Monolithic Application :

- The application is built as a single unit
- It comprises of three layers:

    1.Client side interface(React/Angular)

    2.Server side Application(Springboot/Spring)

    3.Database (MySQL/Oracle)

- Disadvantages:

    Large Code based & lack of Modularity

- To overccome this, Microservices are used.

Microservices :

Microservice architectures are the 'new normal'. Building small, self-contained, ready to run applications can bring great flexibility and added resilience to your code. Spring Boot's many purpose-built features make it easy to build and run your microservices in production at scale.

Microservices are a modern approach to software whereby application code is delivered in small, manageable pieces, independent of others.

Why build microservices?

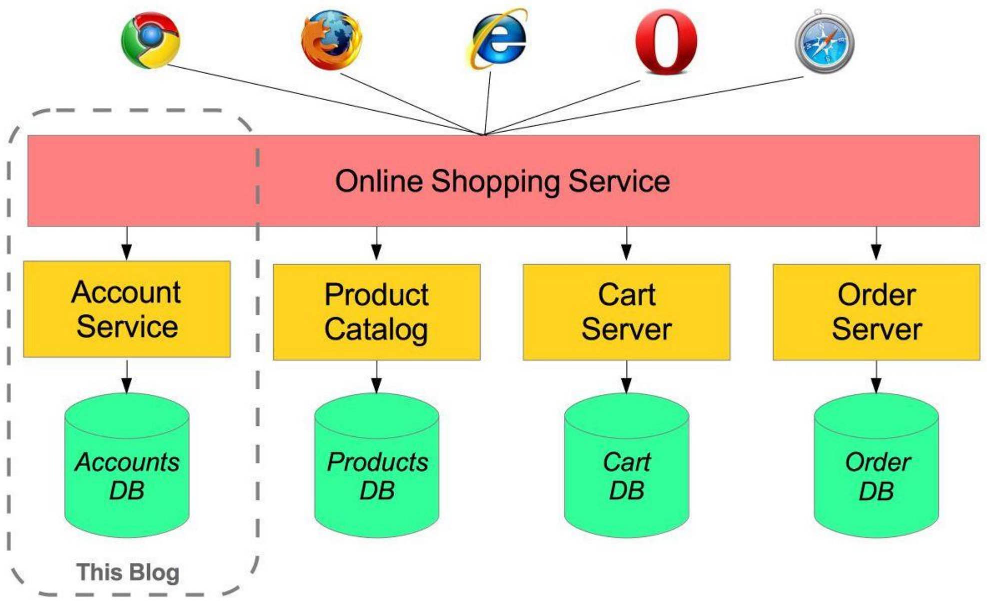
Their small scale and relative isolation can lead to many additional benefits, such as easier maintenance, improved productivity, greater fault tolerance, better business alignment, and more.

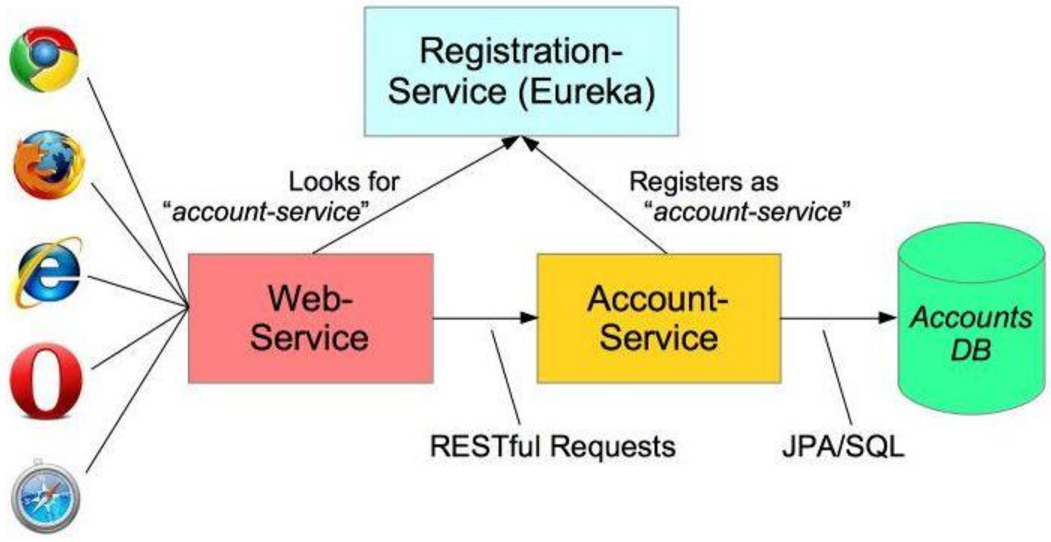
With Microservices Architecture, It's easy to build complex operations.

It will give flexibility to choose technologies & frameworks for each microservices independently.

Microservices allow large systems to be built up from a number of collaborating components. It does at the process level what Spring has always done at the component level: loosely coupled processes instead of loosely coupled components.

For example ,an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts:




Registration-Service (Eureka)
Looks for "account-service"
Registers as "account-service"
Web-Service
Account-Service
Accounts DB
RESTful Requests
JPA/SQL

The Web-Application will make requests to the Account-Service microservice using a RESTful API

How to start with microservices?

1. If you have monolithic application, identify all possible standalone functionalities.

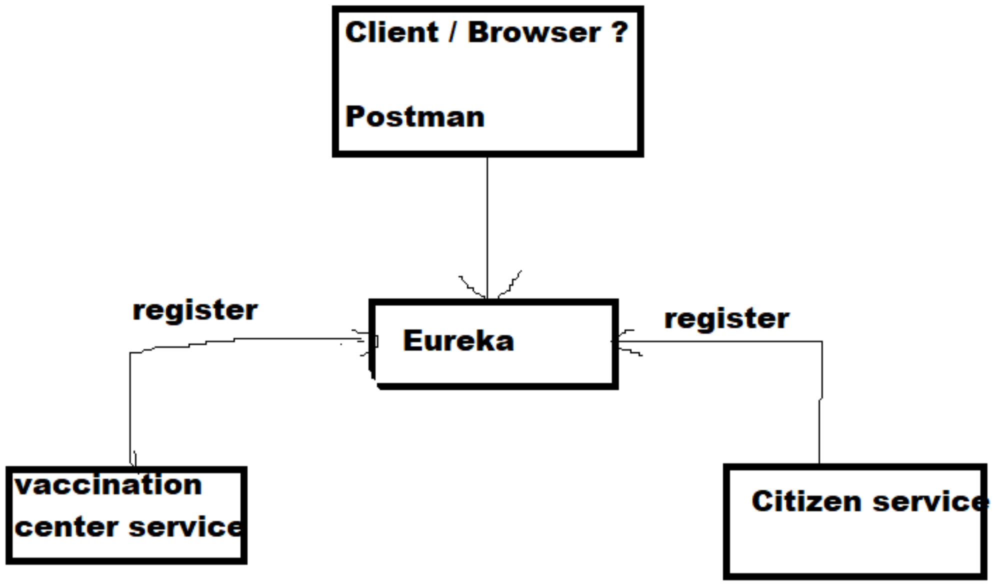
2. Once you have identified them, you need to create standalone projects we will take spring boot to create these microservices.

3. You need them to interact with each other through some ways. It can be REST API or messaging. We are going to use restful architecture for the same.

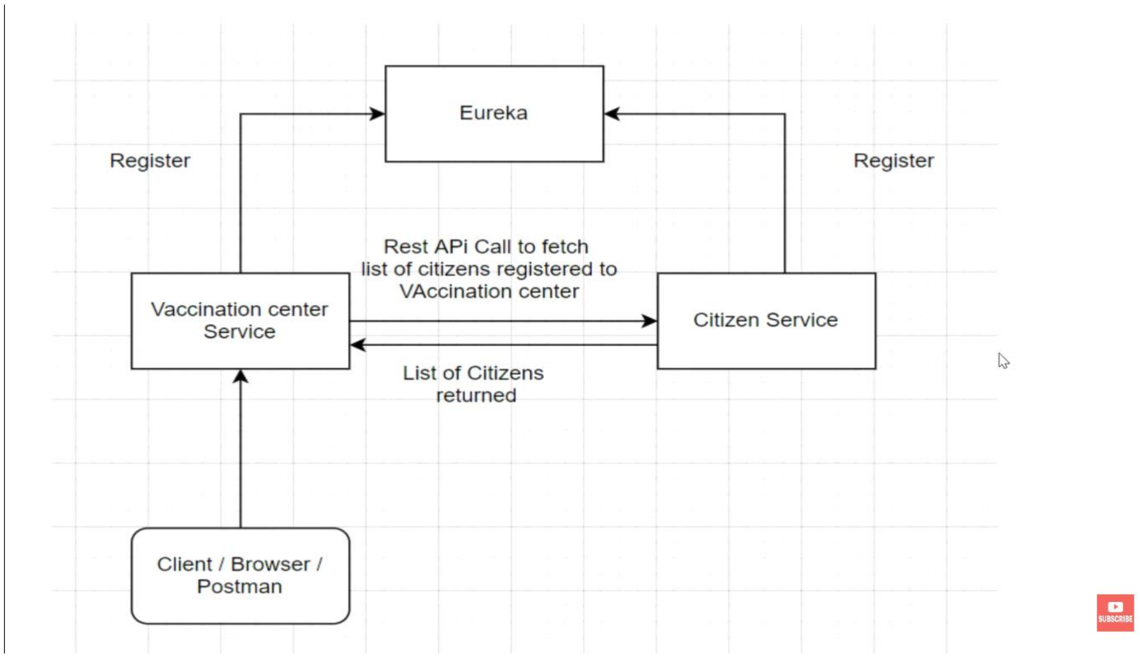
4. But just doing this does not make sure that you have implemented microservice architecture.

You need load balancer, eureka for service discovery(useful during load balancing & cloud deployment), API gateways & many more stuff.

Here we are taking example of vaccination portal in which we will implement to services that are VaccinationCenter service & citizen service.



We will use one database per microservice.


Eureka
Register
Register
Rest APi Call to fetch list of citizens registered to VAccination center
Vaccination center Service
Citizen Service
List of Citizens returned
Client / Browser / Postman

Eureka Sever:

Eureka Server is an application that holds the information about all client-service applications.

Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

Spring Cloud Netflix – Eureka:

client-side service discovery via "*Spring Cloud Netflix Eureka.*"

Client-side service discovery **allows services to find and communicate with each other without hard-coding the hostname and port.** The only 'fixed point' in such an architecture is the *service registry,* with which each service has to register.

One drawback is that all clients must implement a certain logic to interact with this fixed point. This assumes an additional network round trip before the actual request.

With Netflix Eureka, each client can simultaneously act as a server to replicate its status to a connected peer. In other words, a client retrieves a list of all connected peers in a *service registry,* and makes all further requests to other services through a load-balancing algorithm.

To be informed about the presence of a client, they have to send a heartbeat signal to the registry.

We'll implement three micro-services:

- a service registry (Eureka Server)
- a REST service, which registers itself at the registry (Eureka Client)
- a web application, which is consuming the REST service as a registry-aware client (Spring Cloud Netflix Feign Client)

Building a Eureka Server

Eureka Server comes with the bundle of Spring Cloud. For this, we need to develop the Eureka server and run it on the default port 8761.

After downloading the project in main Spring Boot Application class file, we need to add @EnableEurekaServer annotation. The @EnableEurekaServer annotation is used to make your Spring Boot application acts as a Eureka Server.

Implementing a Eureka Server for service registry is as easy as:

1. adding spring-cloud-starter-netflix-eureka-server to the dependencies

2. enabling the Eureka Server in a @SpringBootApplication by annotating it with @EnableEurekaServer

3. configuring some properties

We're telling the built-in *Eureka Client* not to register with itself because our application should be acting as a server.

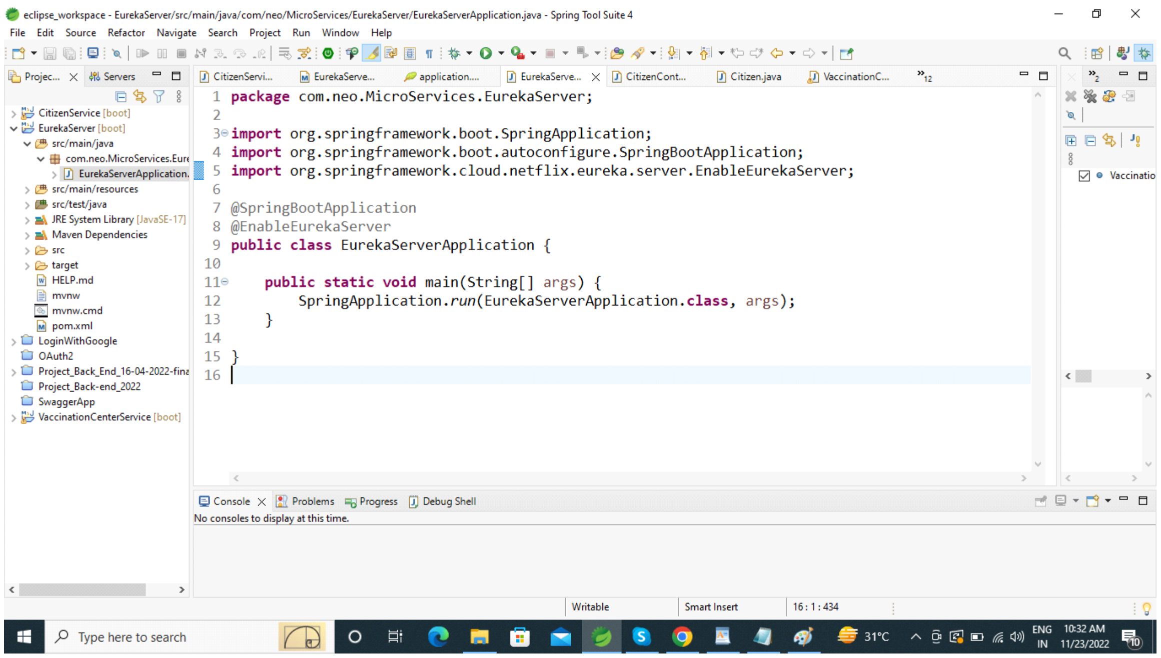
Eureka Client:

For a *@SpringBootApplication* to be discovery-aware, we have to include a *Spring Discovery Client* (for example, *spring-cloud-starter-netflix-eureka-client*) into our *classpath.*

Then we need to annotate a *@Configuration* with either *@EnableDiscoveryClient* or *@EnableEurekaClient.* Note that this annotation is optional if we have the *spring-cloud-starter-netflix-eureka-client* dependency on the classpath.

The latter tells *Spring Boot* to use Spring Netflix Eureka for service discovery explicitly. To fill our client application with some sample-life, we'll also include the *spring-boot-starter-web* package in the *pom.xml* and implement a *REST* controller.

The code for main Spring Boot application class file is as shown below −

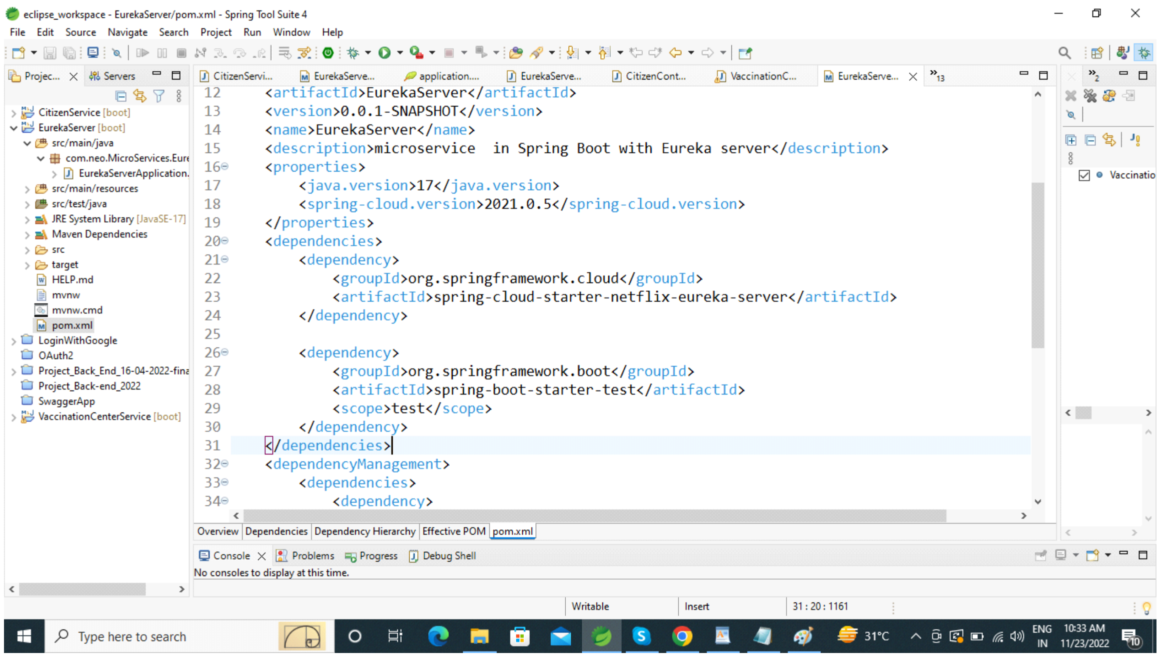


```
package com.neo.MicroServices.EurekaServer;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }

}
```

No consoles to display at this time.

Make sure Spring cloud Eureka server dependency is added in your build configuration file.

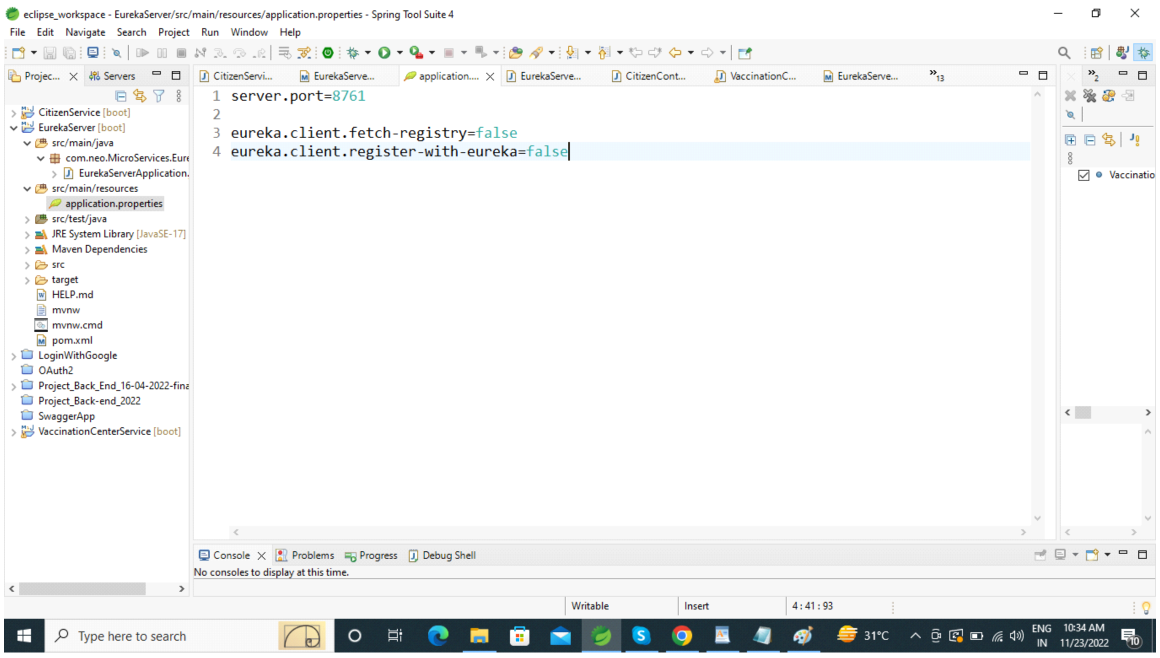
The code for Maven user dependency is shown below –


eclipse_workspace - EurekaServer/pom.xml - Spring Tool Suite 4
File Edit Source Navigate Search Project Run Window Help
CitizenService [boot]
EurekaServer [boot]
src/main/java
com.neo.MicroServices.Eure
EurekaServerApplication.
src/main/resources
src/test/java
JRE System Library [JavaSE-17]
Maven Dependencies
src
target
HELP.md
mvnw
mvnw.cmd
pom.xml
LoginWithGoogle
OAuth2
Project_Back_End_16-04-2022-fina
Project_Back-end_2022
SwaggerApp
VaccinationCenterService [boot]
Overview Dependencies Dependency Hierarchy Effective POM pom.xml
Console Problems Progress Debug Shell
No consoles to display at this time.
Writable Insert 31 : 20 : 1161
Type here to search
31°C
10:33 AM
11/23/2022

```xml
    <artifactId>EurekaServer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>EurekaServer</name>
    <description>microservice  in Spring Boot with Eureka server</description>
    <properties>
        <java.version>17</java.version>
        <spring-cloud.version>2021.0.5</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
```

By default, the Eureka Server registers itself into the discovery. You should add the below given configuration into your application.properties file or application.yml file.

application.properties file is given below –



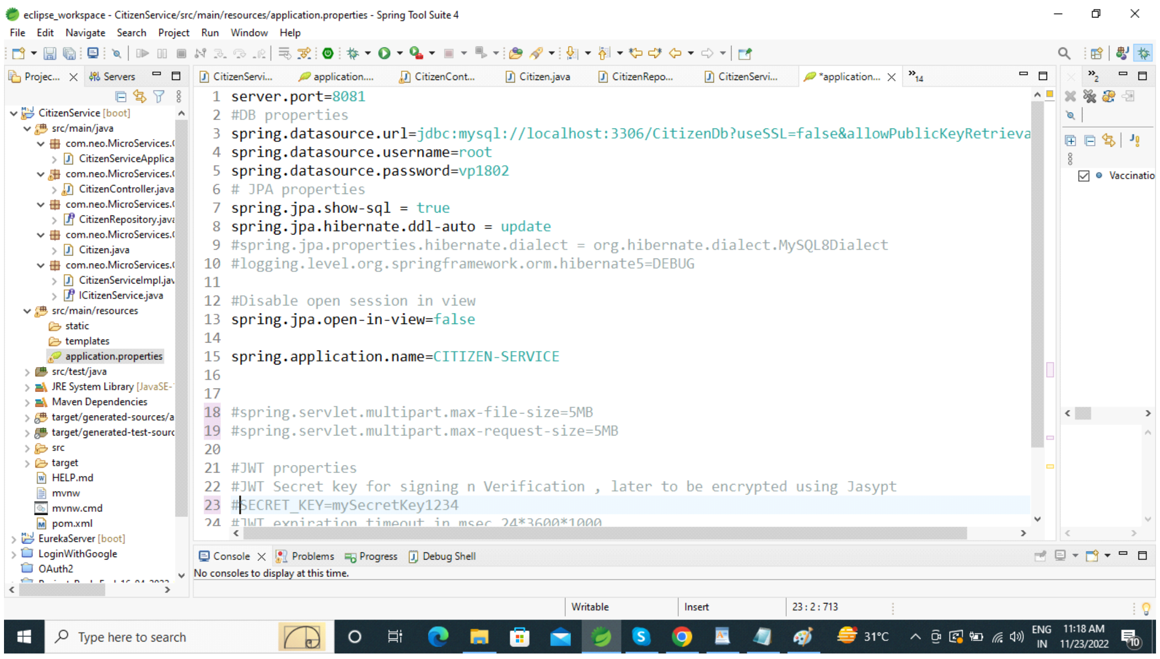
```
server.port=8761

eureka.client.fetch-registry=false
eureka.client.register-with-eureka=false
```

Let's create now Citizen Service-

We will define different port for each microsrvice.

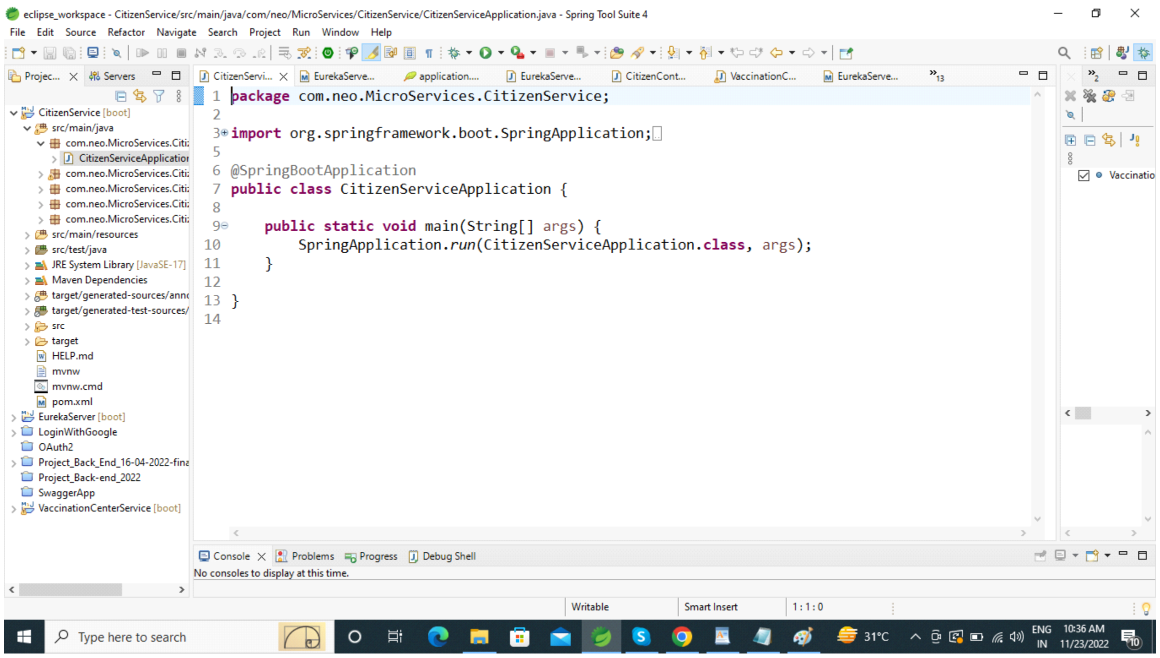
Citizen service will run on port 8081

application.properties file



```
server.port=8081
#DB properties
spring.datasource.url=jdbc:mysql://localhost:3306/CitizenDb?useSSL=false&allowPublicKeyRetrieva
spring.datasource.username=root
spring.datasource.password=vp1802
# JPA properties
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
#spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
#logging.level.org.springframework.orm.hibernate5=DEBUG

#Disable open session in view
spring.jpa.open-in-view=false

spring.application.name=CITIZEN-SERVICE


#spring.servlet.multipart.max-file-size=5MB
#spring.servlet.multipart.max-request-size=5MB

#JWT properties
#JWT Secret key for signing n Verification , later to be encrypted using Jasypt
#SECRET_KEY=mySecretKey1234
#JWT expiration timeout in msec 24*3600*1000
```

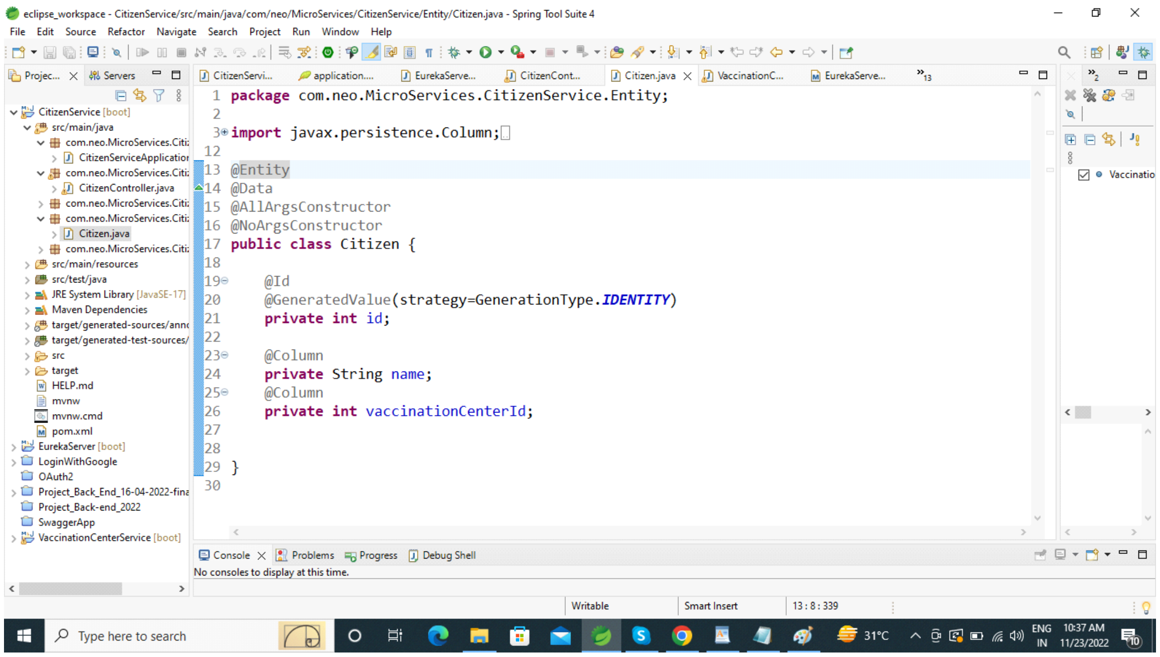
1.Citizen Service Application file


eclipse_workspace - CitizenService/src/main/java/com/neo/MicroServices/CitizenService/CitizenServiceApplication.java - Spring Tool Suite 4

```java
package com.neo.MicroServices.CitizenService;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class CitizenServiceApplication {

	public static void main(String[] args) {
		SpringApplication.run(CitizenServiceApplication.class, args);
	}

}
```

No consoles to display at this time.

2.Create Entity of Citizen


eclipse_workspace - CitizenService/src/main/java/com/neo/MicroServices/CitizenService/Entity/Citizen.java - Spring Tool Suite 4

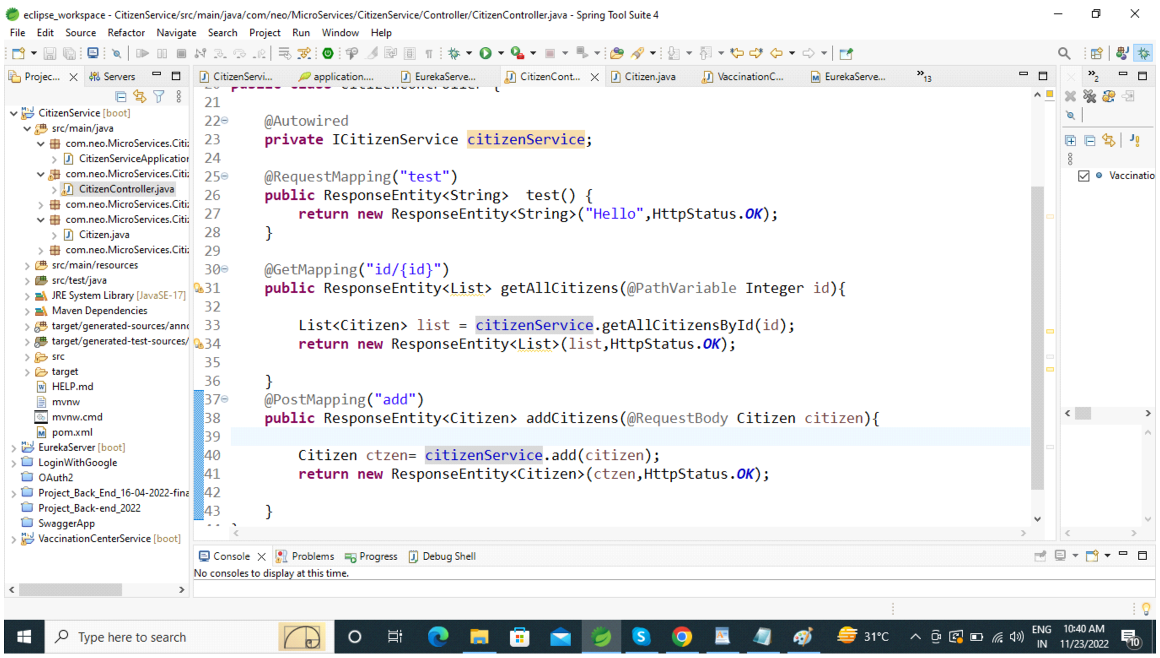
```java
package com.neo.MicroServices.CitizenService.Entity;

import javax.persistence.Column;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Citizen {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private int id;

	@Column
	private String name;
	@Column
	private int vaccinationCenterId;


}
```

No consoles to display at this time.

3. Create Controller class to define APIs .

a. Get the list of citizens

b.add Citizen.



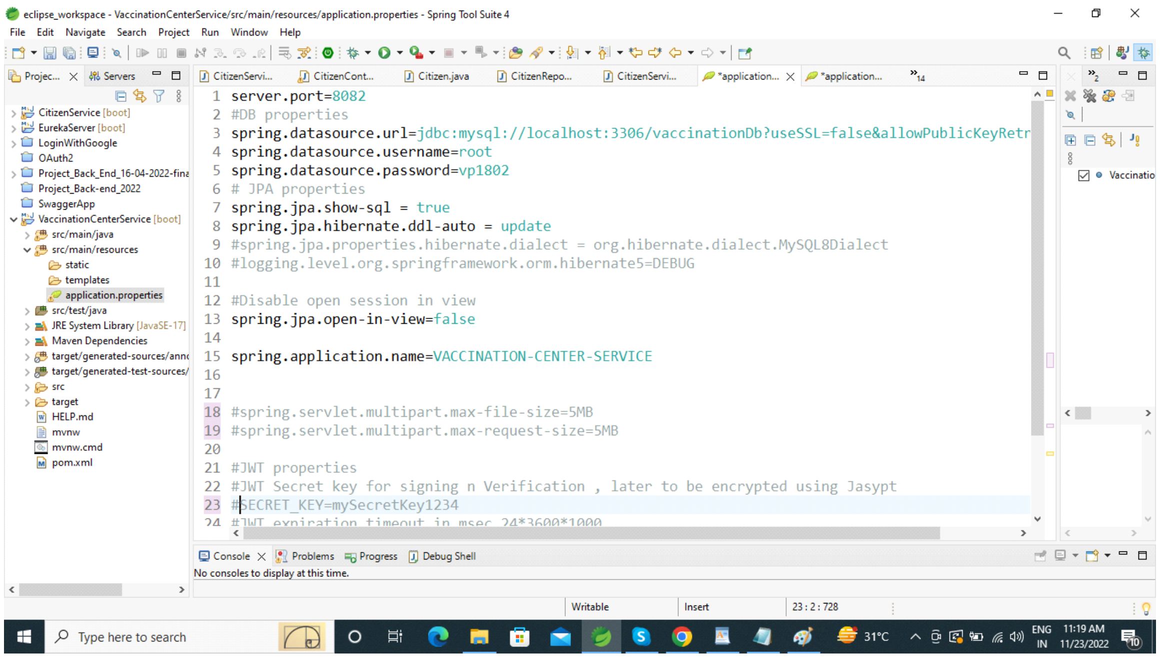
```java
    @Autowired
    private ICitizenService citizenService;

    @RequestMapping("test")
    public ResponseEntity<String>  test() {
        return new ResponseEntity<String>("Hello",HttpStatus.OK);
    }

    @GetMapping("id/{id}")
    public ResponseEntity<List> getAllCitizens(@PathVariable Integer id){

        List<Citizen> list = citizenService.getAllCitizensById(id);
        return new ResponseEntity<List>(list,HttpStatus.OK);

    }
    @PostMapping("add")
    public ResponseEntity<Citizen> addCitizens(@RequestBody Citizen citizen){

        Citizen ctzen= citizenService.add(citizen);
        return new ResponseEntity<Citizen>(ctzen,HttpStatus.OK);

    }
```

To save & fetch details ,create repository intefaces. & service layers to have the business logic.

Let's create now VaccinationCenter Service-

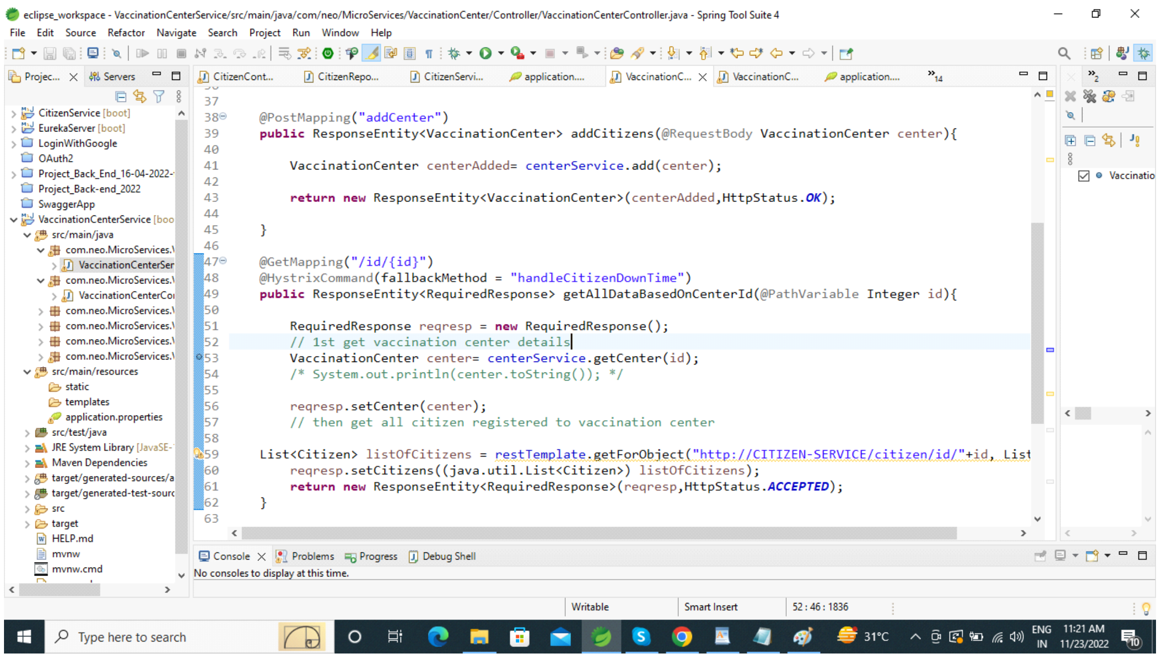
Citizen service will run on port 8082

application.properties file



```
server.port=8082
#DB properties
spring.datasource.url=jdbc:mysql://localhost:3306/vaccinationDb?useSSL=false&allowPublicKeyRetr
spring.datasource.username=root
spring.datasource.password=vp1802
# JPA properties
spring.jpa.show-sql = true
spring.jpa.hibernate.ddl-auto = update
#spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
#logging.level.org.springframework.orm.hibernate5=DEBUG

#Disable open session in view
spring.jpa.open-in-view=false

spring.application.name=VACCINATION-CENTER-SERVICE


#spring.servlet.multipart.max-file-size=5MB
#spring.servlet.multipart.max-request-size=5MB

#JWT properties
#JWT Secret key for signing n Verification , later to be encrypted using Jasypt
#SECRET_KEY=mySecretKey1234
```

1.VaccinationCenter controller file:



```
    @PostMapping("addCenter")
    public ResponseEntity<VaccinationCenter> addCitizens(@RequestBody VaccinationCenter center){

        VaccinationCenter centerAdded= centerService.add(center);

        return new ResponseEntity<VaccinationCenter>(centerAdded,HttpStatus.OK);

    }

    @GetMapping("/id/{id}")
    @HystrixCommand(fallbackMethod = "handleCitizenDownTime")
    public ResponseEntity<RequiredResponse> getAllDataBasedOnCenterId(@PathVariable Integer id){

        RequiredResponse reqresp = new RequiredResponse();
        // 1st get vaccination center details
        VaccinationCenter center= centerService.getCenter(id);
        /* System.out.println(center.toString()); */

        reqresp.setCenter(center);
        // then get all citizen registered to vaccination center

    List<Citizen> listOfCitizens = restTemplate.getForObject("http://CITIZEN-SERVICE/citizen/id/"+id, List
        reqresp.setCitizens((java.util.List<Citizen>) listOfCitizens);
        return new ResponseEntity<RequiredResponse>(reqresp,HttpStatus.ACCEPTED);
    }
```

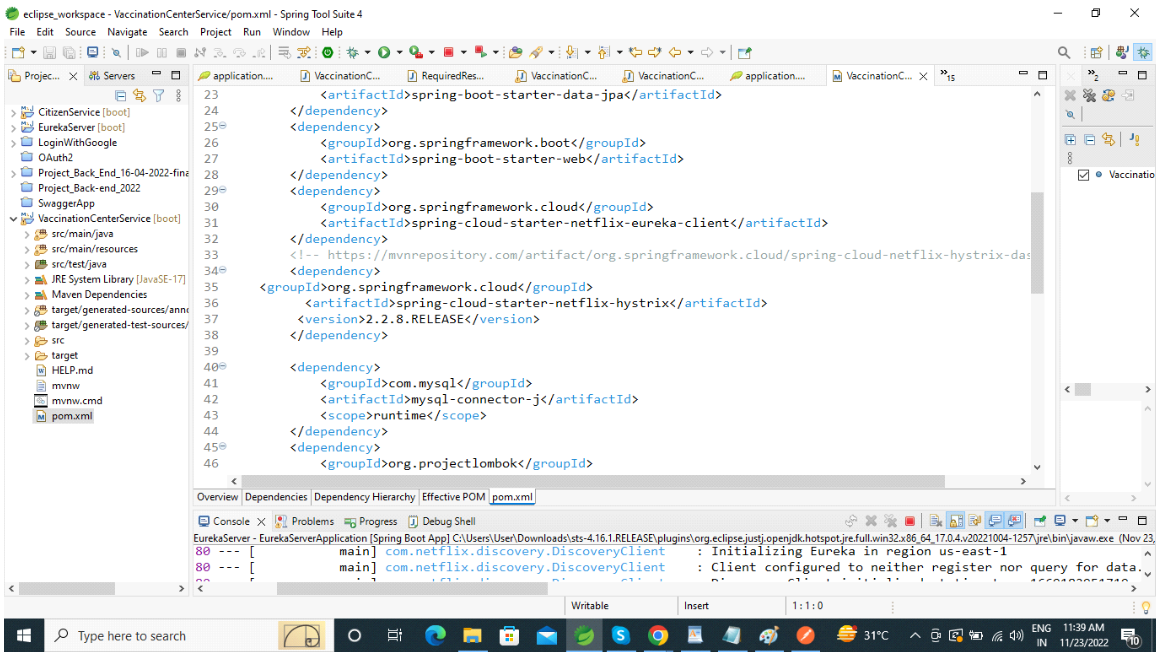
As we know in microservice architecture we have many independent application and some time it is needed that one application communicates with other application to do some operation. So if we want to communicate between two microservice then spring cloud provide us two useful Component.

1.RestTemplate

2.Feign Client

In This Project, we will discuss how two microservice communicate between them using RestTemplate. Make Sure you have your Eureka Server Running and both the application is registered on that. To use RestTemplate in our application we have to follow below steps:

1.Add required dependency in your pom.xml file.



```
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-netflix-hystrix-das
        <dependency>
    <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
         <version>2.2.8.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
```

```
EurekaServer - EurekaServerApplication [Spring Boot App] C:\Users\User\Downloads\sts-4.16.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20221004-1257\jre\bin\javaw.exe (Nov 23,
80 --- [           main] com.netflix.discovery.DiscoveryClient    : Initializing Eureka in region us-east-1
80 --- [           main] com.netflix.discovery.DiscoveryClient    : Client configured to neither register nor query for data.
```

2.Create Two MicroService project. In this we create two project one is MicroService-1(CitizenService) and another is MicroService-2(VaccinationCenter Service).

3.Give ApplicationName , Port and Eureka server path in your application.properties file.

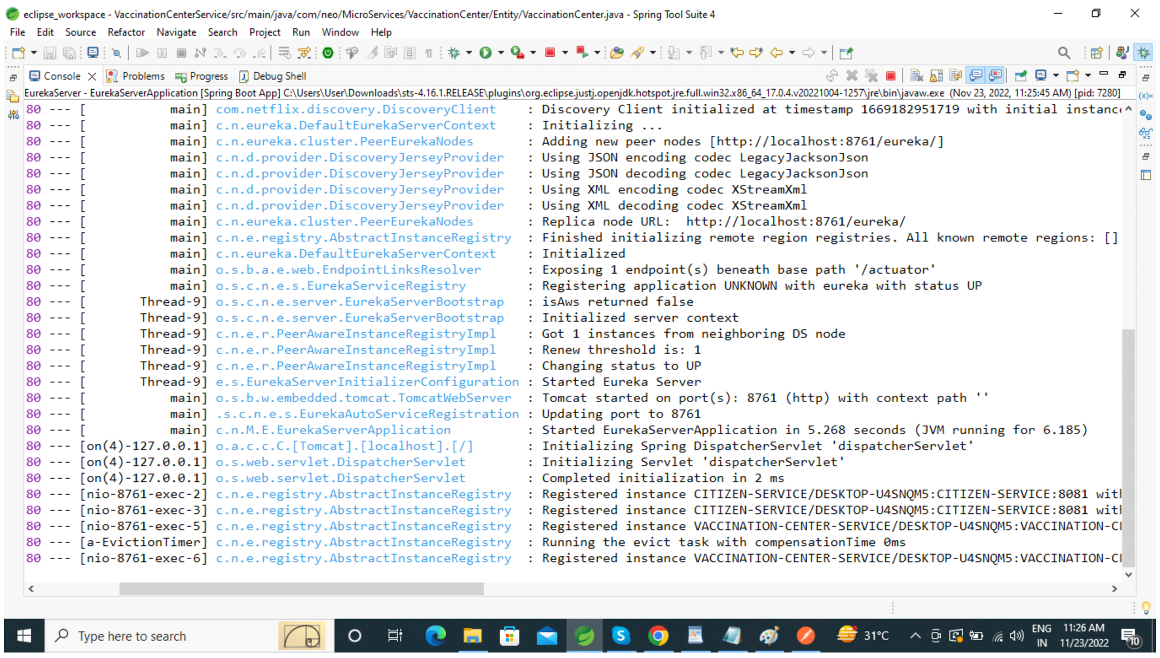
4.Create a Bean for RestTemplate because we can't auto-wired RestTemlate directly. So we create a configuration class and initialize a Bean for RestTemplate.

5.Autowired RestTemplate in your Service class to call Rest End Point of another application.

For Calling a Get Request of MicroService-2 we use below syntax:

List<Citizen> response = restTemplate.getForObject("http://localhost:8081/citizen/id"+id,List.class);

To fetch the combine result of list of citizens & vaccination center Id..we will define a POJO to get it.

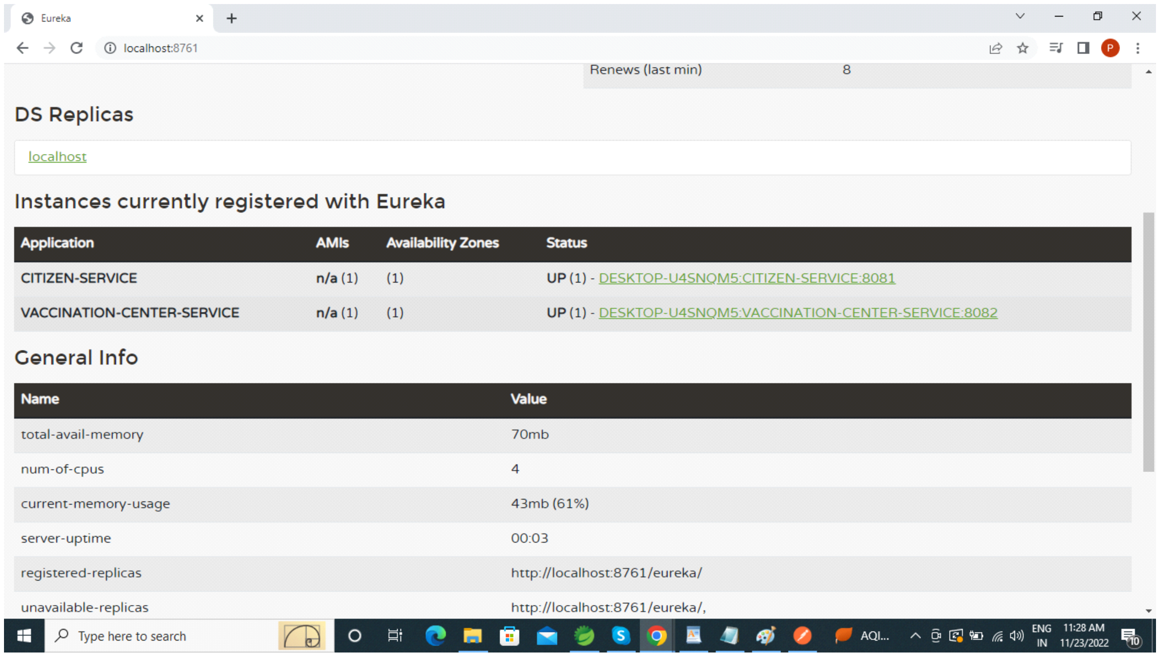


eclipse_workspace - VaccinationCenterService/src/main/java/com/neo/MicroServices/VaccinationCenter/Entity/VaccinationCenter.java - Spring Tool Suite 4

EurekaServer - EurekaServerApplication [Spring Boot App] C:\Users\User\Downloads\sts-4.16.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20221004-1257\jre\bin\javaw.exe (Nov 23, 2022, 11:25:45 AM) [pid: 7280]

```
80 --- [           main] com.netflix.discovery.DiscoveryClient    : Discovery Client initialized at timestamp 1669182951719 with initial instanc
80 --- [           main] c.n.eureka.DefaultEurekaServerContext    : Initializing ...
80 --- [           main] c.n.eureka.cluster.PeerEurekaNodes       : Adding new peer nodes [http://localhost:8761/eureka/]
80 --- [           main] c.n.d.provider.DiscoveryJerseyProvider   : Using JSON encoding codec LegacyJacksonJson
80 --- [           main] c.n.d.provider.DiscoveryJerseyProvider   : Using JSON decoding codec LegacyJacksonJson
80 --- [           main] c.n.d.provider.DiscoveryJerseyProvider   : Using XML encoding codec XStreamXml
80 --- [           main] c.n.d.provider.DiscoveryJerseyProvider   : Using XML decoding codec XStreamXml
80 --- [           main] c.n.eureka.cluster.PeerEurekaNodes       : Replica node URL:  http://localhost:8761/eureka/
80 --- [           main] c.n.e.registry.AbstractInstanceRegistry  : Finished initializing remote region registries. All known remote regions: []
80 --- [           main] c.n.eureka.DefaultEurekaServerContext    : Initialized
80 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 1 endpoint(s) beneath base path '/actuator'
80 --- [           main] o.s.c.n.e.s.EurekaServiceRegistry        : Registering application UNKNOWN with eureka with status UP
80 --- [       Thread-9] o.s.c.n.e.server.EurekaServerBootstrap   : isAws returned false
80 --- [       Thread-9] o.s.c.n.e.server.EurekaServerBootstrap   : Initialized server context
80 --- [       Thread-9] c.n.e.r.PeerAwareInstanceRegistryImpl    : Got 1 instances from neighboring DS node
80 --- [       Thread-9] c.n.e.r.PeerAwareInstanceRegistryImpl    : Renew threshold is: 1
80 --- [       Thread-9] c.n.e.r.PeerAwareInstanceRegistryImpl    : Changing status to UP
80 --- [       Thread-9] e.s.EurekaServerInitializerConfiguration : Started Eureka Server
80 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8761 (http) with context path ''
80 --- [           main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8761
80 --- [           main] c.n.M.E.EurekaServerApplication          : Started EurekaServerApplication in 5.268 seconds (JVM running for 6.185)
80 --- [on(4)-127.0.0.1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
80 --- [on(4)-127.0.0.1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
80 --- [on(4)-127.0.0.1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 2 ms
80 --- [nio-8761-exec-2] c.n.e.registry.AbstractInstanceRegistry  : Registered instance CITIZEN-SERVICE/DESKTOP-U4SNQM5:CITIZEN-SERVICE:8081 wit
80 --- [nio-8761-exec-3] c.n.e.registry.AbstractInstanceRegistry  : Registered instance CITIZEN-SERVICE/DESKTOP-U4SNQM5:CITIZEN-SERVICE:8081 wit
80 --- [nio-8761-exec-5] c.n.e.registry.AbstractInstanceRegistry  : Registered instance VACCINATION-CENTER-SERVICE/DESKTOP-U4SNQM5:VACCINATION-C
80 --- [a-EvictionTimer] c.n.e.registry.AbstractInstanceRegistry  : Running the evict task with compensationTime 0ms
80 --- [nio-8761-exec-6] c.n.e.registry.AbstractInstanceRegistry  : Registered instance VACCINATION-CENTER-SERVICE/DESKTOP-U4SNQM5:VACCINATION-C
```

Eureka server :



localhost:8761

Renews (last min) 8

DS Replicas

localhost

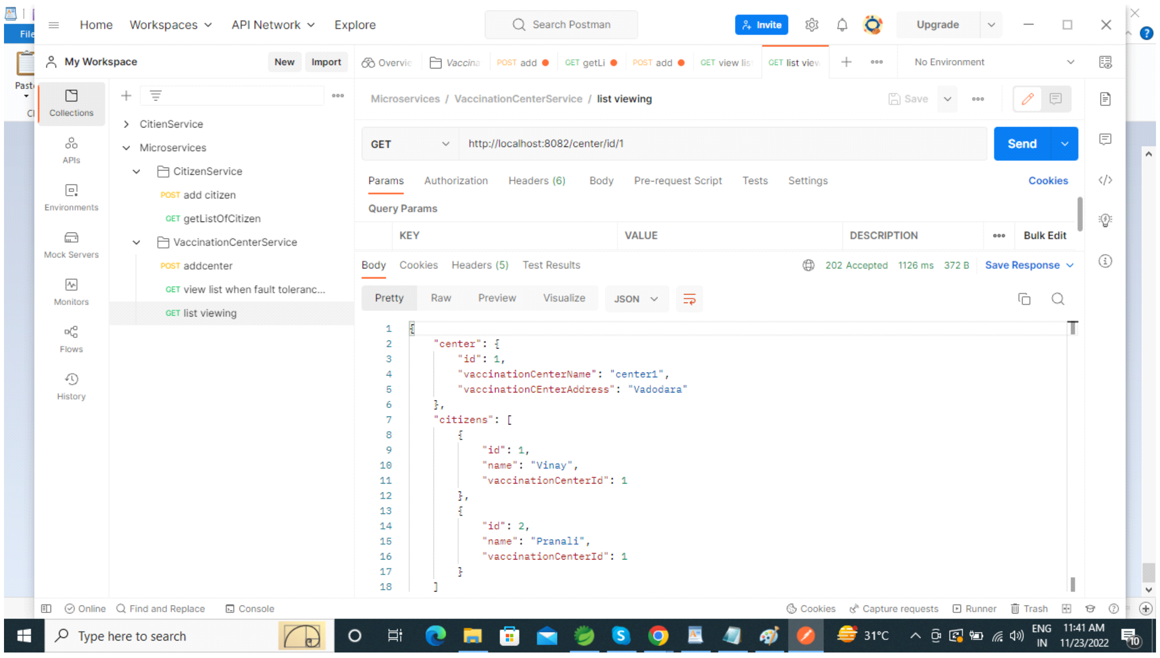
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CITIZEN-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-U4SNQM5:CITIZEN-SERVICE:8081
VACCINATION-CENTER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-U4SNQM5:VACCINATION-CENTER-SERVICE:8082

General Info

Name	Value
total-avail-memory	70mb
num-of-cpus	4
current-memory-usage	43mb (61%)
server-uptime	00:03
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,

postman result:



GET http://localhost:8082/center/id/1

```
{
    "center": {
        "id": 1,
        "vaccinationCenterName": "center1",
        "vaccinationCEnterAddress": "Vadodara"
    },
    "citizens": [
        {
            "id": 1,
            "name": "Vinay",
            "vaccinationCenterId": 1
        },
        {
            "id": 2,
            "name": "Pranali",
            "vaccinationCenterId": 1
        }
    ]
```

Here we are getting combine result of vaccination center details & citizens who have registered to that center just by hitting one query.

Now what if one of the service is down, it will give you error. To overcome this,we implemented fault tolerance in microservices.

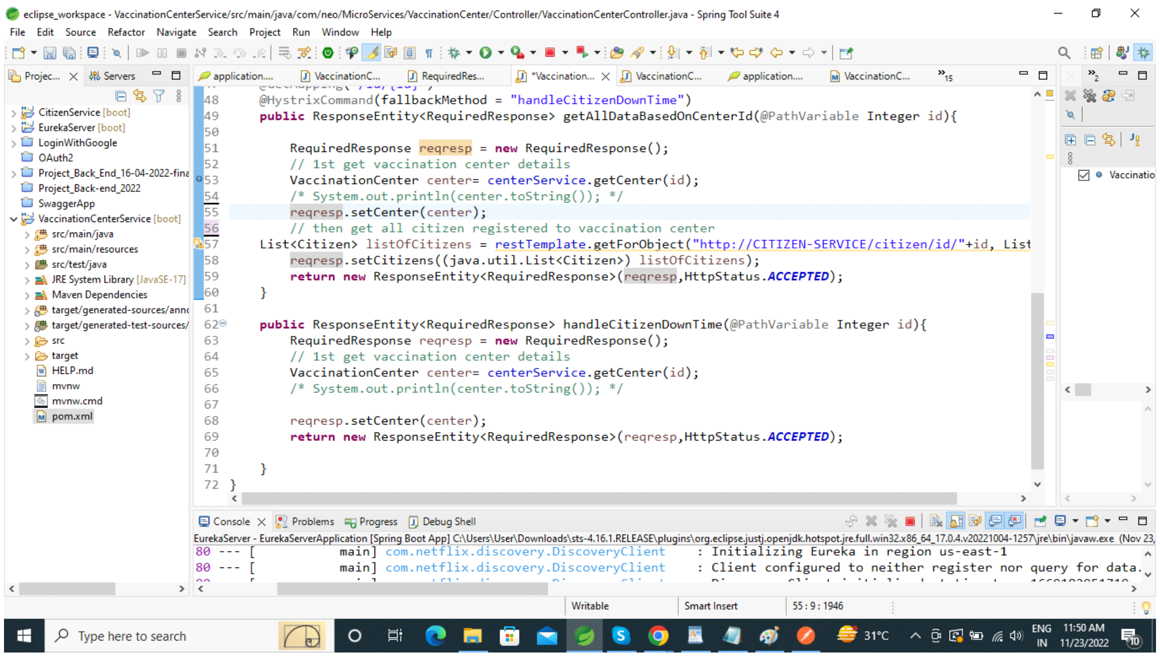
Fault Tolerance in Microservices:

1. If one of the services is down, fault arises.

2. for eg. if citizen service is down, it will show error for other service .

3. Fault Tolerance is the property that enables a system to continue operating properly in the event of failure of some of it's components.

4. We need to configure a circuit breaker design pattern which says that whenever particular microservice is down, stop calling that service & rather call a fallback method that will show how to configure.



eclipse_workspace - VaccinationCenterService/src/main/java/com/neo/MicroServices/VaccinationCenter/Controller/VaccinationCenterController.java - Spring Tool Suite 4

```java
    @HystrixCommand(fallbackMethod = "handleCitizenDownTime")
    public ResponseEntity<RequiredResponse> getAllDataBasedOnCenterId(@PathVariable Integer id){

        RequiredResponse reqresp = new RequiredResponse();
        // 1st get vaccination center details
        VaccinationCenter center= centerService.getCenter(id);
        /* System.out.println(center.toString()); */
        reqresp.setCenter(center);
        // then get all citizen registered to vaccination center
    List<Citizen> listOfCitizens = restTemplate.getForObject("http://CITIZEN-SERVICE/citizen/id/"+id, List
        reqresp.setCitizens((java.util.List<Citizen>) listOfCitizens);
        return new ResponseEntity<RequiredResponse>(reqresp,HttpStatus.ACCEPTED);
    }

    public ResponseEntity<RequiredResponse> handleCitizenDownTime(@PathVariable Integer id){
        RequiredResponse reqresp = new RequiredResponse();
        // 1st get vaccination center details
        VaccinationCenter center= centerService.getCenter(id);
        /* System.out.println(center.toString()); */

        reqresp.setCenter(center);
        return new ResponseEntity<RequiredResponse>(reqresp,HttpStatus.ACCEPTED);

    }
}
```

```
EurekaServer - EurekaServerApplication [Spring Boot App] C:\Users\User\Downloads\sts-4.16.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20221004-1257\jre\bin\javaw.exe (Nov 23,
80 --- [           main] com.netflix.discovery.DiscoveryClient    : Initializing Eureka in region us-east-1
80 --- [           main] com.netflix.discovery.DiscoveryClient    : Client configured to neither register nor query for data.
```

Hystrix Example for real impatient

Hystrix configuration is done in four major steps.

1.Add Hystrix starter and dependencies.

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
</dependency>
```

2.Add @EnableCircuitBreaker annotation

3.Add @EnableHystrixDashboard annotation

4.Add annotation @HystrixCommand(fallbackMethod = "myFallbackMethod")

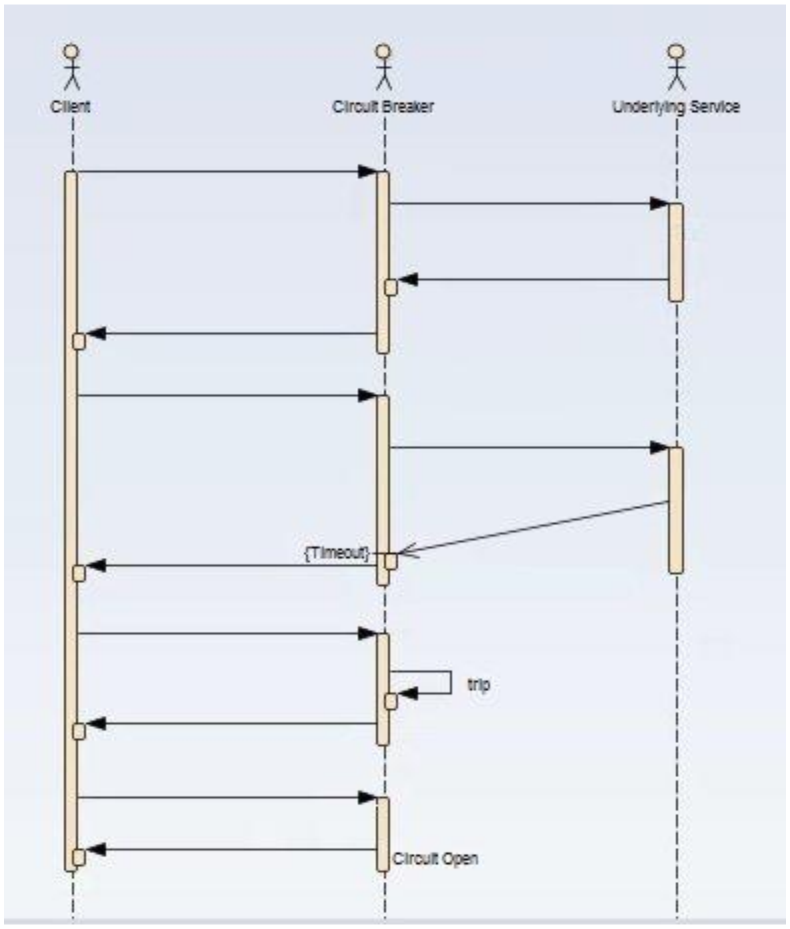
What is Circuit Breaker Pattern?

If we design our systems on microservice based architecture, we will generally develop many Microservices and those will interact with each other heavily in achieving certain business goals. Now, all of us can assume that this will give expected result if all the services are up and running and response time of each service is satisfactory.

Now what will happen if any service, of the current Eco system, has some issue and stopped servicing the requests. It will result in timeouts/exception and the whole Eco system will get unstable due to this single point of failure.

Here circuit breaker pattern comes handy and it redirects traffic to a fall back path once it sees any such scenario. Also it monitors the defective service closely and restore the traffic once the service came back to normalcy.

So circuit breaker is a kind of a wrapper of the method which is doing the service call and it monitors the service health and once it gets some issue, the circuit breaker trips and all further calls goto the circuit breaker fall back and finally restores automatically once the service came back .



Hystrix Circuit Breaker Example

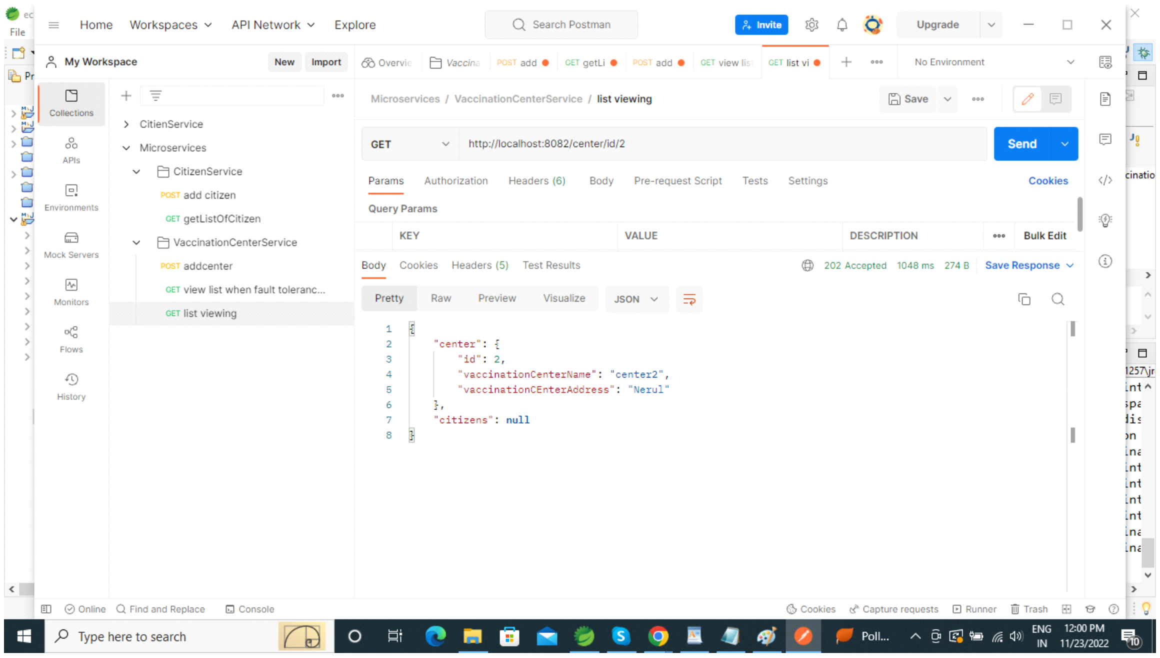
To demo circuit breaker, we will create following two microservices where first is dependent on another.

Citizen Microservice – Which will give some basic functionality on Citizen entity. It will be a REST based service. We will call this service from VaccinationCenter Service to understand Circuit Breaker. It will run on port 8081 in localhost.

VaccinationCenter Microservice – Again a simple REST based microservice where we will implement circuit breaker using Hystrix. Citizen Service will be invoked from here and we will test the fall back path once Citizen service will be unavailable. It will run on port 8082 in localhost.

Now we already know that VaccinationCenter service is calling Citizen service internally, and it is getting Citizen details from that service. So if both the services are running, VaccinationCenter service is displaying the data returned by Citizen service as we have seen in the VaccinationCenter service browser output above. This is CIRCUIT CLOSED State.

Now let us stop the Citizen service by just pressing CTRL + C in the Citizen service server console (stop the server) and test the VaccinationCenter service again from browser. This time it will return the fall back method response. Here Hystrix comes into picture, it monitors Citizen service in frequent interval and as it is down, Hystrix component has opened the Circuit and fallback path enabled.



```
GET http://localhost:8082/center/id/2
```

```
202 Accepted   1048 ms   274 B
```

```
{
    "center": {
        "id": 2,
        "vaccinationCenterName": "center2",
        "vaccinationCenterAddress": "Nerul"
    },
    "citizens": null
}
```

Here in above image, we can see as Citizen service is down we will get the result of vaccination center details only.

Microservice API Gateway Implementation:

What is API Gateway?

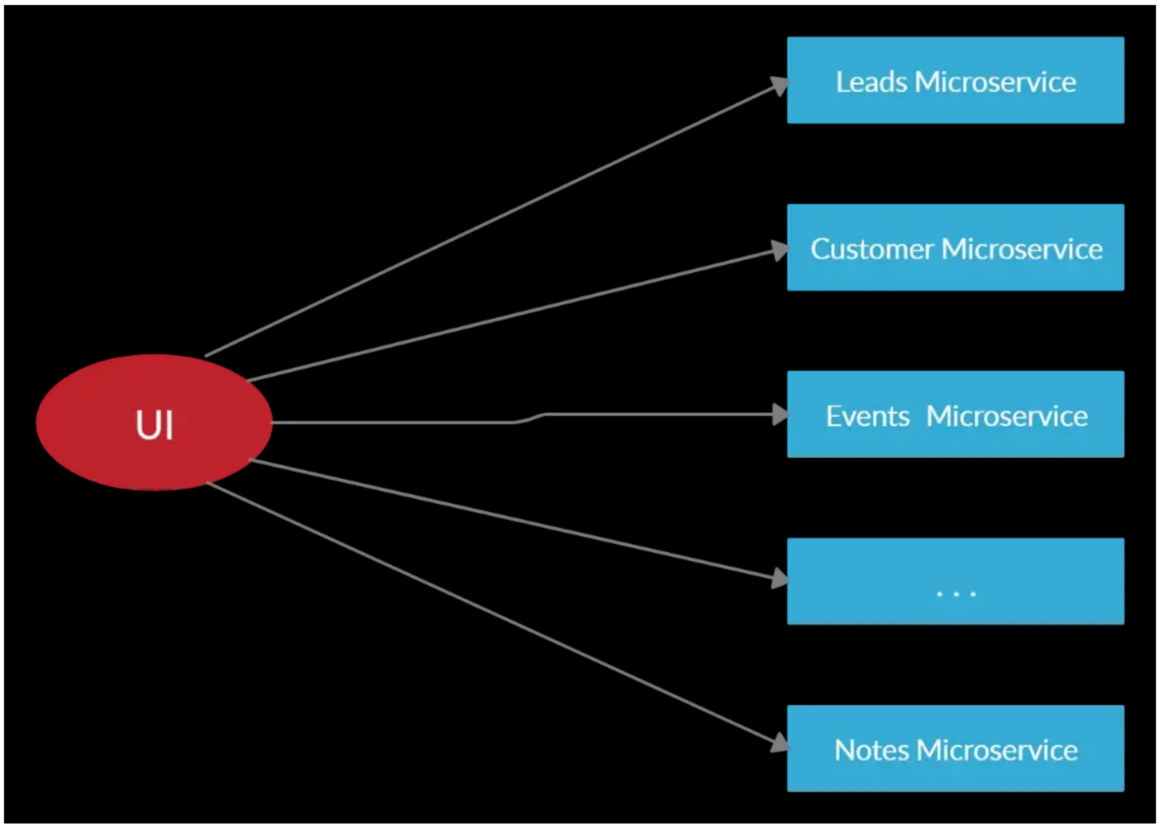
In plain terms, API Gateway is enhanced reverse proxy with more advanced capabilities including orchestration and added security and monitoring capabilities. Netflix Zuul, Amazon API Gateway, Apigee and of course Spring Cloud Gateway are few of well known api gateway implementation.

Why Use an API Gateway

The essence of microservice architecture is decomposing monoliths into fine grained services.

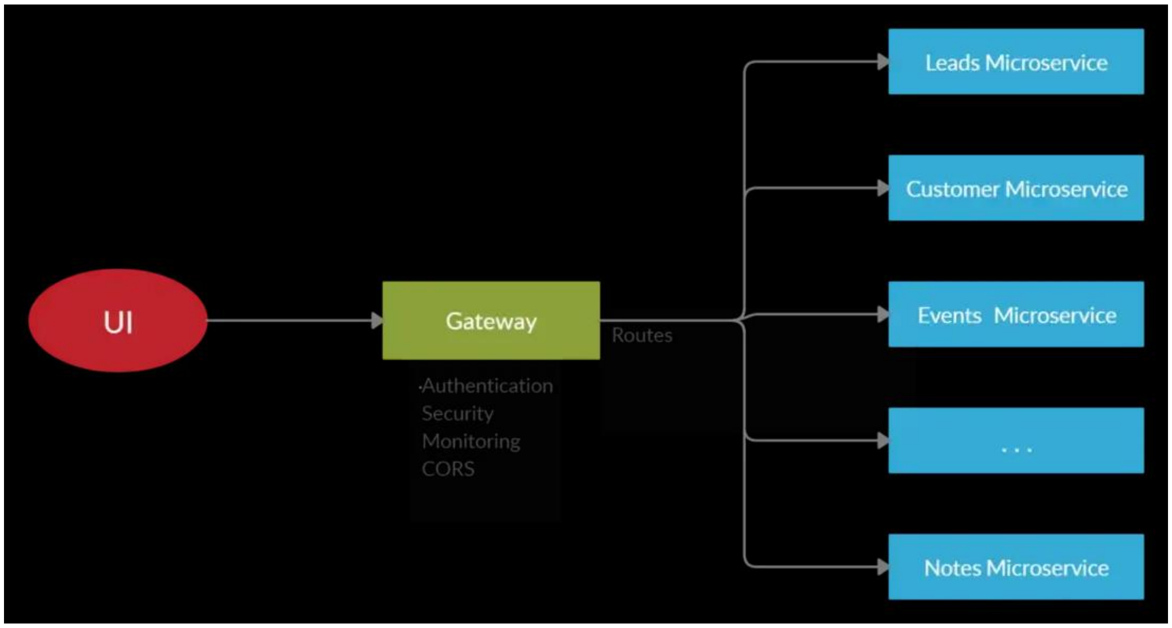
Each of these “micro” services can be deployed and scaled independently. While this pattern has many advantages, it also brings numerous challenges. One of the foremost of these challenges is the increased complexity which in turn leads to a lack of maintainability.

Imagine a scenario in which you have to implement a UI for a complex CRM (Customer Relationship Management) application. In order to populate different portions of this UI and perform different user operations, you may have to invoke dozens of microservices. Also, UI has to be aware of the network locations (host and port) of all these microservices. In addition, if the network location of a microservice changes (which tends to happen quite frequently in cloud), UI will also have to be updated. To state the obvious, it’s a very bad design.



In addition, a typical web application should have the support for monitoring, authentication, security, CORS and etc. If the UI is directly invoking the microservices, the aforementioned cross cutting concerns should be implemented in each and every microservice separately. A simple change in CORS policy or authentication mechanism would force changes in all the microservices, which is inefficient and error prone.

In order to have a more robust approach, we have to implement a single point of entry or a gateway for all the incoming traffic to microservices. UI will always send requests to the gateway and in turn the gateway will forward the requests to relevant microservices. Essentially, gateway acts as a middle-ware between the UI and the microservices. The following figure illustrates this concept.



The two prime advantages of this approach are as follows.

1.UI doesn’t need to be aware of the network locations of individual microservices. Instead, UI only needs to know the network location of the gateway. Gateway will route the incoming requests to relevant backend services.

2.Cross cutting concerns such as authentication, security, monitoring, CORS and etc will be handled by the gateway. Whenever a change is required in any of these aspects, that change can be made in a single place (gateway) to affect all the microservices.

Why not ZUUL? Why Spring Cloud Gateway?

1. Zuul is a blocking API. A blocking gateway API makes use of many threads as the no. of incoming

requests. So this approach is more resource intensive. If no thread are available to process incoming requests then the request has to wait in queue.

2. Spring cloud gateway is a non-blocking API. When using non-blocking API, a thread is always available to process the incoming request. These requests are then processed asynchronously in the background & once completed the response is returned. So no incoming request never gets blocked when using spring cloud gateway.

How it works?

Client make requests to spring cloud gateway. If the gateway Handler mapping determines that a request matches a route, it is sent to the Gateway Web Handler. This Handler runs the request through a filter chain that is specific to the request.

The reason the filters are divided by the dotted line is the filters can run logic both before & after the proxy request is sent. All "pre" filter logic is executed. Then the proxy request is made, the "post" filter logic is run.

URIs defined in between routes without a port get default port values of 80 & 443 for the HTTP & HTTPs URIs respectively.

Spring Cloud Gateway features:

Built on Spring Framework 5, Project Reactor and Spring Boot 2.0

Able to match routes on any request attribute.

Predicates and filters are specific to routes.

Circuit Breaker integration.

Spring Cloud DiscoveryClient integration

Easy to write Predicates and Filters

Request Rate Limiting

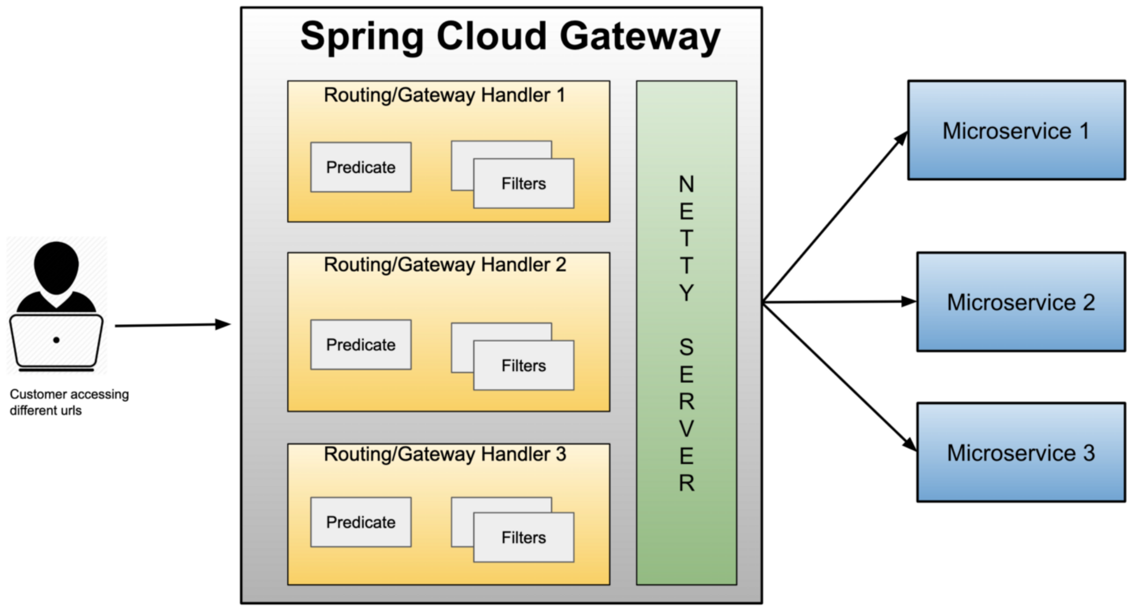
Path Rewriting

So, what is Spring Cloud Gateway?

Spring Cloud Gateway is API Gateway implementation by Spring Cloud team on top of Spring reactive ecosystem. It provides a simple and effective way to route incoming requests to the appropriate destination using Gateway Handler Mapping.

And Spring Cloud Gateway uses Netty server to provide non-blocking asynchronous request processing.

Below is high level flow of how request routing works in Spring Cloud Gateway:



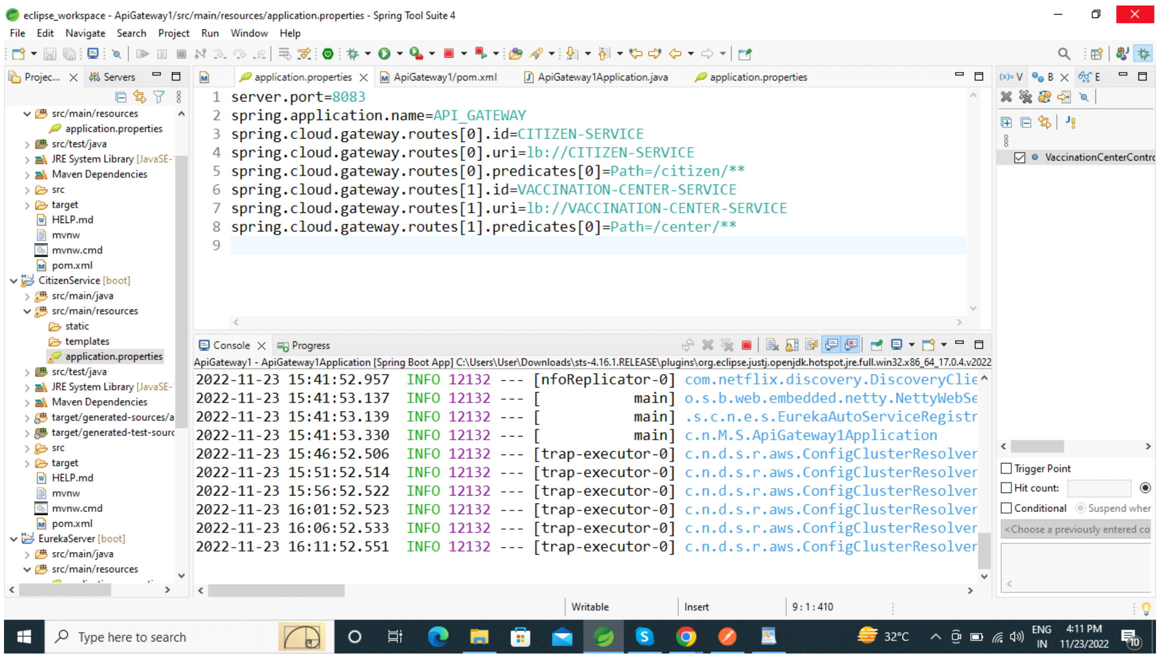
Spring Cloud Gateway consists of 3 main building blocks:

Route: Think of this as the destination that we want a particular request to route to. It comprises of destination URI, a condition that has to satisfy — Or in terms of technical terms, Predicates, and one or more filters.

Predicate: This is literally a condition to match. i.e. kind of “if” condition..if requests has something — e.g. path=blah or request header contains foo-bar etc. In technical terms, it is Java 8 Function Predicate

Filter: These are instances of Spring Framework WebFilter. This is where you can apply your magic of modifying request or response. There are quite a lot of out of box WebFilter that framework provides. But of course, we are talking about Spring Framework.

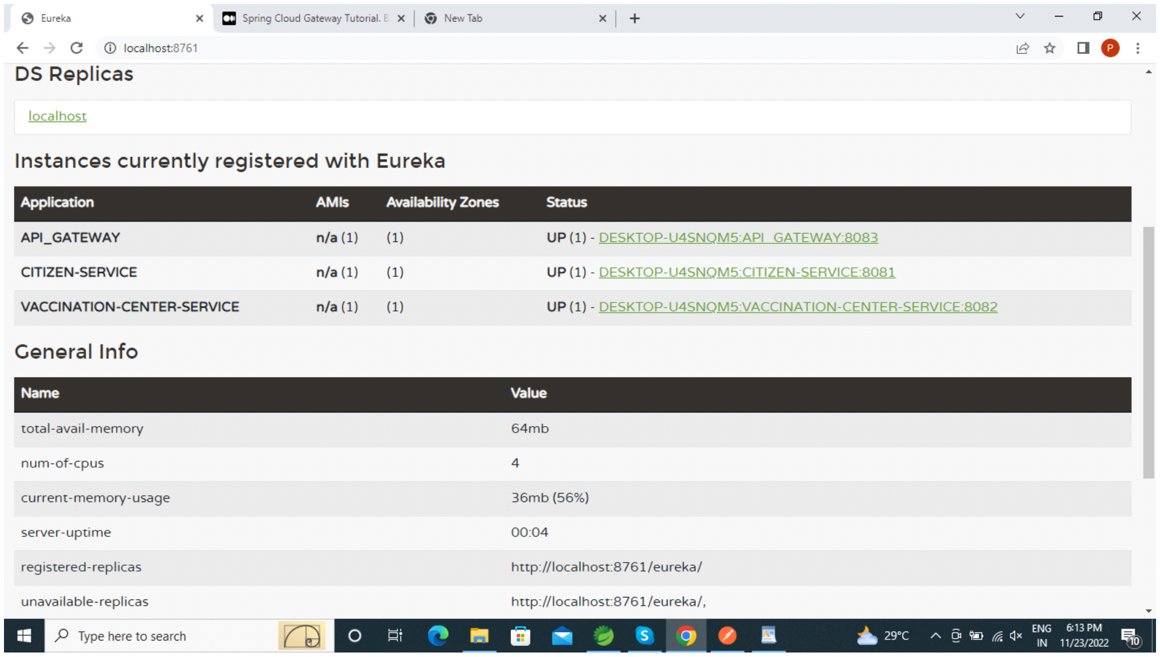
Let's create API gateway :



```
server.port=8083
spring.application.name=API_GATEWAY
spring.cloud.gateway.routes[0].id=CITIZEN-SERVICE
spring.cloud.gateway.routes[0].uri=lb://CITIZEN-SERVICE
spring.cloud.gateway.routes[0].predicates[0]=Path=/citizen/**
spring.cloud.gateway.routes[1].id=VACCINATION-CENTER-SERVICE
spring.cloud.gateway.routes[1].uri=lb://VACCINATION-CENTER-SERVICE
spring.cloud.gateway.routes[1].predicates[0]=Path=/center/**
```

```
ApiGateway1 - ApiGateway1Application [Spring Boot App] C:\Users\User\Downloads\sts-4.16.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v2022
2022-11-23 15:41:52.957  INFO 12132 --- [nfoReplicator-0] com.netflix.discovery.DiscoveryClie
2022-11-23 15:41:53.137  INFO 12132 --- [           main] o.s.b.web.embedded.netty.NettyWebSe
2022-11-23 15:41:53.139  INFO 12132 --- [           main] .s.c.n.e.s.EurekaAutoServiceRegistr
2022-11-23 15:41:53.330  INFO 12132 --- [           main] c.n.M.S.ApiGateway1Application
2022-11-23 15:46:52.506  INFO 12132 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver
2022-11-23 15:51:52.514  INFO 12132 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver
2022-11-23 15:56:52.522  INFO 12132 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver
2022-11-23 16:01:52.523  INFO 12132 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver
2022-11-23 16:06:52.533  INFO 12132 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver
2022-11-23 16:11:52.551  INFO 12132 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver
```

using all routing in application.properties of API gateway, we get the result as-



localhost:8761

DS Replicas

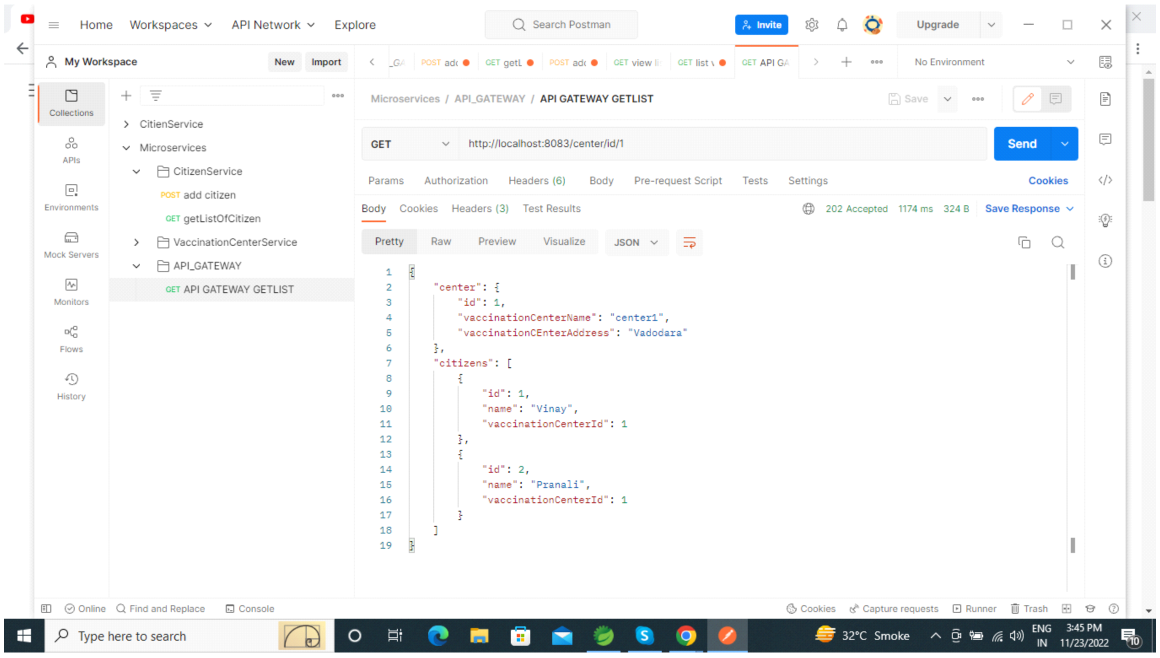
localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API_GATEWAY	n/a (1)	(1)	UP (1) - DESKTOP-U4SNQM5:API_GATEWAY:8083
CITIZEN-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-U4SNQM5:CITIZEN-SERVICE:8081
VACCINATION-CENTER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-U4SNQM5:VACCINATION-CENTER-SERVICE:8082

General Info

Name	Value
total-avail-memory	64mb
num-of-cpus	4
current-memory-usage	36mb (56%)
server-uptime	00:04
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,

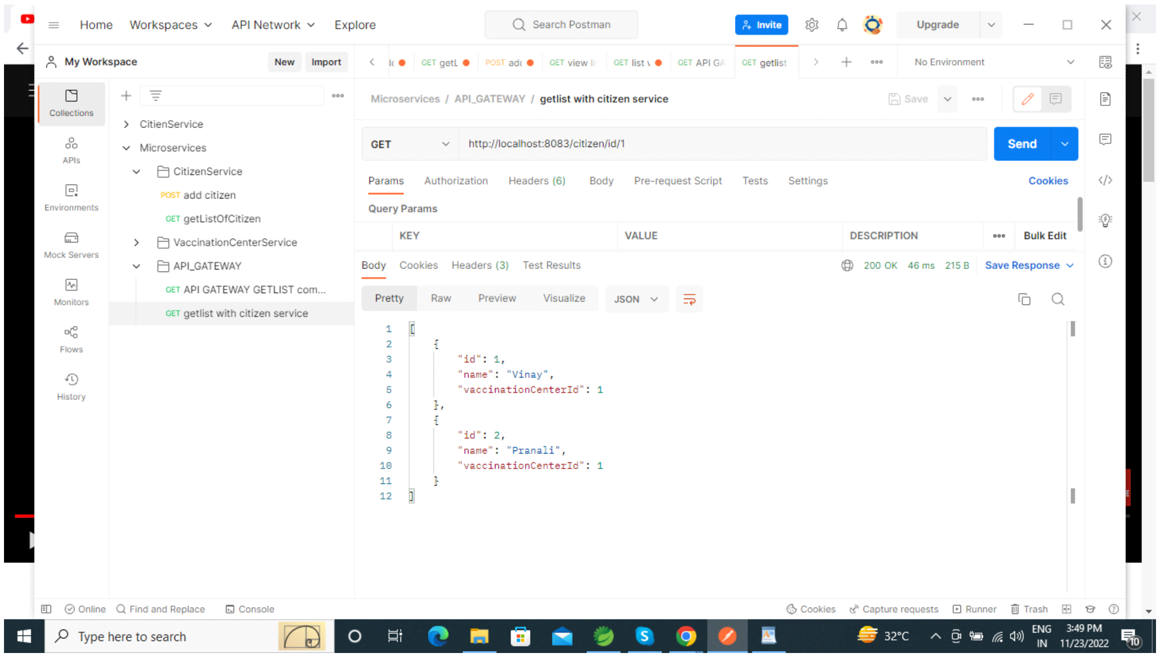


Microservices / API_GATEWAY / API GATEWAY GETLIST

GET http://localhost:8083/center/id/1

202 Accepted 1174 ms 324 B

```
{
    "center": {
        "id": 1,
        "vaccinationCenterName": "center1",
        "vaccinationCEnterAddress": "Vadodara"
    },
    "citizens": [
        {
            "id": 1,
            "name": "Vinay",
            "vaccinationCenterId": 1
        },
        {
            "id": 2,
            "name": "Pranali",
            "vaccinationCenterId": 1
        }
    ]
}
```



Microservices / API_GATEWAY / getlist with citizen service

GET http://localhost:8083/citizen/id/1

200 OK 46 ms 215 B

```
[
    {
        "id": 1,
        "name": "Vinay",
        "vaccinationCenterId": 1
    },
    {
        "id": 2,
        "name": "Pranali",
        "vaccinationCenterId": 1
    }
]
```

Reference:-

https://youtube.com/playlist?list=PLyHJZXNdCXsd2e3NMW9sZbto8RB5foBtp