Spring Micro-services

Pre-requisites

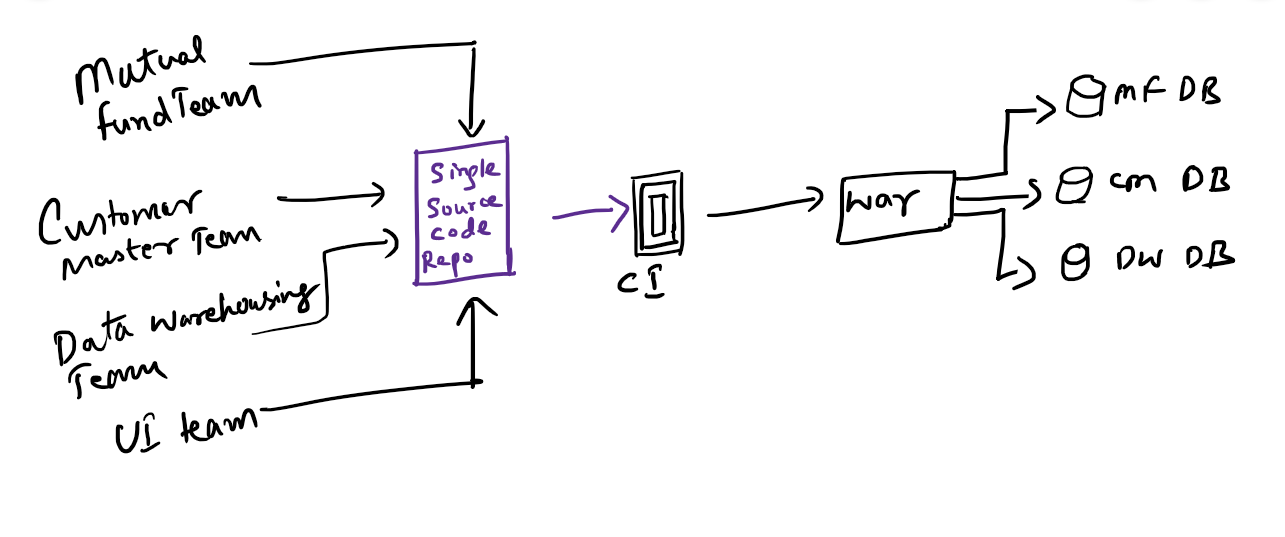
* Spring Boot
* Spring REST

Software Industries has been evolved with lot of changes and approaches while delivering applications to their customers starting from stand-alone to distributed applications, one such change is that industries adopting to Microservices (came to market in 2014) while building applications.

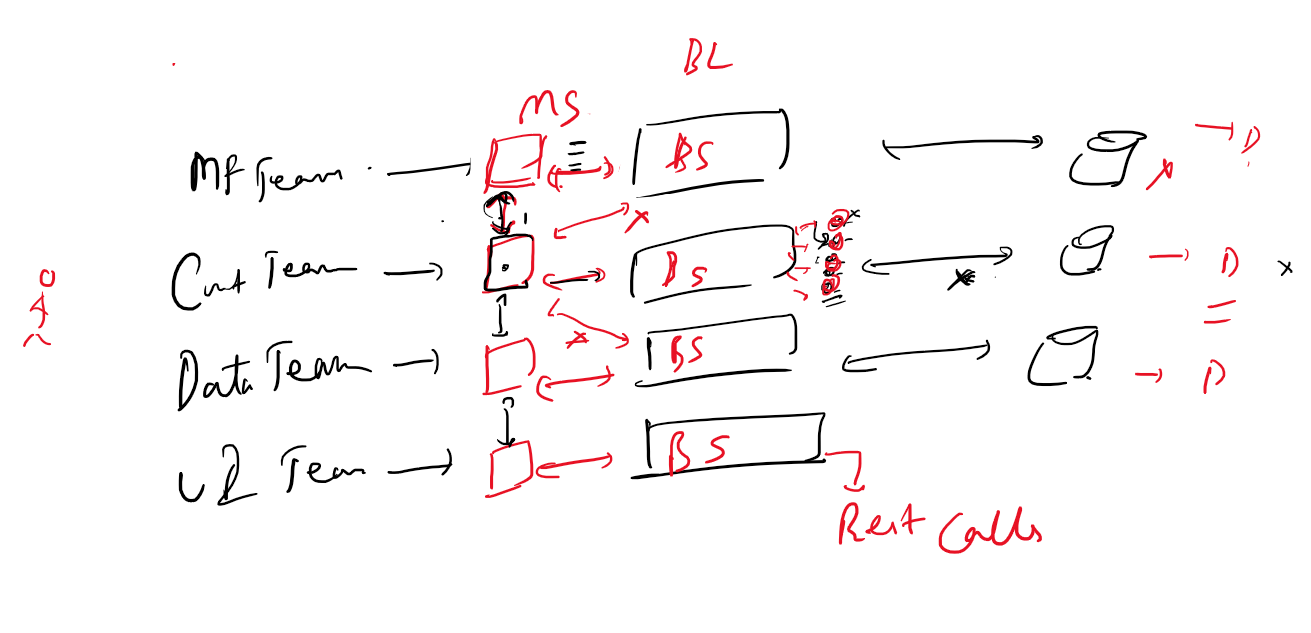
Microservices are loosely coupled services that are independent from other services of same or different applications

Monolithic vs Microservices architecture

Monolithic architecture



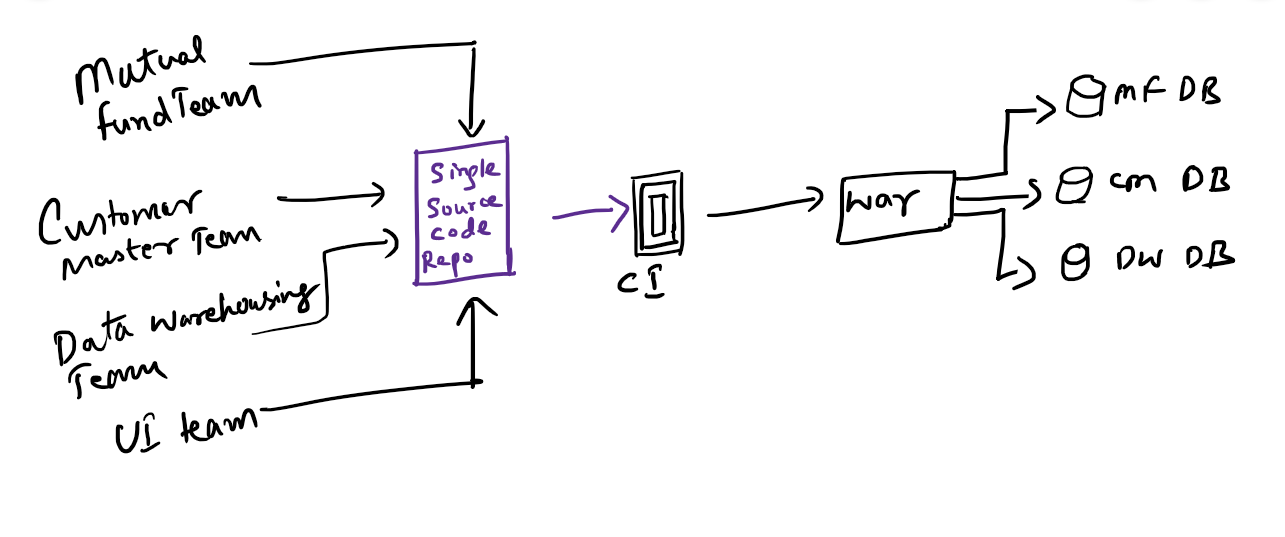
Microservices architecture



What’s a Microservice?

Before the concept of Microservices evolved, most web-based applications were built using a monolithic architecture style.

In a monolithic architecture, an application is delivered as a single deployable unit (like WAR or EAR). However an application wouldn’t be having a single module, it would have multiple modules and each module will have separate responsibilities of task to be done and each module is developed by a separate team but a single code repository with continuous integration.



The entire application will also have knowledge of all the datasources used within the application.

Ex: A banking application will have many modules like Mutual Funds, Customer Services, Dataware Housing and many other modules like.

* Investments
* Loan
* Insurance

In Monolithic architecture multiple development teams are forced to artificially synchronize their delivery because their code needs to be built, tested and deployed as an entire unit.

The problem here is that

* When an individual team needed to make a change, an entire application had to be rebuilt, retested and redeployed to modify the old artifact.
* If in the production when the application goes down because of some system failures then all the modules go down.
* Suppose demand increases for some services then the whole application has to be scaled up and scale down when demand decreases this is obvious even if there’s a demand for a single service of an application and cost of servers are expensive

Ex: Increasing booking services in IRCTC during festival times leads to scale whole application

All these led to create a better solution for customers which gave a path to invent Microservices architecture

Steps made path for Micro Services:

1. Traditional SDLC
2. Challenges with traditional SDLC
3. Today’s software needs

Waterfall model approach may not match to the future market at the end of the product.

So you will go with Agile process where you develop the product according to the market trends and change the features if required in Scrum meeting, but in Agile you will have separate teams for development, testing, deployment, monitoring, here each team will not have knowledge on the other teams work.

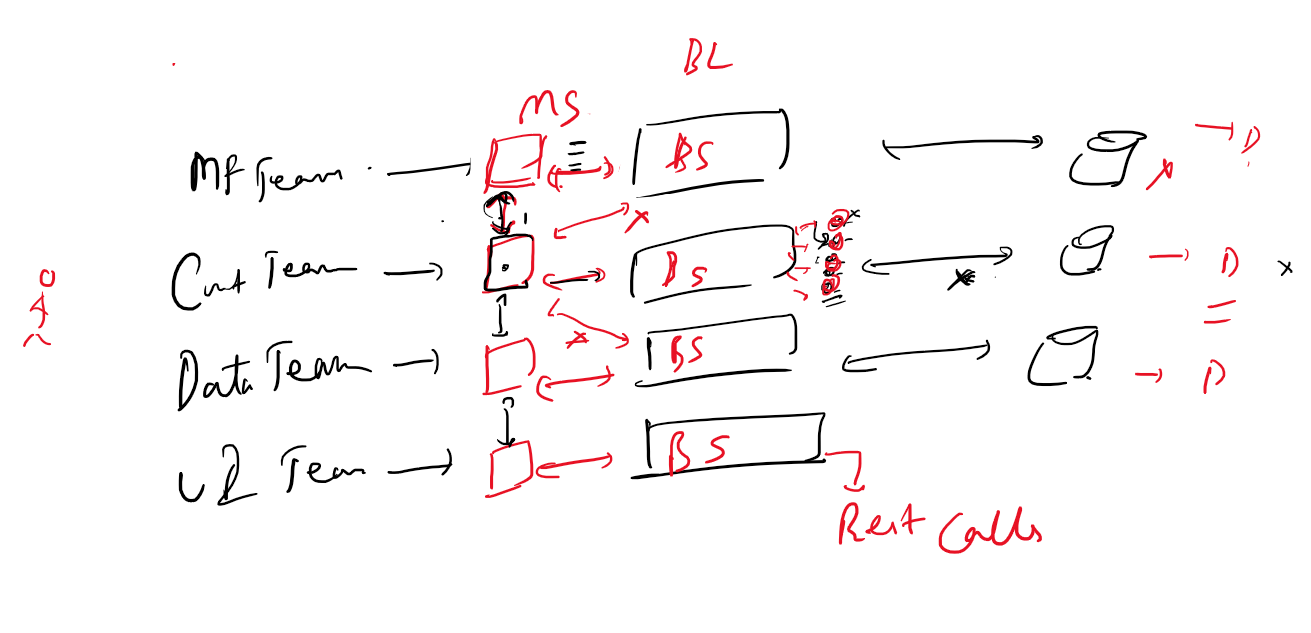
But now the need is, a single person need to have fullstack knowledge in the same agile process where he will have knowledge on all the things like development, testing, deployment, monitoring so that he will take less time for resolving any issues, but everybody will be in the same team that is called as Scrum team and each person may need to have knowledge on the whole process.

Microservices: These are reusable services or components which is responsible to carry out specific business task and can be developed and built independently from other Microservices.

A single application can be decomposed into multiple services which are responsible to perform some business task and each team works on a particular service and have separate code repository and deployed in a separate container. This adds more flexibility to the changes because other services need not to be rebuilt or tested when a particular service is changed and also scaling can be done only to the service which has high traffic.

When multiple services are deployed on different machine each service will have idea of their respective databases and these services can communicate only through HTTP REST calls

Each database is private to the micro-service, one micro-service can’t directly access database of another micro-service , it must access only through its corresponding micro-service.



Spring Projects for Microservices

Spring has its own benefits and it has evolved as per the industry trend. It has two projects which are used to build Microservices

1. Spring Boot
2. Spring Cloud

Spring Boot: It is used to quickly create production grade spring applications. It takes minimum effort for developers to create spring applications with spring boot. It gives production environment in the development environment like property files for configurations, servers comes in the code only this takes less time for production, testing and development team to add configurations based on their profile.

Spring Cloud:

It helps in providing the tools to quickly build the distributed applications with common design patterns like:-

Service Discovery  
Discovery Client  
Circuit Breaker  
Load Balancing

Using this coordination between the distributed applications can be simplified.

With spring cloud you can simplify the development of distributed applications with simple annotations & configurations.

Spring Cloud uses the Spring Boot project to quickly develop the applications, we must use the compatible spring boot projects to work with spring cloud.

Spring cloud uses many spring boot features:

* Starter Projects
* Compatible version of Spring Boot for Spring Cloud
* Creating an executable jar/war
* Property configuration
* Passing the configurations from command line at runtime
* Deploying the war on external server
* Actuators endpoints: health, metrics, beans
* MVC implementation: Service, DAO
* JPA Repository
* REST based services
* Curl commands
* Working on GIT

In spring cloud we are going to deal with:

* Externalized Configuration: This makes a configuration which is remotely available for multiple applications, you can apply security like Encryption by using JCE (Java Cryptographic Extension), You can also force applications to pass the credentials while accessing the remote location
* Microservices: Service Discovery, Discovery Client, Load-Balancing, we need to know some design patterns, like Development patterns, resilience pattern, routing pattern.
* Circuit Breaker: Hystrix (Will be Deprecated), Resilience4j
* Zuul: Common door for all the incoming request to your service discovery, dynamic routing, enables adding cross cutting cocernts
* Security: To protect the resources with authentication & autorization

Conclusion

To make Micro Services to work together you need to follow some guidelines like

1. Configuration server: Common configuration for all the Micro Services
2. Eureka: Registry server for all the micro-services
3. Ribbon: To distribute the load on multiple instance of same micro services
4. Hystrix: Circuit breaker for tripping failed services/non-performing services
5. Zuul: API Gateway for securities, logging
6. Sleuth: Log tracing, gives id for the flow of request & response
7. Zipkin: Log aggregator

Steps for creating Micro Services

1. First Create Eureka Registry
2. Next Create two simple service that is a client which registers into Eureka Registry
3. Next Create a config server which will also registers into Eureka Registry
4. Create Config client to use the config server
5. Create profiles to use different configuration’s based on the environments like development, testing, production
6. Register the Config server as a separate micro-service in a Eureka & use this micro-service from another micro-service

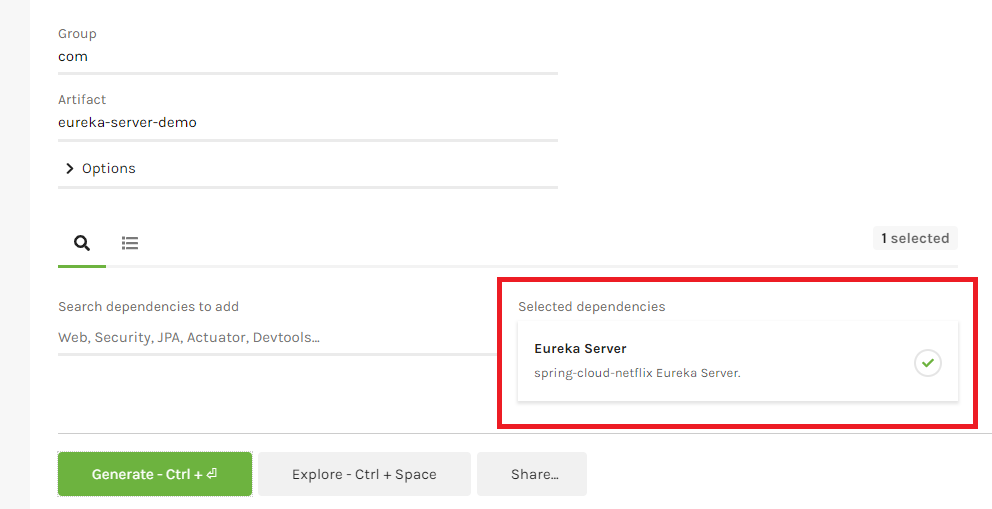
Creating the Service Discovery & Registration

We will use Nexflix Eureka service registry which is eureka server

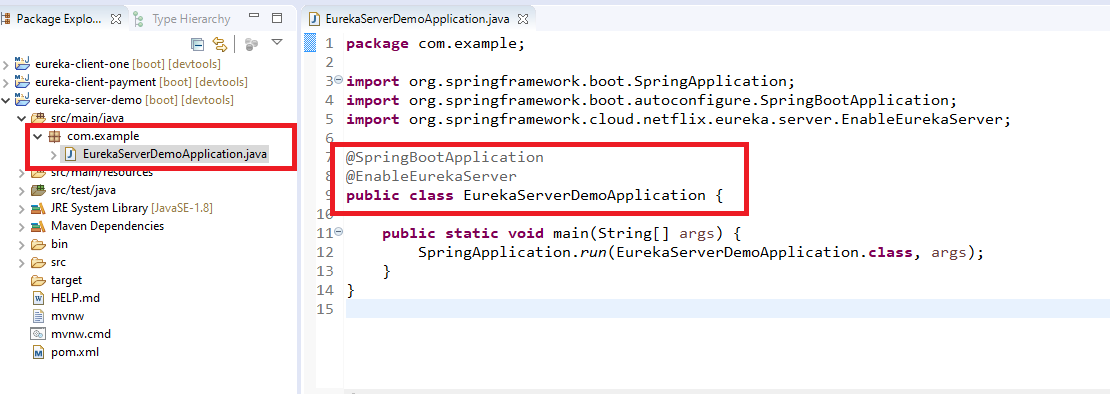
All the Micro Services will be registered in the eureka server

It’s better to create project from Spring Initializr, because Eureka server is not the only dependency may need.

You need only Eureka Sever dependency to be chosen from the spring intializr,

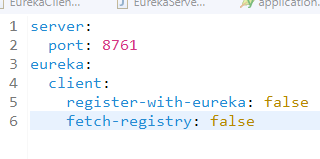


Export the project



Create application.yml & delete application.properties.

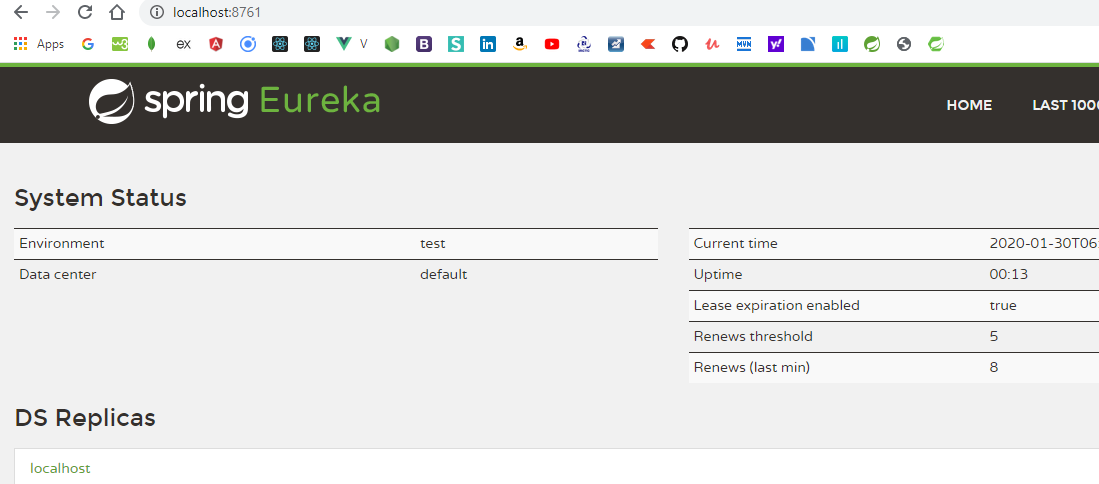
application.yml



* The default port Eureka Client’s registers with the Eureka Registry Server in 8761, hence start the Eureka Server at 8761.
* Eureka Server must not register itself as a client & also must not fetch from the registry hence make those false as the default fetch-registry & register with eureka will be true

Run the main() to start the Eureka Registry

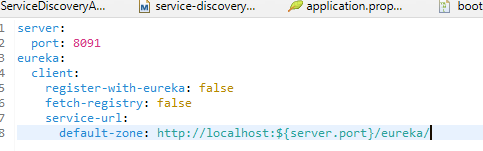
Output:



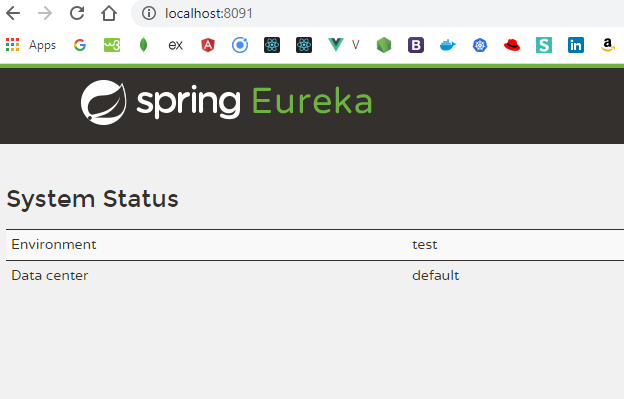
Now you can create services which are treated as clients and Micro Services to register with the Eureka Server, you can name the client in bootstrap.yml instead of application.yml.

Running the Eureka server in different port

application.yml



Output:



Note: client is not registering in other ports, need to resolve the issues

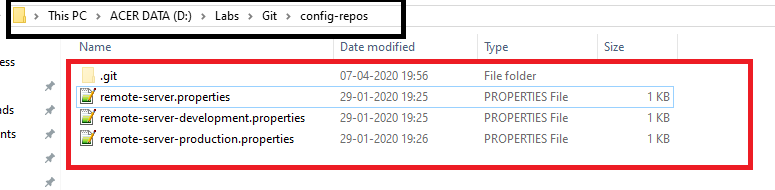
Spring Cloud Config Server

Spring Cloud Config provides an externalized configuration support for client & server side in a distributed system, you can create a configuration file which is accessible to multiple services on cloud, you can have GIT or Local repository for cloud configuration

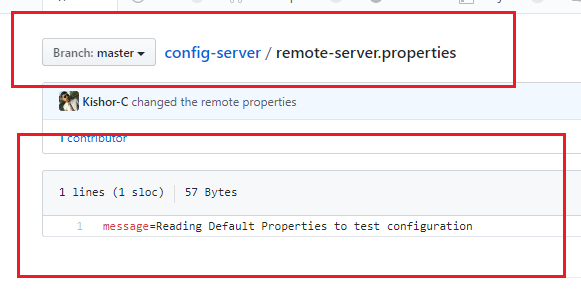
Create a git repository

Note: Ensure you have done commit whether its local or remote repository

Below are the three files in local repository

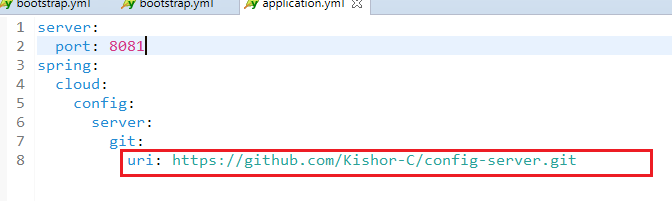


remote-server.properties

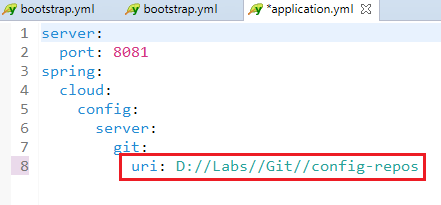


You can push this to remote repository and use the remote uri or local repository uri in the application which need these properties for multiple clients i.e.,

Git Remote URI



Git Local URI



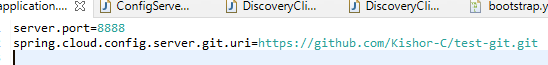
Exercise:

Use configuration server and fetch the configurations for Service Discovery and Microservices

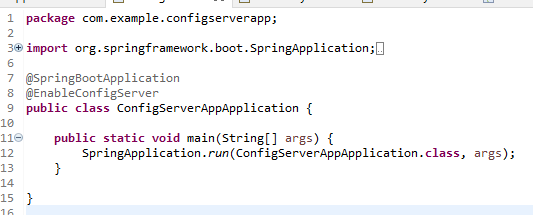
Note: Configuration Server must not be registered as a microservice because it is not treated as microservice it’s a configuration server

Steps

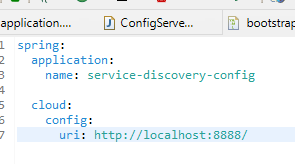
1. Create Configuration Server to access git
   1. application.properties



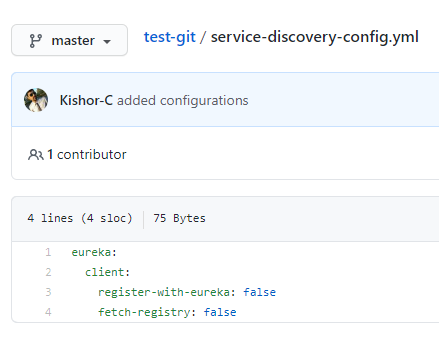
* 1. ConfigServerApplication



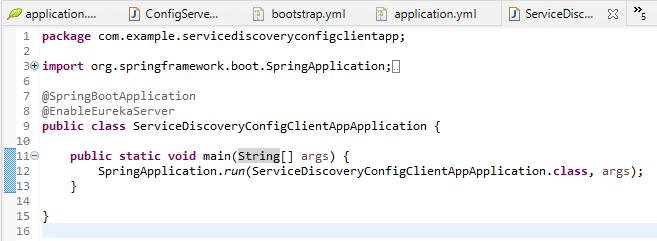
1. Create Service Discovery with bootstrap.yml access configuration server and fetches a configuration file
   1. bootstrap.yml



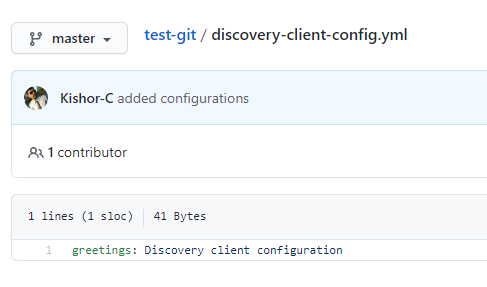
* 1. push service-discovery-config.yml with some service discovery configuration



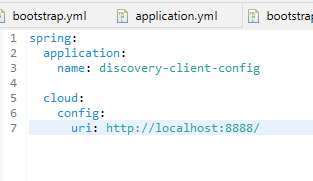
* 1. in application.yml set port = 8761
  2. EnableEurekaServer



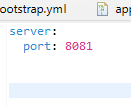
1. Create configuration client and make it a microservice
   1. create & push discovery-client-config.yml to git



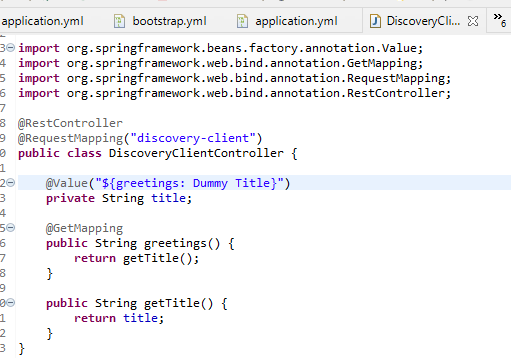
* 1. bootstrap.yml



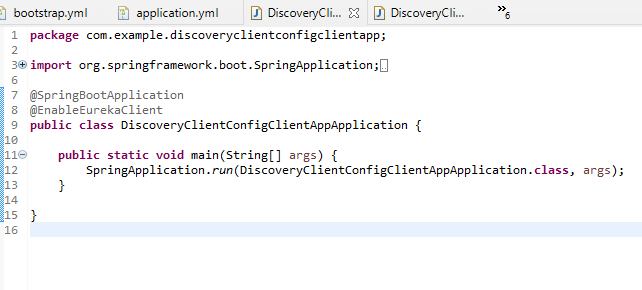
* 1. application.yml



* 1. Controller

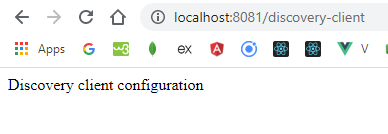


* 1. Microservice Application



Launch Configuration Server then Service Discovery and then Microservice, you can locate service discovery & microservice fetching configurations from configuration server in their logs

Output:



Communication between the Microservices

We are creating 2 Micro Services client-one & client-two and client-two will not call any business logics of client directly, the only way it will call is through REST calls, i.e., client-two calls rest api’s of client one, this is because every service will have unique database & they are private to the each microservice.

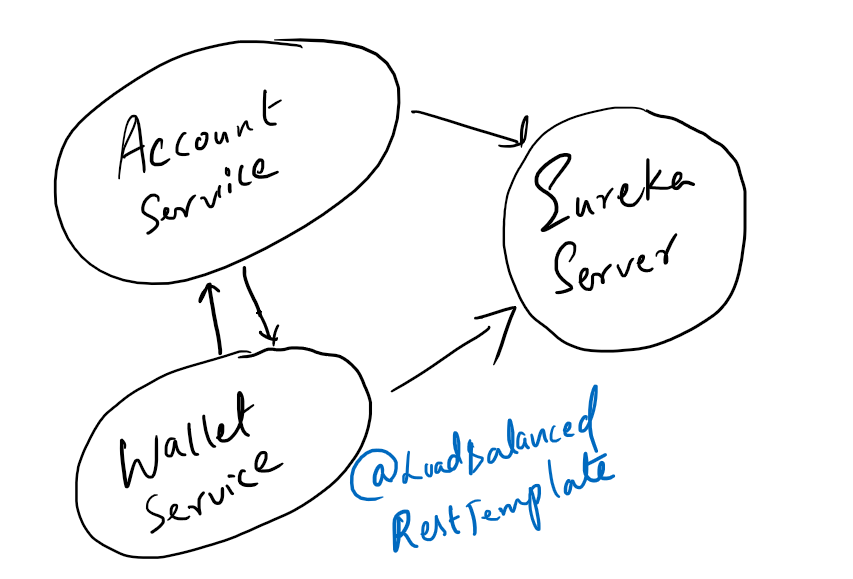
The two Micro Services we are creating will not interact with any database to simplify the learning of Micro Services.

1. The first Micro Service will get the balance of the customer account & this balance might be called by other services or modules like RD account, FD account, Paytm, Zerodha and many more, some services may be deployed in multiple containers in a docker & some may be running in a different container also.
2. The second Micro Service will call the first Micro Service to get the balance to open RD account or FD account or make some payment through wallet (wallet balance may be 30% of the actual balance, amount debited from balance will be debited in account in our case just to understand the Micro Service communication & we are not following any rules like having database, adding amount to wallet & etc)

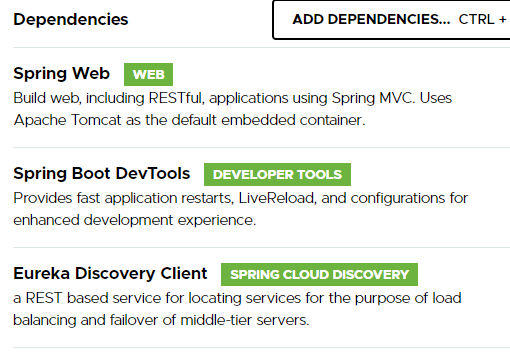
Creating 2 microservices that communicates with each other

1. account-service
2. wallet-service

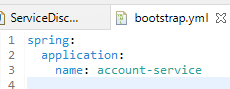
Here account-service will have a Transaction instance that is returned to the wallet-service and updates the wallet balance, both the microservices must be registered in the service discovery, wallet-service uses logical name of the account-service instead of the physical address



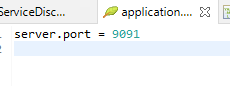
Add below dependencies to the account-microservice



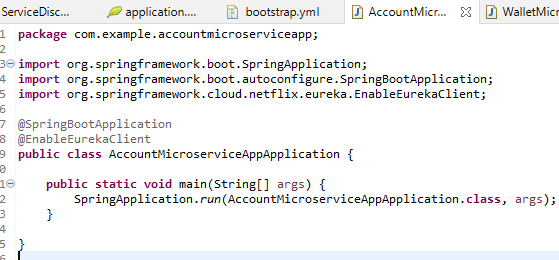
bootstrap.yml



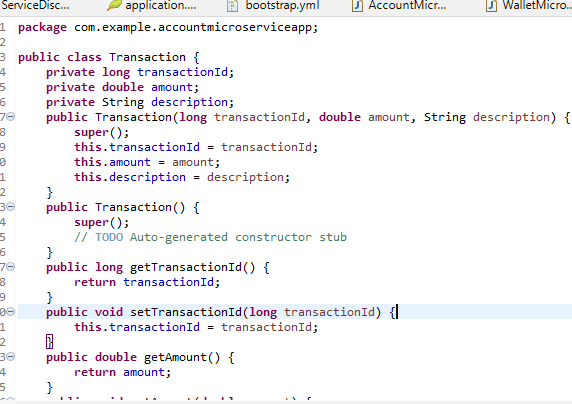
application.properties



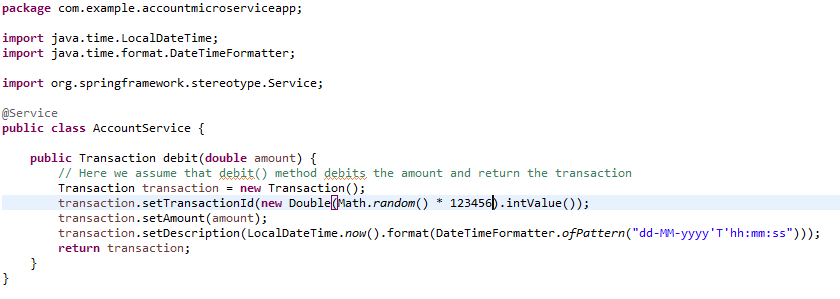
AccountMicroServiceApplication.java



Transaction.java



AccountService.java



AccountController.java



Create another Microservice which uses RestTemplate with @LoadBalanced (Ribbon) to call Account Service to equally distribute the load on multiple instances of Account Services

Create a project: wallet-service

Add below dependencies  


Create a RestTemplate instance to be maintained by the container & add the @LoadBalanced annotation

@LoadBalanced: It is a Ribbon that equally distributes the load on multiple instance of same service if exists

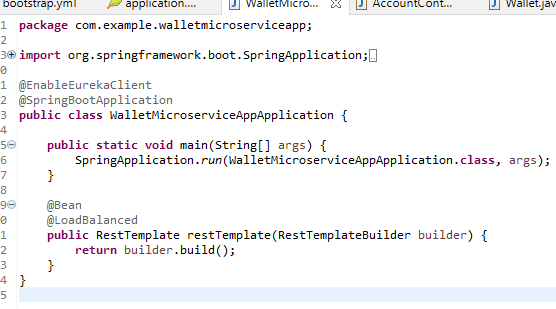
We are using load balance enabled RestTemplate which parses URL passed to its method and will round-robin balance all requests among all the service instances.

Note: a RestTemplate class must use @LoadBalanced if it is using the logical name of the service instead of actual service location because the service location is completely abstract and @LoadBalanced caches these informations and balances all requests going to the remote service, hence if you miss out the @LoadBalanced and if RestTemplate is using logical name then you will get an exception, so it does make sense using the @LoadBalanced when you are using the logical name of the remote service.

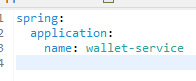
Note:

1. Since @LoadBalanced annotation caches the service location at the client side, the consumer call directly goes to the service instance without the service discovery
2. These caches will periodically refresh to be aware of new instance of the service and these caches are invalidated if in case there are any unavailable service instances and creates new local cache
3. LoadBalancing is one of the powerful concept implemented in microservice after the service discovery

WalletMicroserviceAppApplication



bootstrap.yml

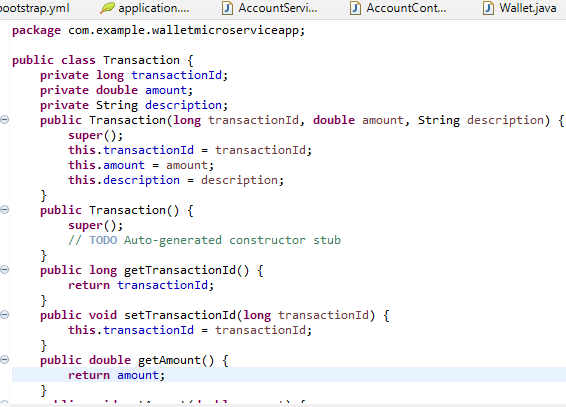


application.properties



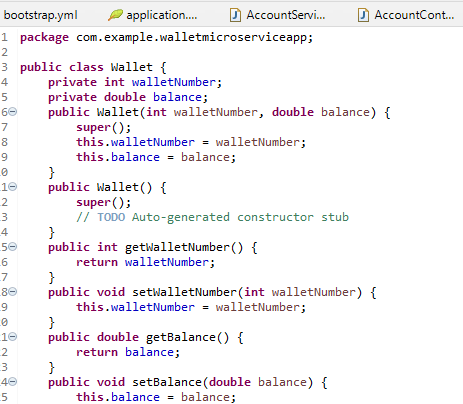
Create a model matches to the structure of the response coming from the account-service i.e., transaction instance in JSON format

Transaction.java



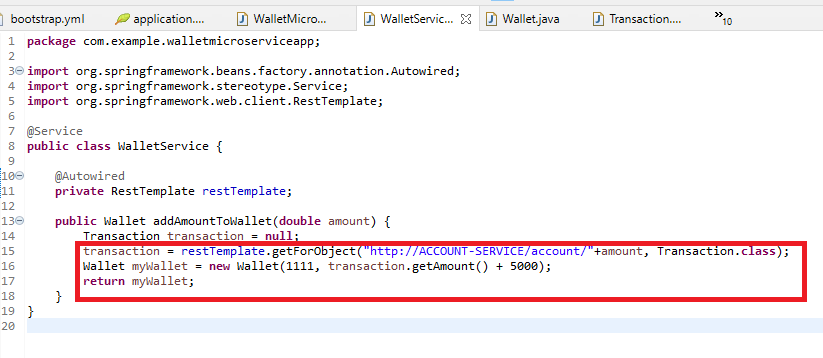
Create Wallet model that which is updated from the account-service response

Wallet.java

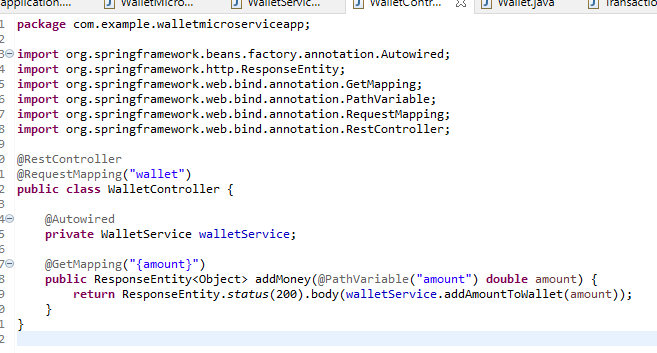


Create WalletService that calls the account service using the logical name and updates the wallet balance

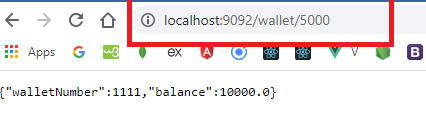
WalletService.java



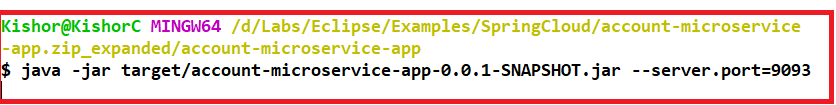
WalletController.java



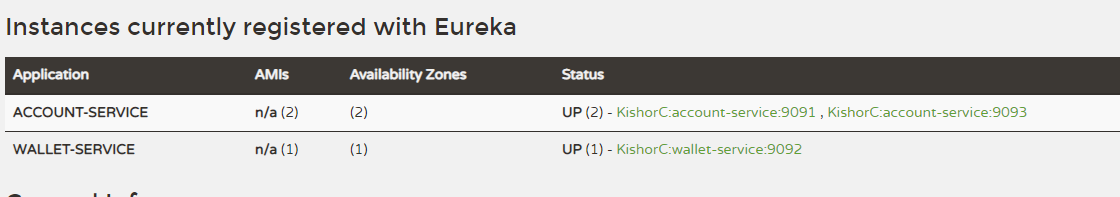
Output:



How to see @LoadBalanced is equally distributing the load



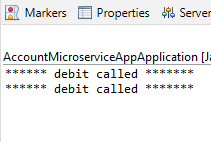
Now look at Eureka you will see two instances of Account Service



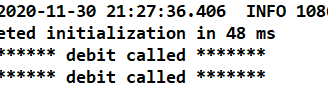
Send multiple requests to the wallet service that calls account service you must be able to see logs in both

You can see equal times the debit called in both the server console

Eclipse output



Command prompt output



Note:

You can scale up or scale down the services based on the requirement say for example,

1. In IRCTC application you may want the application to handle more requests at the time of festival seasons like deepavali or during tatkal time in that case you can scale up and make more application instances of it available and remove those instances when the time is over or when traffic is reduced.
2. In Income Tax portal the traffic will be more during the financial year end so to handle more traffics you can scale up by running more instances of the application & remove them when traffic reduces

Traditional service location resolution model using DNS and a load balancer

Here you will be having a service resolution layer where DNS name is used by service consumer and a routing table will have entries of all the services and their physical locations, the load balancer will query the routing tables when a service consumer needs to call a service, you will also have a secondary load balancer which constantly pings to check the availability of the primary load balancer, it will take over the primary load balancer ip address if necessary, but the problem they all should be in the same application infrastructure and it makes unreliable as its a single point of failure with limited horizontal scaling and they are statically managed i.e. most load balancers aren’t designed for rapid registration and de-registration of services as it has to be done in the database using some 3rd party API’s and it’s a complex task

In Cloud based these issues are completely resolved with Service Discovery and Load Balance

Service Discovery will be made highly available in the cluster and service lookups can be shared across multiple nodes in the service discovery, each node in the service discovery cluster shares the state of service instance.

Load balanced are locally cached at the client side so if in case service discovery is unavailable the client can still call the services because of local caching however there will be log traces which keeps generating at all the service because all the services keeps pinging every 30seconds to state they are available to the service discovery, but the benefits is the client could able to still communicate to the services even if the service discovery is unavailable for some moment

Service Discovery must be made available so that it can register the new instances and also make service consumer to locally cache these new instances when periodically the services ping the service discovery.

Note: New instances may not show up quickly because it needs a confirmation of 3 heart beats by default

Note: eureka.instance.leaseRenewalIntervalInSeconds this property can be used to change the default value other than 30, but in production better to keep the default considering the internal computations in the server

Service sending heartbeats to the service discovery is necessary so that service discovery can de-register or register the instances based on its availability, we must understand here that once the client registers and a local cache is created without service discovery consumers can call the other services

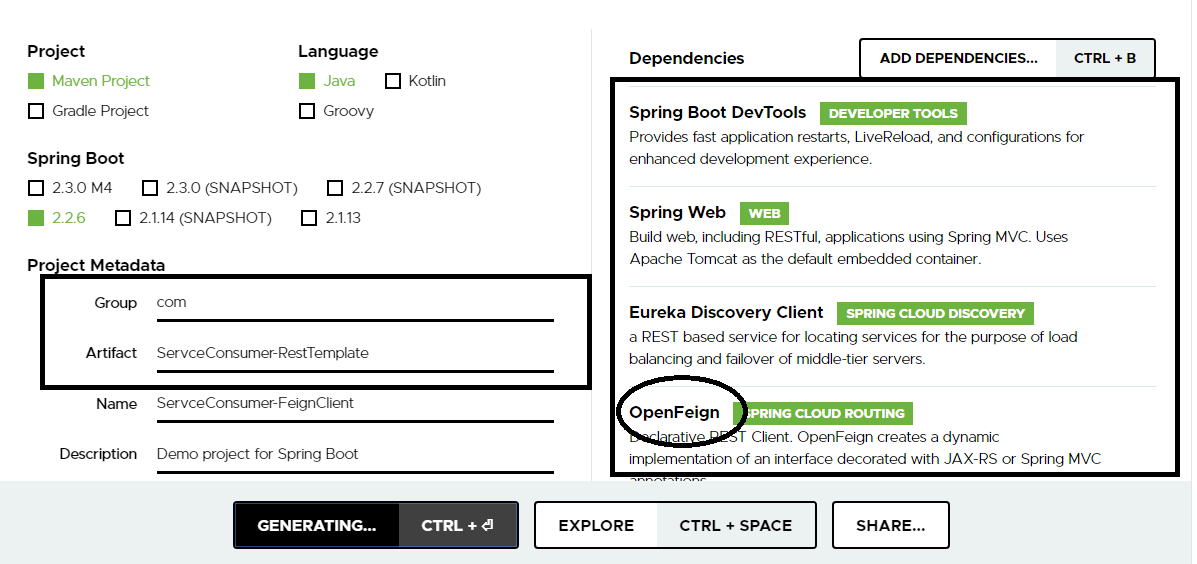
RestTemplate is not reusable, you had to write the application url in every call or you must create a builder uri with the base URL and if you want to use in another client to call the same microservice then you had to create the RestTemplate again so if you want to use the client calling instance you have to use the Feign client.

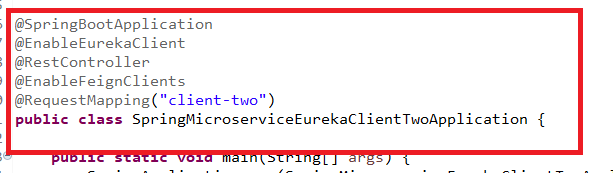
You must build multiple RestTemplate for multiple microservices however the Feign client allows you to create multiple interfaces for multiple microservice calls

Feign Client

Feign Client is a discovery-aware Spring RestTemplate using interfaces to communicate with endpoints. This client is registry server-aware client. It will be used as the Discovery-server-aware Spring RestTemplate using an interface with service endpoints to communicate, and these interfaces will be automatically implemented at runtime. The Spring cloud Netflix feign client is using services-names instead of service-urls.

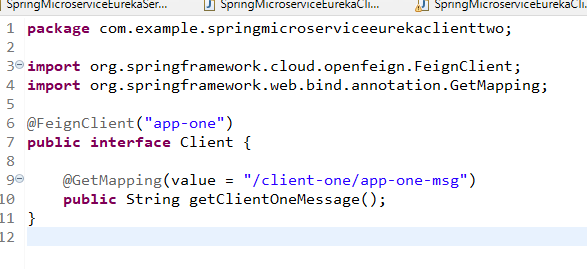
Feign already uses Ribbon, so if you are using @FeignClient then this section also applies.





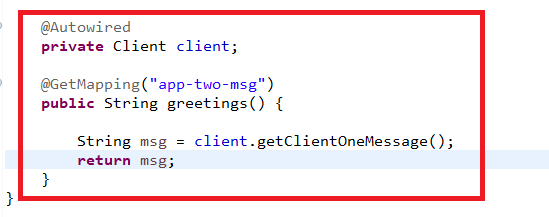
@EnableFeingClients: scans the interfaces that have Feign clients and creates a Client-side LoadBalancing internally

Feign Client



The interface is implemented through proxies and you need to mention the application name in the @FeignClient and application api’s on top of the methods.

The service will call the microservice using getClientOneMessage()

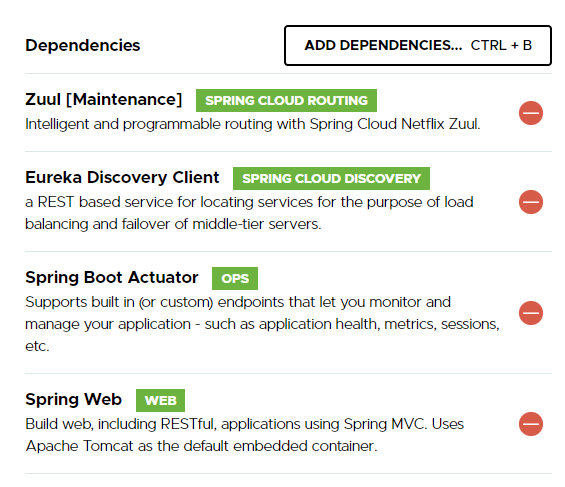


Gateway API: Zuul

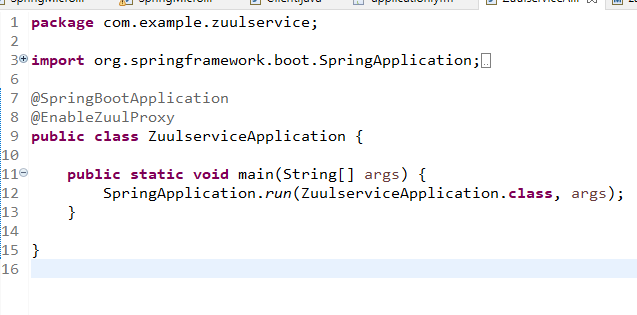
Common gate way for all the microservice calls, you can intercept the request and perform operations with filters like pre-processing & post processing, you can apply logging, tracking, security and so on.

Every service need not to implement these capabilities like logging, tracking as its very difficult and it wouldn’t be consistent as each team may use different algorithms, hence you need a gate way that is a common point to perform all the tasks for all the microservices.

Dependencies:

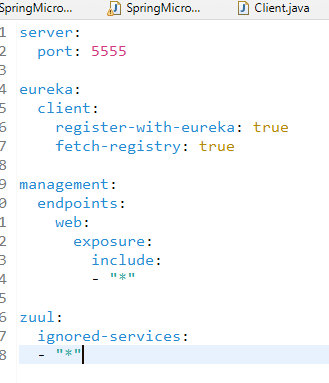


@EnableZuulProxy: This sets up a zuul endpoint and installs reverse proxy filters so that all the request can pass through this and route to appropriate destination



You must fetch service discovery and register zuul service so that all the services can go through the gate way api

application.yml

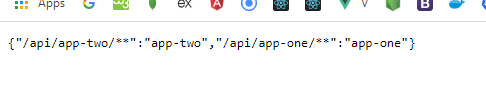


Here Zuulservice registers with eureka and ignores all the services to pass through the zuul gateway it’s like blocking.

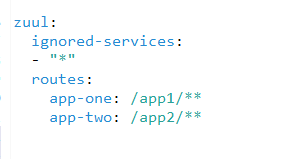
Adding a prefix to the services



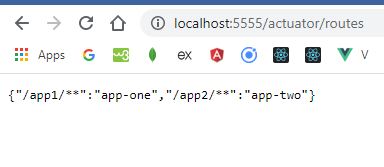
Output:



Making all the services to use a different URI



Output:



Circuit Breaker with Resilience4j:

Here you will see the way you handle failed services

Resilience patterns with Resilience4j

All systems, especially distributed systems will experience a failure, however most software engineers consider complete failure of a piece of infrastructure or critical service to build the resilient systems.

They focus on clustering, load balancing and segregation of infrastructure into multiple locations.

It’s easy to detect the failure services comparing to the slow services and if a service fails then you can easily route to another instance however detecting the poor performing services and routing around it is not that simple because

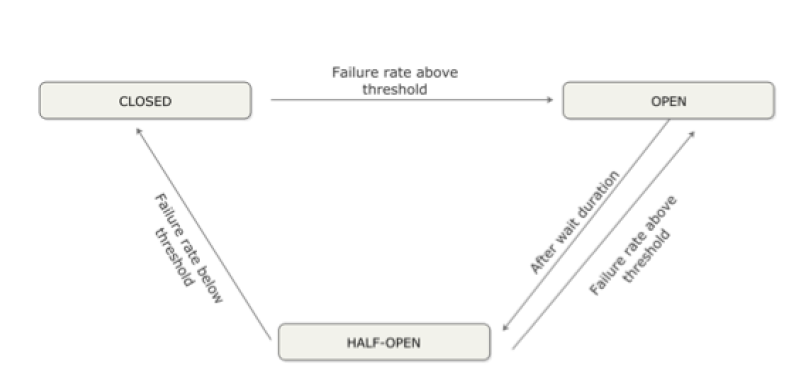
1. Degrading of services can occur at irregular intervals and may take time to build momentum to collapse completely after the resource exhaust its thread pool or connection pool.
2. Calls to remote services are synchronous and don’t cut short a long running call, the caller of the service has no concept of timeout as it’s a remote and call they had to wait for response, though you can think of keeping a timeout facility it’s not known by the consumer how long should it wait because the remote service can respond to the call in different pace depending on its performance & traffic
3. Applications are designed to deal with complete failures of remote services not partial degradations

What is harmful about poorly behaving services is the consumer will utilize resources until it gets a response and more problem occur when there is a cascading of calls from one service to another service and so on, it leads to cascading effect which can bring down an entire application ecosystem as all other consumers resource get exhausted when they wait for the response, what we need is an immediate response/fail fast so that intermediate service calls don’t keep waiting, especially in cloud environment we can’t say when some service could go down or slow, hence you must give an alternative response like a fallback mechanism for poorly behaving services until it becomes available, so that all the services get those alternative response the moment any service goes down and it is made available until the remote services comes up by constantly checking for remote service availability.

Resilience4j finite state machine

Resilience4j will use a state machine while calling remote services to pass all the requests to the remote service and block all the requests to remote service based on 3 states

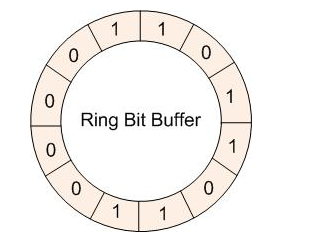
1. closed
2. open
3. half-open

  
A circuit break must be in CLOSE state or atleast HALF-OPEN state to call the remote service, if it is in OPEN state it means that the remote service is down and it has met the failure rate above the threshold, after some duration circuit will not fully open rather it partially opens to ensure the remote service can take calls, it needs successive calls to the remote service to CLOSE else it goes to OPEN state.

Once the circuit breaker is in OPEN state all the call goes to the fall back method without calling the exact method that is invoking the remote service.

A fallback method can give a different result for the consumers for some time until the remote service recovers and giving consumer some time for remote service to recover.

Ring Bit Buffer

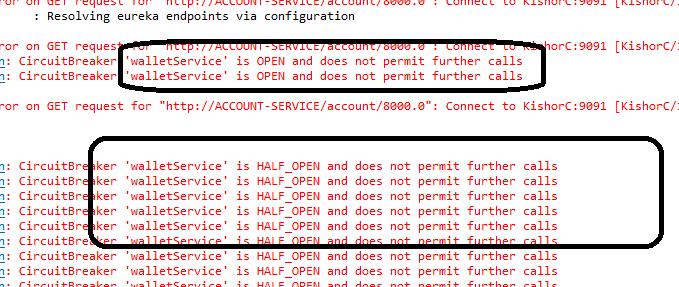


This is the ring bit buffer which stores 0 or 1 for success or failure which is used by the circuit breaker, the ring bit buffer must be full before calculating the failure rate, suppose ring bit buffer size is 10, then atleast 10 calls must be evaluated before evaluating the failure rate, suppose there are only 9 calls the circuit breaker will not trip open even if all 9 calls are failed.

The state of the circuit breaker changes from closed to open when the failure rate is above the configured threshold, then all calls are rejected for a configured time duration, CircuitBreaker throws CallNotPermittedException when the circuit is open.

After some duration is elapsed circuit breaker state changes from OPEN to HALF\_OPEN and allows configurable number of calls to see if the remote service is still unavailable or has become available again, the circuit breaker uses a another configured Ring Bit Buffer to evaluate the failure rate in the HALF\_OPEN state. if the failure rate is above the threshold then it goes back to OPEN state, If the failure rate is equal or below to the threshold the state changes back to CLOSED state.

Below are the logs you get when the circuit is tripped.



Implementing the circuit breaker with fallback method

Let us add the circuit breaker dependency in the wallet-service which is calling the account service

pom.xml



Actuator: to check the status of the circuit breaker in the production

Resilience4j: to use circuit breaker implementation

AOP: To call the fall back methods without actual invocation

WalletService.java

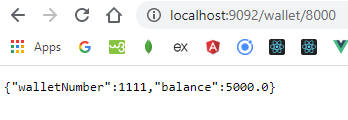


If the Account-Service is down, circuit breaker calls the fallback method as well as the remote service, but after certain threshold is met it OPEN the circuit so that all the call goes directly to the fallback method without calling the actual method however after some duration circuit will be HALF\_OPEN and based on the success it will CLOSE the circuit, however all these are the configurations you have to do in application.yml file of wallet-service

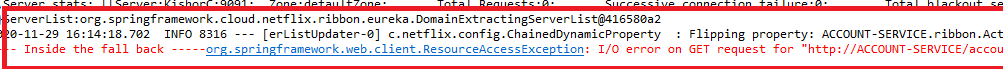
application.yml



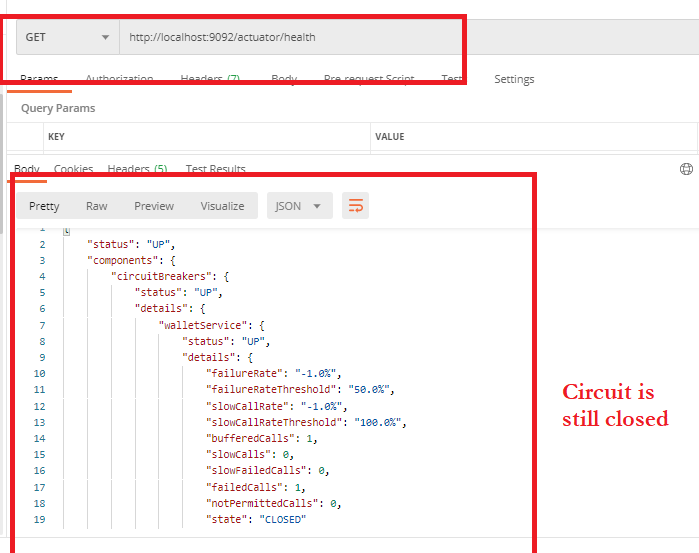
Stop the account service and look for the console and the health of the circuit breaker



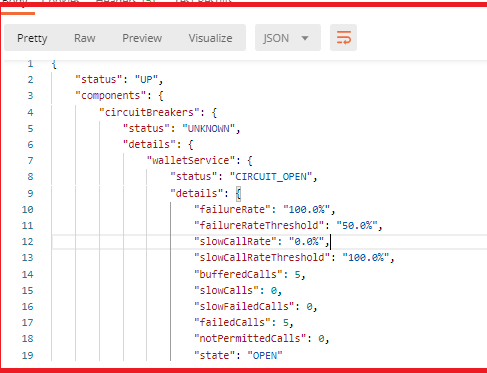
The above output is because of ResourceAccessException which occurred due to the failed service and goes to the fallback method after some seconds which you can notice when browser waits for a while



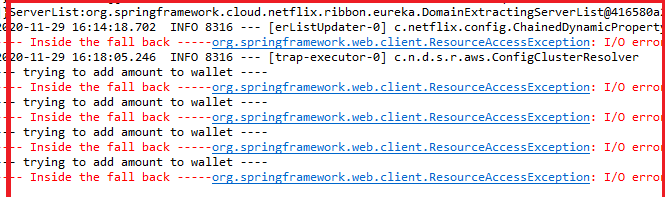
You can see the circuit breaker state in the wallet service actuator health from actuator/health that shows



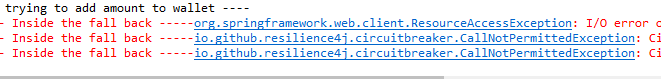
Make few more requests to the failing service, since the threshold is 5, the circuit OPENS and TRIPS at the failedCalls: 5, you can see that in the health



You can see in the console that call happened five times to the real method, however once circuit OPENS the scenario will be different



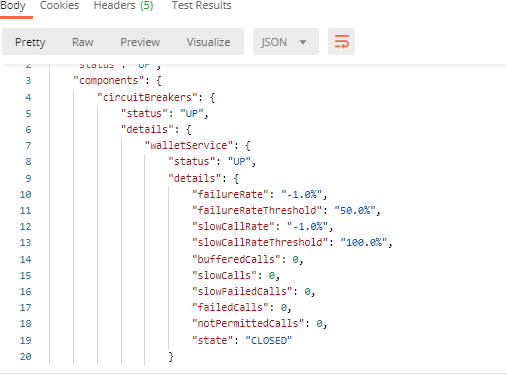
Now if you call the same service you don’t get the log of real method that is calling remote service



Directly the fallback is getting executed the moment the circuit is OPEN, but the circuit will HALF\_OPEN after 50s according to the configuration, but 3 consecutive failure calls will again OPEN the circuit, and 3 consecutive success calls will CLOSE the circuit from HALF\_OPEN



Now you start the account-service and see the state, but on the 3rd call the circuit will close

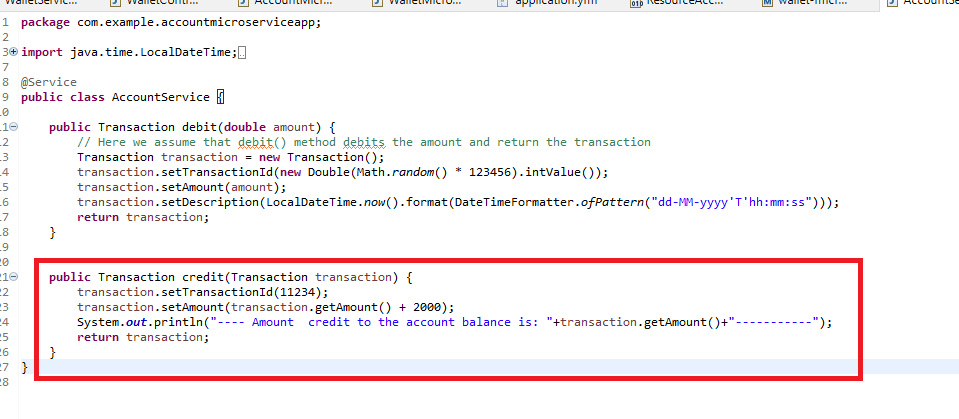


Add circuit breaker to multiple remote service calling methods is similar to the above implementation, it must have a separate fallback method.

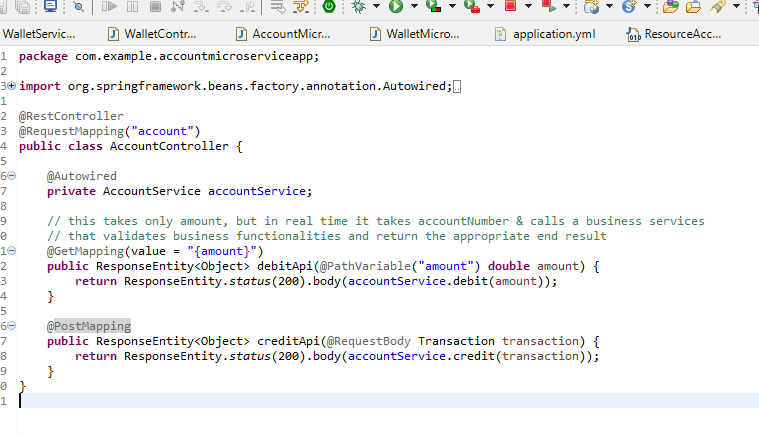
When you add configuration of circuit-breaker to another method of wallet service you get the response of health actuator



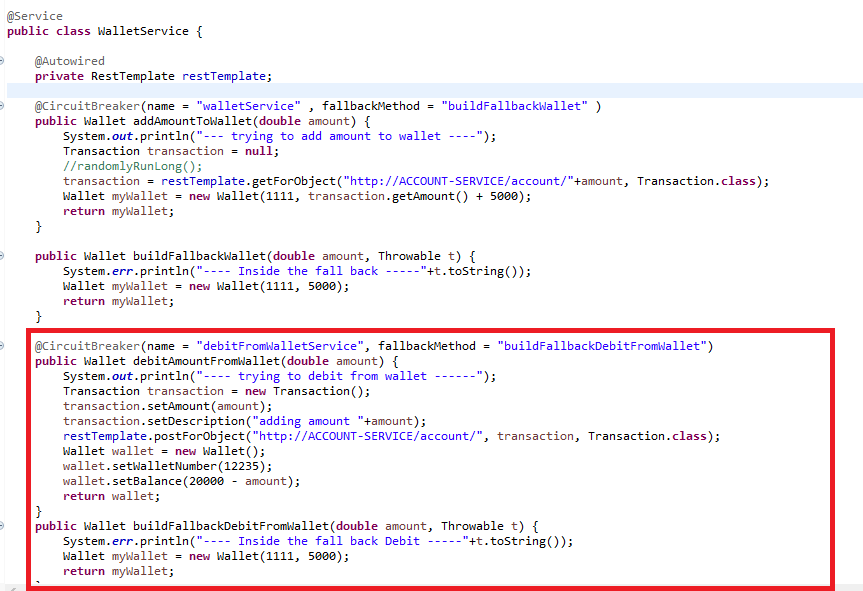
Modify the AccountService to receive the amount



Create a rest endpoint in the AccountController



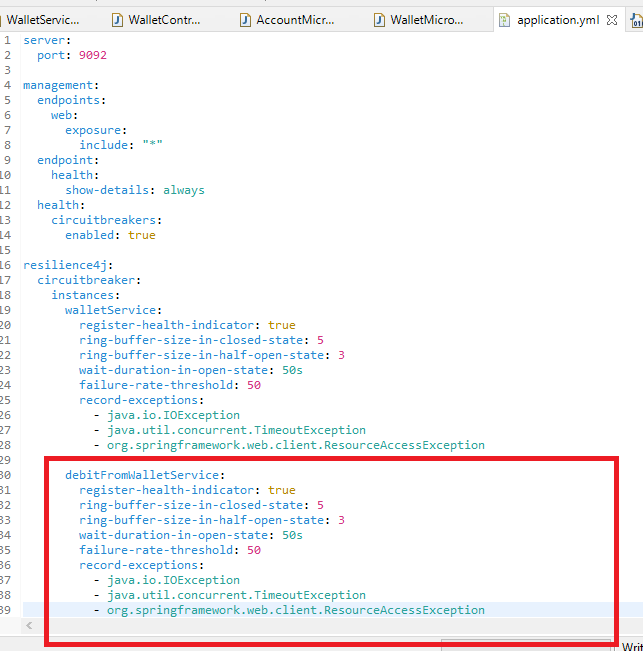
Add the CircuitBreaker & Fallback in the consumer wallet-service



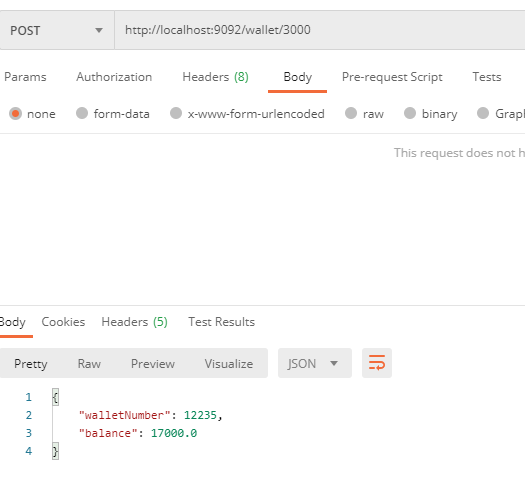
Create a rest end point in WalletController



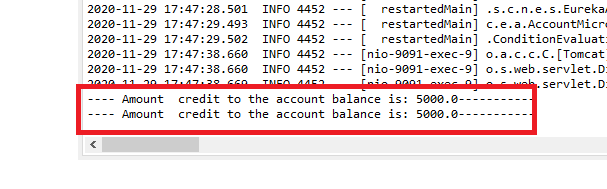
Add the configuration of CricuitBreaker in the application.yml



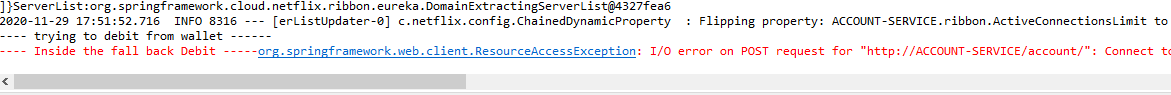
Send request to the wallet service you will see the logs in the Account service



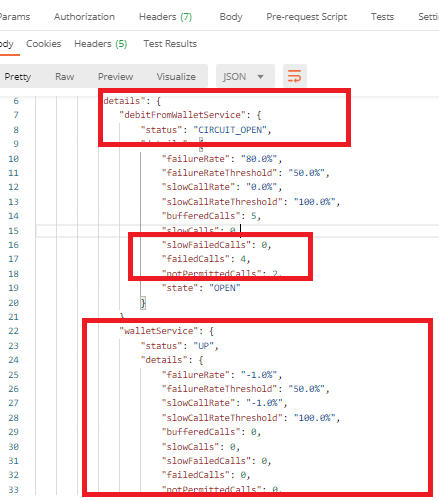
Output in the account-service



Stop the account service and check the circuit breaker state



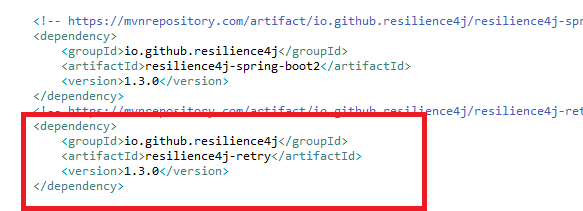
Fallback is called, you can see the state from the actuator health



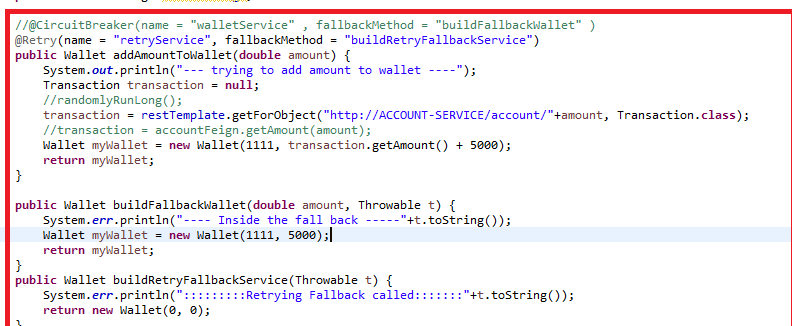
Resilience4j Retry:

You can make few attempts to call failed services after some wait duration.

pom.xml



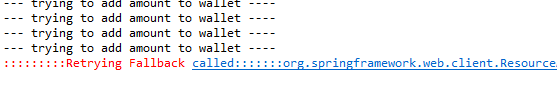
Comment @CircuitBreaker and add @Retry on top the method



application.yml

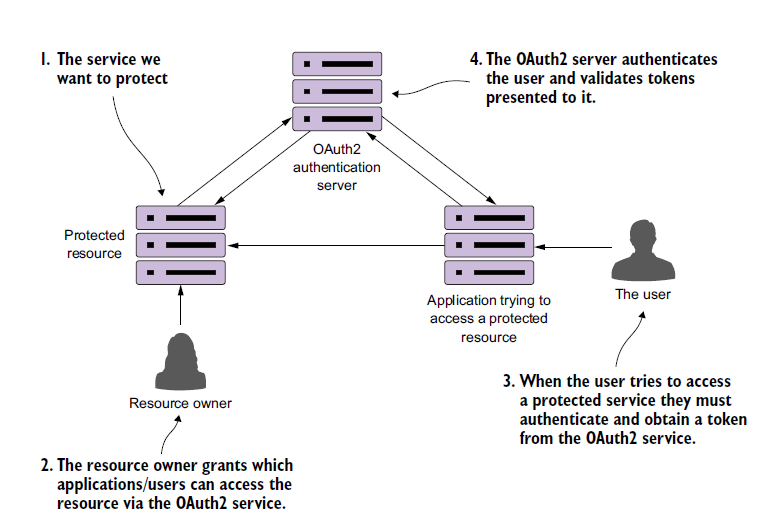


Stop the account service and call from wallet service you will see retries 4 times and then the fallback getting executed.



Securing your microservices

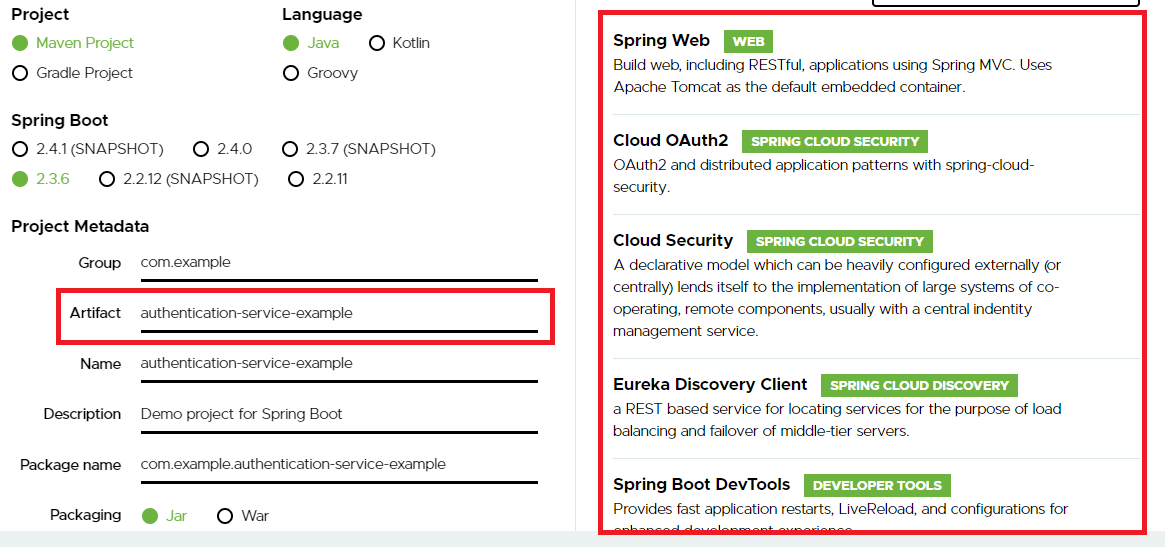
* OAuth2 is a token-based authentication framework to authenticate users.
* OAuth2 ensures that each microservice carrying out a user request doesn’t need to be presented with user credentials with every call.
* Each application that wants to call our services needs to be registered with our OAuth2 authentication service.
* Each application will have its own application name and secret key.
* User credentials and roles are in memory or a data store and accessed via Spring security.
* Spring Cloud Security supports the JSON Web Token (JWT) specification.
* JWT defines a signed, JSON standard for generating OAuth2 tokens.
* JWT carries user informations who are authenticated with a digital signature so that OAuth server can check whether anyone tampered it
* JWT follows standards, but OAuth2 tokens don’t follow any standards like digital signature, extensibility, using private key & public key for encryption & decryption.
* Hence using JWT in OAuth2.0 is better instead of using only OAuth service tokens, as JWT can carry user informations including their roles so OAuth doesn’t need to check to user is valid instead it can only check whether no tampering in the signature
* JWT must not carry sensitive information’s as it can be decoded with any online Api’s ex: <https://www.jsonwebtoken.io/>
* JWT must only carry tokens which will have user name and the roles which is not at all a threat and any hacker doing tampering or changing roles can be easily identified as the signature doesn’t match against the generated server when you tamper.



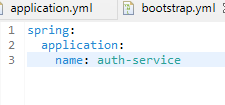
Components used in the OAuth2 service

1. OAuth2 Authentication Server: Authenticates the application & user using Authentication Manager, generates the token based on the authentication & verifies the token when in need
2. User: Who tries to access the protected resources (Microservices)
3. Application: Client application used by the end user, itself will have few parameters get authenticated by OAuth server
4. Protected Resource: Services that can be accessed by valid user who has permissions
5. Resource Owner: Grants permissions to access the resources via OAuth2 service

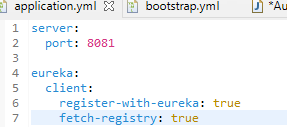
Project Dependencies



bootstrap.yml



application.yml



Creating an Authentication Server

We need to create an endpoint that are called by the services which are protected resources to validate the token and retrieve the user information

AuthenticationServiceExampleApplication.java



Registering the applications that can use protected services

OAuth2Config.java



configure(ClientDetailsServiceConfigurer) defines which clients are going to register our service

configure(AuthorizationServerEndpointsConfigurer) defines different components used within the AuthorizationServerConfiguration, it tells spring to use the default authentication manager and user details service that comes up with spring.

{noop}password: To make our code work with plain text password we to use this format, this delegates the password to a NoOpPasswordEncoder automatically.

withClient() and secret(): provides the name of the application that we are registering along with a secret that will be presented when the my-client application calls our OAuth2 server to receive an OAuth2 access token.

Configuring my-client users

The AuthenticationManagerBean is used by Spring Security to handle authentication.

The UserDetailsService is used to hold user information that will be returned from Spring Security.

The configure() method is where we’ll define users, their passwords, and their roles.

WebSecurityConfigurer.java

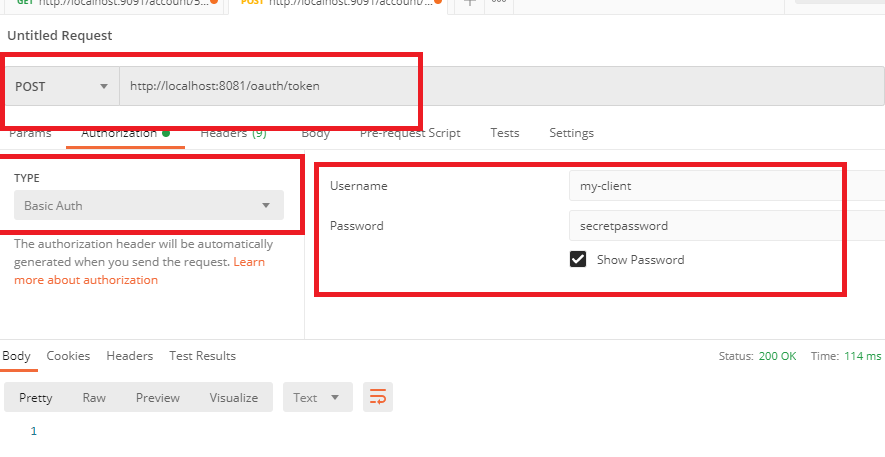


AuthenticationManagerBean and UserDetailsService are used to configure the /oauth/token and /user.

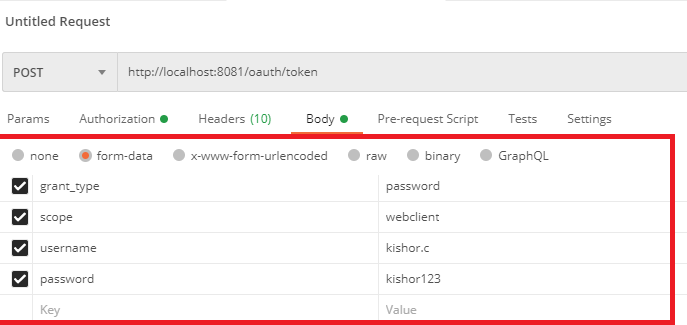
Now user can use application (my-client) to authenticate however the application passes few extra inputs in the form(username, password), hence through postman we can send request with all the required inputs to get the token which is used in every request header to access the service, if the token is not provided then you will get Unauthorized error.

Postman setup

Setting up the application credentials with Basic Authentication type

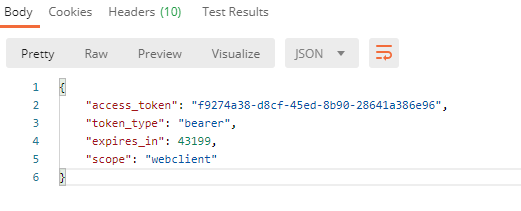


Setting up the user credentials, grant type & scope



When requesting an OAuth2 token, the user’s credentials are passed in as HTTP form parameters to the /oauth/token endpoint.

The JSON payload that’s returned from /oauth/token call.



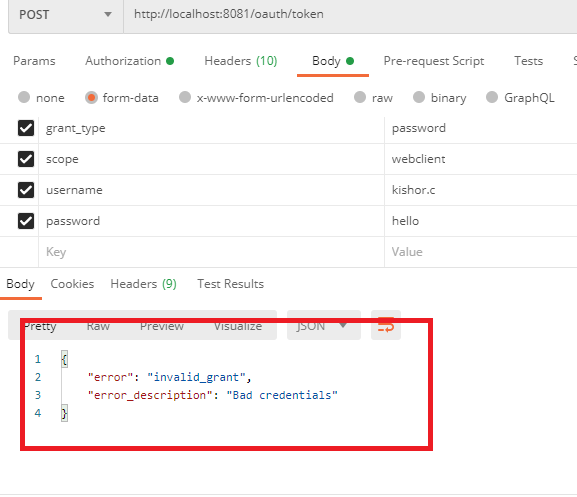
access\_token: it is the authentication token presented with each call.

token\_type: the type of OAuth2 access token being generated

expires\_in: number of seconds before the access token expires

scope: defined scope for which the token is valid.

If the credentials are wrong you will get the bad credentials error



When user wants to access the service he passes only username & password and also he doesn’t need to enter the application username and password as he will be already using the registered application.

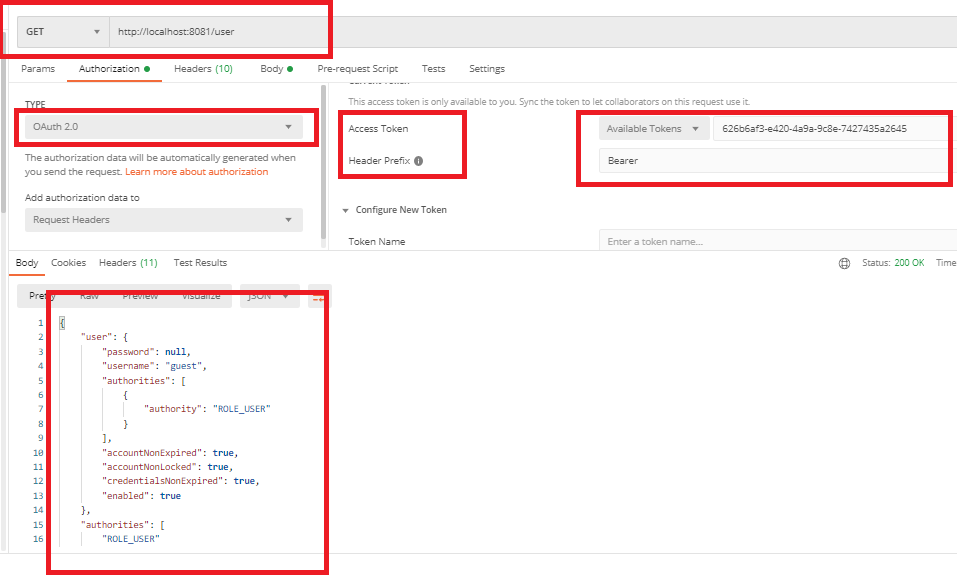
You will get different access token for guest username & password



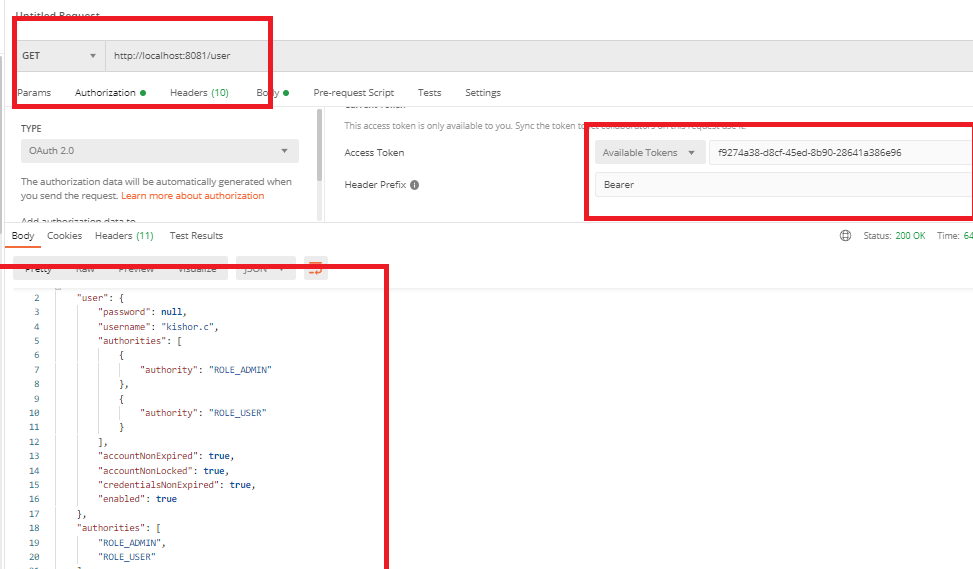
The access token is passed through HTTP headers on each call to the services that are protected.

Now we have a valid OAuth2 access token from the OAuth service, the microservices that are protected resources are going to call the authentication service /user endpoint to validate the token and retrieve the user information.

Let us try to call /user endpoint with the generate authentication token to see the user information like Kishor.c or guest.

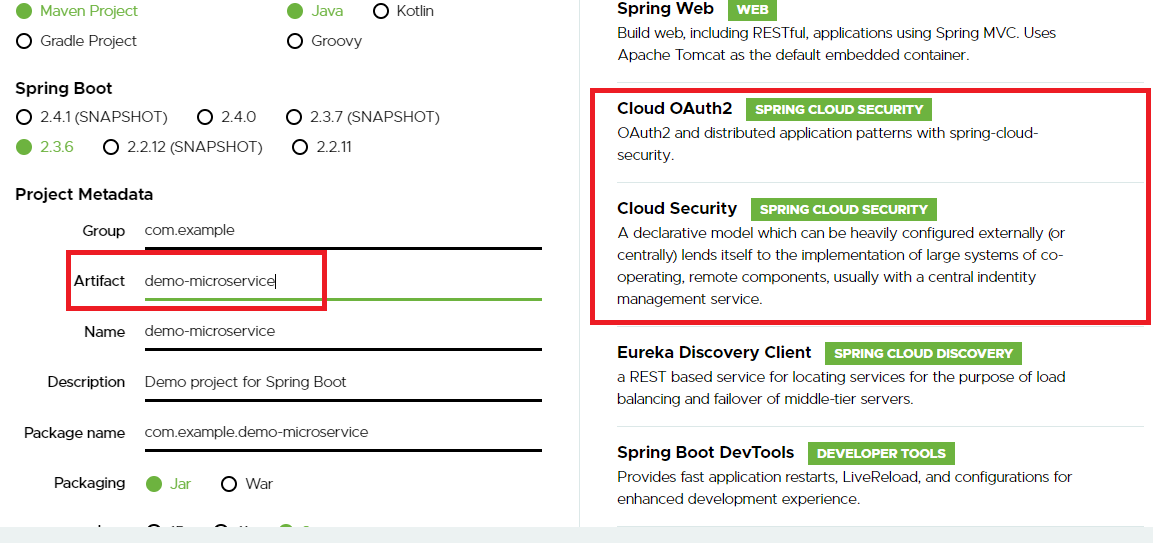


You can see the ROLE\_USER and we also have a token for ADMIN & USER authorities for username Kishor.c



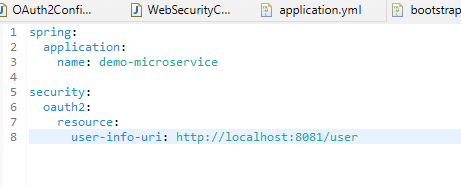
Microservices can use the endpoint /user to validate the token for each call, for that microservice need to be protected resource and use the /user endpoint url.

Adding dependencies to demo-microservice pom.xml

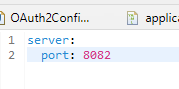


Adding the OAuth2 Authorization server location to validate the token

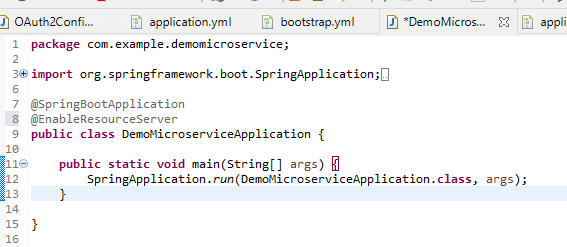
bootstrap.yml



application.yml



Making our service protected resource

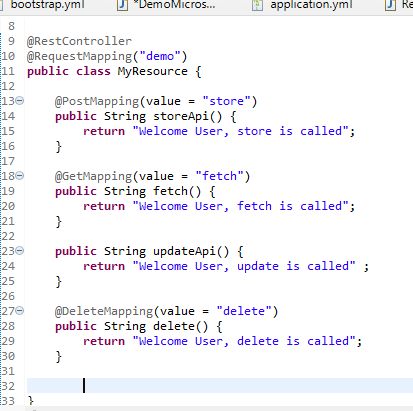


@EnableResourceServer: It does following work for us

1. Enforces a filter that intercepts all incoming calls to the service.
2. Checks if there’s an OAuth2 access token present in the incoming calls HTTP request header.
3. Calls back URL defined in the security.oauth2.resource.user-info-uri to see if the token is valid.
4. Once it knows if the token is valid, it applies any access control rules over whom and what access a service.

Let us create some endpoints in our microservice

MyResource.java

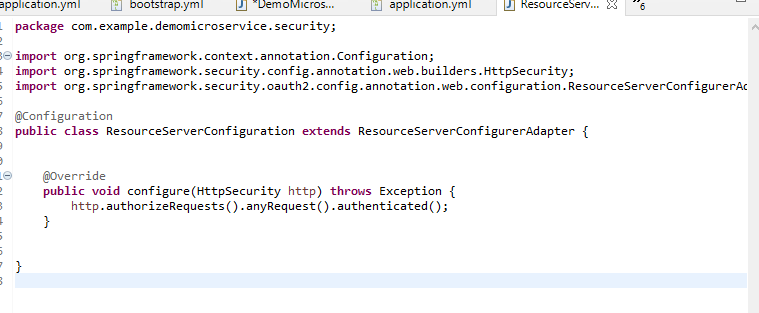


Defining the access control rules over the service

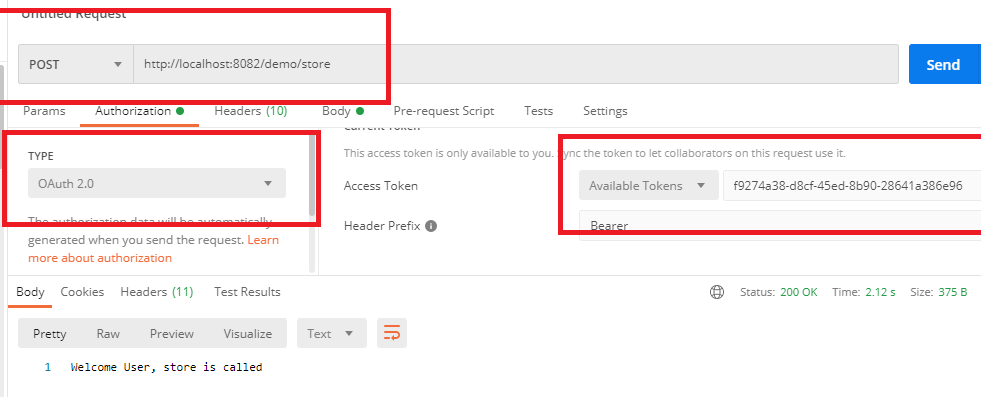
You can define here only authenticated users can access a service URL and users with a specific role can access a service URL.

As of now we will define only the authenticated user can access our microservice

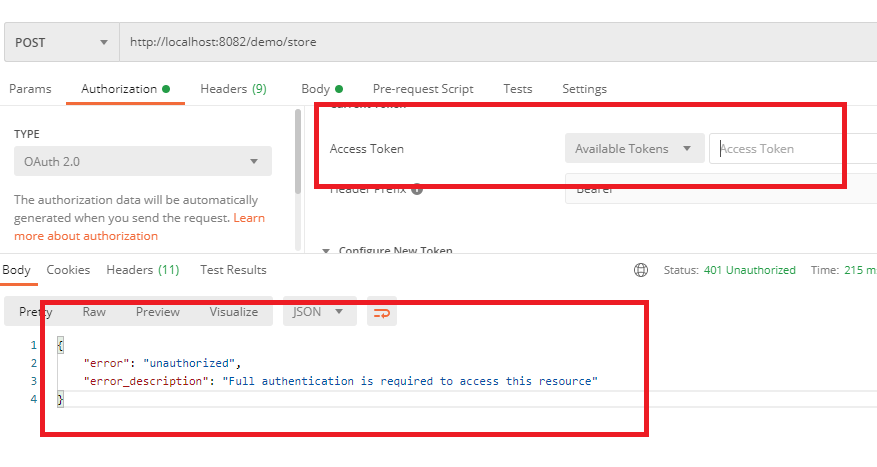
ResourceServerConfiguration.java

The configure() method defines that access to any url in the microservice has to be to the authenticated users only.

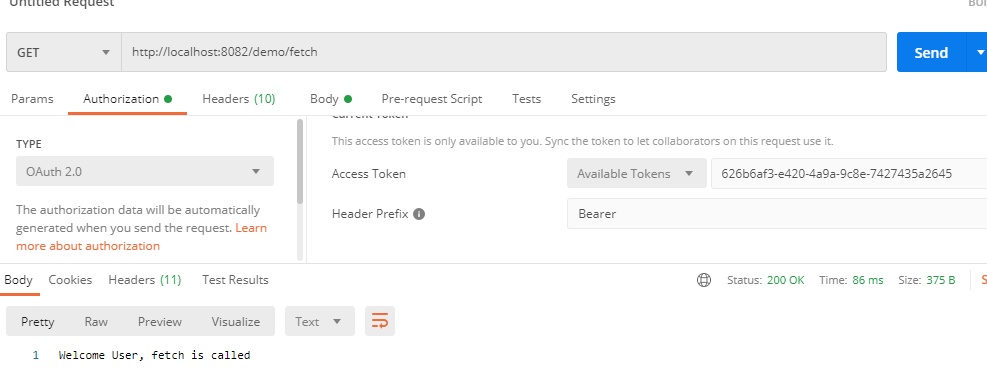
Calling some endpoints with access token



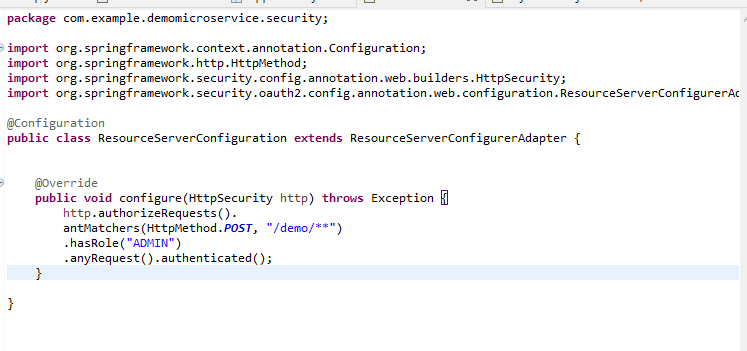
ResourceServerConfiguration allows all the authenticated users to access any endpoints, however if token is not provided you will get 401 unauthorized error



You can access all the uri’s of the microservice if the token is provided

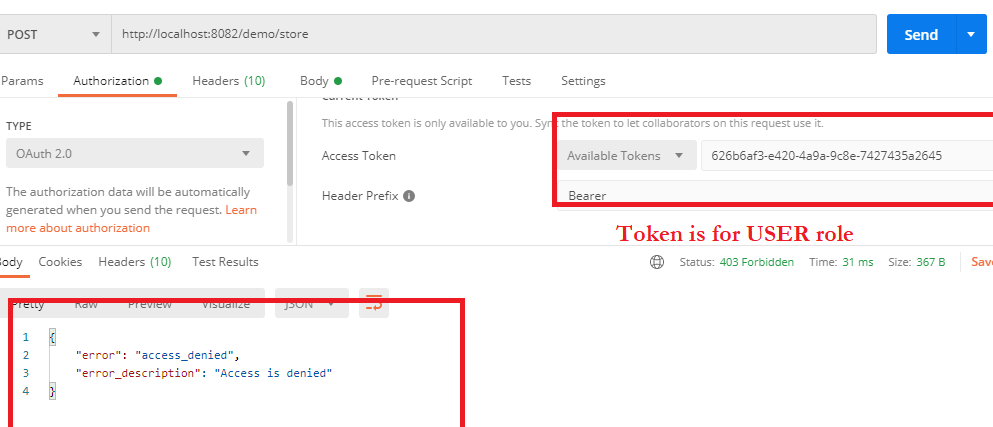


Protecting the service via a specific role

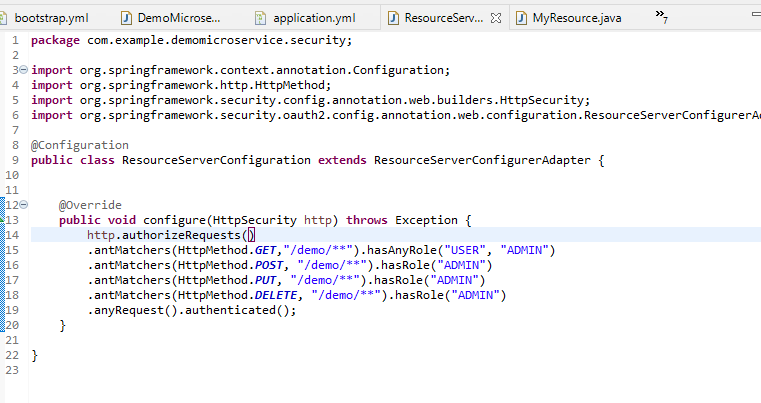


antMatchers can take varying URI’s separated by comma, you can also pass ‘\*’ to match any uri, like /\*/account-service/\*\*

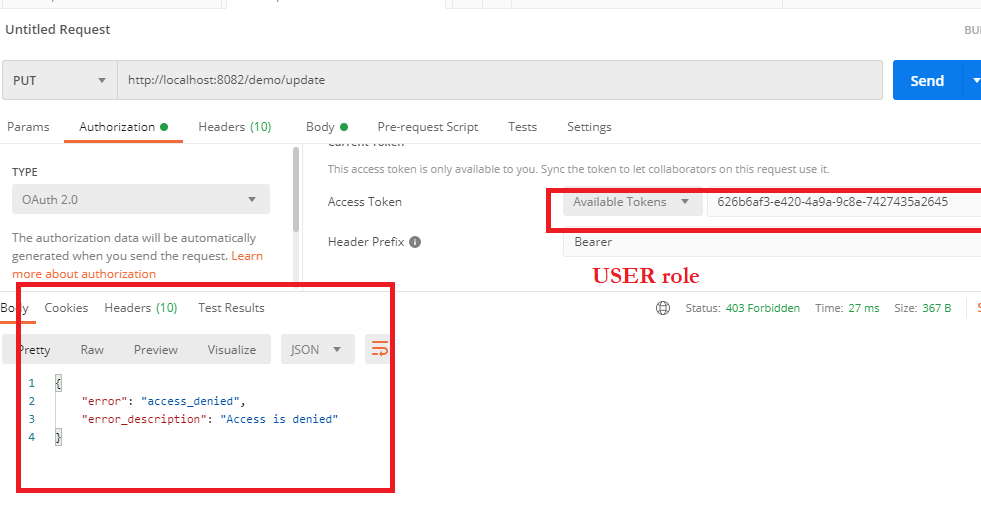
Accessing the resource protected for ADMIN from USER role



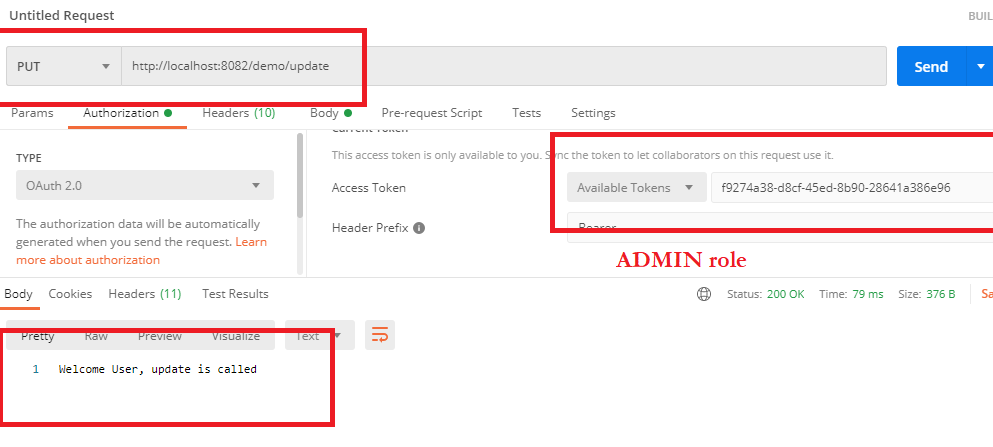
Protecting USER role to access only HTTP GET

Output:

USER role gives the access\_denied



ADMIN role allows you to access



Microservice communication using tokens:OAuth2RestTemplate

RestTemplate can’t use tokens in each request call hence you must use OAuth2RestTemplate which can make OAuth2 authenticated Rest requests with the token

Javascript Web Tokens and OAuth2

OAuth2 is a token-based authentication framework but ironically it doesn’t provide any standard for how the tokens in its specification are to be defined. To rectify lack of standards around OAuth2 tokens, a new standard emerged which is Javascript Web Tokens (JWT). JWT is an open standard proposed by Internet Emerging Task Force (IETF) that attempts to provide a standard structure for OAuth2 tokens. JWT tokens are

* Small: encoded to Base64 and can be easily passed via a URL, Http Headers or an HTTP post parameter
* Cryptographically signed: A JWT token is signed by the authenticating server that issues it. This means you can be guaranteed that the token hasn’t been tampered with.
* Extensible: It can have additional information before the token is sealed that can be decrypted by the receiving service.

How JWT works

The JWT token is created using a **secret string** that is **stored on a OAuth Server**. Next, the server sends that JWT back to the client which will store it either in a cookie or in local storage.

[](https://i.stack.imgur.com/UXiHK.png)

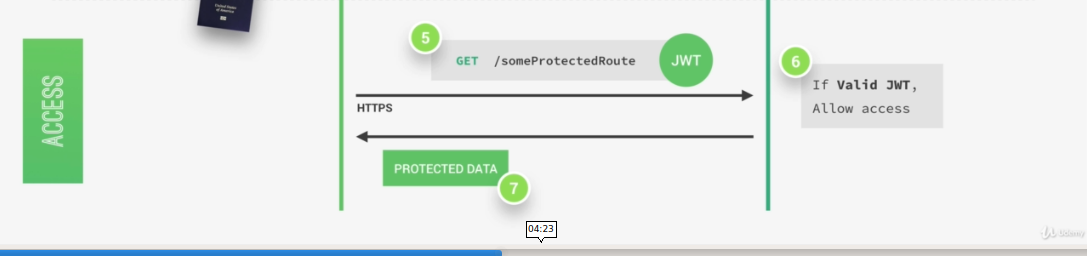
The user is authenticated now and basically logged into our application without leaving any state on the server (i.e., nothing is stored in the server side).

So the server does in fact not know which user is actually logged in, but of course, the user knows that he's logged in because he has a valid Json Web Token which is a bit like a passport to access protected parts of the application.

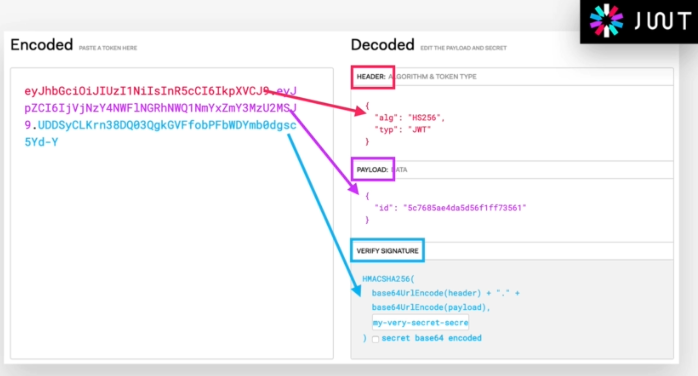
A user is logged in as soon as he gets back his unique valid Json Web Token which is not saved anywhere on the server. And so this process is therefore completely stateless.

Then, each time a user wants to access a protected resource like his user profile data, for example. He sends his Json Web Token along with a request, so it's a bit like showing his passport to get access to that route.

Once the request hits the server, our app will then verify if the Json Web Token is actually valid and if the user is really who he says he is, well then the requested data will be sent to the client and if not, then there will be an error telling the user that he's not allowed to access that resource.

[](https://i.stack.imgur.com/tTxqp.png)

All this communication must happen over https, So secure encrypted Http in order to prevent that anyone can get access to passwords or Json Web Tokens. Only then we have a really secure system.

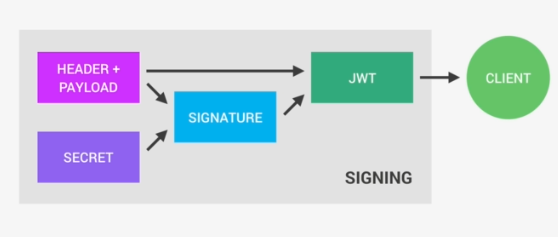
[](https://i.stack.imgur.com/0FpIC.png)

So a Json Web Token looks like left part of this screenshot is an encoding string made up of three parts.

* Header: Some metadata about the token itself
* Payload: Data that we want, it’s encoded(Base64) not encrypted
* Signature: Signing the JWT using header & payload

The payload is the data **So anyone will be able to decode them and to read them**, we cannot store any sensitive data in here. But that's not a problem at all because in the third part, so in the signature, is where things really get interesting. The signature is created using the header, the payload, and the secret key that is saved on the server.

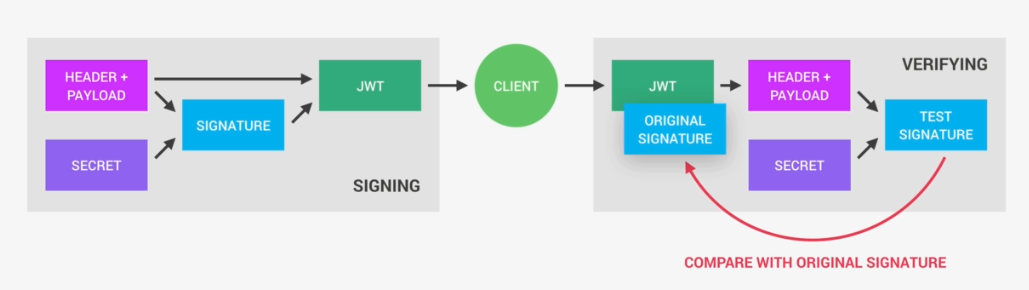
And this whole process is then called **signing the Json Web Token**. The signing algorithm takes the header, the payload, and the secret to create a unique signature. So only this data plus the secret can create this signature, all right? Then together with the header and the payload, these signature forms the JWT, which then gets sent to the client.

[](https://i.stack.imgur.com/bOHqZ.png)

Once the server receives a JWT to grant access to a protected resource, it needs to verify it in order to determine if the user really is who he claims to be. In other words, it will verify if no one changed the header and the payload data of the token. So again, this verification step will check if no third party actually altered either the header or the payload of the Json Web Token.

So, how does this verification actually work? Well, it is actually quite straightforward. Once the JWT is received, the verification will take its header and payload, and together with the secret that is still saved on the server, basically it creates a test signature.

But the original signature that was generated when the JWT was first created is still in the token, right? And that's the key to this verification. Because now all we have to do is to compare the test signature with the original signature. And if the test signature is the same as the original signature, then it means that the payload and the header have not been modified.

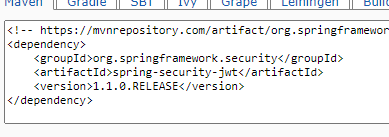
[](https://i.stack.imgur.com/b2dzI.png)

if they had been modified, then the test signature would have to be different. Therefore in this case where there has been no alteration of the data, we can then authenticate the user. And of course, if the two signatures are actually different, well, then it means that someone tampered with the data like changing the roles from ADMIN to SUPER ADMIN. Usually by trying to change the payload. But that third party manipulating the payload does of course not have access to the secret, so they cannot sign the JWT. So the original signature will never correspond to the manipulated data. And therefore, the verification will always fail in this case. And that's the key to making this whole system work. It's the magic that makes JWT so simple, but also extremely powerful.

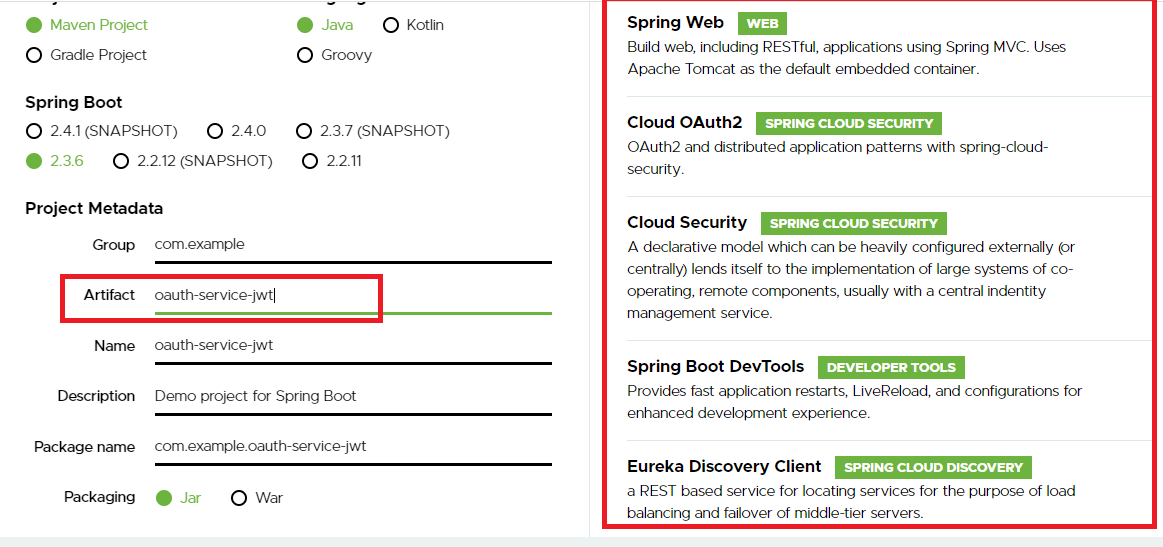
Creating JWT from OAuth2.0

OAuth2.0 access tokens are just some hexadecimal strings that doesn’t have any user data so you can configure OAuth2.0 to generate JWT which follows a standard compare to normal OAuth2.0 token

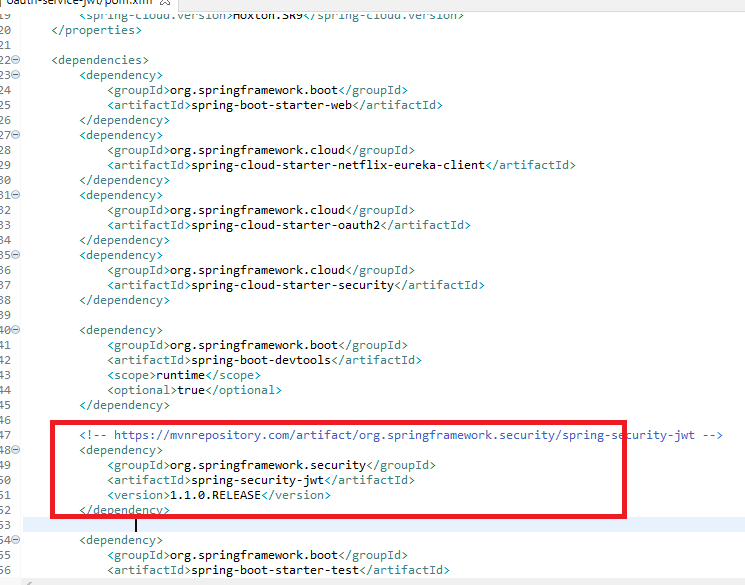
Note: we can modify the existing authentication service just by adding below dependency



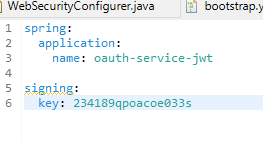
Other Dependencies required for new project are



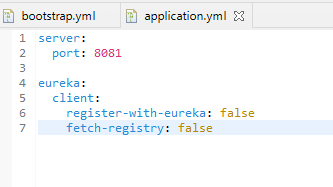
Below is the entry in pom.xml



bootstrap.yml

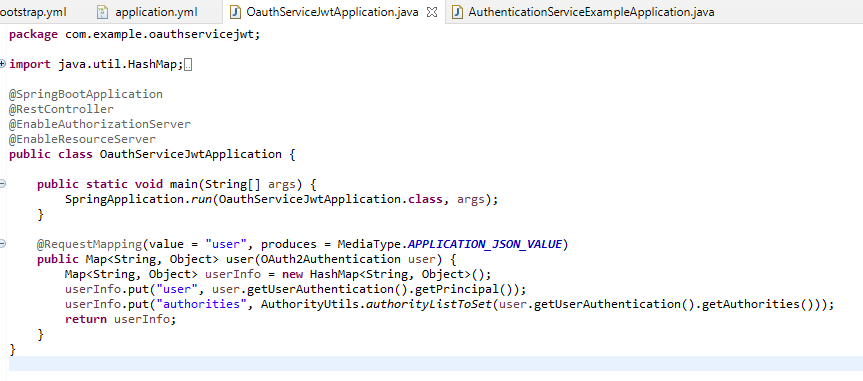


application.yml

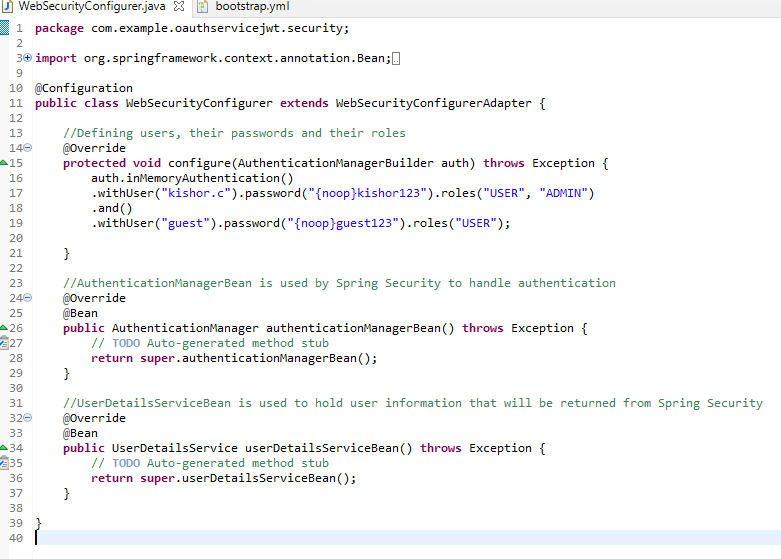


No Changes in the @SpringBootApplication class

OAuthServiceJwtApplication.java



No changes in the WebSecurityConfigurer.java



Configuration to read signing.key ServiceConfig.java



Creating Jwt enhancers JwtTokenStoreConfig.java



TokenStore is the persistence interface for OAuth2 tokens which is necessary at the time of translation

JwtAccessTokenConverter acts as the translator between Jwt encoded token values and OAuth2 authentication information

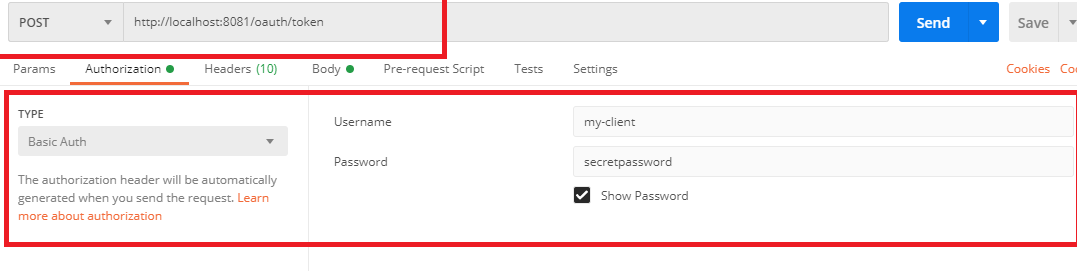
Creating the oauth2 to use JWT

JwtOAuth2Config.java

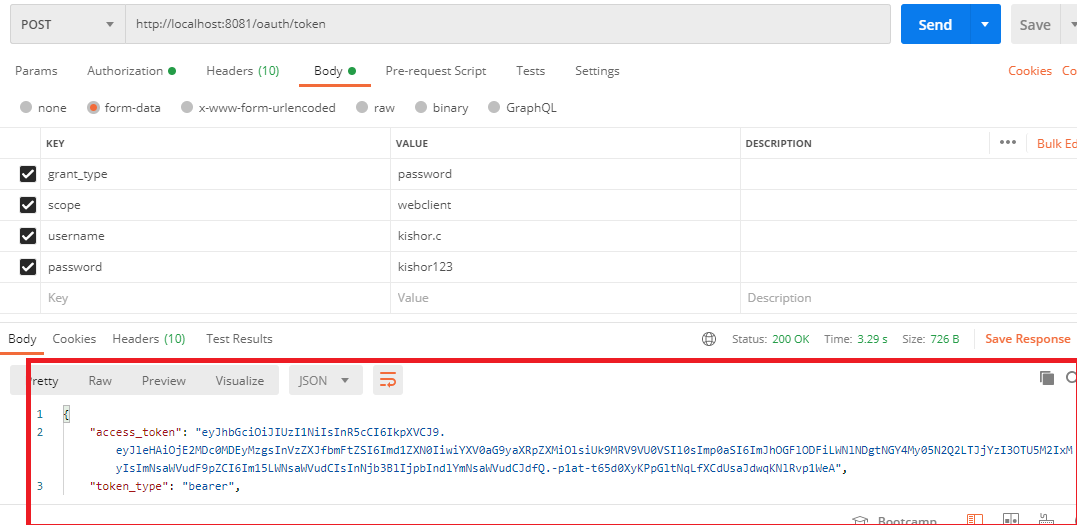


Obtaining JWT by passing the inputs with POSTMAN

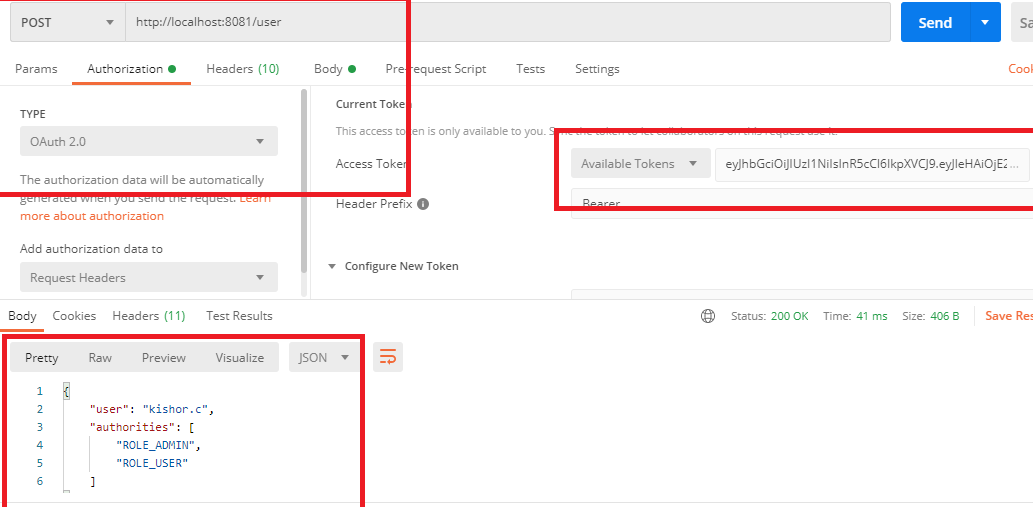
Client configuration



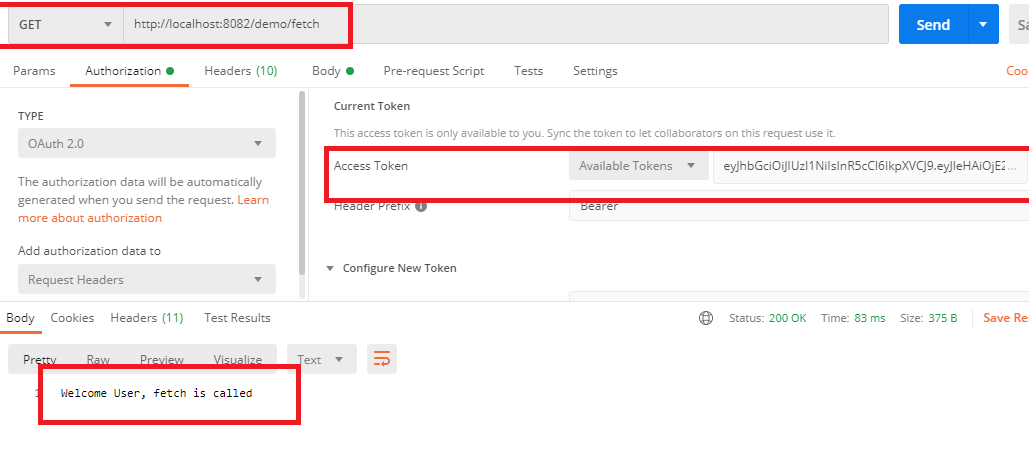
User configuration



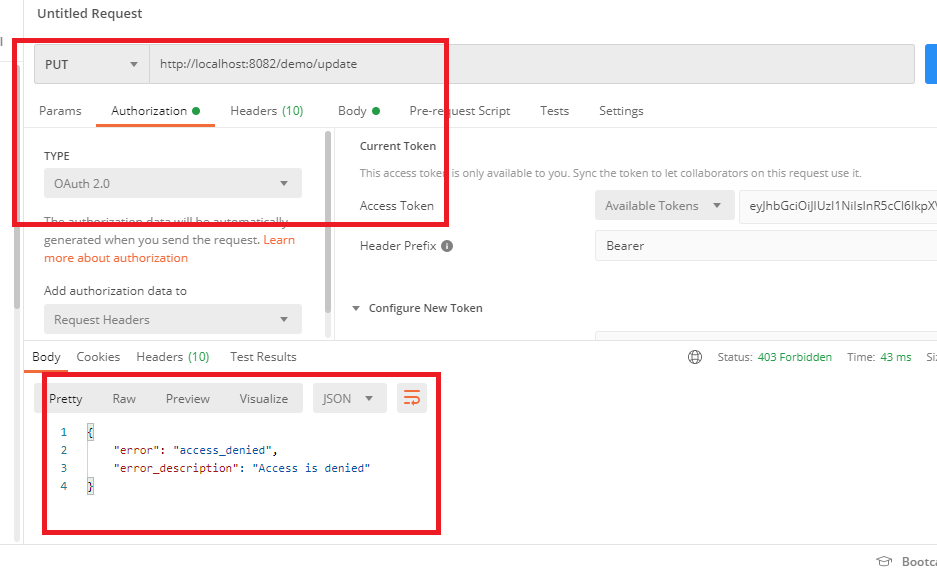
Use the access\_token and send request to the /user endpoint you can see the user details



Call the same demo microservice with the generated JWT access token it should work.



When you use guest credential token and call the update you will get access denied error as below:



Distributed log tracing using Sleuth & Zipkin

It’s obvious that when too many microservices calling each other logging will be challenging as each microservice will store the logs separately, we need to know the flow of the request and response, like identifying or tracing the request & response which passed through what all the microservices and if at any place the request failed we need to identify in which microservice the error occurred.

In distributed systems viewing logs of each microservice would be challenging as we need to compare each microservice logs and combine them together and identify the sequence of the request flow, manually doing so is challenging hence we have Sleuth & Zipkin.

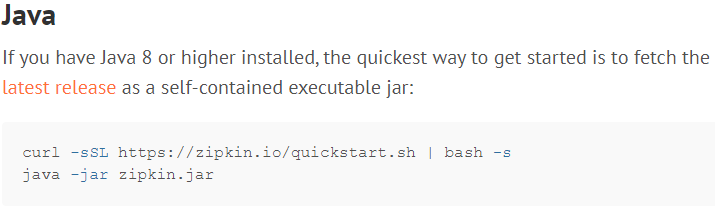
Sleuth: Which takes care of generating some unique trace id’s & span id’s for the request that passed through all the microservice and then with the help of some tools like Zipkin or ELK you can aggregate the logs and view them.

Zipkin: It is a distributed tracing system. It shows multiple logs generated by separate microservice from a single request in one log by collecting the logs from the Sleuth.

Note: Zipkin must be downloaded from the below url & execute the jar which opens the Zipkin system in 9411 port

URL to download Zipkin:

<https://zipkin.io/pages/quickstart>



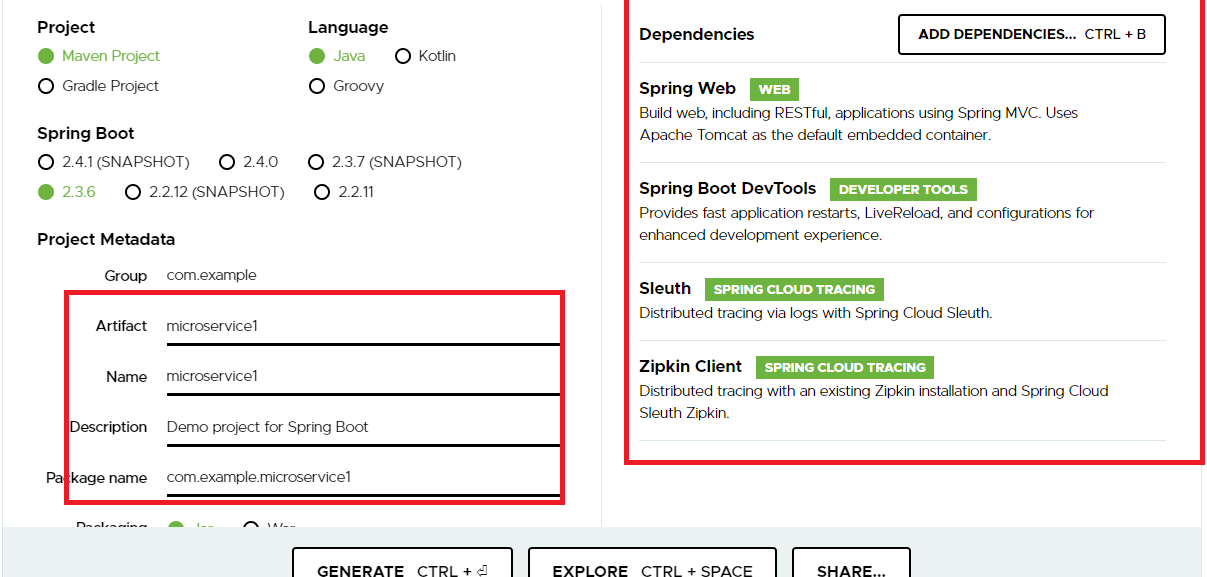
Understanding Zipkin

We will create 4 microservices, where each microservice calls the another microservice except the last one i.e.,

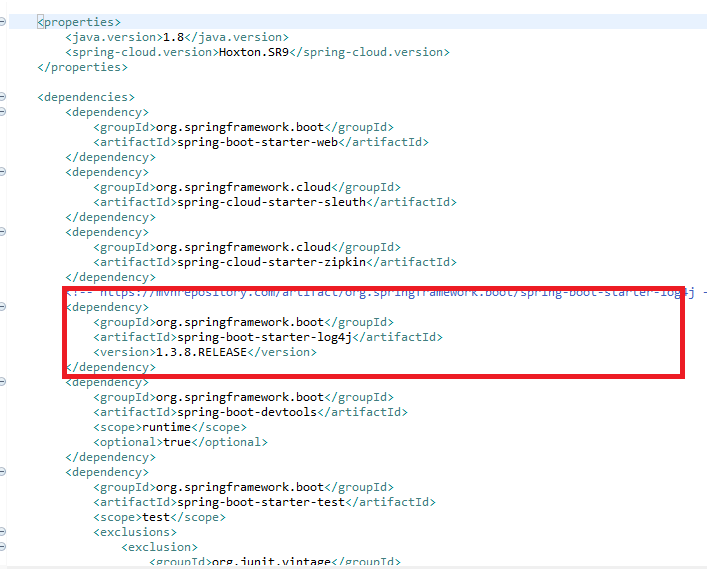
microservice1 -> microservice2 -> microservice3 -> microservice4

The response generated from microservice4 is passed to 3 to 2 to 1 and microservice1 shows the response, in every microservice we log simple information.

Create 4 microservices with below dependencies



log4j is not present in the Spring Initializr we will through maven.



Create 4 projects with all these dependencies and name the projects as

* microservice1
* microserivce2
* microservice3
* microservice4

You just need to use RestTemplate in 3 microservices and call the other microservices and microservice4 doesn’t use RestTemplate as it doesn’t call any microservice.

In all the 4 projects we will have same code except the changes in the rest endpoint, class, method & log names

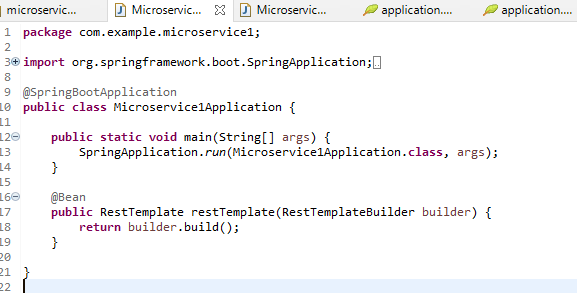
First Microservice

In application properties we have commented Zipkin so that the project will not send the logs to Zipkin, however by default Zipkin is enabled

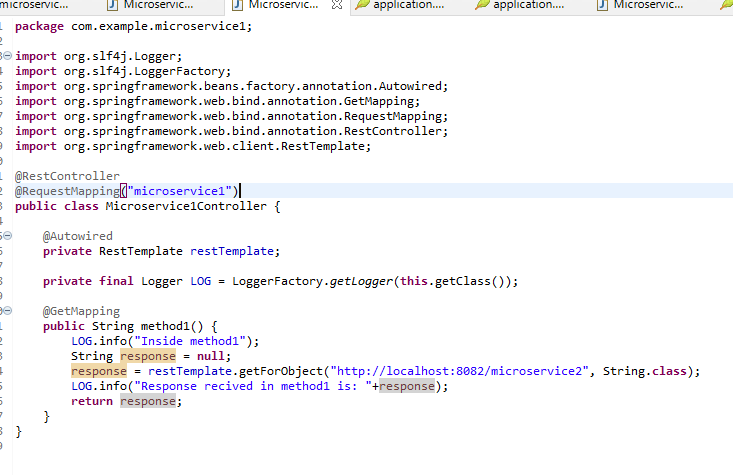
application.properties



Microservice1 Application



Microservice1Controller calls the Microservice2 logs the info



That’s it you can now create other 3 microservices using same procedure but change, application properties, the @RequestMapping uri and call a different microservice

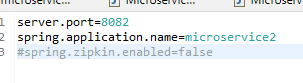
i.e.,

* URI for First Microservice is /microservice1, so you create Second Microservice with /microservice2, and so on.
* Log changes in each microservice
* RestTemplate in First Microservice calls /microservice2 so Second Microservice calls /microservice3 and Third Microservice calls /microservice4, but Fourth Microservice doesn’t call any rather it returns response to third microservice which returns to second that returns to the First Microservice

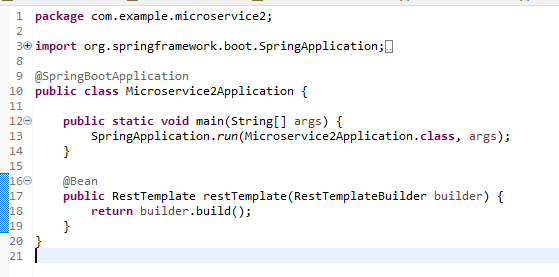
Second Microservice

Follow the same procedure but make changes accordingly

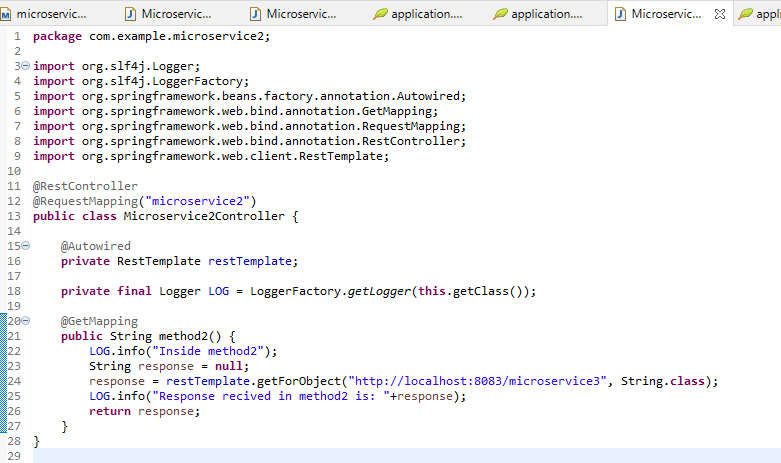
application.properties



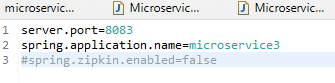
Microservice2Application



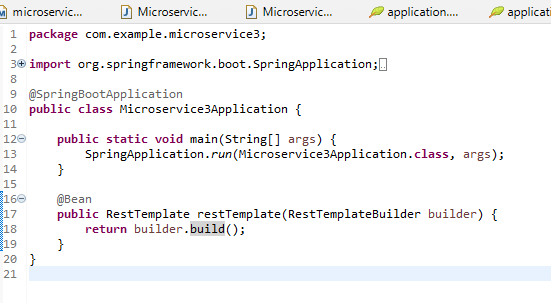
Microservice2Controller logs the info & calls 3rd microservice



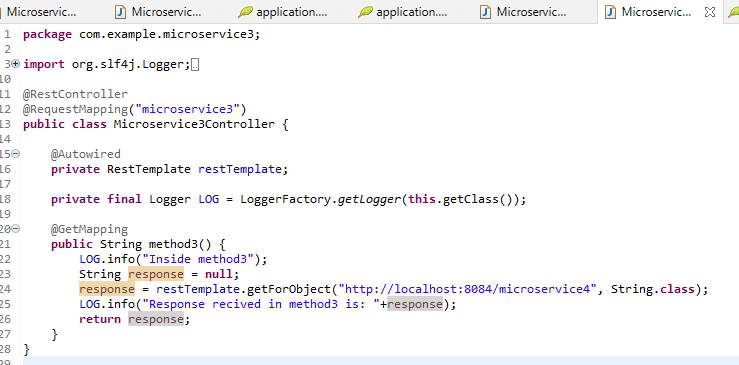
Third Microservice change properties & other things necessary



Microservice3Application

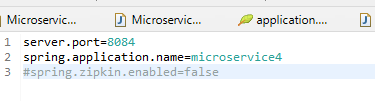


Microservice3Controller that logs info & calls microservice4

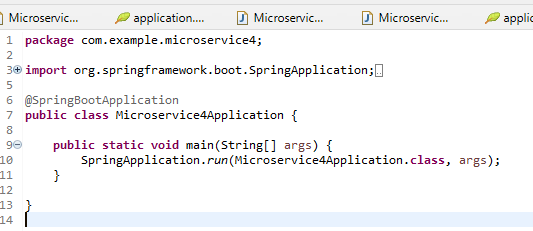


The Fourth microservice doesn’t call any rather returns the response

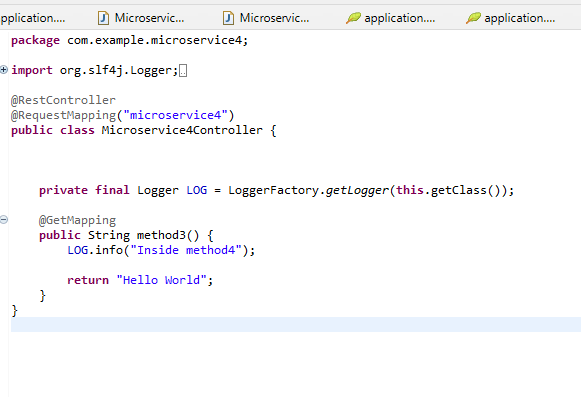
application.proerties



Microservice4Application



Microserivce4Controller that returns response



Now run all the microservices,

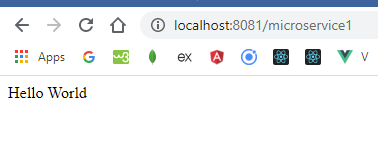
Before making a request execute the zipkin jar

Note: Zipkin port is 9411



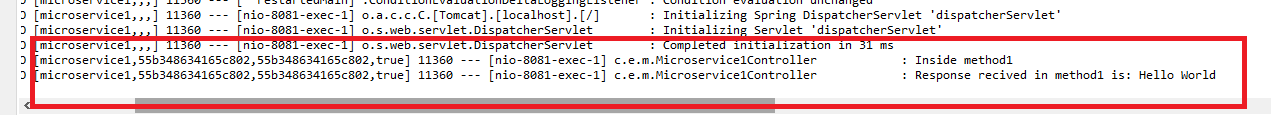
Once all the 4 microservices are launched you can make a request to microservice1

Output:



You can see different logs in each console of the microservice

Micrservice1 log



Microservice2 log



Microservice3 log



Microservice4 log



In all the logs you will see the information in the following format

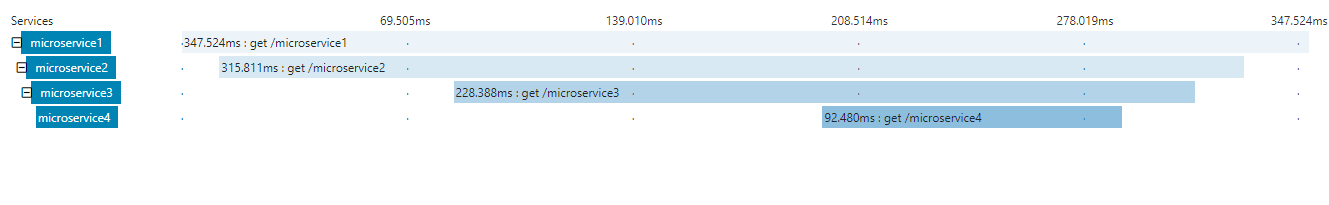
**[unique-traceId, span-id-for-a-particular-method, Zipkin-enabled]**

TraceId is unique for a request so you will see same id in all the 4 logs

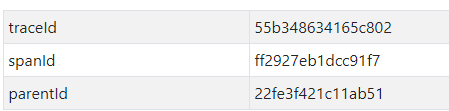
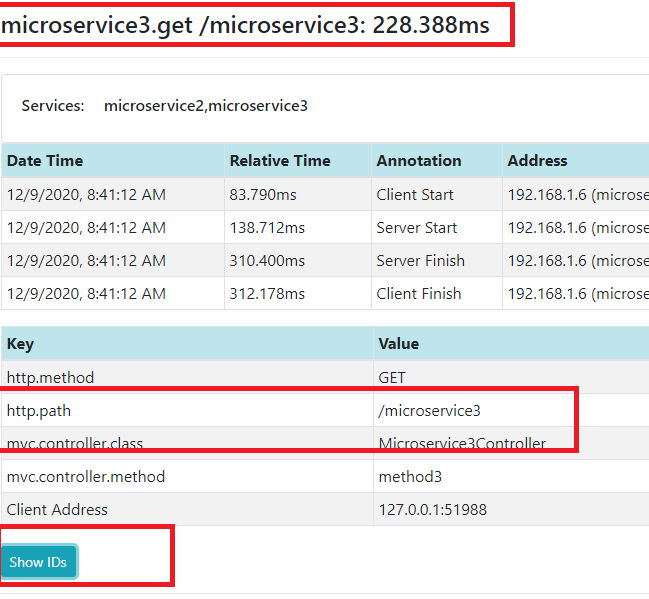
span-id is unique for a particular microservice method so it’s different to each microservice but it is a parent id for next microservice call that you can zee in Zipkin.

Open Zipkin in <http://localhost:9411/>

Click on Find Traces you will see the logs and its sequential flow



Click on any microservice you will see the details of which microservice called what are their log id’s



parented is the span-id of the microservice that called this microservice