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

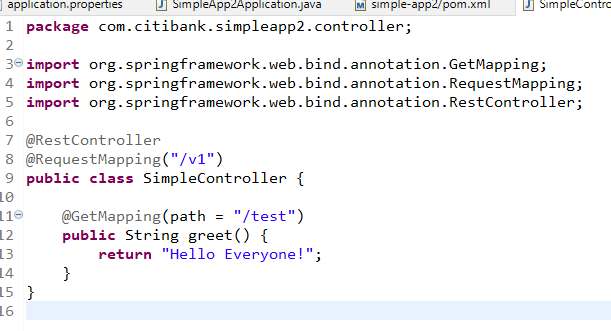
REST stands for Representational State Transfer which helps two applications written in different technologies to exchange the data in a common format like JSON, XML, CSV & etc.

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



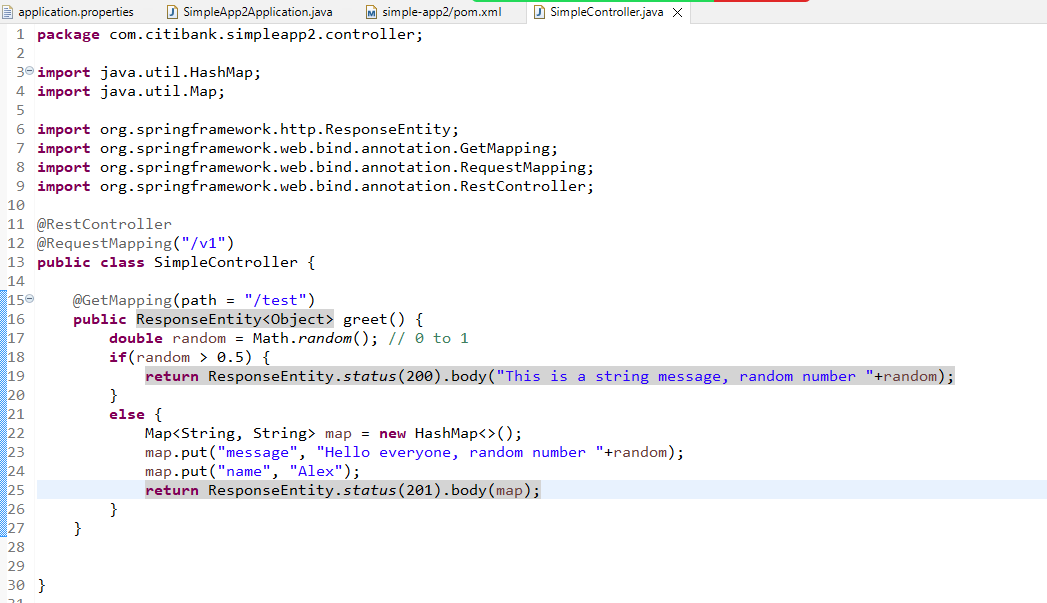
The above code only returns text data & by default the status code will be 200, but we can customize the response status so that we can send different status codes like 200, 201, 204, 401, 404 and so on

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

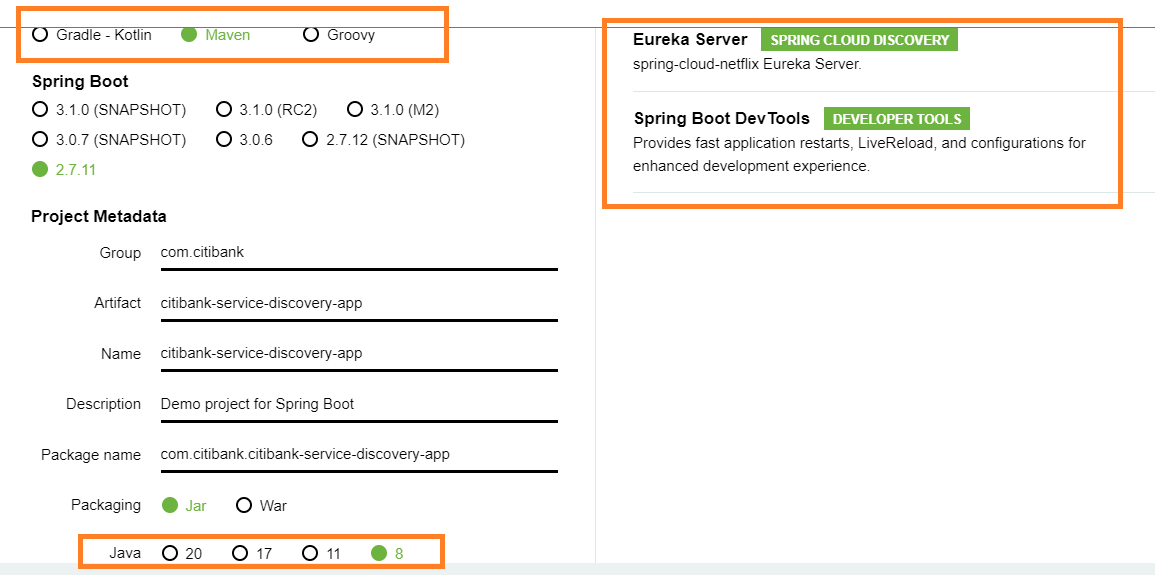
1. Spring Cloud: It provides all the necessary tools & design patterns to implement microservices like Service discovery (Eureka Server), Discovery Client (Eureka Client), Client Side Load Balancer, Circuit Breaker Pattern and etc.
2. Spring Boot: It takes care of auto-configurations based on the library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

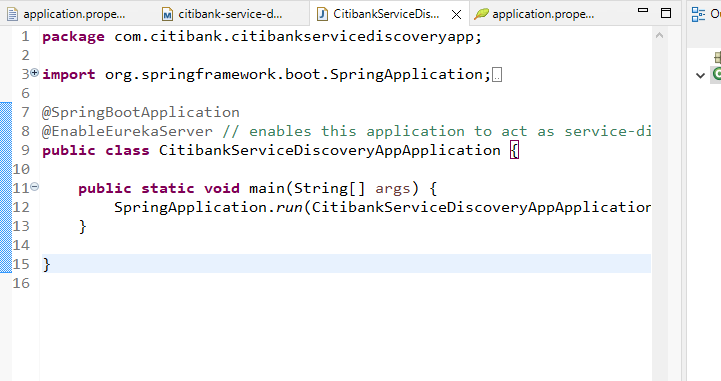
Service Discovery

1. Eureka Server
2. Dev tools

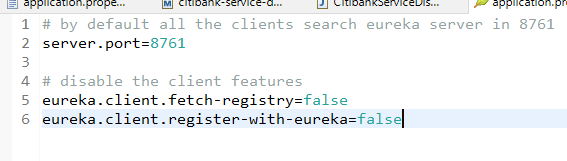
Note: Eureka Server depends on Eureka Client which means service discovery will try to automatically register itself as a microservice because of spring boot, we need to disable the client feature in service discovery in application.properties



Enabling Eureka Server



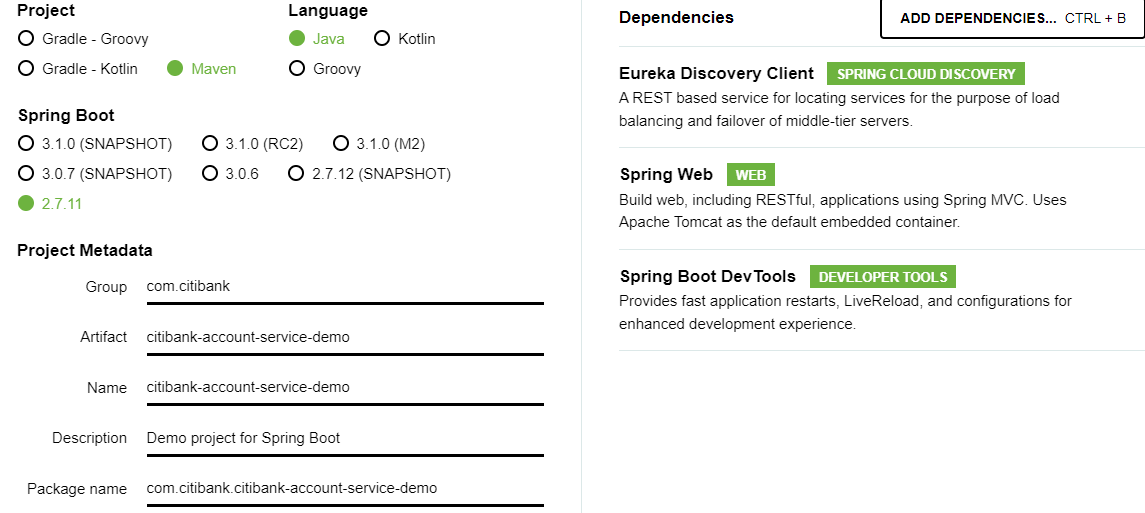
application.properties

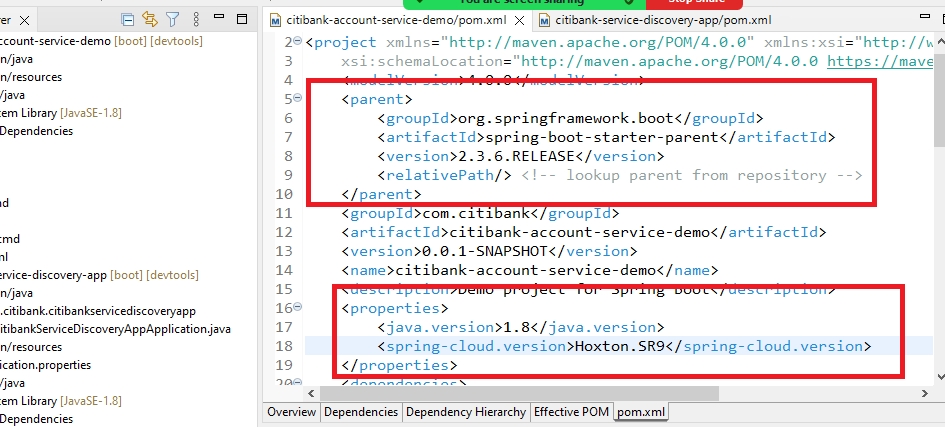


Creating clients for service discovery i.e., microservices which needs to have instance-id

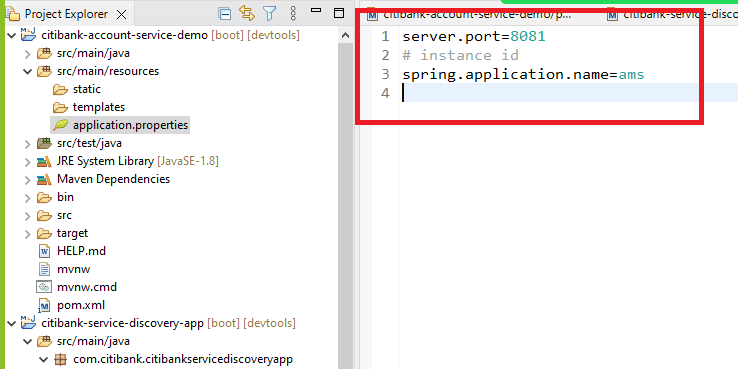
Dependencies

* Eureka Client
* Web
* Devtools

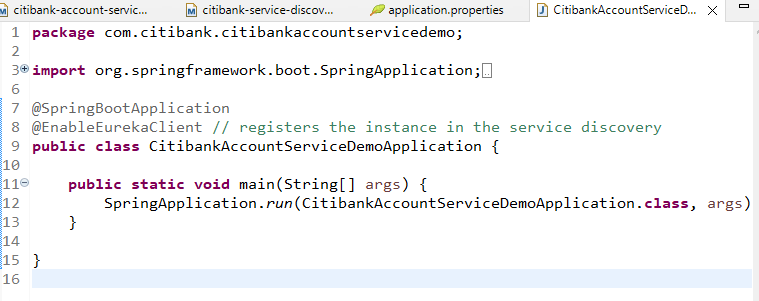




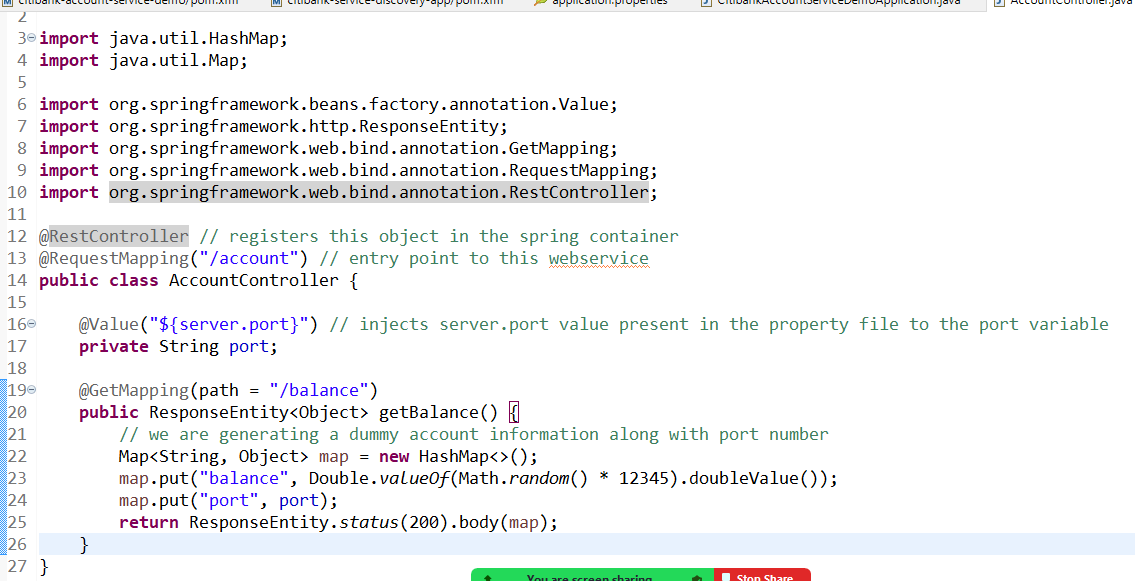
application.properties



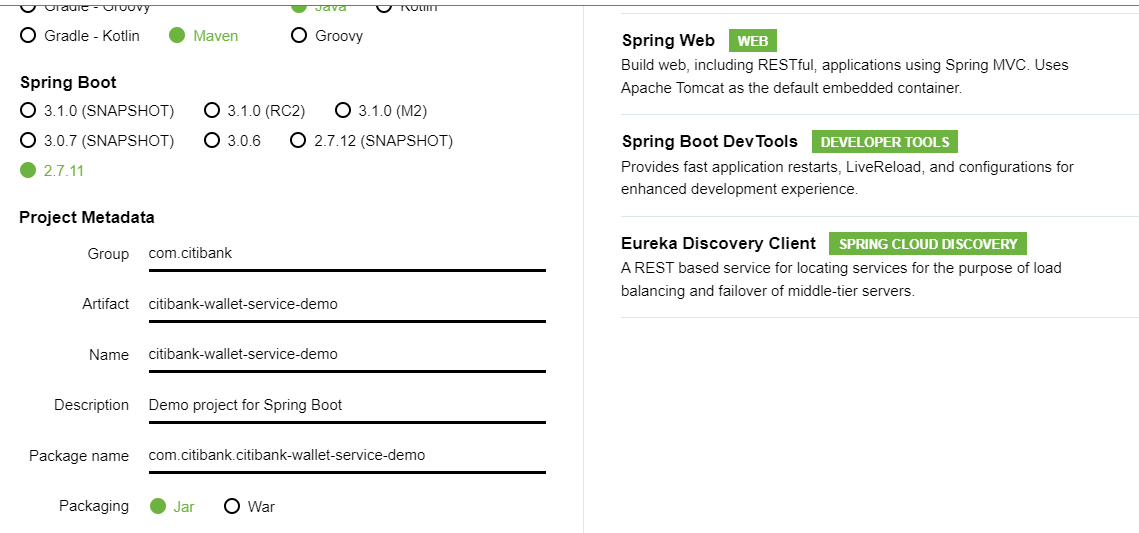
Enable the project to act as a microservice



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



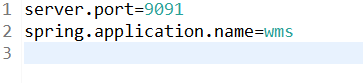
Wallet Microservice



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

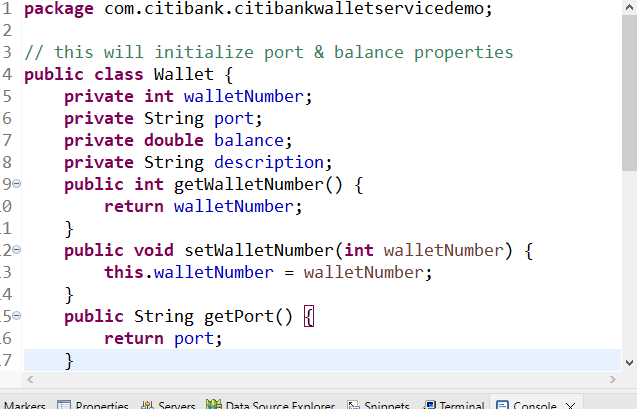


application.properties



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

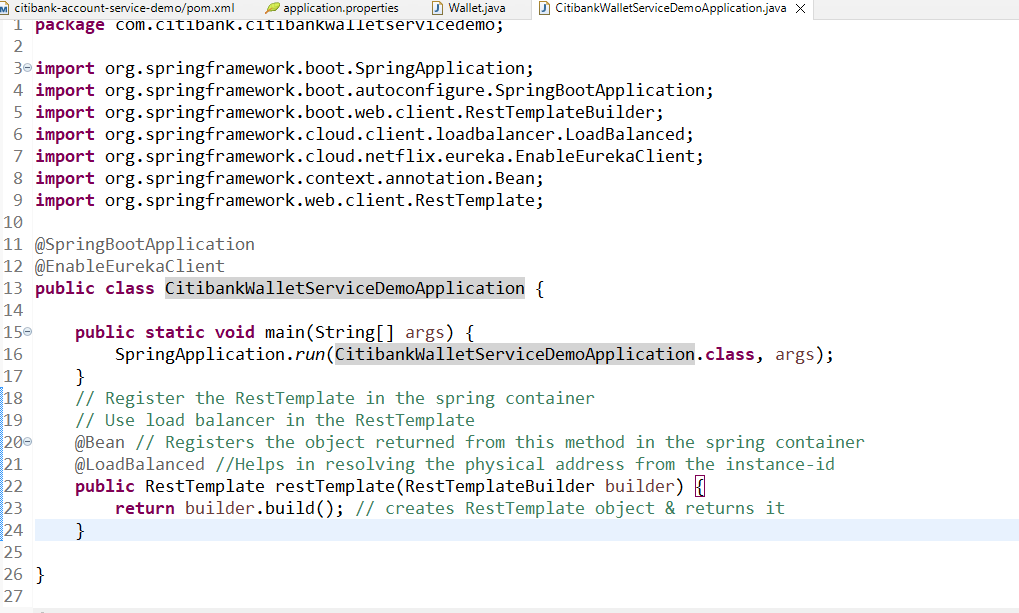
Wallet.java



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

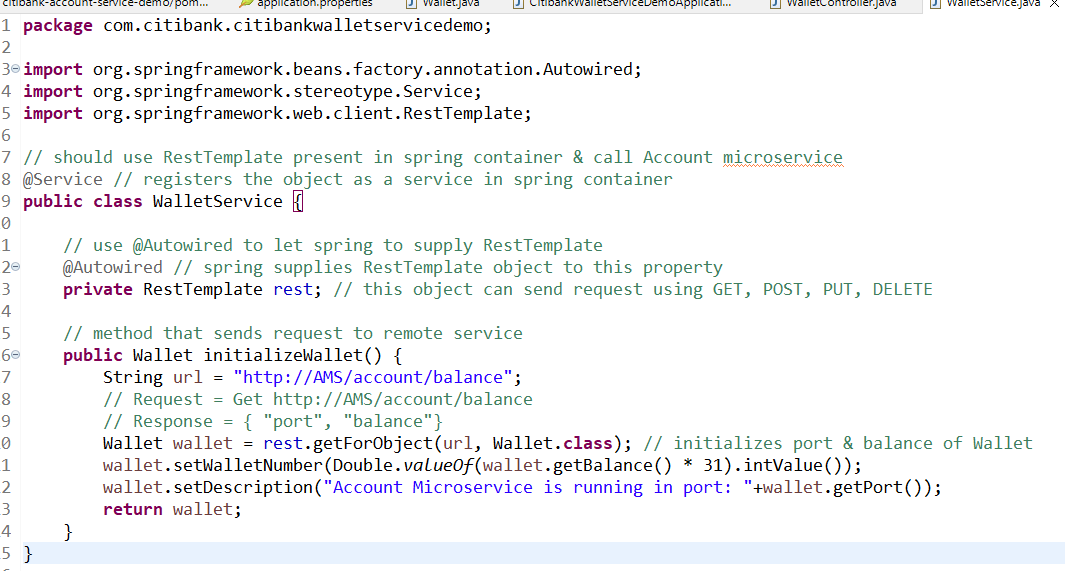
1. Register the RestTemplate in spring container so that it is available through the application
2. Bind the RestTemplate with Ribbon client i.e., load balancer so that load balancer can resolve the physical address using the IP address.



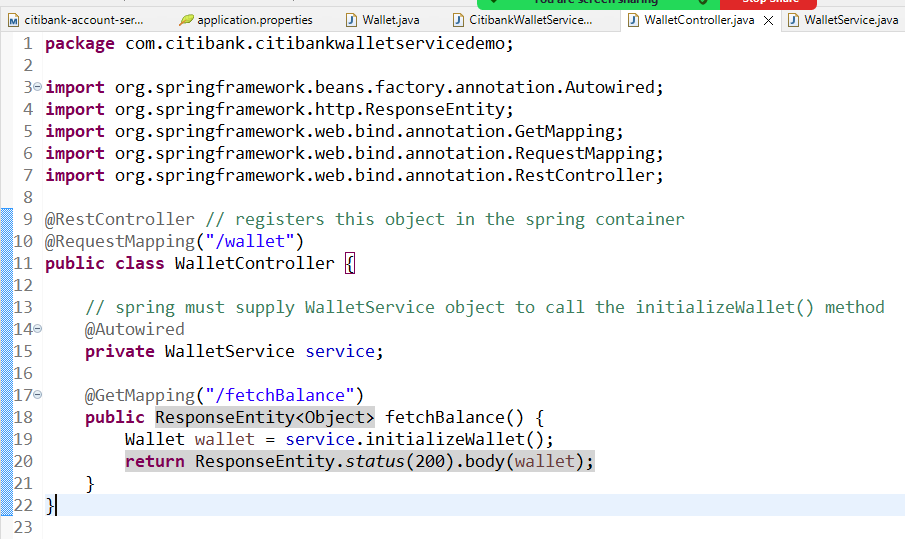
Since calling a microservice from another microservice is a business logic, we need a service layer in our code, this service layer is accessed by the controller layer, hence we need to create 2 classes

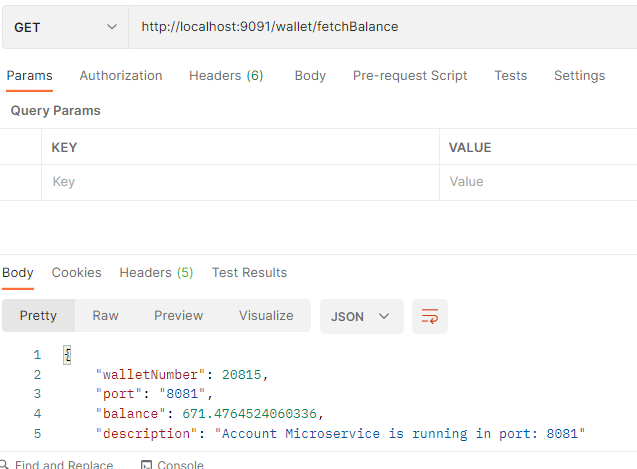
1. WalletService
2. WalletController

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)