Microservices

Microservices are small services that can be independently developed, built and deployed.

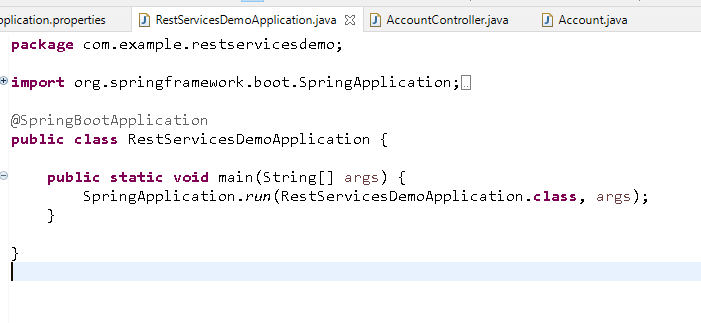
Spring provides a module called spring microservices which allows you to develop microservices.

Pre-requisites

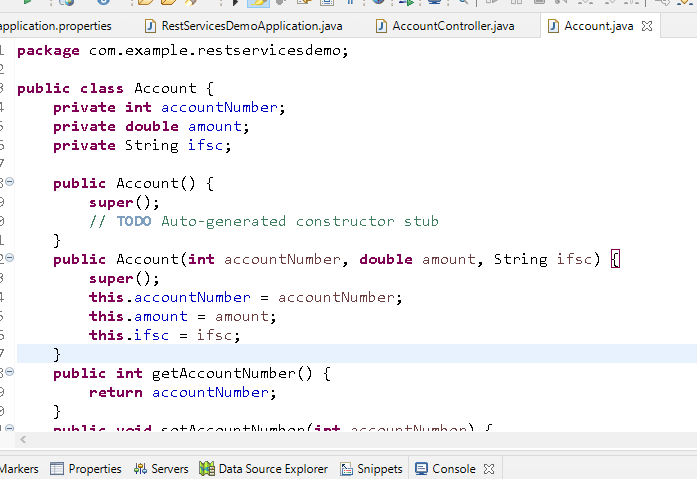
* Java
* Spring Boot
* Spring REST
* Spring Initializr project

Simple Spring REST project with Spring Boot

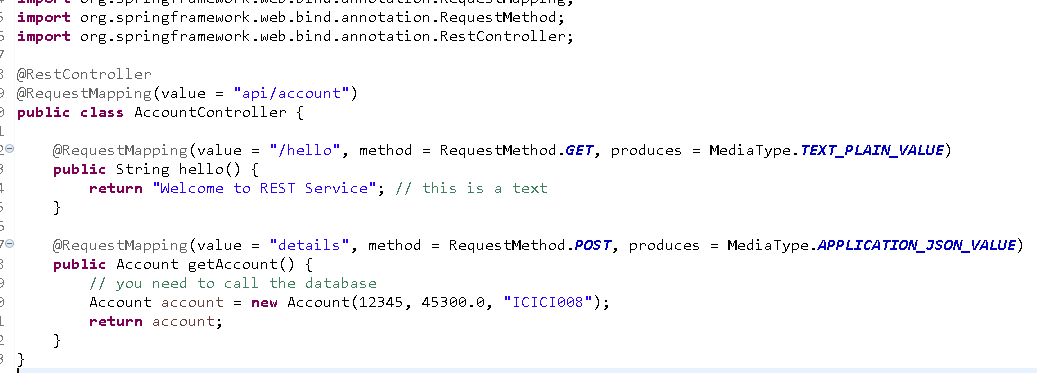
@SpringBootApplication



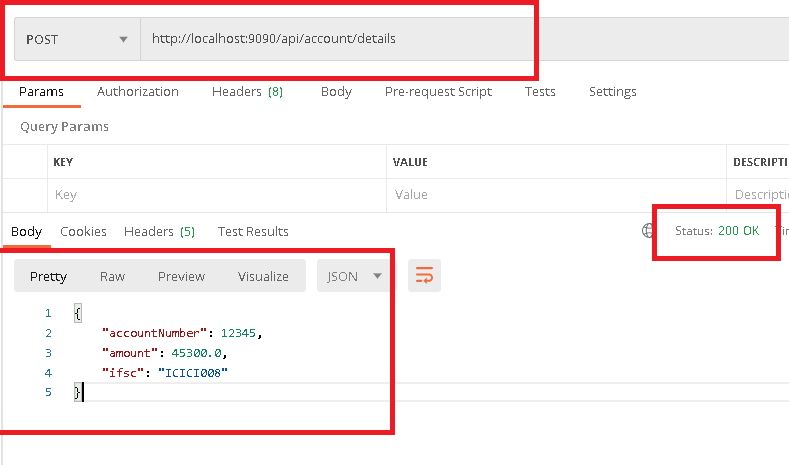
Account.java



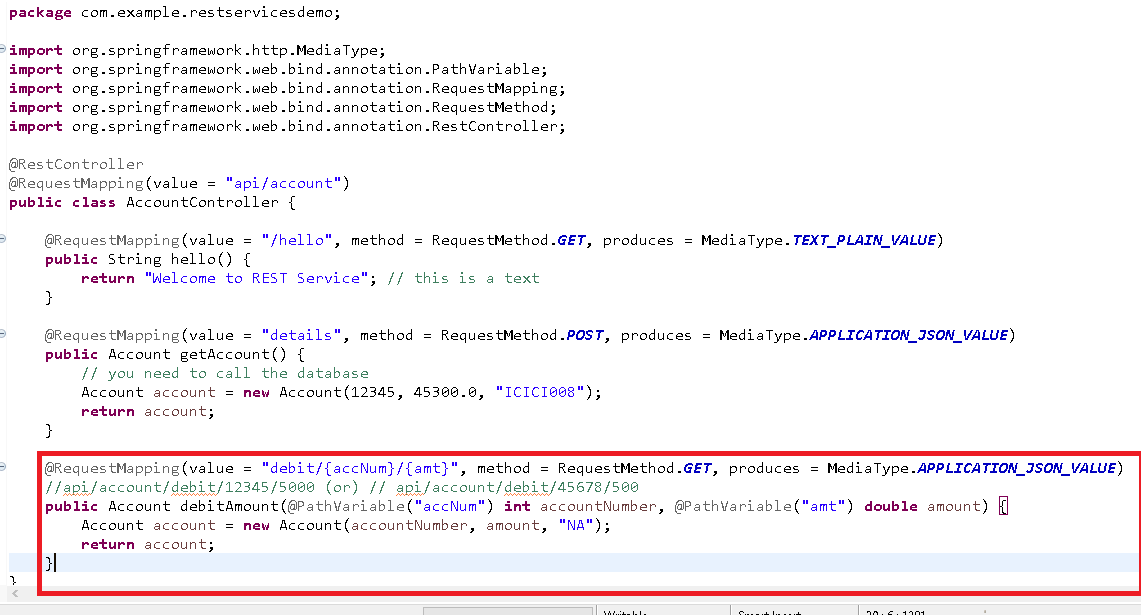
AccountController.java



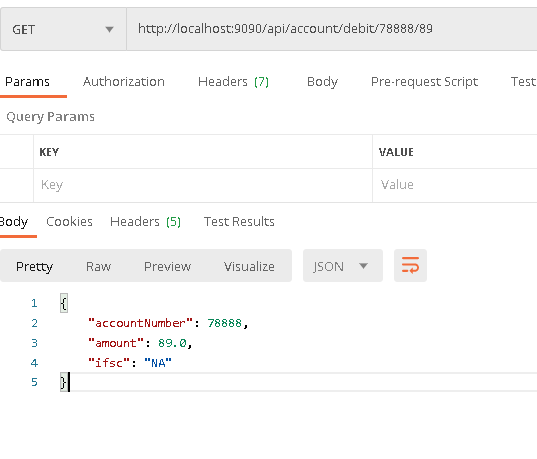
Output:



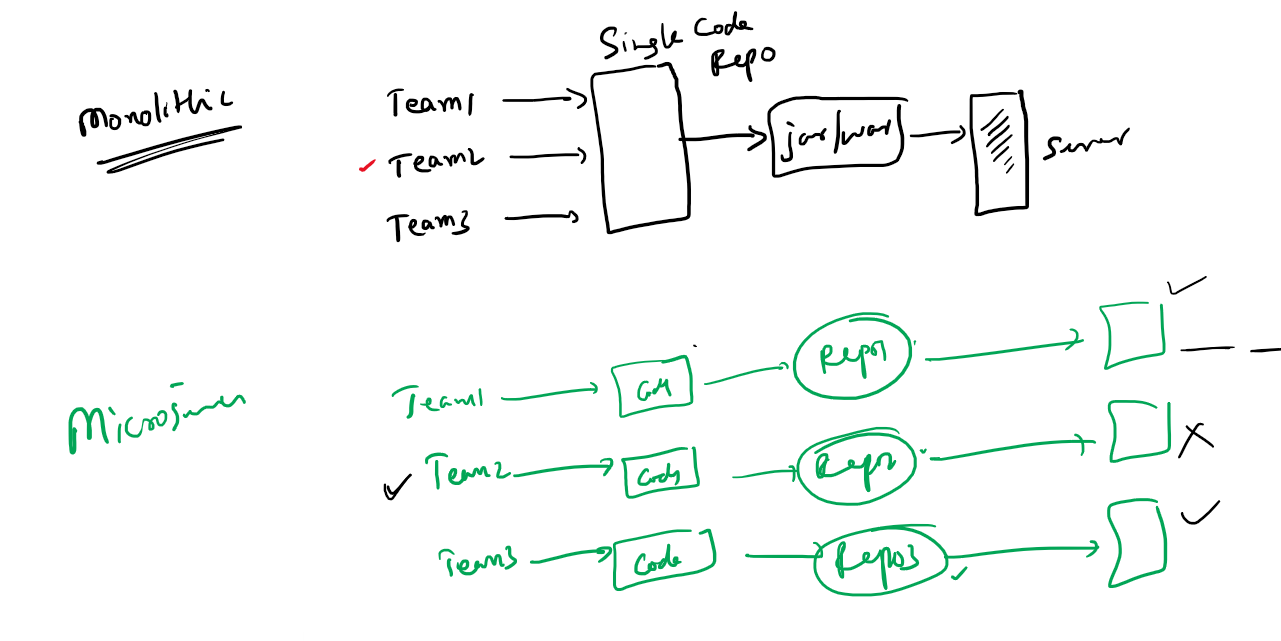
Client can send input via a text box in mobile or web or from credit card machine and so on, you must know how to extract those inputs in the rest service



Output



Monolithic vs Microservice architecture



Spring Framework integrated with Netflix OSS to make microservices developed through spring

Spring Framework released a module called *Spring Cloud* which is dependent on *Spring Boot*

Using these two projects you can quickly develop microservices with simple annotations.

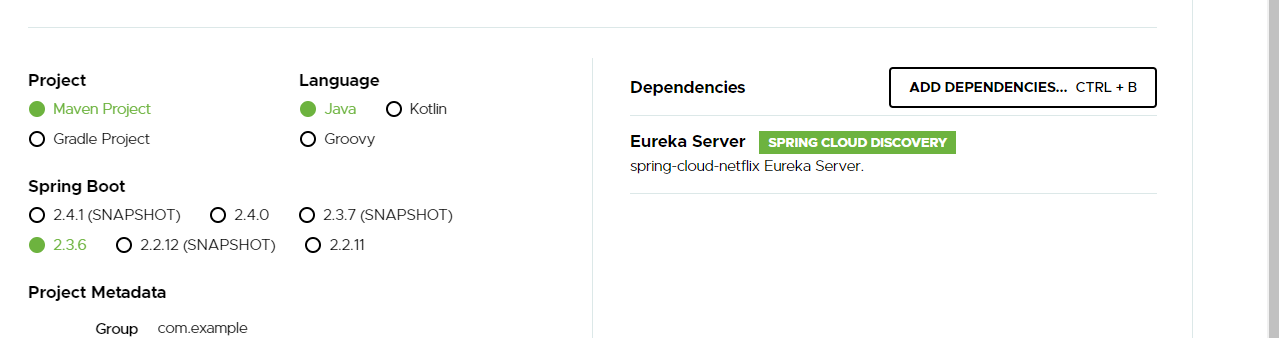
Steps involved in microservices

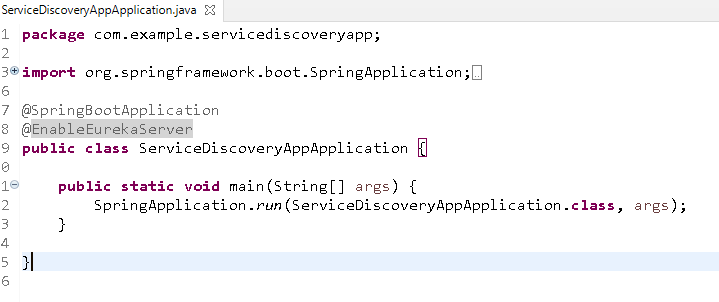
1. Service Discovery: where all the microservice registers to this registry so that microservices can locate other microservices
2. Discovery Clients: These are microservices that are called as clients which registers with the Service Discovery
3. Configuration Server:
4. Circuit Breaker:
5. API Gateway:

Service Discovery: It is implemented by Netflix and in spring you will use Eureka Server for Service discovery

Discovery Clients: It is implemented by Netflix and in spring you will use Eureka clients to register with Eureka server

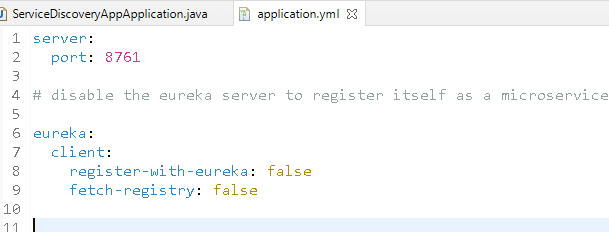
Creating Service Discovery





@EnableEurekaServer: creates a service discovery where all the @EnableEurekaClient would be registered.

Service Discovery acts like a client as well so you must disable few properties in the application.yml file

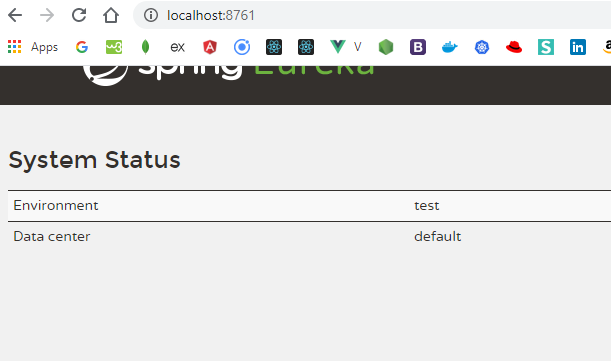


server.port: 8761, this is the default port all your microservice will register with the service discovery

eureka.client.register-with-eureka: false, this disables service discovery to register itself in its registry

eureka.client.fetch-registry: false, this disables client to fetch informations from service discovery

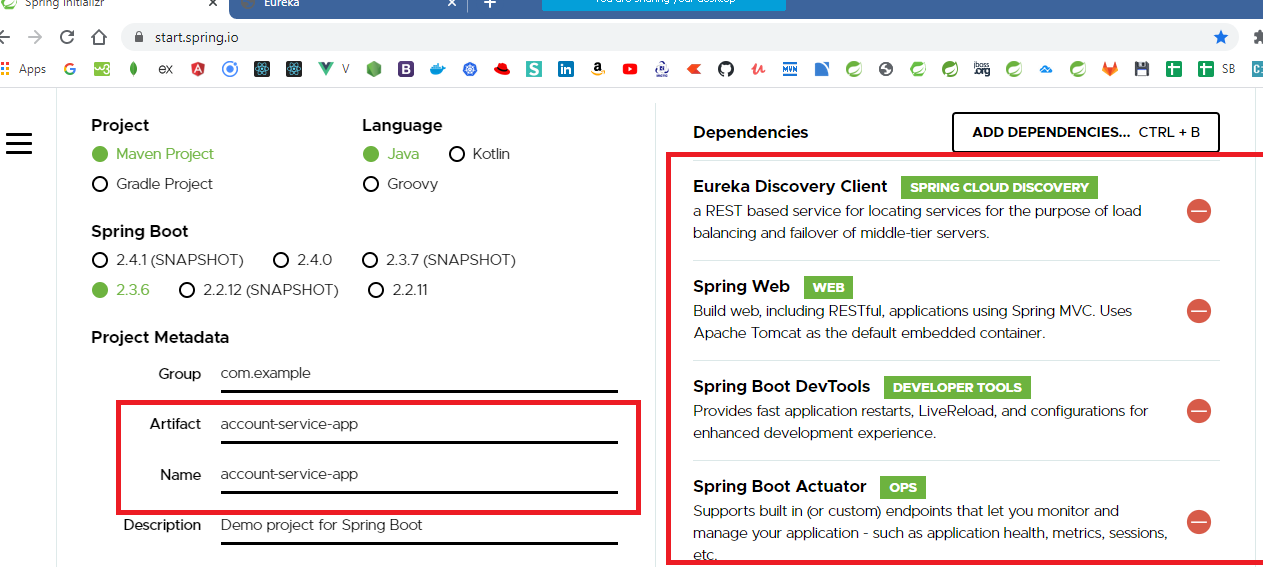
Output:

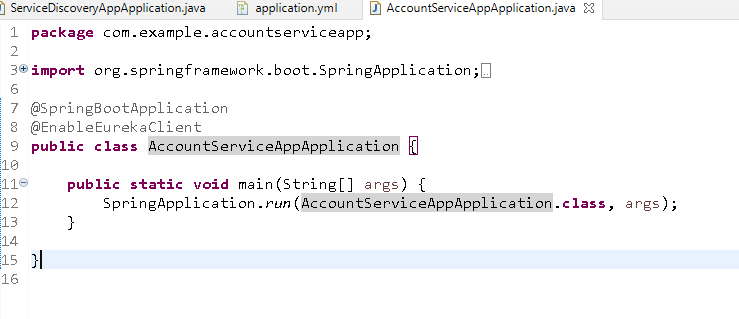


This dashboard shows all the registered microservices

Creating a microservice

1. Web
2. Eureka Client
3. Devtools (optional)
4. Actuator (optional)

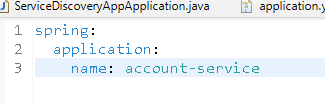




Now this is a microservice that tries to register with Service Discovery running in 8761 port

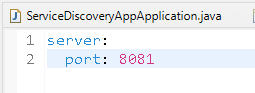
However every microservice needs a logical name that helps other microservices to communicate

bootstrap.yml

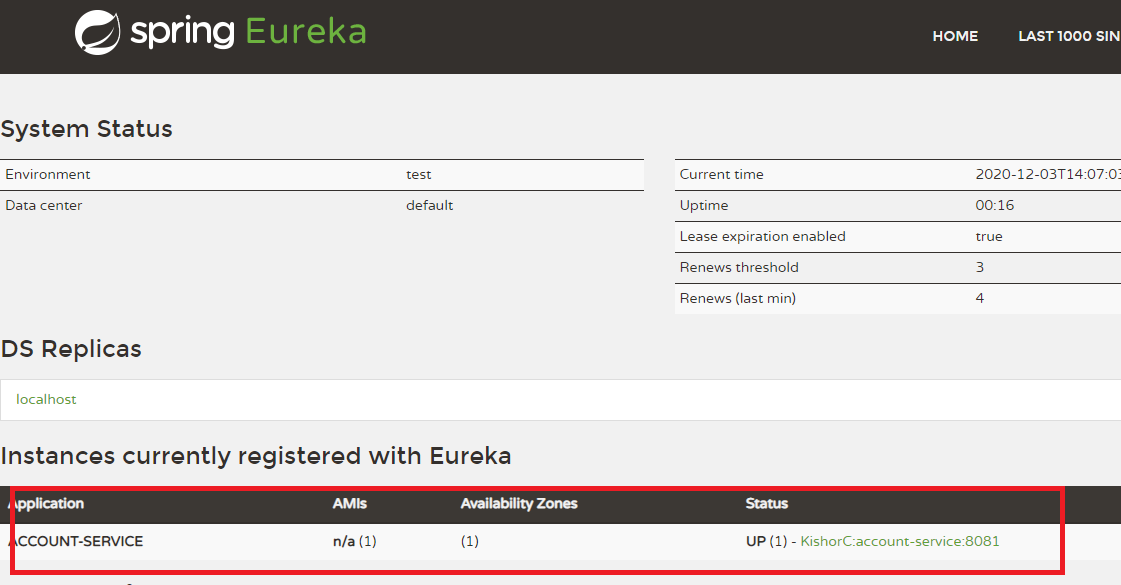


This is the file that is loaded before application.yml, you keep some configurations that should be loaded before application.yml, like application names, profiles, configuration servers url and so on

application.yml



Output:

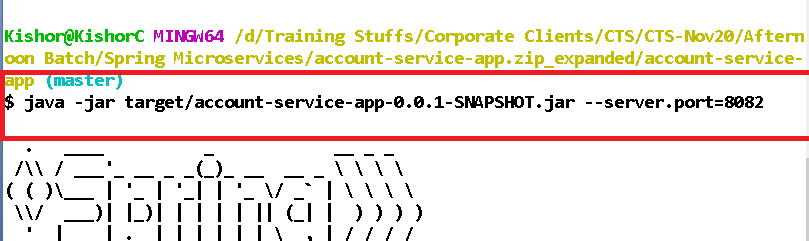


Eureka has only one instance of Account-Service if you need to multiple instances of account service then you need to launch this service in another port other than 8081, because the instances created in local machine.

You can use below commands from your project location

mvn package

java -jar target/file-name.jar --server.port = 8082



This will show up in the eureka dashboar, we are running one instance in eclipse & other in command prompt of account service

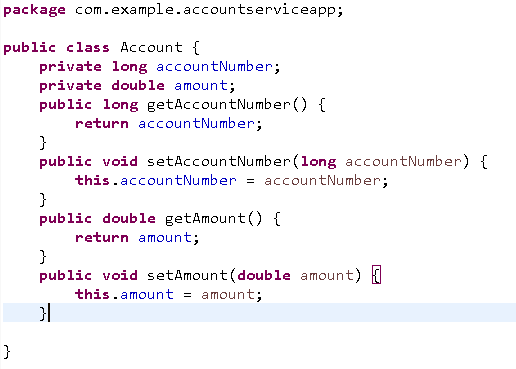


Exercise:

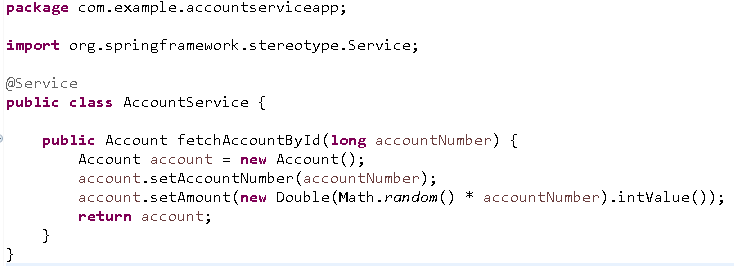
Create another microservice named paytm service and register with eureka with a different application name

Store this exercise in cts-hands-on repository in afternoon folder with another folder named microservice

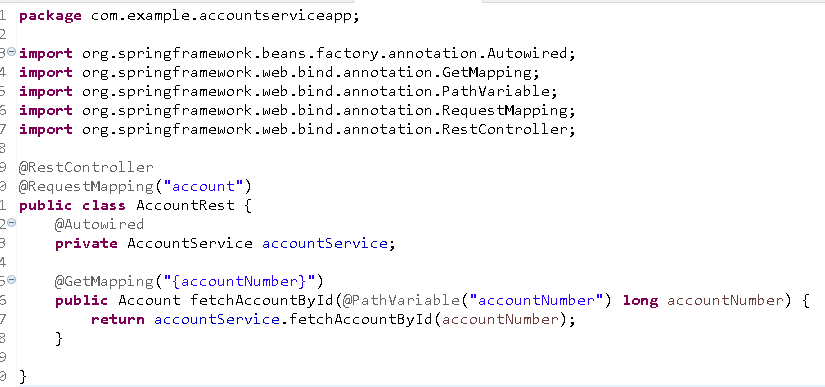
Account.java



AccountService.java



AccountRest.java



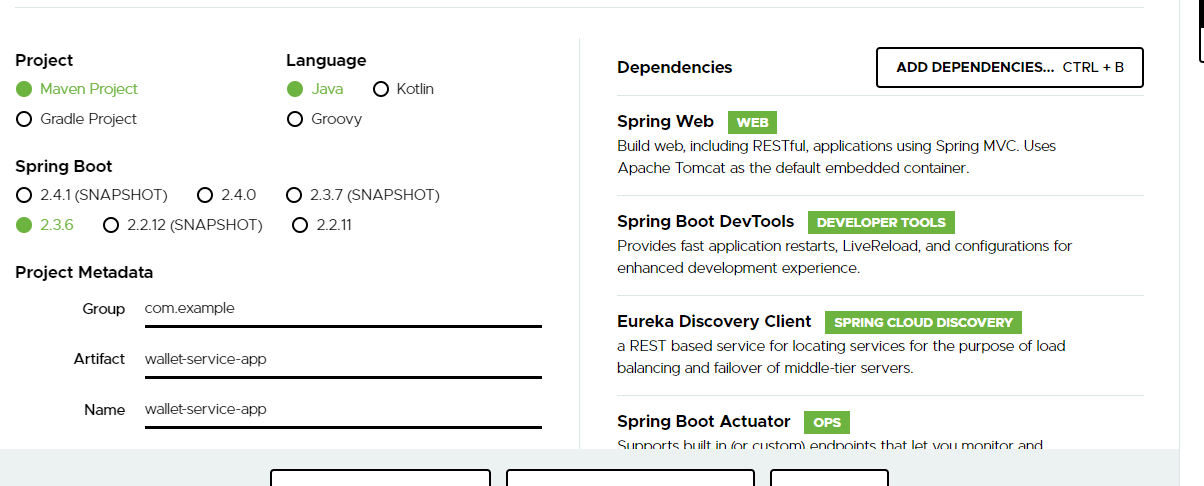
Here the REST endpoint to access account service is <http://localhost:8081/account>, however the microservice which has to communicate will not have idea about the other microservice location, so they will use the logical name of the microservice registered in the service-discovery, i.e., **ACCOUNT-SERVICE**

Communication between the microservices

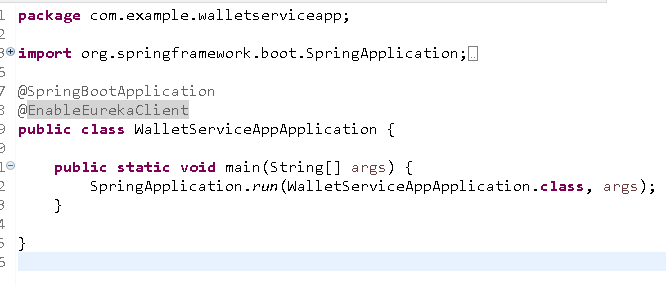
When any microservice has to communicate with other microservices they have to use

* service-id or logical name registered with service-discovery
* instance that can make REST calls, for ex: In Spring you have RestTemplate

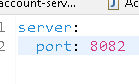
Wallet Service communicates with Account Service



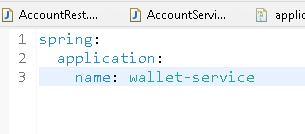
Add Eureka Client annotation



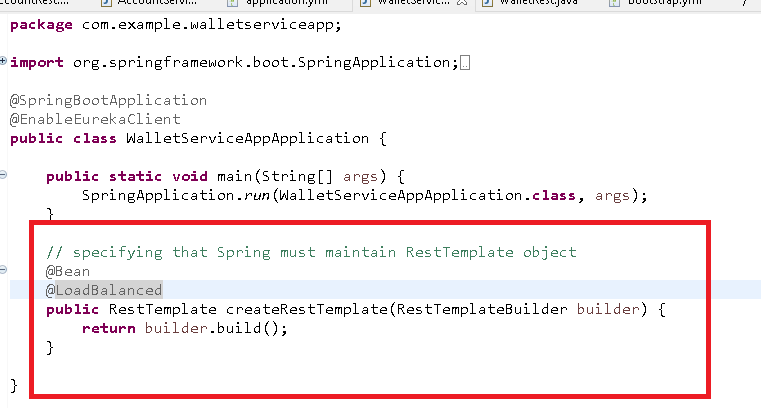
application.yml



bootstrap.yml

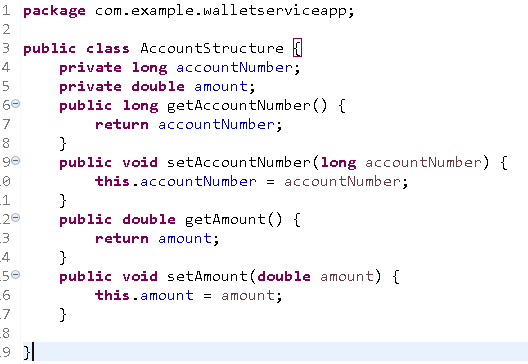


Creating RestTemplate instance with LoadBalanced

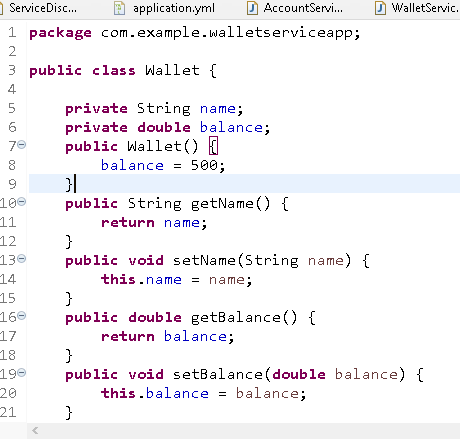


@LoadBalanced creates the load balanced backed RestTemplate object

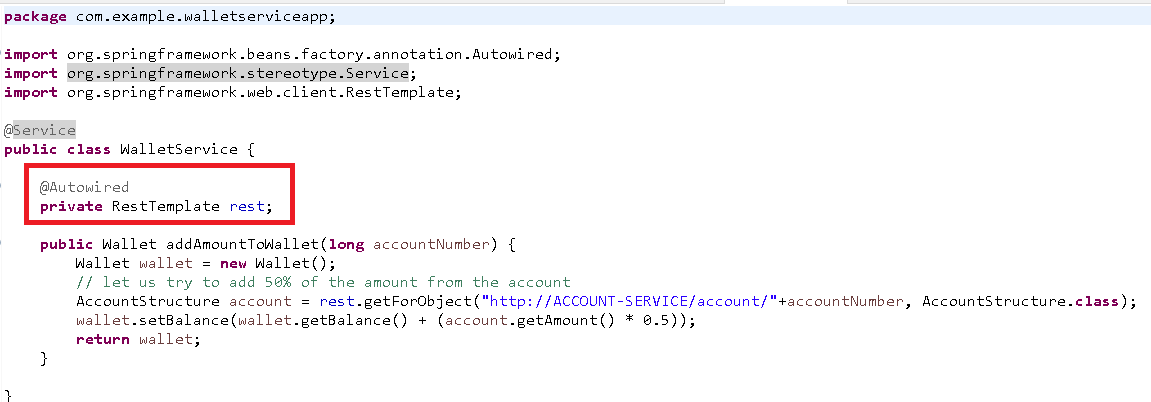
AccountStructure.java: This must match to json structure



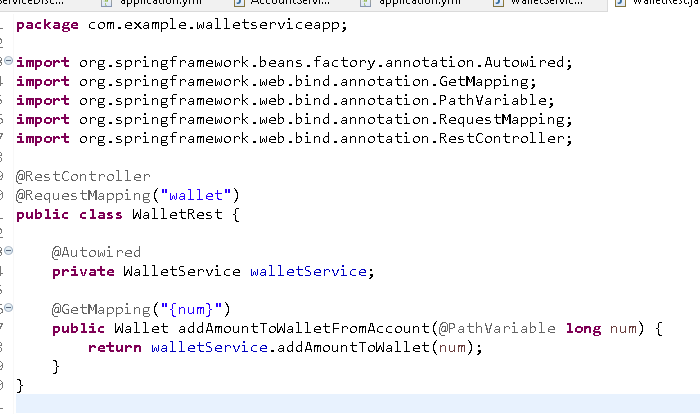
Wallet.java



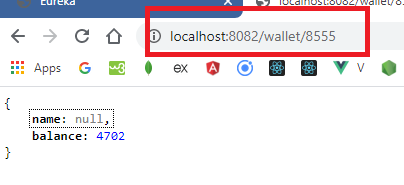
WalletService.java



WalletRest.java



Output:



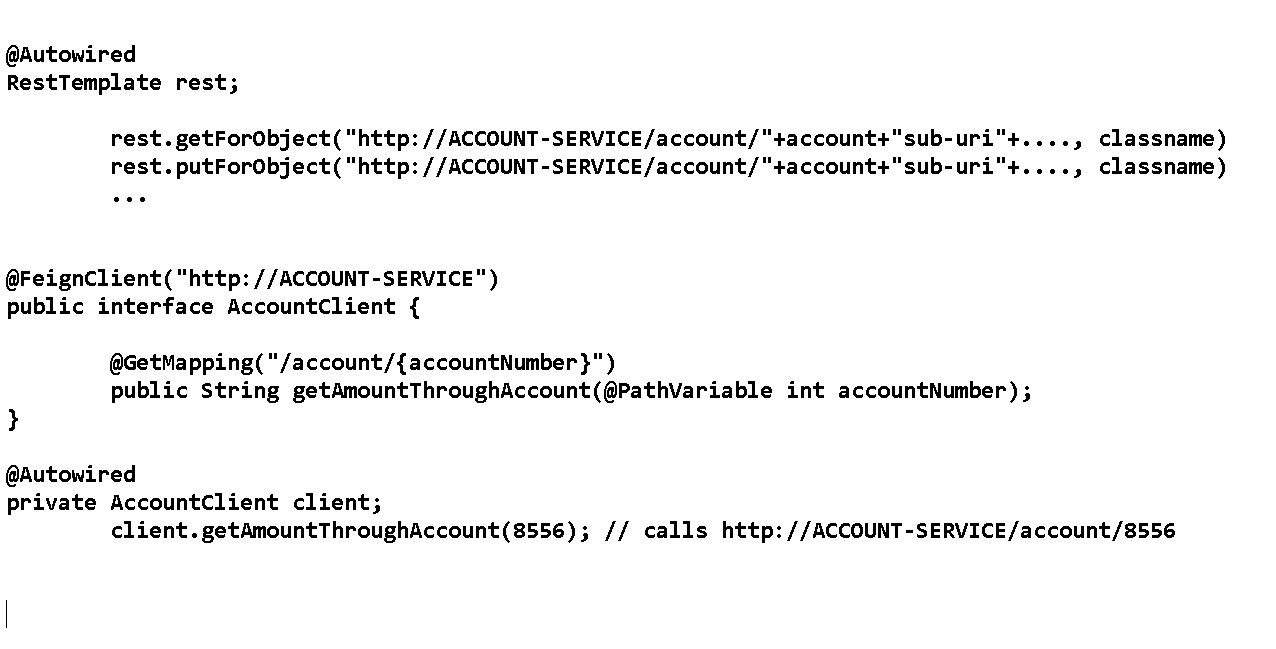
Note: name is null because we didn’t initialize it in wallet object, <http://localhost:8082/wallet/8555>

Sends request to

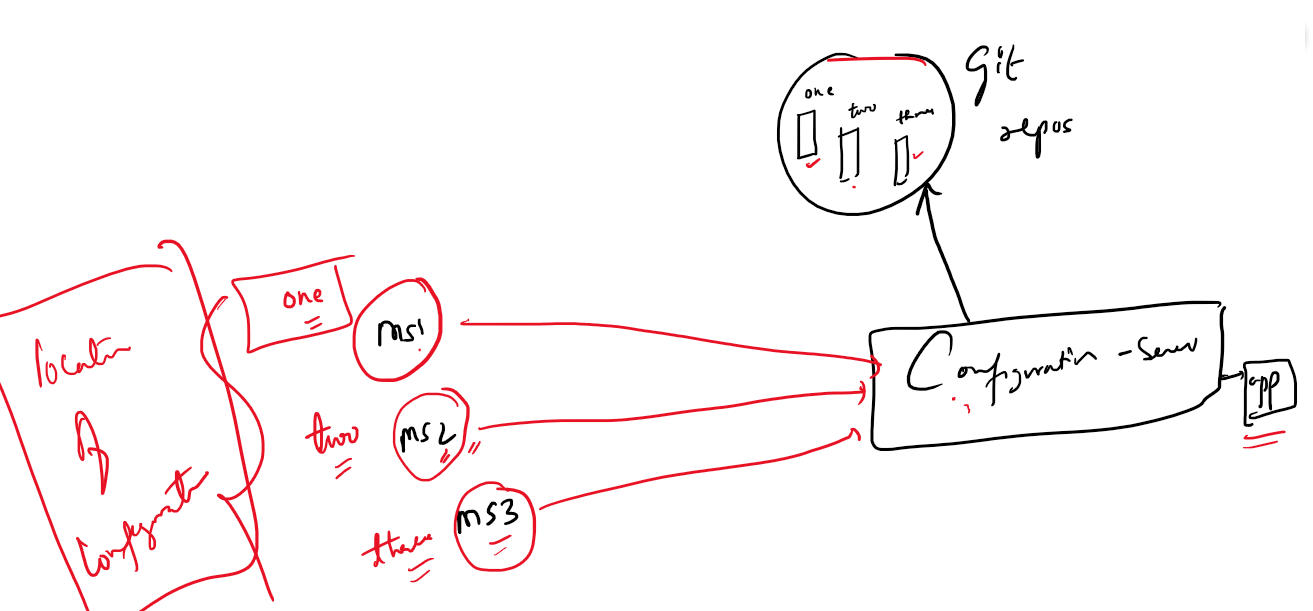
<http://account-service/account/8555>

Feign Clients: This is used as an alternative to the RestTemplate to call the webservices

It is reusable compare to RestTemplate, because it is going to be used with the help of interfaces with some methods that maps to appropriate microservice



Configuration Server: It maintains the configuration files for multiple microservices which can fetched from the configuration server, these configurations you can keep in the GIT.



Configuration server is an application that will know the location of the configuration files and needs microservice to mention the configuration file it needs to fetch.

Configuration Server will know the GIT location

Microservices will know the Configuration Server location

Configuration Server needs only one dependency

* config server

Microservices that connects to configuration server acts as configuration client it needs a dependency

* config client
* web
* actuator
* devtools

We will create 4 configuration files that can be loaded based on the profiles [developer, tester, production, default]

content of client-one.yml

title: This is client one configuration

content of client-one-developer.yml

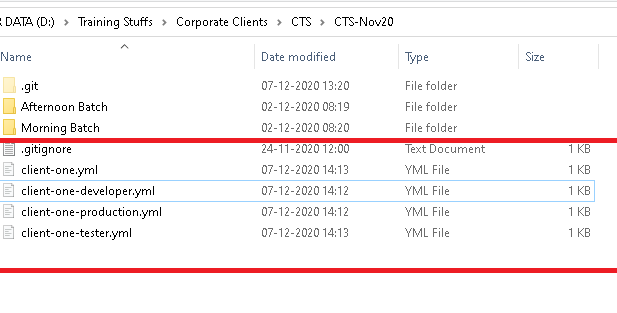
title: This is client one configuration for developer

content of client-one-tester.yml

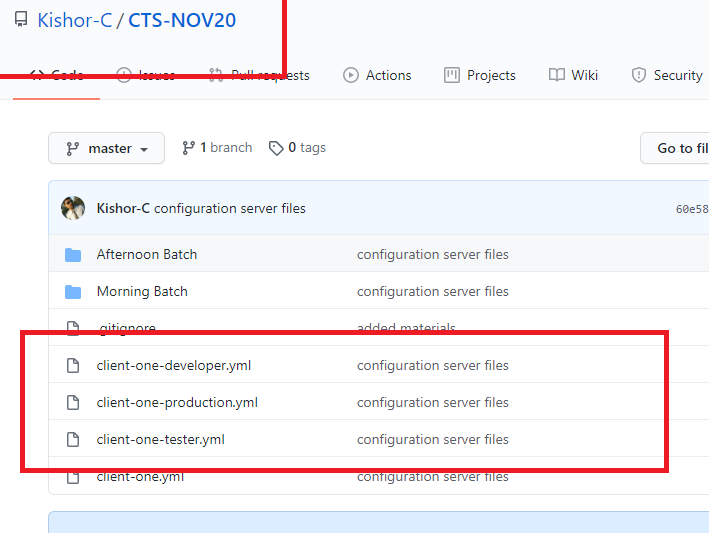
title: This is client one configuration for tester

content of client-one-production.yml

title: This is client one configuration for production



All these configuration files (yml) has single property title, that has to be read by config-clients by connecting to the config-server



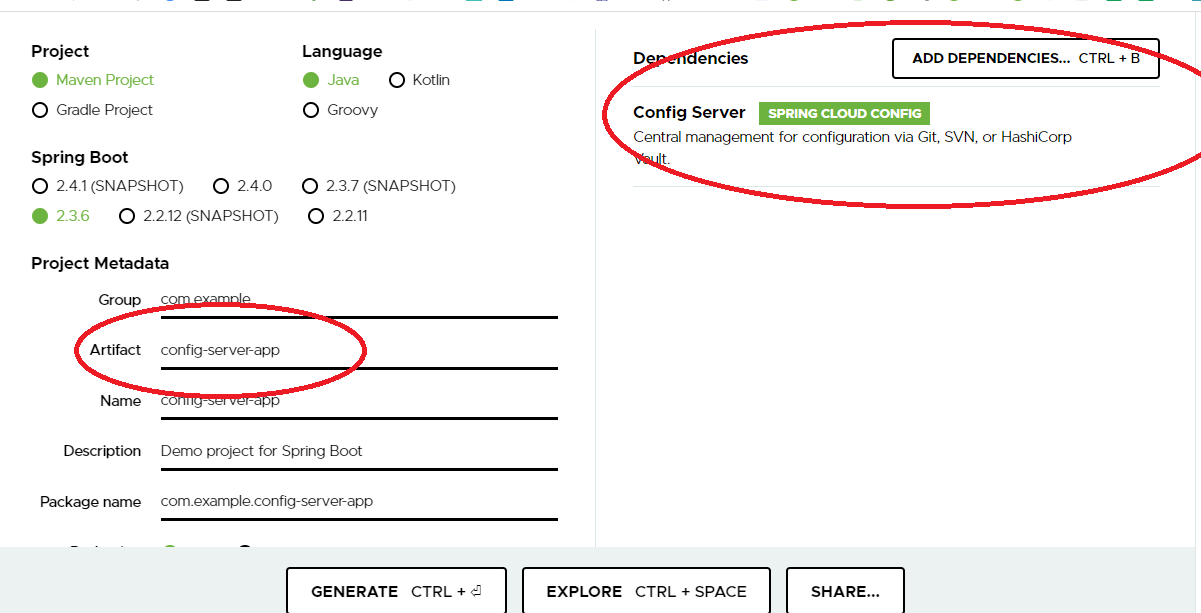
Configuration server will use this GIT URI in application.yml file

URI: <https://github.com/Kishor-C/CTS-NOV20.git>

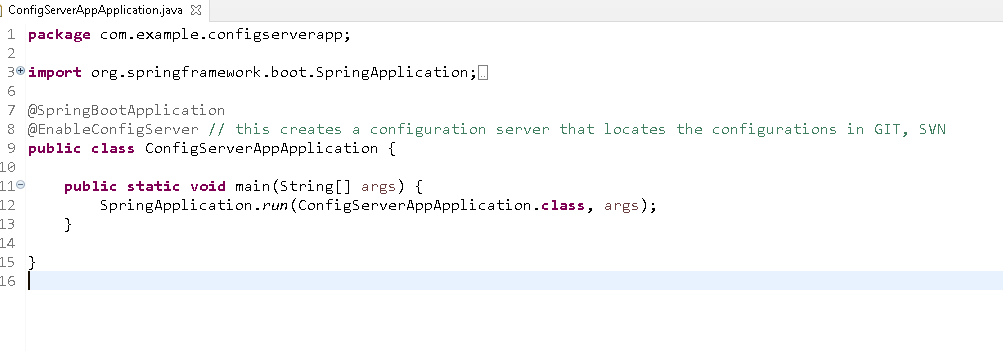
Configuration clients will use the Configuration server URI and mentions the application name as client-one so that client-one.yml file will be loaded, suppose the clients mentions the profile as developer then client-one-developer.yml file will be loaded.

Note: you have to mention them in the bootstrap.yml

Creating configuration server

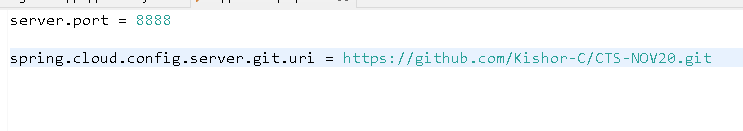


Creating the configuration server



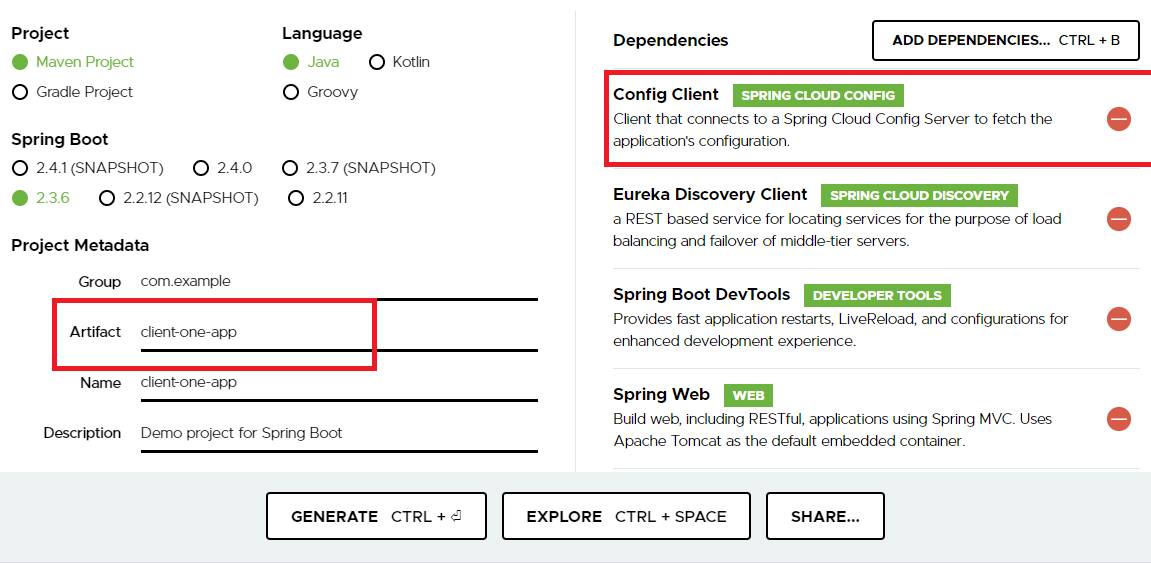
Specifying the uri of the git repository and also the port of the configuration server

application.properties

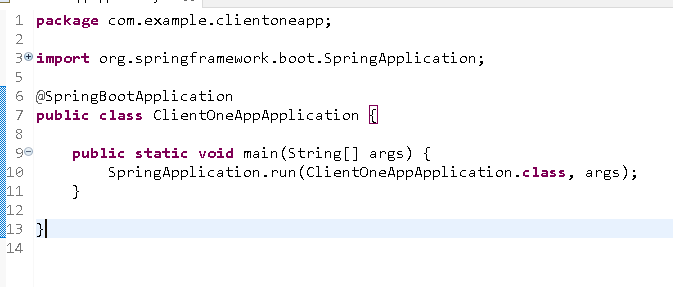


Now this is the configuration server which locates the configuration in the GIT repository, now all the config-clients must use configuration server location to fetch the configuration files

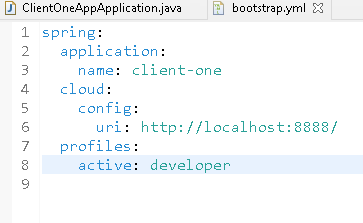
Create a config-client that can be a micro-service if needed, you must mention the config-server location in the bootstrap file



No change in the main class, but you can add @EnableEurekaClient if needed.



bootstrap.yml

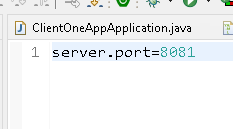


spring.application.name: client-one configuration file will be fetched, however because of profiles you get different file

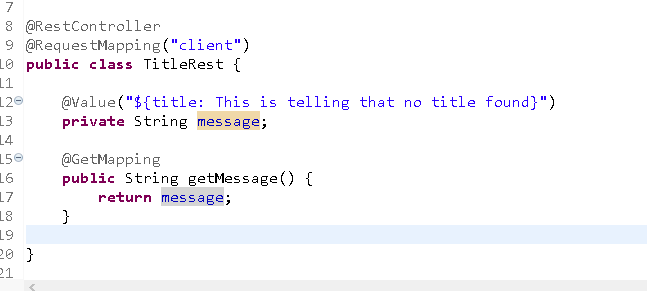
spring.profiles.active: client-one-developer.yml file will be fetched

spring.cloud.config.uri: location of the configuration server

application.properties

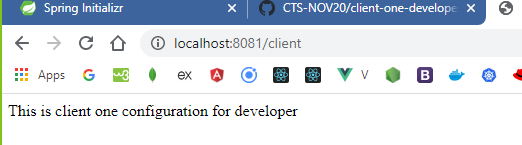


Since all the configuration files have title property we need to extract it using @Value annotation.



This rest end point only shows the title value of the configuration file.

Output:



Exercise:

1. Store the datasource configurations in GIT and fetch from configuration server
2. Microservice will fetch the datasource configurations and connects to MYSQL
3. Microservice need to have Service Layer, DAO layer (JpaRepository) and controller (RestController)
4. Microservice should also register with Service Discovery
5. Microservice must perform 4 operations
   1. store employee: id must be auto-generated
   2. update salary based on id
   3. fetch all employees
   4. fetch employee by id
   5. delete employee by id
6. Use common URI /employee but different http methods (PUT, DELETE, POST, GET) to perform CRUD operations

Fault tolerance (Circuit Breaker) with Hystrix & Resilience4j

Whenever a microservice communicates with another microservice chances of fault tolerance is more, because if a microservice sends request to another and it sends to some other, then if any of the microservice is down then other microservices need to wait for the response, this may lead to cascading of failures leading all other microservices to wait and incoming requests keep filling that can lead to exhaust the resource in every dependent microservice that is waiting for the response, so a microservice need to fail fast instead of waiting for the response, so that it can call some other fallback method instead of sending request to failed services giving time for failed services to recover.

Circuit breaker pattern is implemented by Hystrix but it will be deprecated soon, so you can use another library Resilience4j.

The different is in Hystrix you will configure circuit breaker in java code, however in Resilience4j you can configure through java code or through property files.

@HystrixCommand(fallbackMethod = “callFallback”, failedRequests=6, waitDuration = “50s”)

public void microservice1() {   
 // calling microservice2  
}   
public void callFallback() {   
// gives some other response  
}

The same thing you can do with Resilience4j where you can configure the things in property files which is better than hystrix

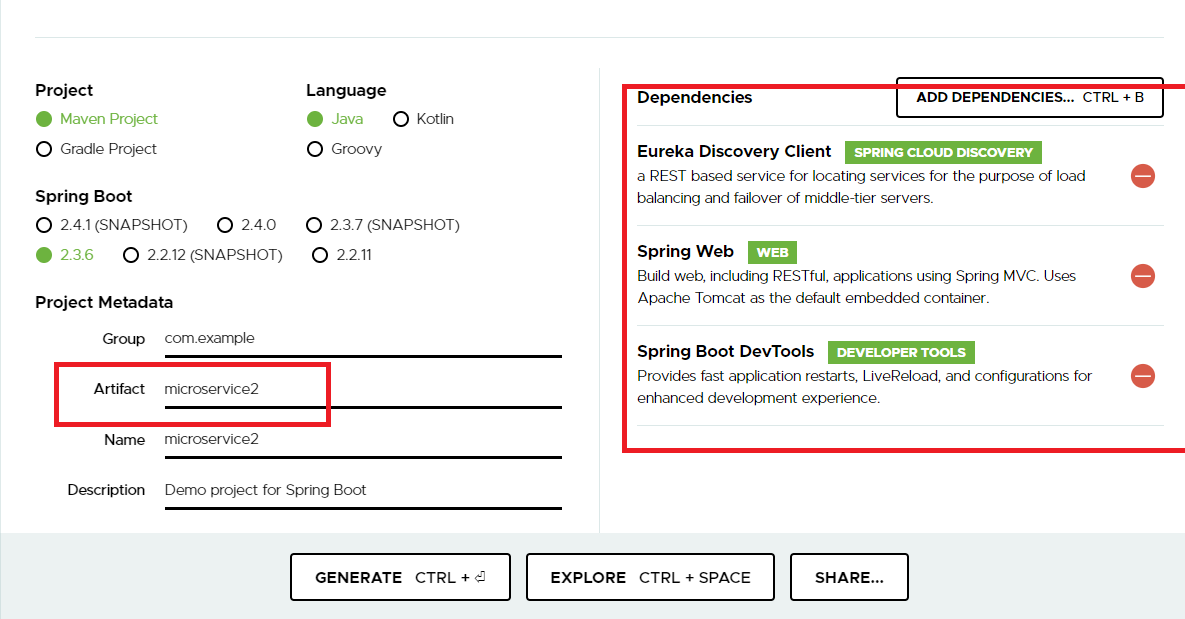
We will create 2 projects

1. microservice1
2. microservice2

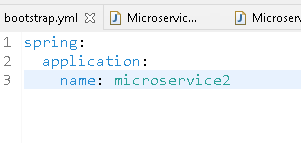
microservice1 & microservice2 registers with service discovery & microservice1 can use Hystrix to enable circuit breaker, because it calls microservice2

microservice2 is called from microservice1 so we will add hystrix to microserivce1.

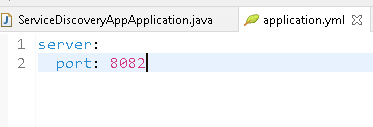
Firstly we will create microservice2 & then microservice1



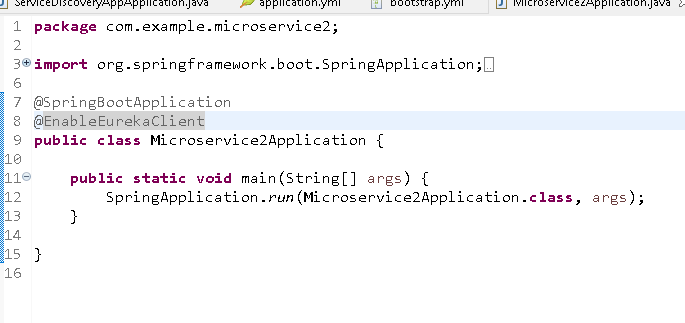
bootstrap.yml



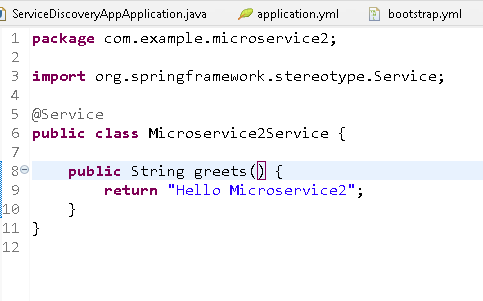
application.yml



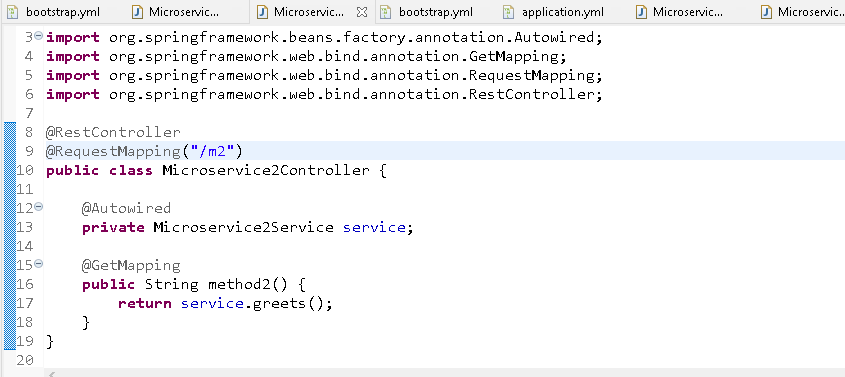
@EnableEurekaClient in the @SpringBootApplication



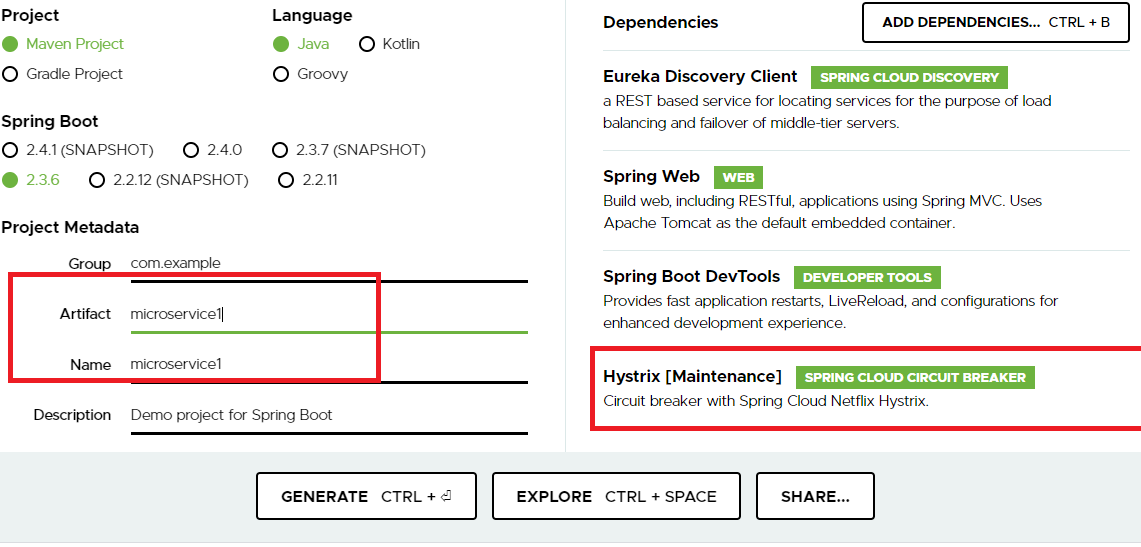
We will create a service that returns Hello Microservice2



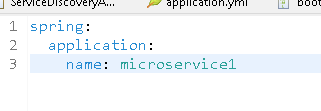
We will also create a rest endpoint that calls the service



Create microservice1 to call microserivce2



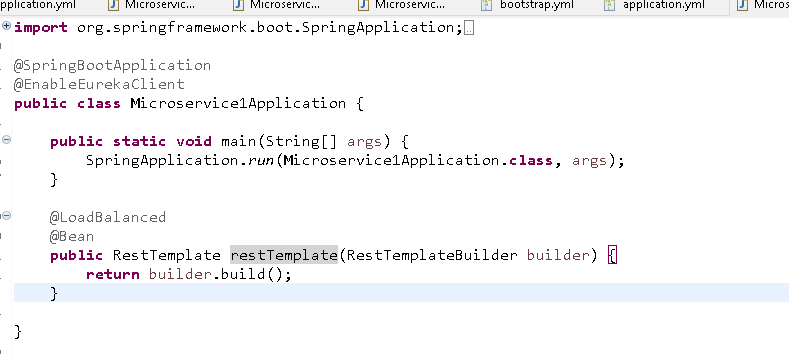
bootstrap.yml



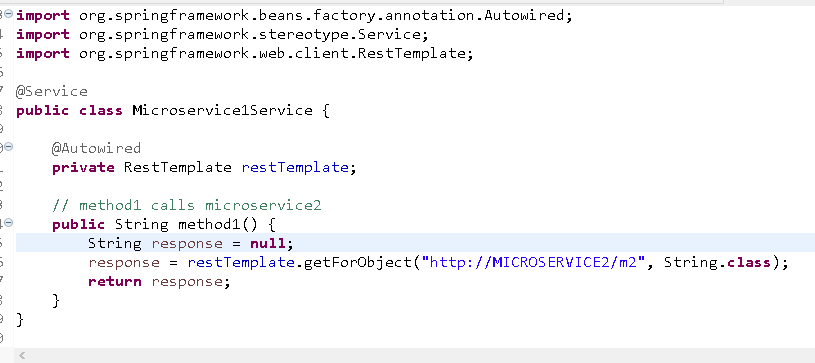
application.yml



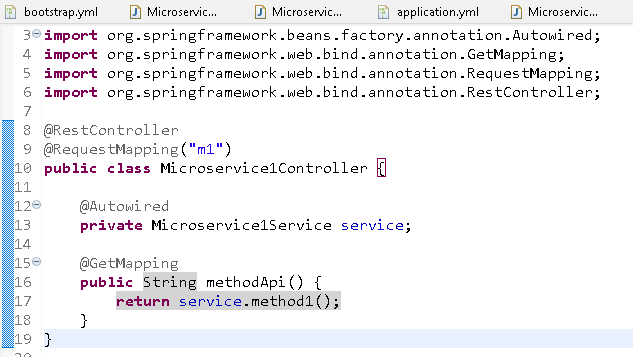
Add @EnableEurekaClient & create RestTemplate



Now create a service that calls microserivce1 later you can add hystrix commands

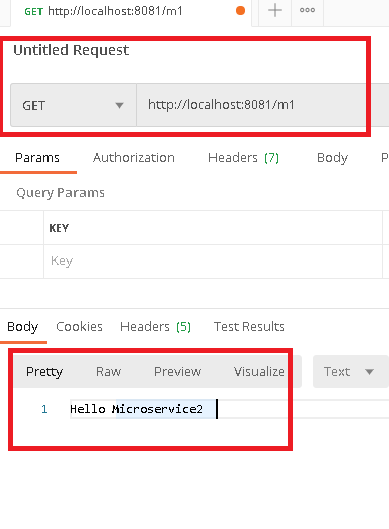


Create a Restendpoint to call the microserivce2 from microservice1

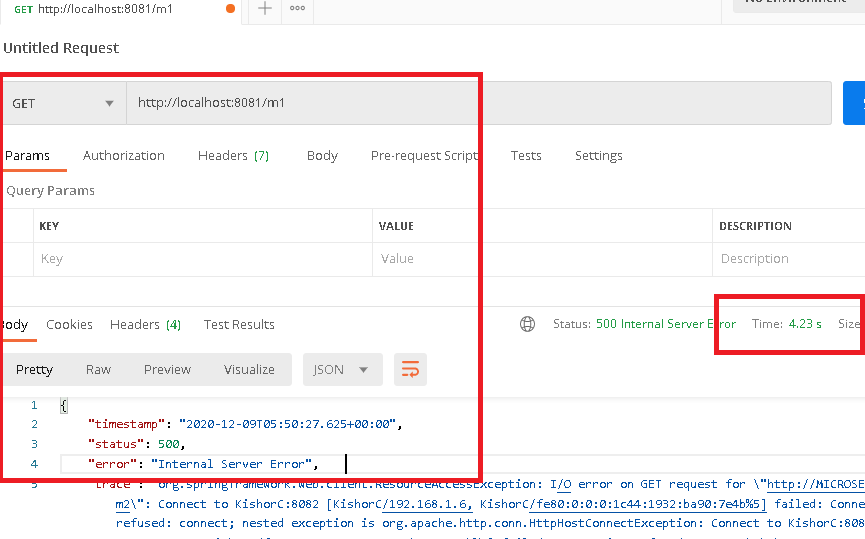


service.method1() is calling microservice2 and gets a response.

Testing from postman



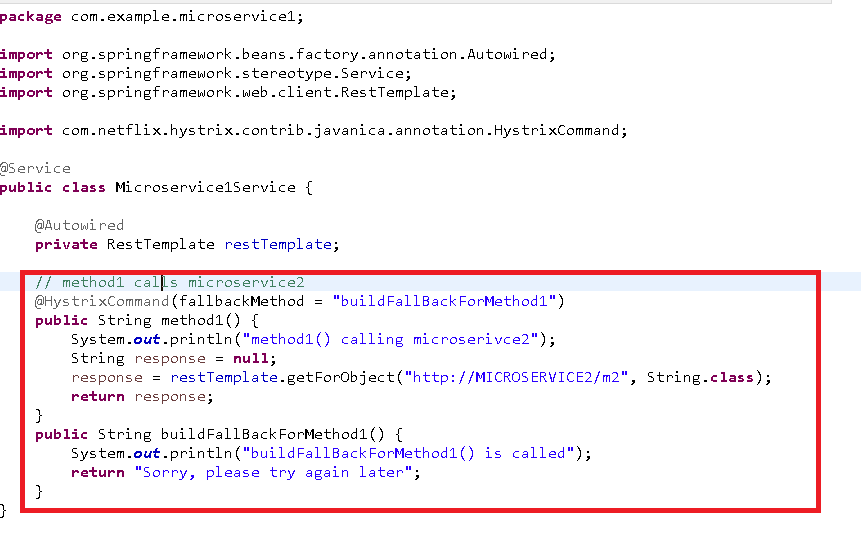
We are getting the response from microservice2, but now we will stop microservice2 and see what happens



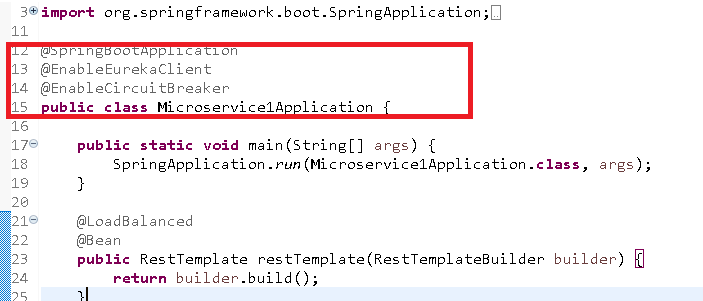
Here microserivce1 waited for 4seconds & gave an exception response, but if more number of requests wait this way chance of resource unavailability in the microservice1 can occur and microserivce1 also goes down moreover other microservices would be waiting for microserivce1 response, when microservice1 is waiting for microserivce2 response, hence you must add a fallback method in microservice1 which gives an alternate response when remote service is down, you will create a circuit breaker which will fail fast and open the circuit & give the response after certain number of failed requests.

Now you can add a circuit breaker @HystrixCommand that mentions the fallback method that is called when remote microservice2 is down.

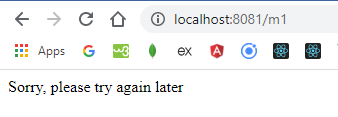
Microservice1Service.java



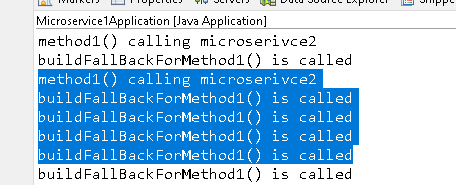
Add @EnableCircuitBreaker in the application



Output:



You are getting response from fallback, and also you can see in console not everytime the request is going to method1, its directly going to fallback.



The console shows that you are getting response directly from fallback when the circuit is open.

Api Gateway:

It is a common front door for all the incoming requests that needs to contact the microservices, all the clients will use common end point name so that it will be easy for the clients to use this single end point name.

Api gateway can track each requests, and do a reverse proxy (accepting the request with one uri and routing to different uri).

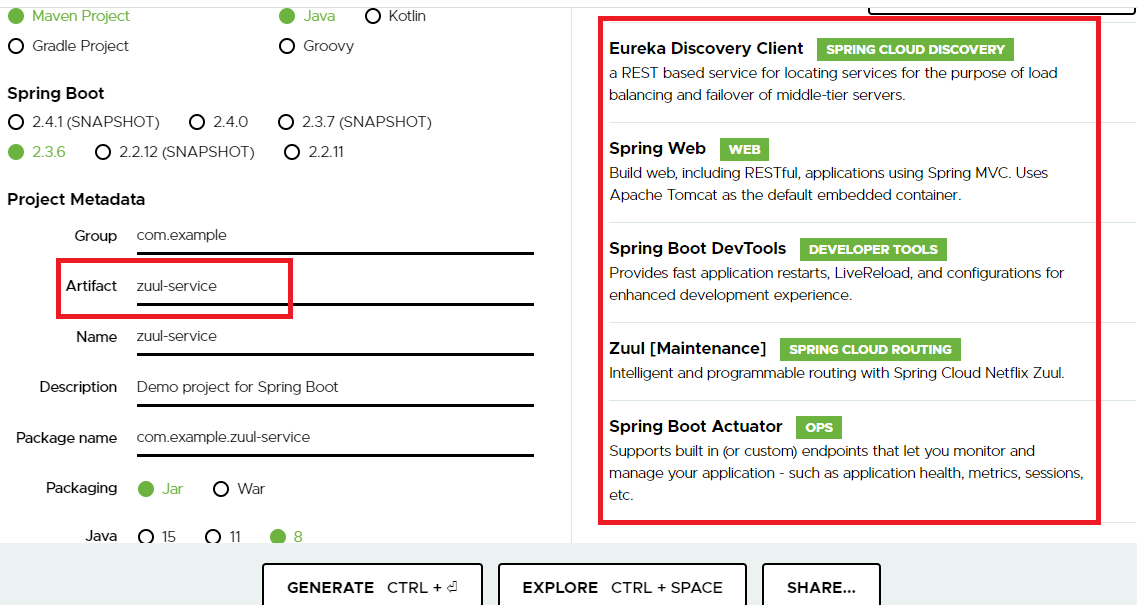
Zuul is the library that acts a API gateway

Zuul also has load balancer that can equally distribute loads coming to the same instance, which can’t be avoided when multiple clients uses their own load balancers, All the requests from multiple microservices/clients must come to the API gateway & it will decide how to distribute.

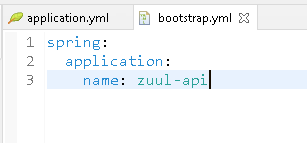
Note: You can configure zuul to register with Service Discovery so that it will know how many instances of service is available in Service Discovery

How to create API gateway using Zuul:

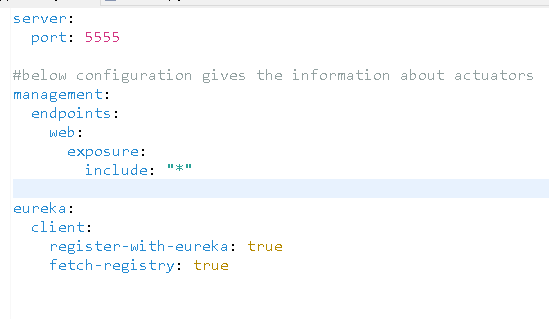
Since zuul should know how many microserives are registered with service discovery we will registering zuul also in the service discovery.



bootstrap.yml



application.yml

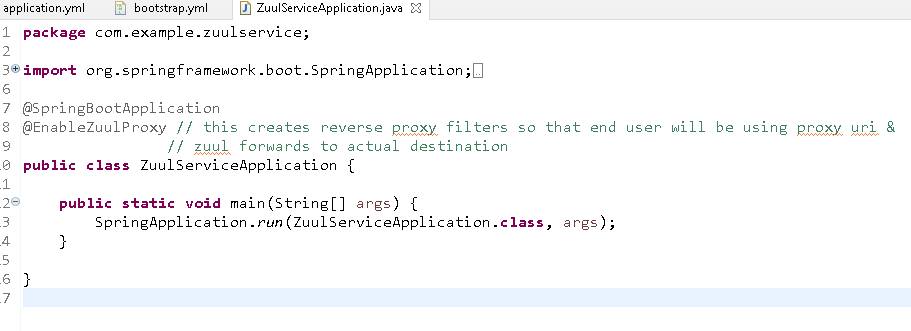


zuul will use 5555 port but it has to be registered in the eureka server so that it will know all the service instance uri’s, we have configured management.endpoitns.web.expsure.include, so that with the help of actuator endpoints you can see application informations

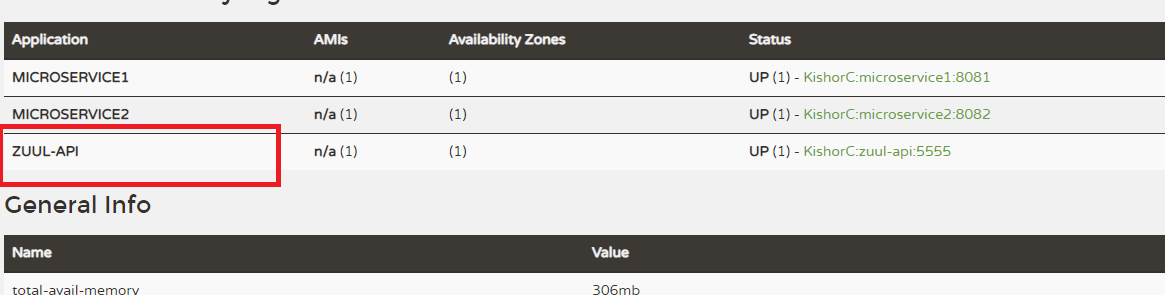
/actuator/routes: this endpoint shows all the proxy uri’s  
/actuator/health: this shows the health of the application i.e, up or down  
/actuator/metrics: this shows the metric informations like cpu usage

/actuator/beans: this shows all the beans instantiated in your application.

Now you need to create Zuulproxy



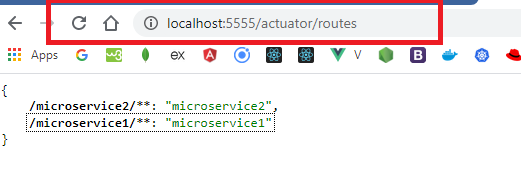
You can see Zuul has registered in eureka dashboard



Now you can see the proxy uri’s for your microservices

By default the proxy uri’s will be same as the microservice logical name

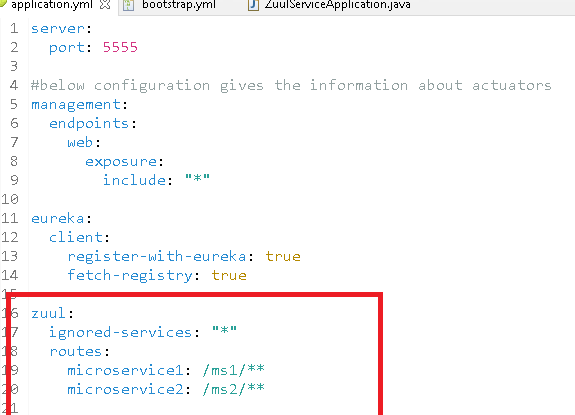
Through zuul you can check the actuator/routes endpoints to see the proxy uri’s



/microservice2/\*\* means it can have hierarchical paths, like microservice/fetch/100

You can notice that any request coming from microservice2/\*\* goes to the microserivce2

application.yml



Now microservice1 & microservice2 can be access via Zuul API gateway as

<http://localhost:5555/ms1>

<http://localhost:5555/ms2>

In real time you will have a domain name for the localhost:5555 and microservices are not directly accessible because they run in intranet, so it wouldn’t be accessible through domain name, you will give domain name only for api gateway.

anybody want to access microservice1 or microservice2 need to use domain name of zuul gateway, because it will be public

i.e.,

<http://www.zuul.com/ms1>

<http://www.zuul.com/ms2>

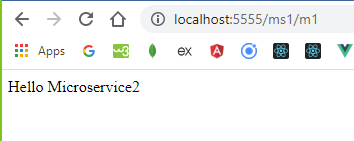
However using

<http://microservice1/>

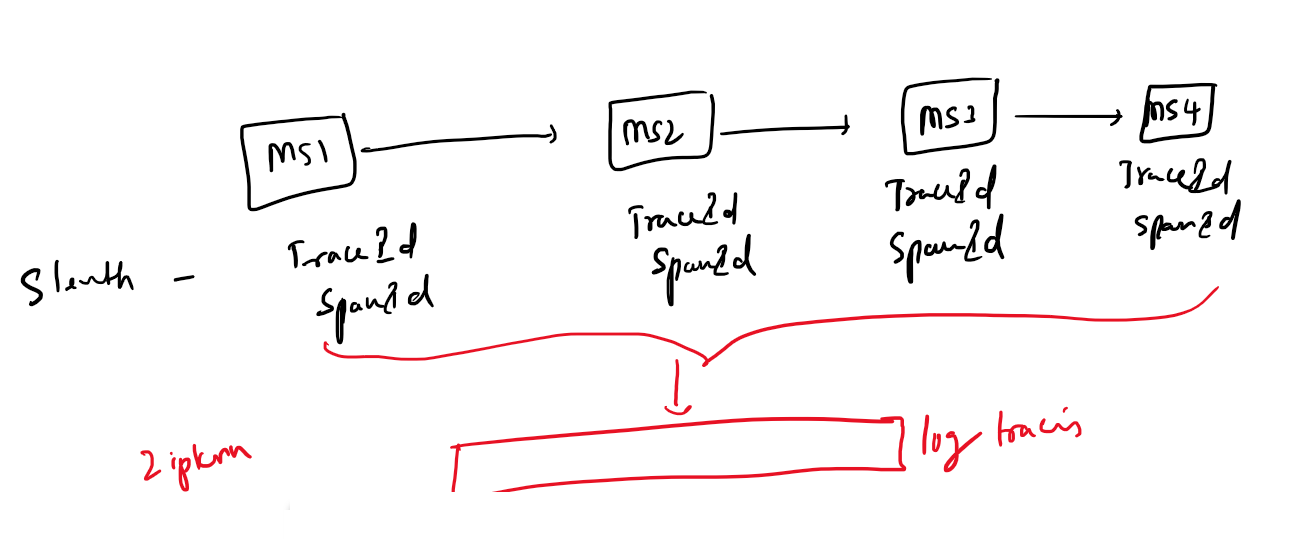
doesn’t work in real time.

Let us try to access the microservice1 through zuul and check if it works

<http://localhost:5555/ms1/m1>



Distributed tracing with Sleuth & Zipkin



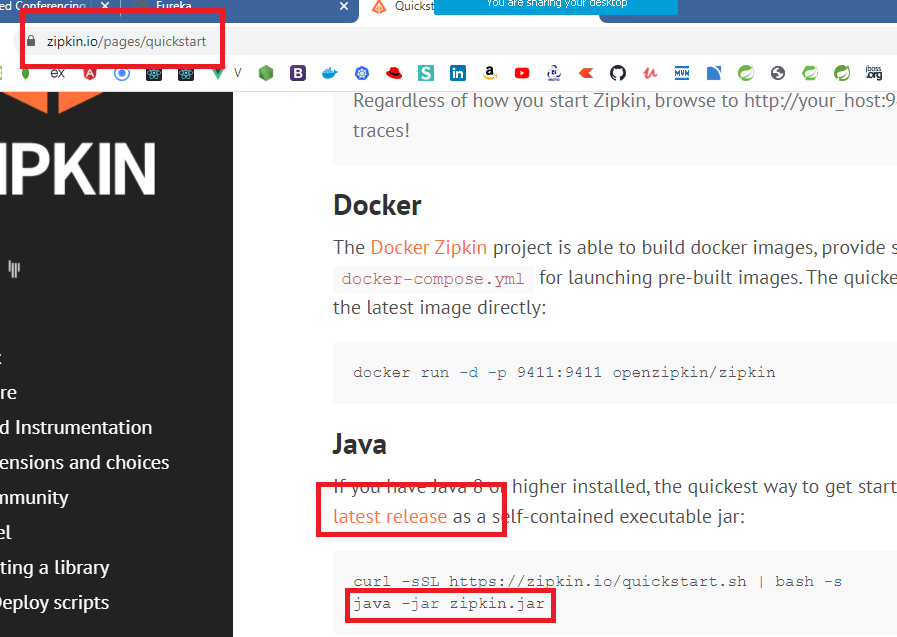
Sleuth gives trace id & span id for every calls made to the microservice.

* traceid: it is unique for every request
* spanid: it is unique for a particular microservice method

Zipkin aggregates all these trace id & span id and shows the tracing of these logs as a single log.

You need to download a Zipkin executable jar and run so that you can see the distributed tracing.

Location:



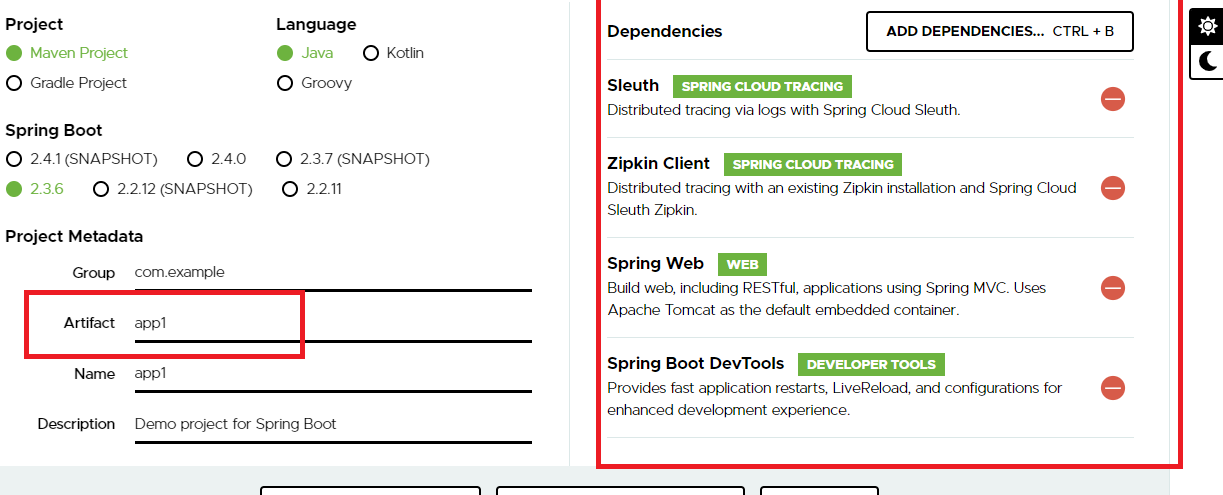
*java -jar Zipkin.jar*

Zipkin runs in 9411 port

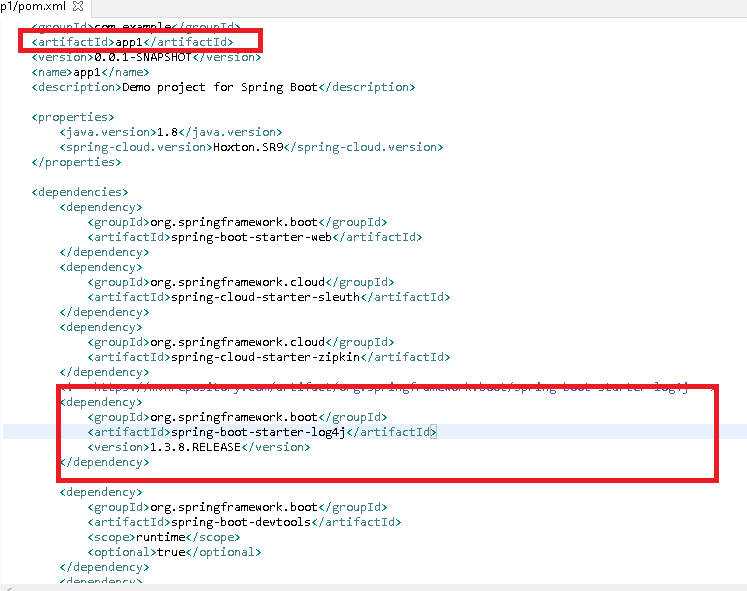
Firstly we need to create some microservices and interact with them and also we need logger libraries to be added.

We will be creating app1, app2, app3, app4 microserivces that will have same code with few changes i.e., class name, method name & uri’s

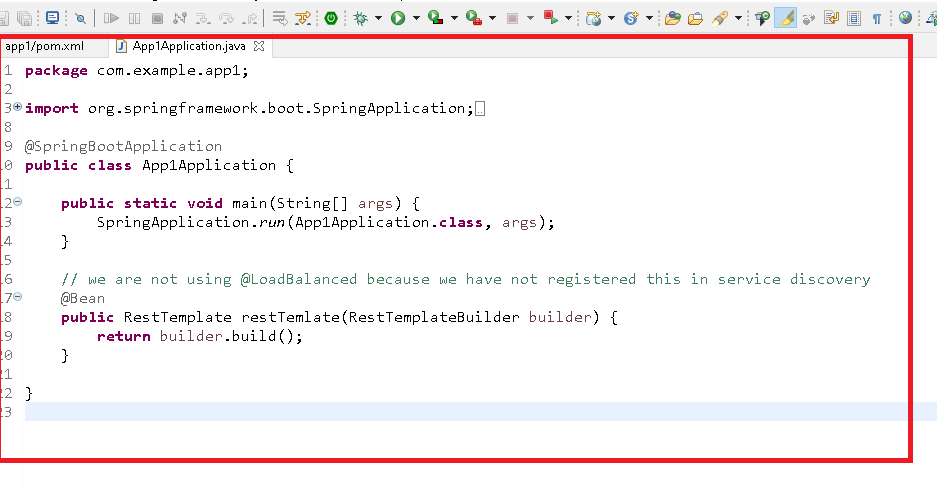
Here *app1 -> app2 -> app3 -> app4* ultimately app1 should receive the response



We need to add another library log4j which must be added from maven



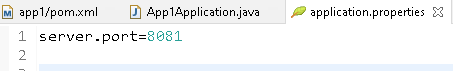
App1Application.java



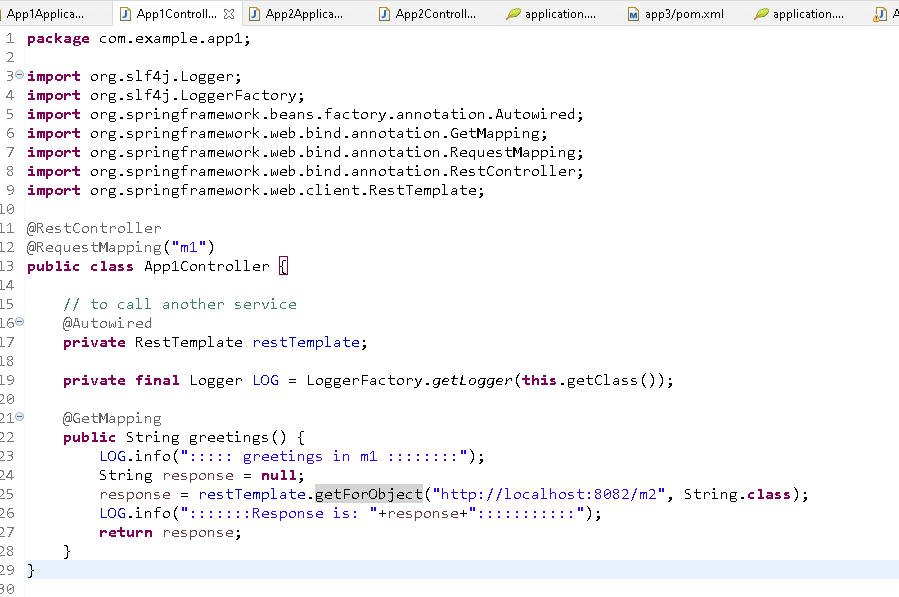
Here all the other projects can be created with the same code but RestTemplate is not required in app4.

We need each applications to run in different port

application.properties



A controller in app1



This is it, now the sleuth will generate the trace-id & span-id and Zipkin will aggregate these logs, but Zipkin should be running (executable jar must be executed)

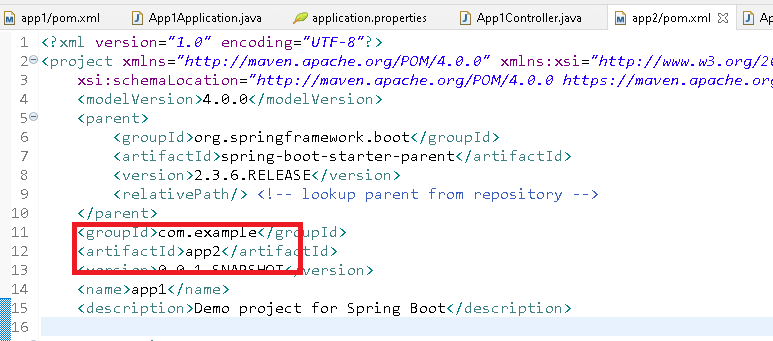
You copy the same project & do the necessary changes

app1 -> App1Application.java -> 8081 -> /m1   
app2 -> App2Application.java -> 8082 -> /m2 -> artifact-id

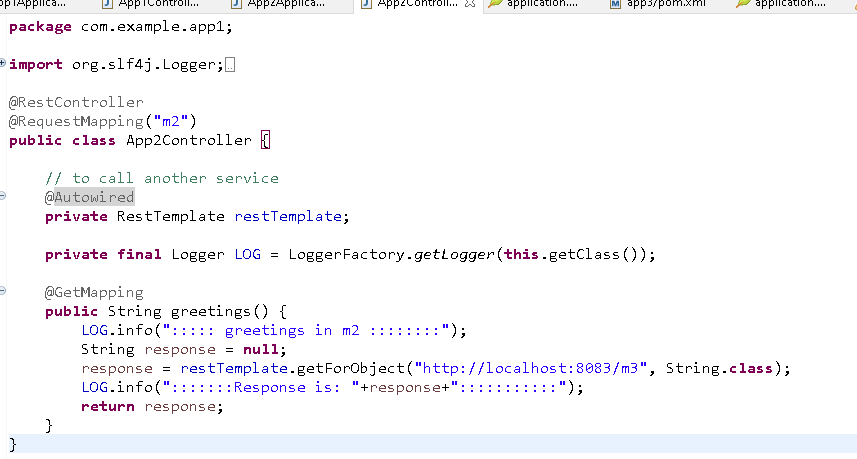
and so on

Copy the project app1 and paste it asks to rename, so rename to app2 and do these necessary changes to app3 and app4.

app2/pom.xml



App2Application.java



App2Controller.java

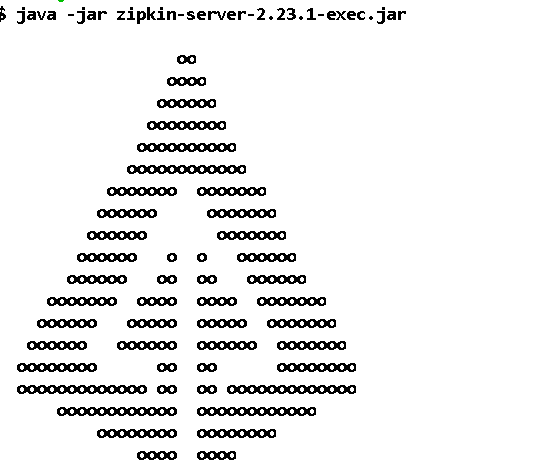


lastly application.properties



Do this for other projects, but here I’m only using 3 projects

Update all the projects & start the Zipkin by running the downloaded jar.



Then start all the application and send request to /m1 you must see the logs in the Zipkin as well as in console also.

Output in Zipkin ui

