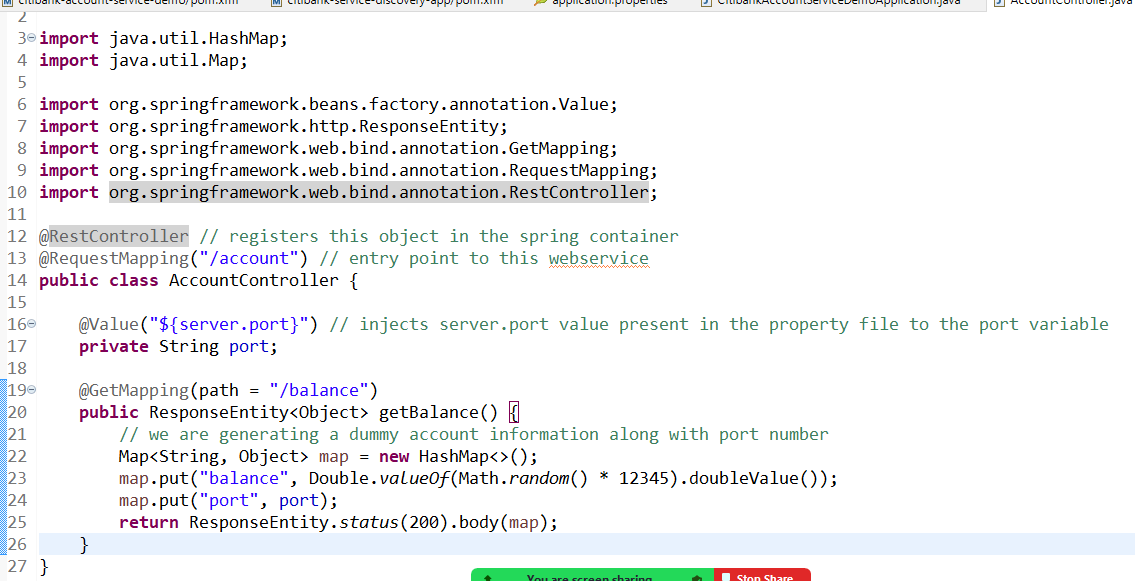
application.properties



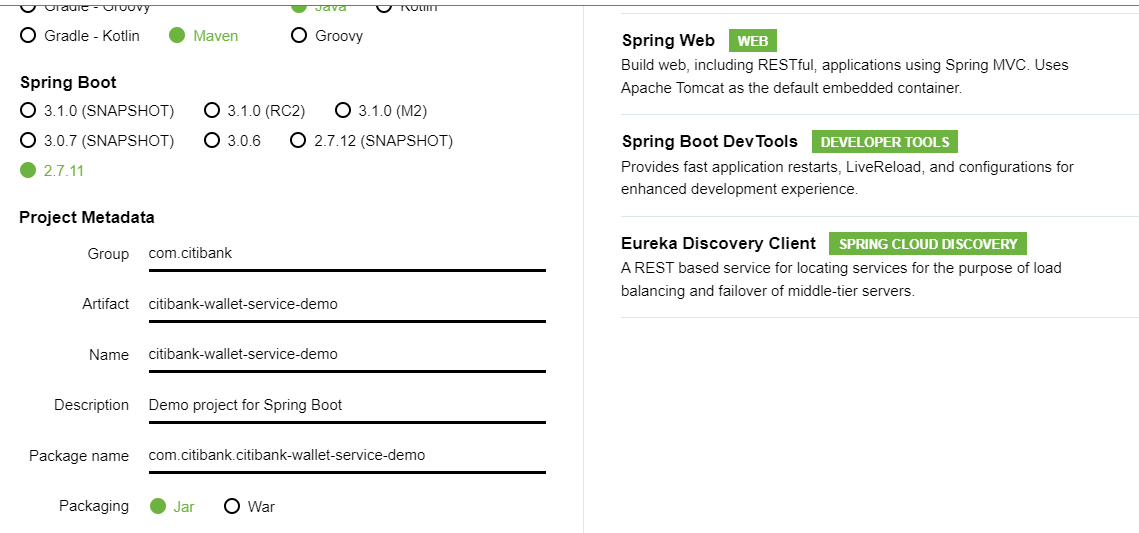
Enable the project to act as a microservice



Controller code that generates random balance and the port number of the instance



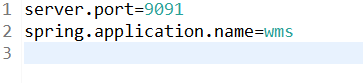
Wallet Microservice



change the pom.xml to use spring boot 2.3.6.RELEASE & spring cloud Hoxton.SR9

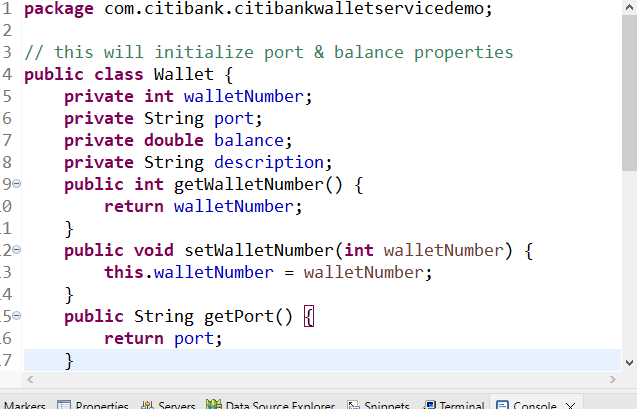


application.properties



We need to create WalletController, WalletService & a model to represent the remote service data i.e., Wallet

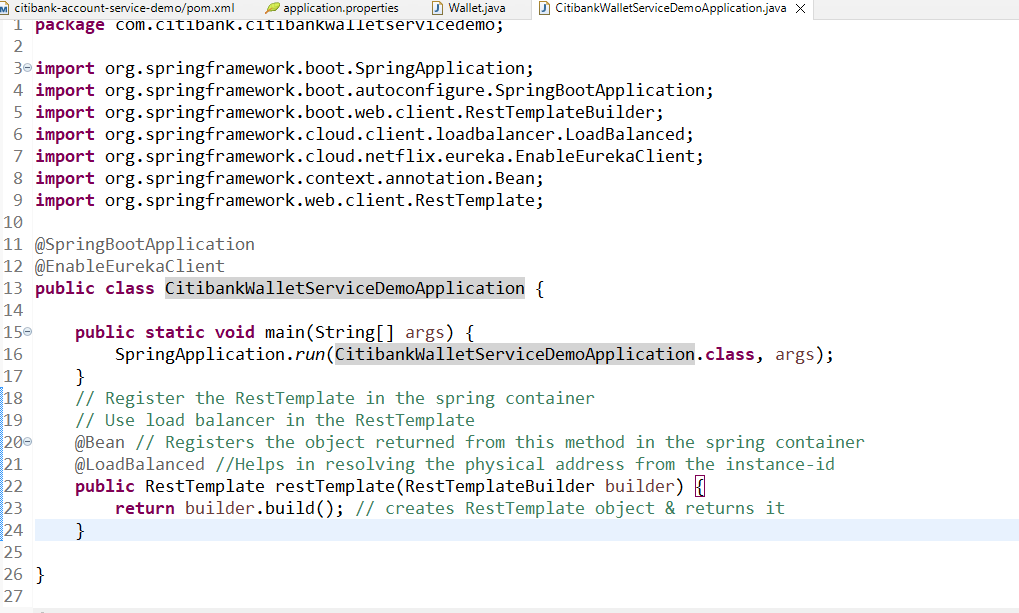
Wallet.java



We need to use RestTemplate to access the remote service (i.e., Account Microservice Service), but the RestTemplate cannot resolve the Physical address using instance-id, hence we need to wrap RestTemplate with Ribbon client (client side load balancer), which takes care of resolving the physical address from the instance-id

Things to create in our application

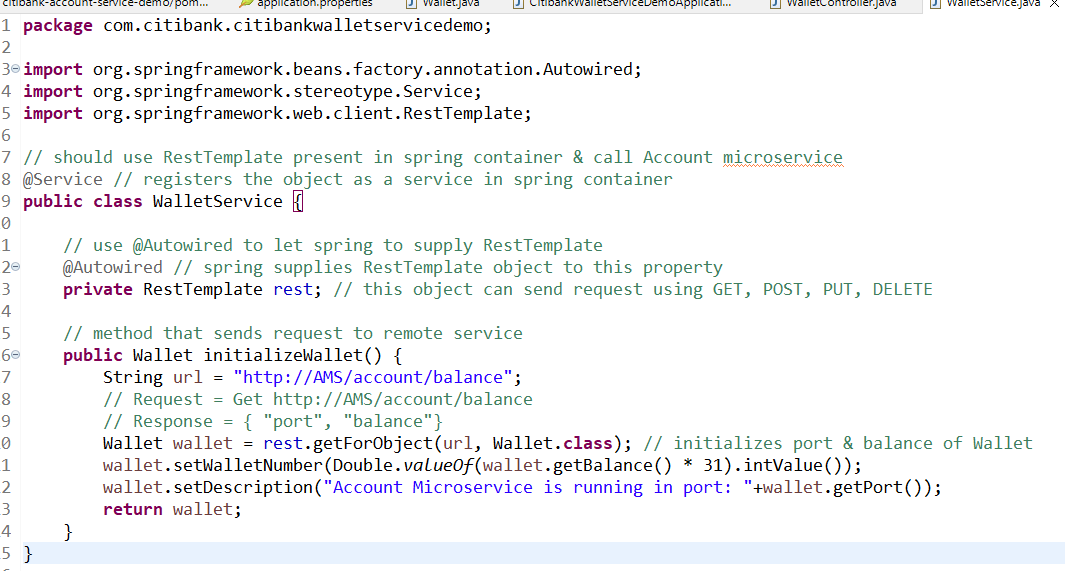
1. Register the RestTemplate in spring container so that it is available through the application
2. Bind the RestTemplate with Ribbon client i.e., load balancer so that load balancer can resolve the physical address using the IP address.



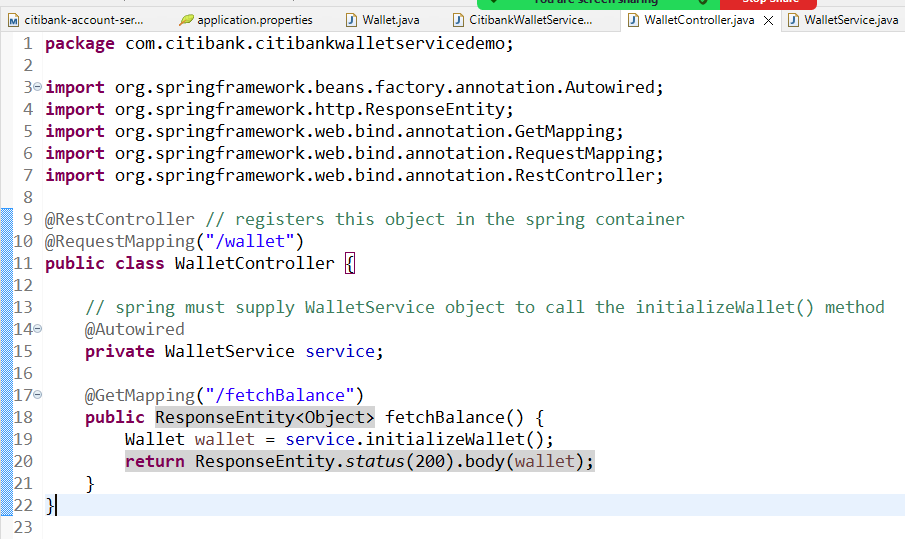
Since calling a microservice from another microservice is a business logic, we need a service layer in our code, this service layer is accessed by the controller layer, hence we need to create 2 classes

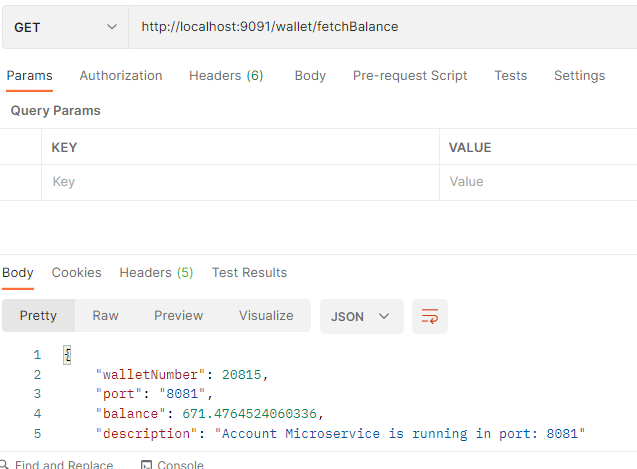
1. WalletService
2. WalletController

WalletService.java



WalletController.java





Create multiple instances of account microservice by changing the port number like 8082, 8083 (java -jar filename.jar --server.port=8082)

* Observe the Eureka Dashboard you must see those number of instances in account microservice
* From Wallet Microservice send multiple requests, you will see the response coming from different account service instance (Load balancer distributing the request)
* Stop the service discovery and send request from the wallet microservice you must still see the response (load balancer caches the instance it is communicating)

Todays Agenda

1. Using feign client to communicate with the microservice
2. Using Circuit Breaker Pattern
3. API Gateway

FeignClient: It is provided spring cloud to interact with the remote microservices, it internally uses client side load balancer & it is reusable

In Rest Template

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

In Feign Client

@FeignClient(“http://AMS/”)  
public interface AccountClient {   
 @GetMapping(“/account/balance”)  
 public Wallet fetchBalance()  
}

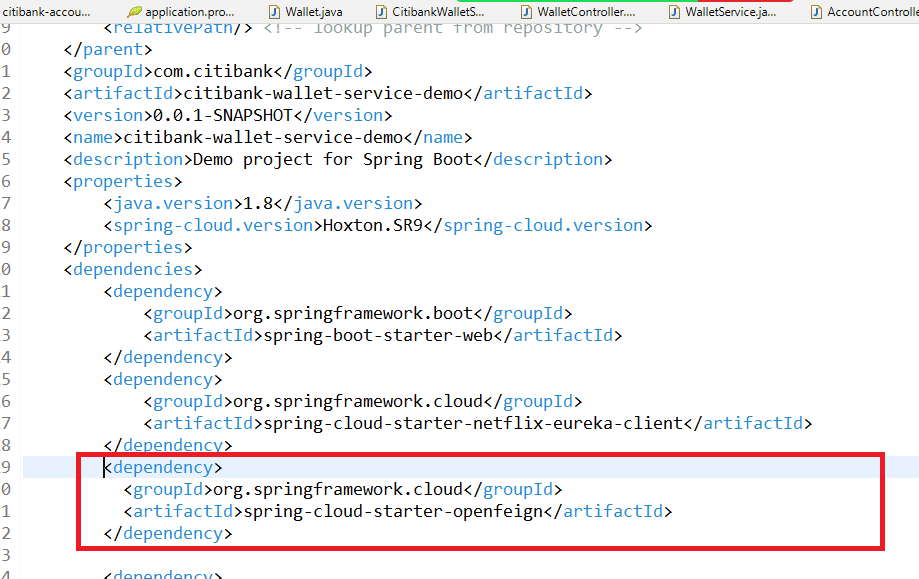
To call the remote service

accountClient.fetchBalance(); // Get: http://AMS/account/balance

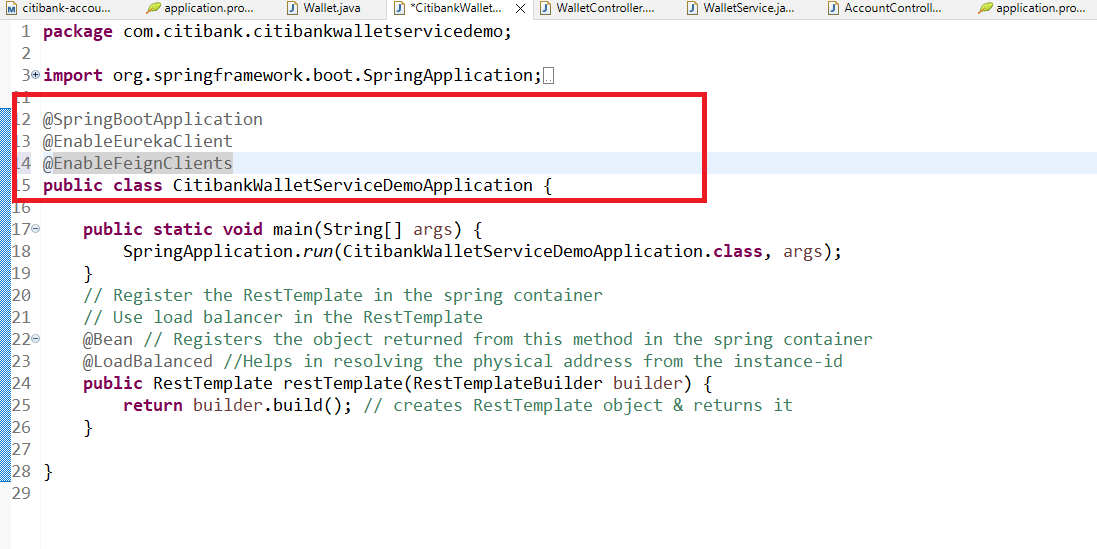
Note: Feign Client interface we don’t have to implement, spring boot implements automatically using an annotation called @EnableFeignClients, which you need to use in your application

To use Feign Client we need below library

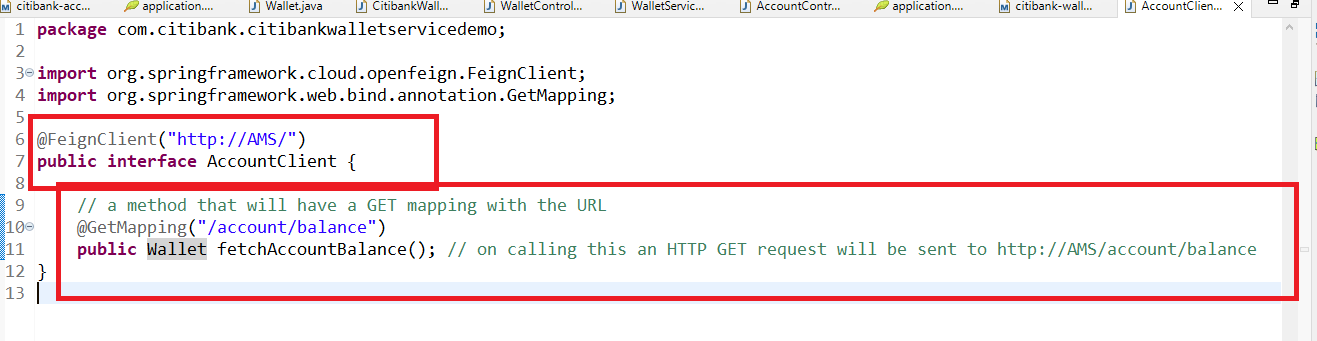
* Open Feign



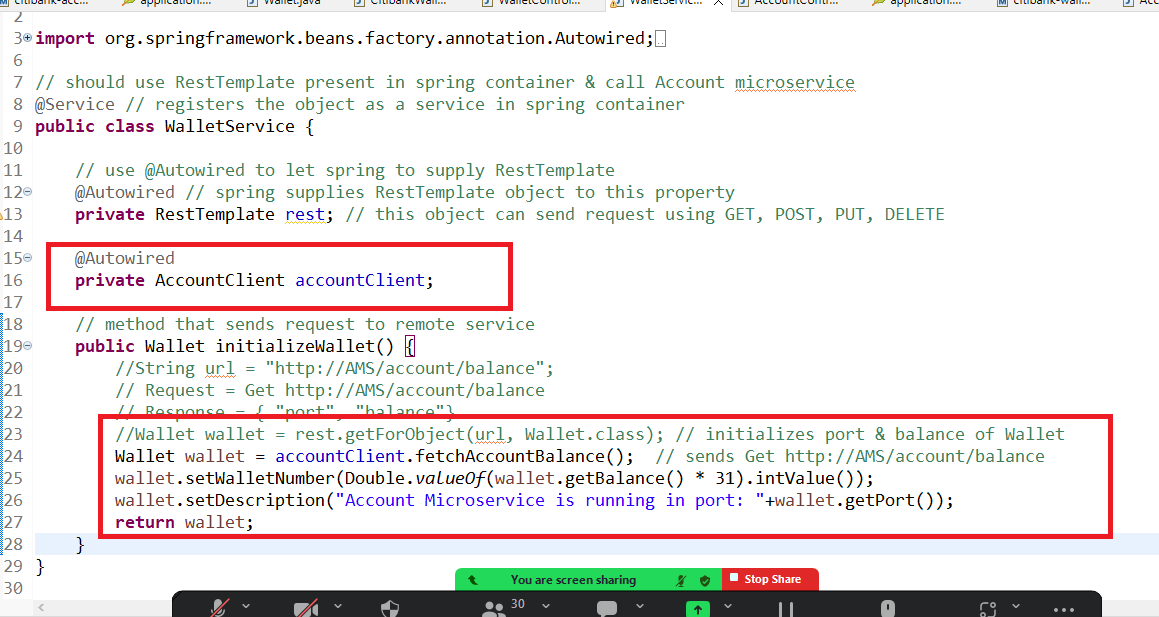
We need to use @EnableFeignClients in our application so that it can detect interfaces having @FeignClient and provide implementation automatically to such interfaces & register the object of these implementations in the spring-container.



Create an interface with @FeignClient



Now you just need to use @Autowired on AccountClient & call the fetchAccountBalance access the remote service



Send an hard-coded balance like 5000, 6000 etc from the Account Microservice and create a method in the Wallet Microservice called addAmount and have a initial value of the wallet amount as 500 so that when you call addAmount it must get the amount from the account & add it to the wallet so that the amount will be added to the wallet

i.e.,

AccountMicroservice must response as { “balance” : 5000 }

WalletMicroservice must use that 5000 & add to the initial balance so that your total wallet balance will be 5500, final output must be { walletNumber, port, balance: 5500 }

Note: Use @PutMapping to perform this operation in both the microservices & do this with FeignClient interface

Circuit Breaker

It is one of the design pattern applied when the remote services are down the client services wouldn’t go down by checking how much failed request the client can tolerate, when the circuit breaker is applied the connection wouldn’t be there between the microservices upto some time, and client can constantly check the remote service status and make a connection.

Circuit Breaker uses 3 states to interact with the remote service

1. CLOSE
2. OPEN
3. HALF\_OPEN

Close State: Client sends request to the remote service

Open State: Client doesn’t send request to the remote service instead a fallback method will be called that gives the alternate response

Half Open State: Client sends request to the remote service, this state is a decision making state to either change to Open or Close state, but it makes decision with limited requests

To add circuit breaker pattern we have libraries like Hystrix, Resilence4j

Hystrix is under maintenance because it follows programmatic approach for circuit breaker configuration.

Resilence4j is another form that is built on Hystrix it supports declaration approach for circuit breaker configuration.

Client Microservice

It will have the method that sends request to the remote service & a fallback method that is executed when request fails to get response

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

// initializeWallet is executed when the Circuit is in Close or Half Open state

// fallback is directly executed when the Circuit is OPEN

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

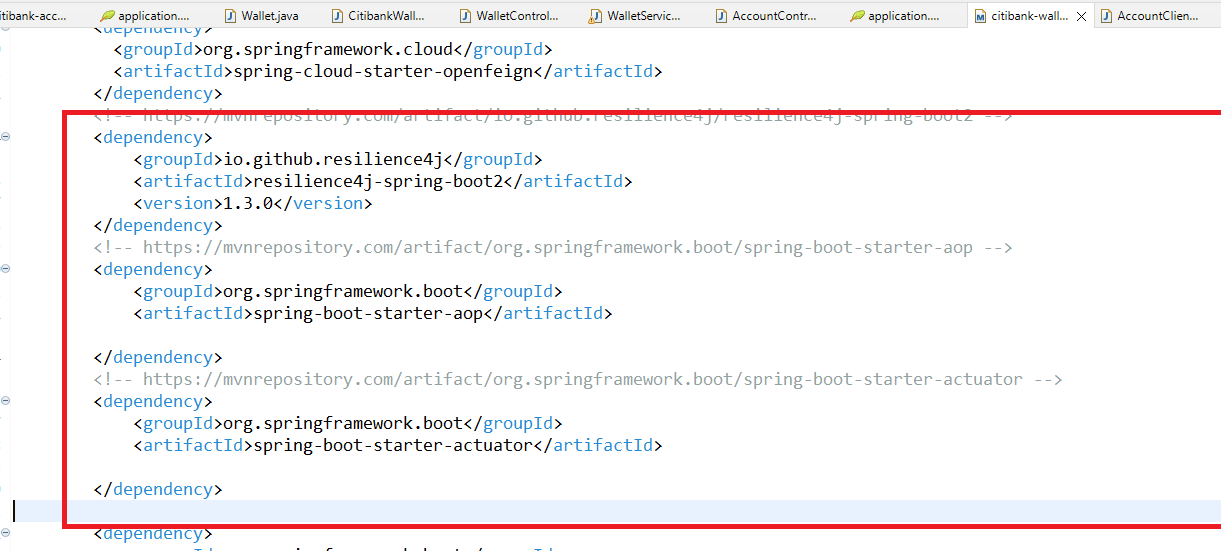
application.properties

We must write circuit breaker configuration like threshold limit, wait duration, request limit in close, in half-open state and etc.

Circuit Breaker uses something called Ring Bit Buffer to track the request status, based on the ring bit buffer the circuit break understands how many requests are failing or succeeding.

Dependencies we need to add in the wallet microservice

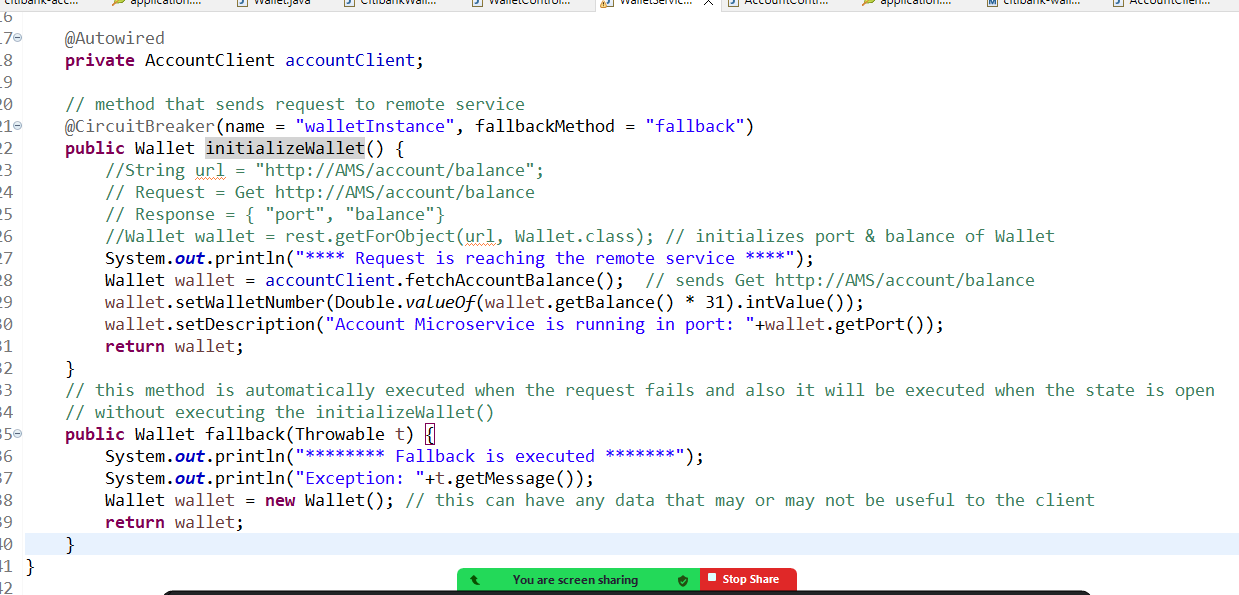
1. Resilience4j from io.github group-id
2. Spring AOP from spring boot starter group-id: we need this as fallback must be automatically executed without explicitly invoking
3. Actuator: This is required to monitor the application, this will give status, health information’s of the application



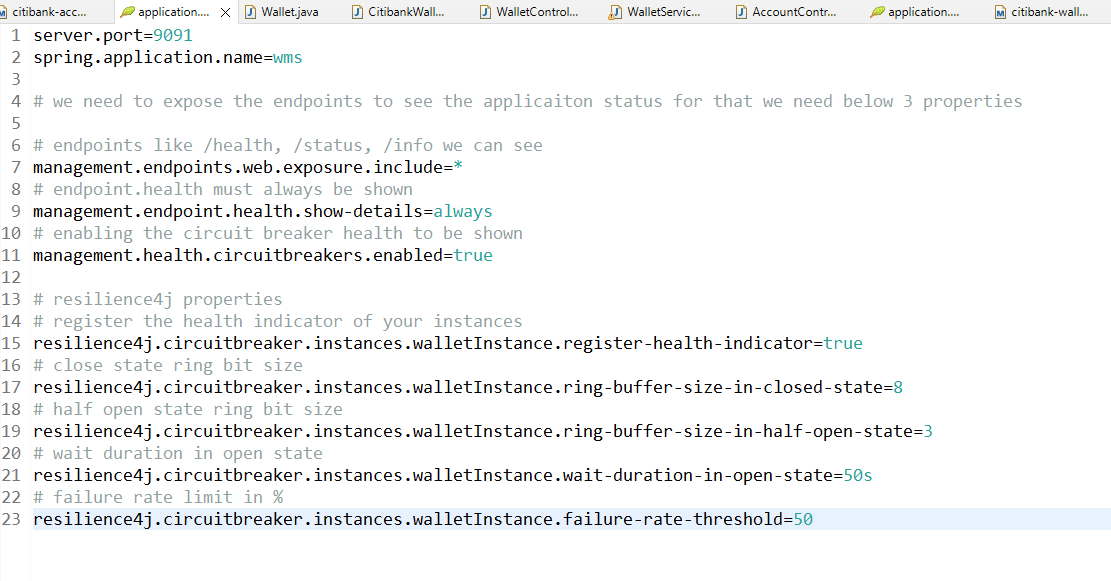
Steps to add the circuit breaker to our application

1. Have only one instance of remote service for wallet service
2. Use @CircuitBreaker on the method that calls the remote service
3. Create a fallback method that will be executed if the remote service request fails
4. In application properties enable all the circuit breaker features

WalletService.java

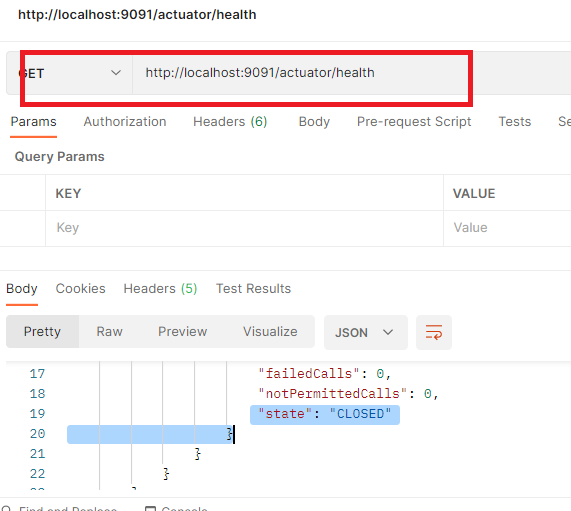


application.properties



Things to observe

1. Send few requests to account service from wallet service
2. You can use /actuator/health to see the circuit breaker properties in wallet service i.e., http://localhost:9091/actuator/health
3. Stop the account service
4. Send few requests & observe the console to see the request failing and going to the fallback
5. In wallet microservice you can use /actuator/health to see the circuit breaker informations like state, failedRequests
6. Wait for 50s to see circuit breaker state to go to half\_open from the open state after the failed requests reached the threshold limit
7. Start the remote service & observe that state changes from the half-open to close



Agenda

1. API Gateway - Zuul
2. Securing microservices
3. Deploying the microservices in AWS cloud
4. Distributed Transaction in Microservices - Saga Pattern
5. Testing

API Gateway

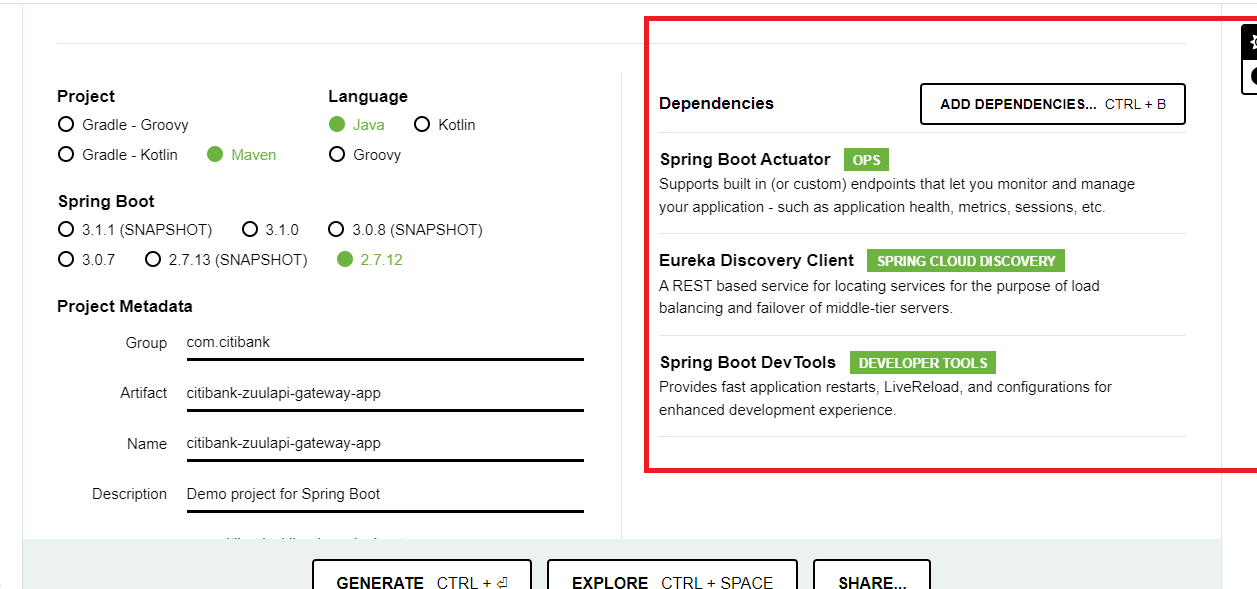
* It is used to achieve reverse proxy feature where client can use one URL and API gateway can route the request to another URL
* You can intercept all the requests & execute pre & post logics i.e., before request reaches to the service you can run some logic & after response is created you run some logic and send that response to the client
* You can expose the public URL of API gateway and run your microservices in private network so that clients wouldn’t have direct access to the service
* You can also monitor the requests

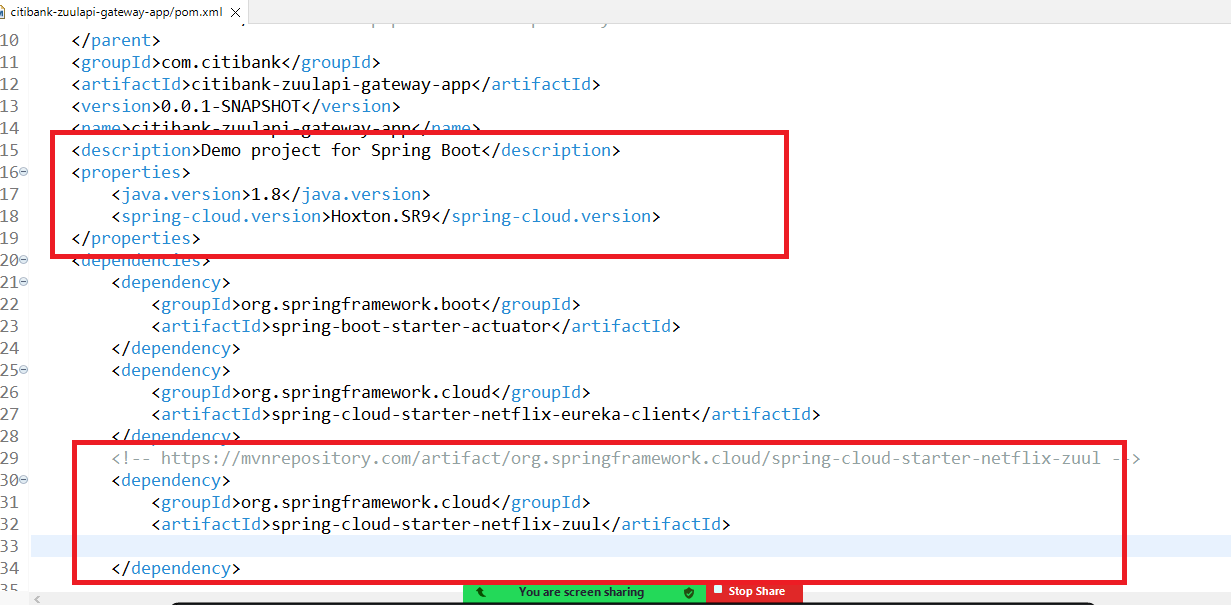
Zuul: It is a library you can use to create API gateway, it gives you an annotation called @EnableZuulProxy.

Note: You must create API gateway a separate project & register in the service discovery so that all the microservices you can access via API gateway

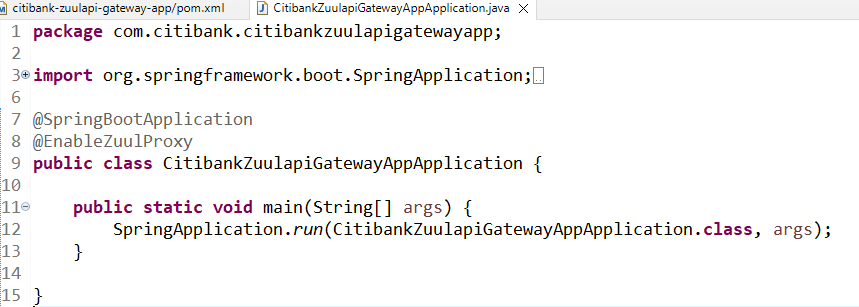
Dependencies

1. Zuul
2. Devtools
3. Actuator
4. Eureka Client

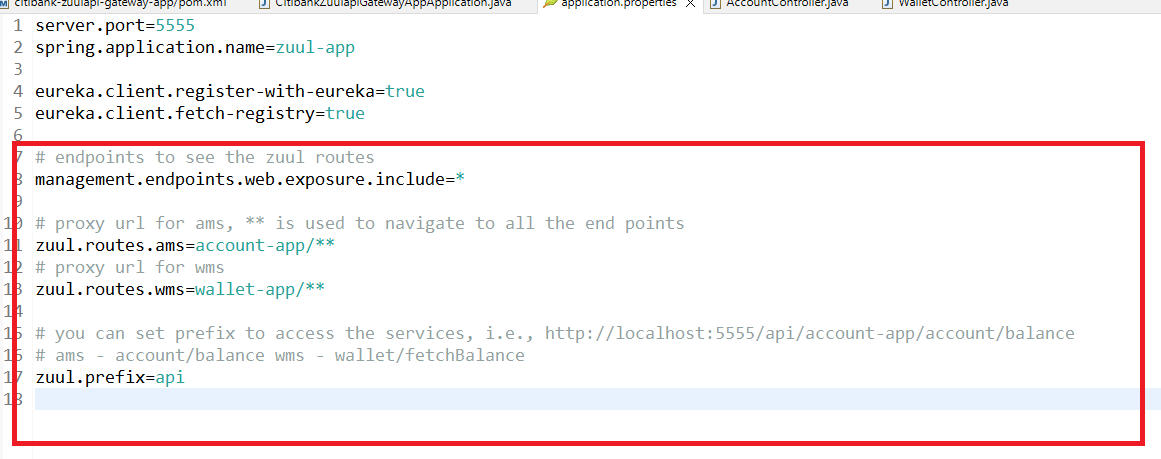




Add @EnableZuulProxy to add API gateway



application.properties



pre filters & post filters

pre filters are executed before the request reaches the service

post filters are executed before the response reaches the client

You need to create @Configuraiton class & extend ZuulFilter



You need to create another Filter that extends ZuulFilter, where filerType will be “post” and same run() method will have a time calculation logic & printing in the logs

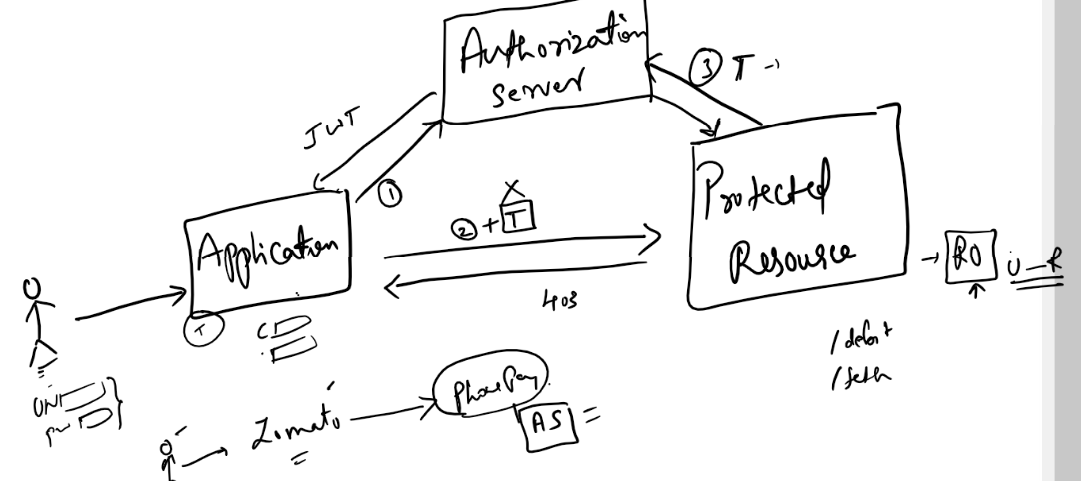
Securing Microservices

Authentication & Authorization (Auth2)

Authentication means to check the user identity

Authorization means to check the user has a permission

OAuth2 it uses 4 components to design secured microservices



When user tries to access some resource behind the scene a request will be sent to the authorization server having user & application credentials in the request header, based on that authorization server generates the token & application would send the request to the resource along with the token, later the resource would send this token to the authorization server to validate & based on the response from the authorization server protected resource can decide whether a request needs to be processed or not.

Resource Owner will be present in the protected resource who decides which type of user has access to what resource

like /store is accessed only by Admin role, /fetch is accessed by both Admin & Guest

Postman acts like user & application which needs to send request to Authorization Server with users & application credentials

Postman needs to send the token given by Authorization Server to the Protected resource

All these things it can do using Authorization Header & Body of the request

We need to use two projects

1. Authorization Server
2. Protected Resource