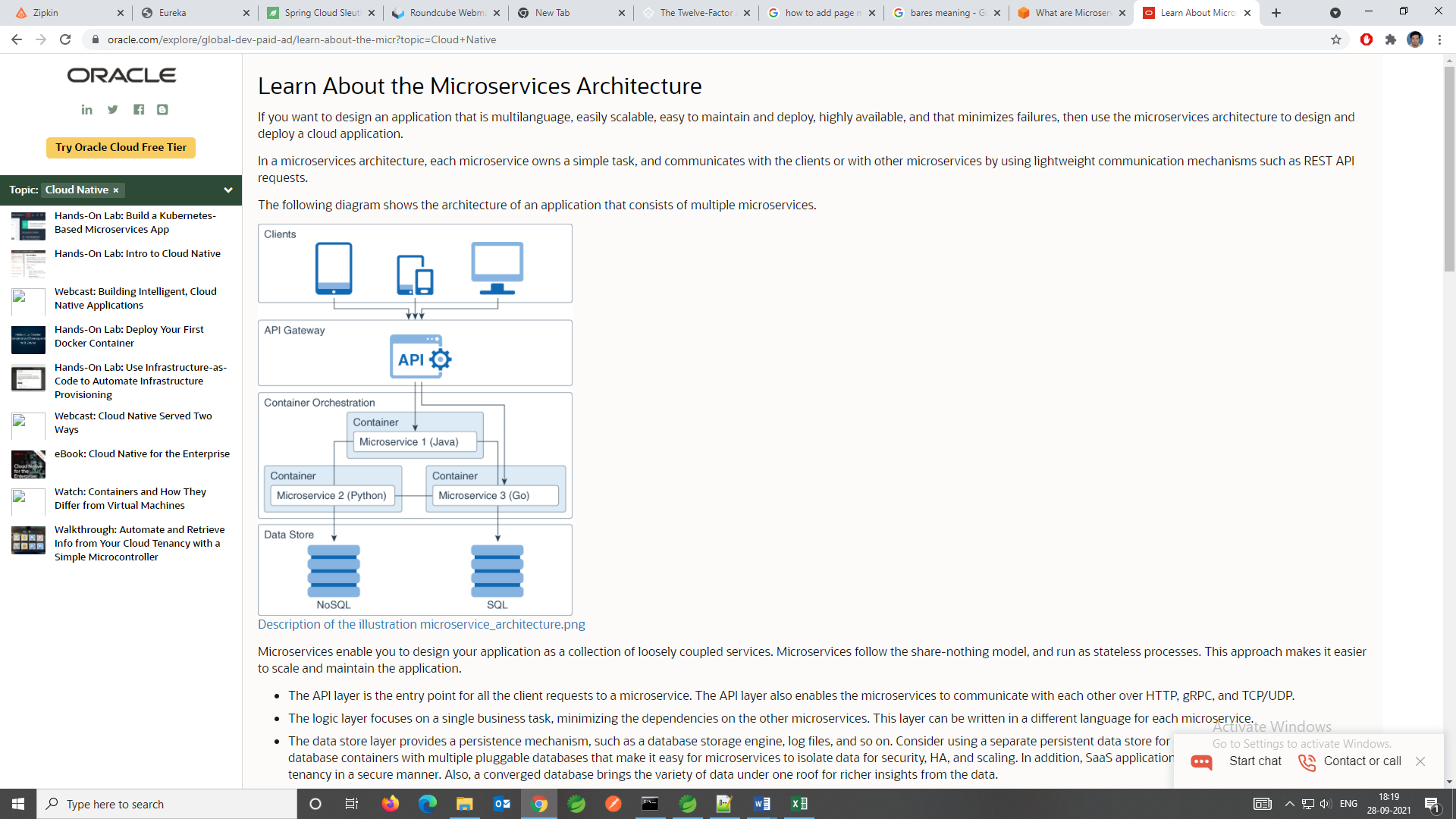
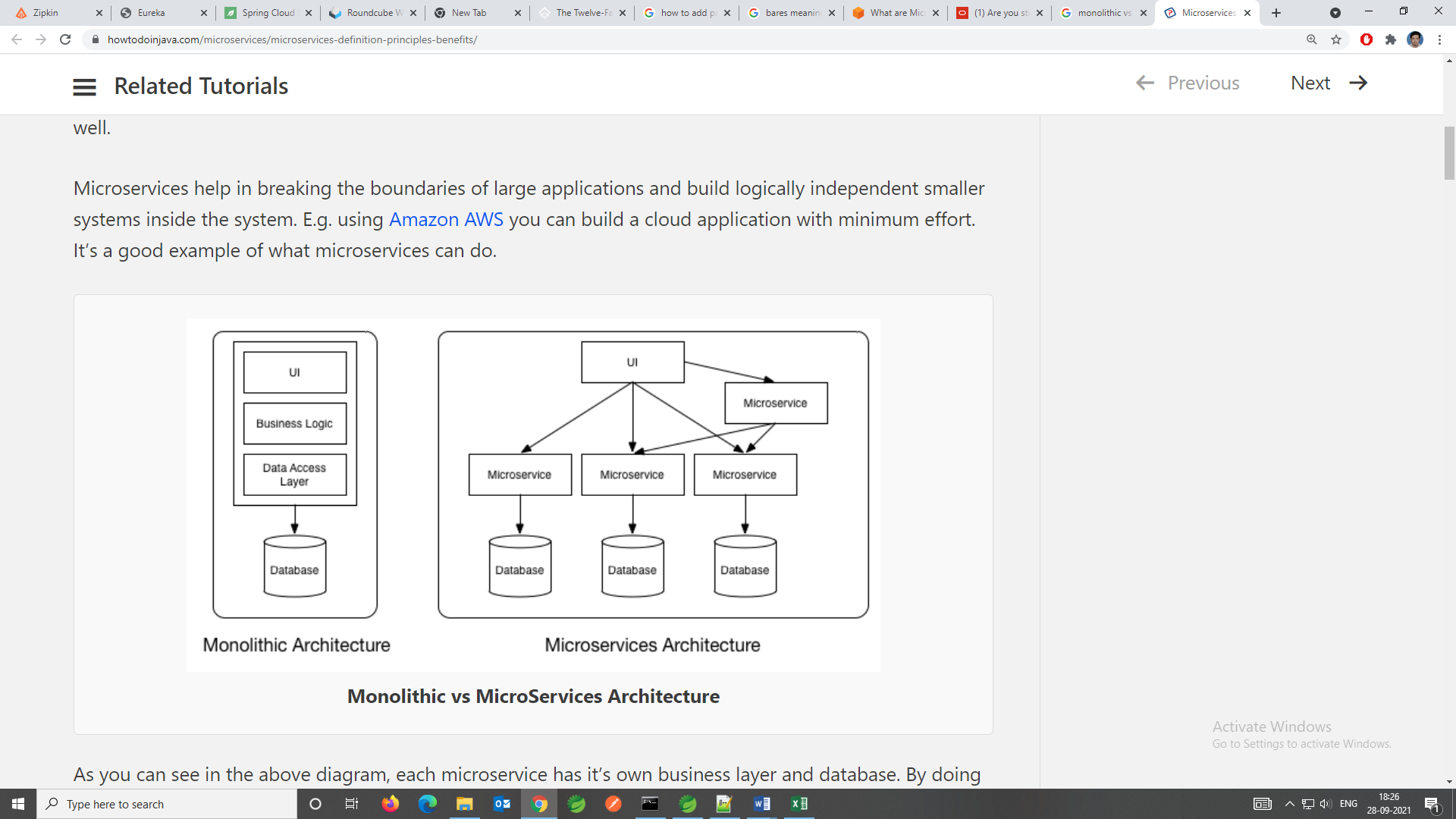
**MICROSERVICES**

Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs. These services are owned by small, self-contained teams.

Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.



|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Microservices Architecture** | **Monolithic Architecture** |
| Unit design | The application consists of loosely coupled services. Each service supports a single business task. | The entire application is designed, developed, and deployed as a single unit. |
| Functionality reuse | Microservices define APIs that expose their functionality to any client. The clients could even be other applications | The opportunity for reusing functionality across applications is limited. |
| Communication within the application | To communicate with each other, the microservices of an application use the request-response communication model. The typical implementation uses REST API calls based on the HTTP protocol. | Internal procedures (function calls) facilitate communication between the components of the application. There is no need to limit the number of internal procedure calls. |
| Technological flexibility | Each microservice can be developed using a programming language and framework that best suits the problem that the microservice is designed to solve. | Usually, the entire application is written in a single programming language. |
| Data management | Decentralized: Each microservice may use its own database. | Centralized: The entire application uses one or more databases. |
| Deployment | Each microservice is deployed independently, without affecting the other microservices in the application. | Any change, however small, requires redeploying and restarting the entire application |
| Maintainability | Microservices are simple, focused, and independent. So the application is easier to maintain. | As the application scope increases, maintaining the code becomes more complex. |
| Resiliency | The application functionality is distributed across multiple services. If a microservice fails, the functionality offered by the other microservices continues to be available | A failure in any component could affect the availability of the entire application. |
| Scalability | Each microservice can be scaled independently of the other services | The entire application must be scaled, even when the business requirement is for scaling only certain parts of the application. |



**PRINCIPALS-OF-MICROSERVICES**

[**I. Codebase**](https://12factor.net/codebase)

One codebase tracked in revision control, many deploys

[**II. Dependencies**](https://12factor.net/dependencies)

Explicitly declare and isolate dependencies

[**III. Config**](https://12factor.net/config)

Store config in the environment

[**IV. Backing services**](https://12factor.net/backing-services)

Treat backing services as attached resources

[**V. Build, release, run**](https://12factor.net/build-release-run)

Strictly separate build and run stages

[**VI. Processes**](https://12factor.net/processes)

Execute the app as one or more stateless processes

[**VII. Port binding**](https://12factor.net/port-binding)

Export services via port binding

[**VIII. Concurrency**](https://12factor.net/concurrency)

Scale out via the process model

[**IX. Disposability**](https://12factor.net/disposability)

Maximize robustness with fast startup and graceful shutdown

[**X. Dev/prod parity**](https://12factor.net/dev-prod-parity)

Keep development, staging, and production as similar as possible

[**XI. Logs**](https://12factor.net/logs)

Treat logs as event streams

[**XII. Admin processes**](https://12factor.net/admin-processes)

Run admin/management tasks as one-off processes

**ARCHITECTURE**

In this architecture we will create two micro-services one is **User-Service** and another one is **Department-Service.** So now **every user will be attached to any one department.** So we will save the department from one service and user from one service. We will create some methods to get the details and add the details.

The Most important part we will create a method which fetch the user details **from user-service** along with that it will fetch the departmentdetails form **department-service.** To that we are going to use rest API (**REST Template**).

Then we will create a **service-registry. All microservices will be connected to that particular service-registry.** So from service-registry we get an idea what are the micro services is connected to the service-registry and their status.

Then we will create an API gateway, which is nothing but a gateway for all the APIs which we have in our application. So all the request should not directly go to the microservices, instead of that it will go through the API gateway.

**Config-Server** Implementation. Let’s say we have 1000 and 1000 microservices and for some of the microservices the configuration is repeated. So it’s not a best practice to add the configuration in all the microservices. Suppose if there is any change then we need to go to all the microservice and we have to change it. So to avoid this staff we will create a **cloud-config-server we will create a GIT repository and all configuration we will save there, and this cloud config-server will read the GIT repository and it will give the configuration to the all the microservices which are connected to it.** So it will be easier for us to maintain the configuration, meaning change it to the all place and it will be available to all the microservices.

**Distributed Login** implementation. So there are lots of micro-services so for identifying the failed microservices we will use zipkin server. So for that we will create a zipkin server and then we will implement the zipkin client and sleuth library. This thigs provide multiple options like “**we can identify which are the services are called what is the trace id and what is the span id**”.

NOTES::

**ZAPKIN:** [Zipkin](https://zipkin.io/) is very efficient tool for **distributed tracing** in [microservices](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/) ecosystem. Distributed tracing, in general, is latency measurement of each component in a distributed transaction where multiple microservices are invoked to serve a single business use case. Let’s say from our application, we have to call 4 different services/components for a transaction. Here with distributed tracing enabled, we can measure which component took how much time.

Internally it has 4 modules –

1. **Collector** – Once any component sends the trace data arrives to Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.
2. **Storage** – This module store and index the lookup data in backend. [Cassandra](https://cassandra.apache.org/), [ElasticSearch](https://www.elastic.co/) and [MySQL](https://howtodoinjava.com/mysql/how-to-installuninstallexecute-mysql-as-windows-service/) are supported.
3. **Search** – This module provides a simple JSON API for finding and retrieving traces stored in backend. The primary consumer of this API is the Web UI.
4. **Web UI** – A very nice UI interface for viewing traces.

**SLEUTH:** [Sleuth](https://cloud.spring.io/spring-cloud-sleuth/) is a tool from Spring cloud family. It is used to generate the *trace id*, *span id* and add these information to the service calls in the headers and MDC, so that It can be used by tools like Zipkin and [ELK](https://howtodoinjava.com/microservices/elk-stack-tutorial-example/) etc. to store, index and process log files. As it is from spring cloud family, once added to the CLASSPATH, it automatically integrated to the common communication channels like –

* requests made with the [RestTemplate](https://howtodoinjava.com/spring-boot2/resttemplate/spring-restful-client-resttemplate-example/) etc.
* requests that pass through a [Netflix Zuul](https://howtodoinjava.com/spring/spring-cloud/spring-cloud-api-gateway-zuul/) microproxy
* HTTP headers received at [Spring MVC](https://howtodoinjava.com/spring-mvc-tutorial/) controllers
* requests over messaging technologies like Apache Kafka or RabbitMQ etc.

**BASIC IMPLEMENTATION**

In basic implementation we are calling one service from another service. So to do that we have to use **Rest template** in both the services.

**Rest Template:**  To consume a RESTful service, Spring provides the RestTemplate class. This allows you **to send HTTP requests to a RESTful server and fetch data** in a number of formats - such as JSON and XML.

To implement that we need to add one dependency

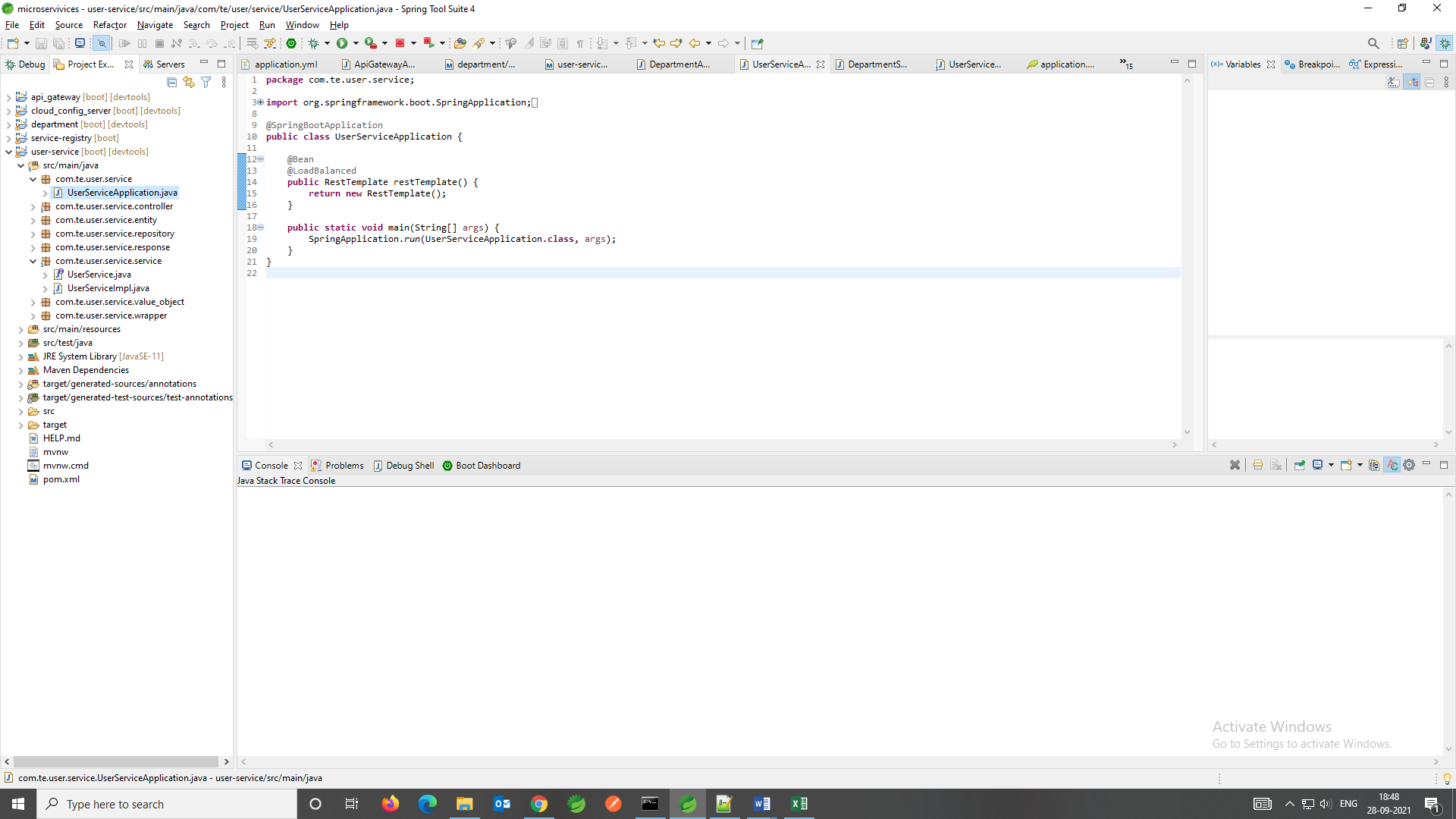
<dependency>

<groupId>org.springframework.cloud</groupId>

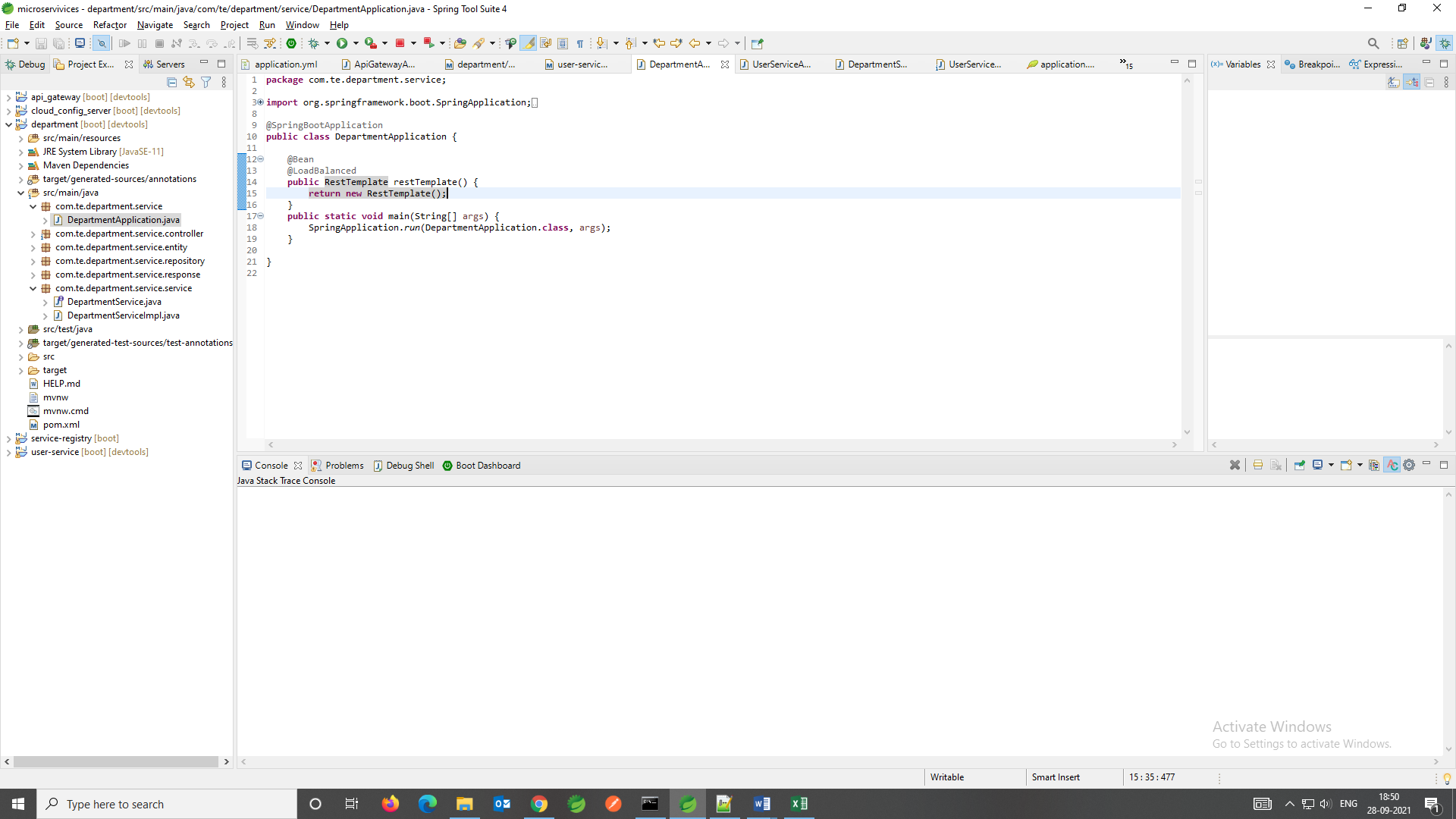
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

Rest Template method in user-service:



Rest Template method in department-service:



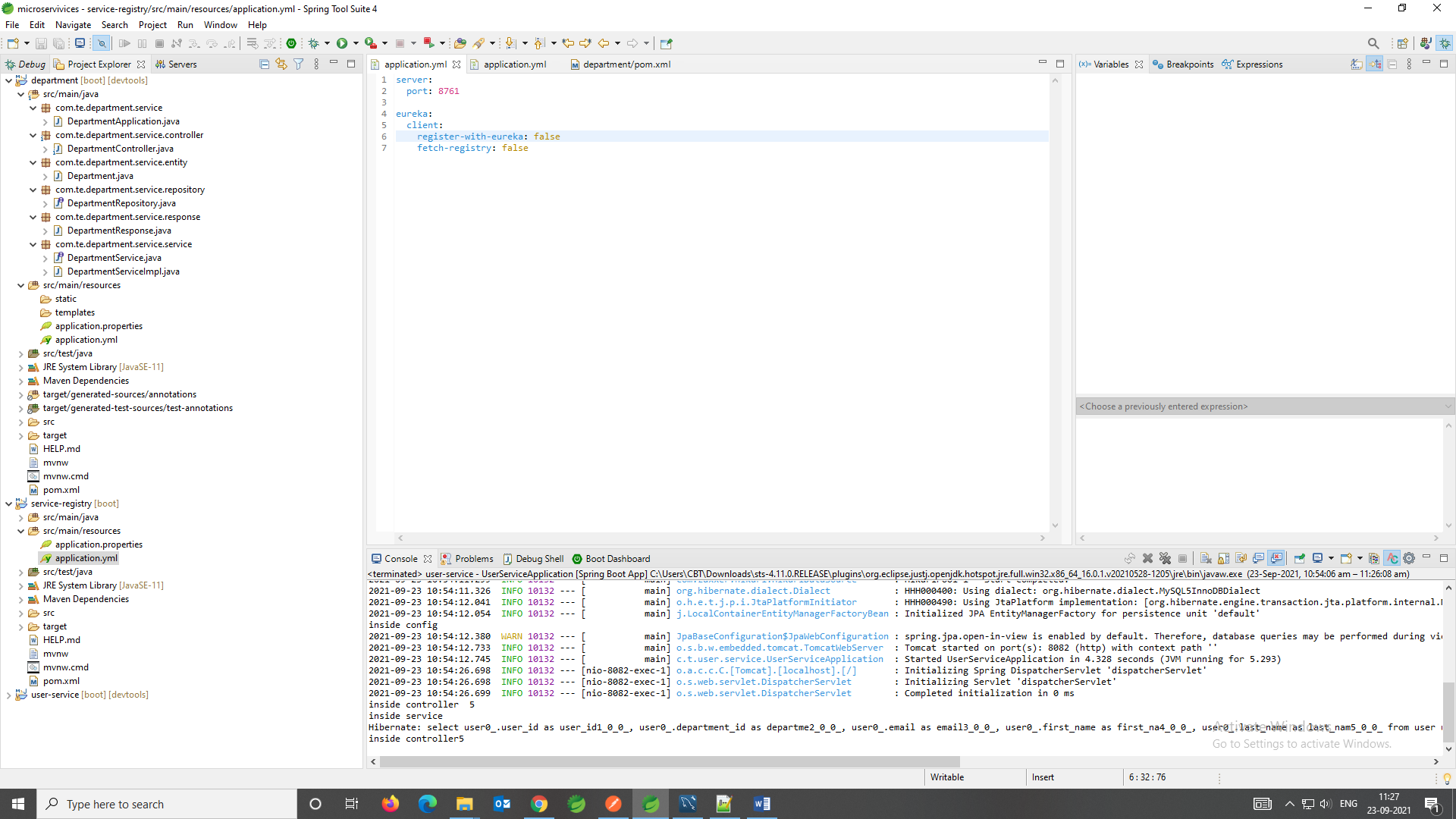
**IMPLEMENTATION FOR “Service-Registry” ,“API-Gateway” &&”Cloud-Config-Server”**

**NOTE**: If an application contains 1000 of micro-services then in case of management it is very difficult.

Now to maintain all the micro-services we create a **service-registry** which is responsible to maintain all the micro-services. So whatever micro-services will create that will have connected to the **service-registry**, so it will take care of everything like (port, URL etc).

**To implement the Service-Registry we are user Eureka Server:**

The Eureka server is nothing **but an implementation of service discovery pattern**, where microservices can register themselves so others can discover them. ... This server is also known as discovery server. The @EnableEurekaServer annotation is used to make your Spring Boot application acts as a Eureka Server.



After this:

For the other services we have to implement the eureka client over there. THERE WE NEED TO GIVE THE “**register-with-eureka=true**” && “**fetch-registry=true**”. So that these two can be consider as a client and they can go ahead and connect with the eureka server here. So for that we have to add some dependency.

1. <spring-cloud.version>2020.0.3</spring-cloud.version>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

1. Dependency management and built informantion

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

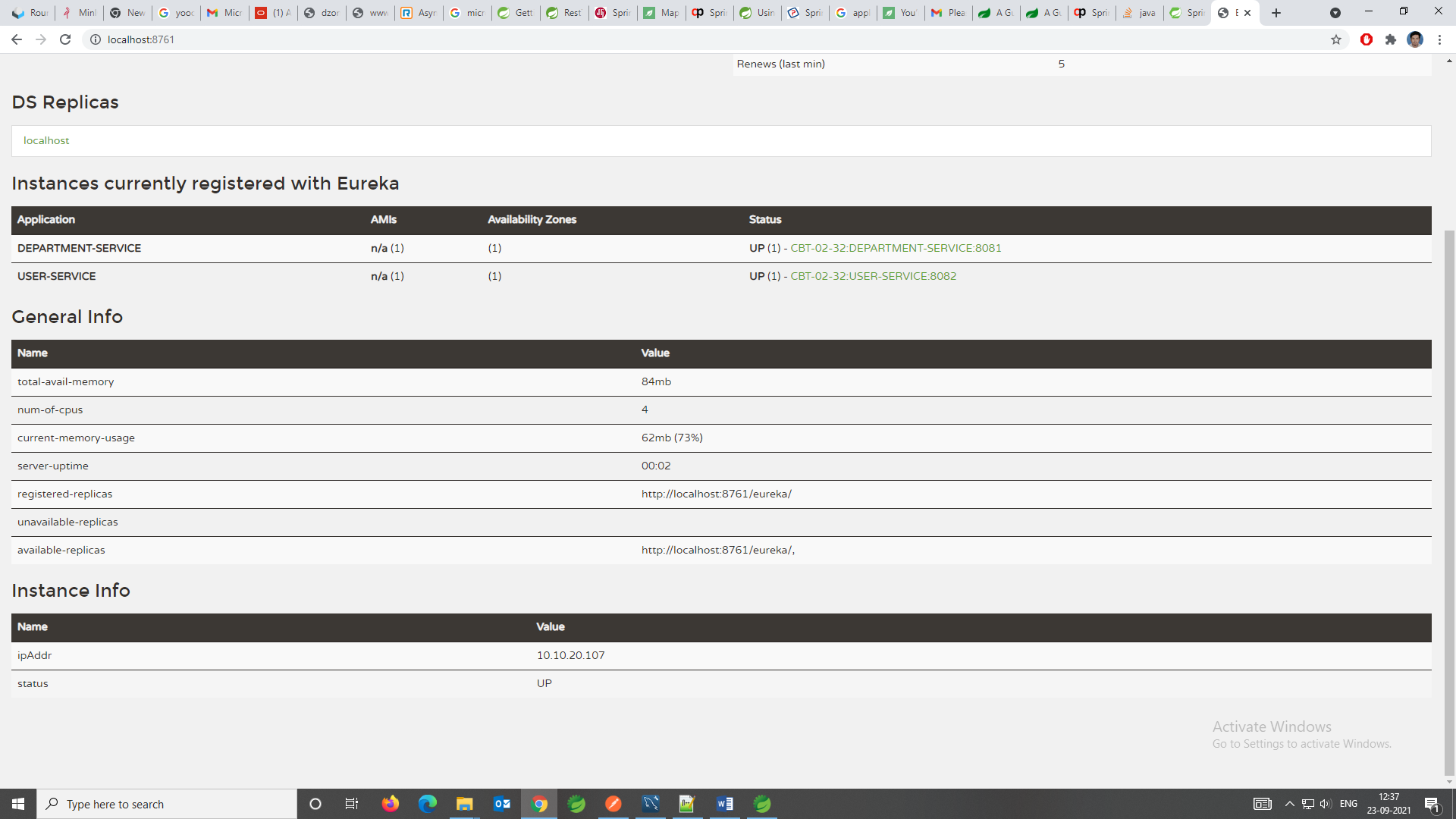
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

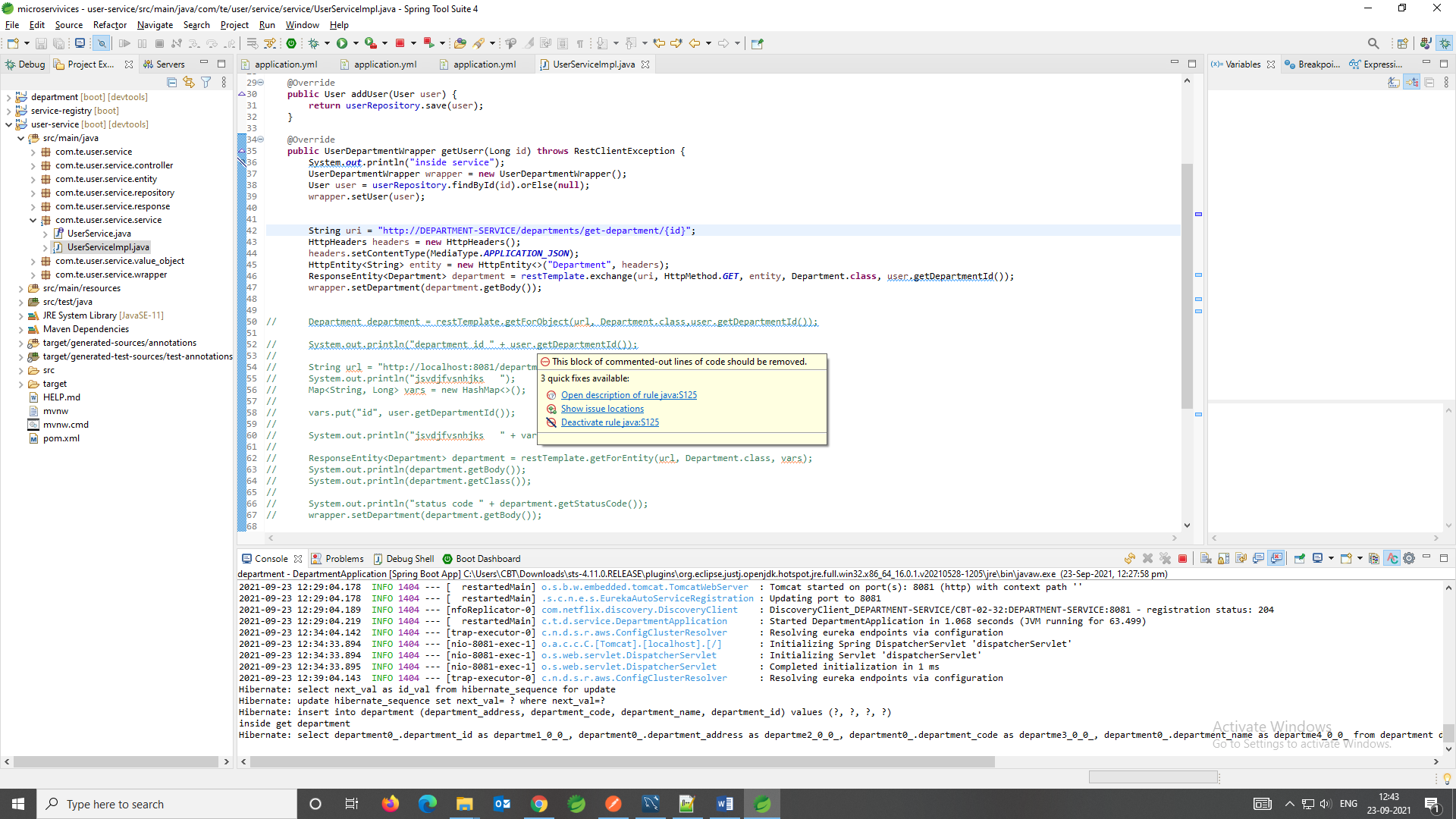
</plugin>

</plugins>

</build>



Consider we have a multiple instances of the service and those services are working on a deferent ports and different URL so it’s quite difficult to get the correct URL information. So to resolve this one we will chain the URL information and we will give the application name.



Now after this situation if you hit the get user API then it will throw and exception

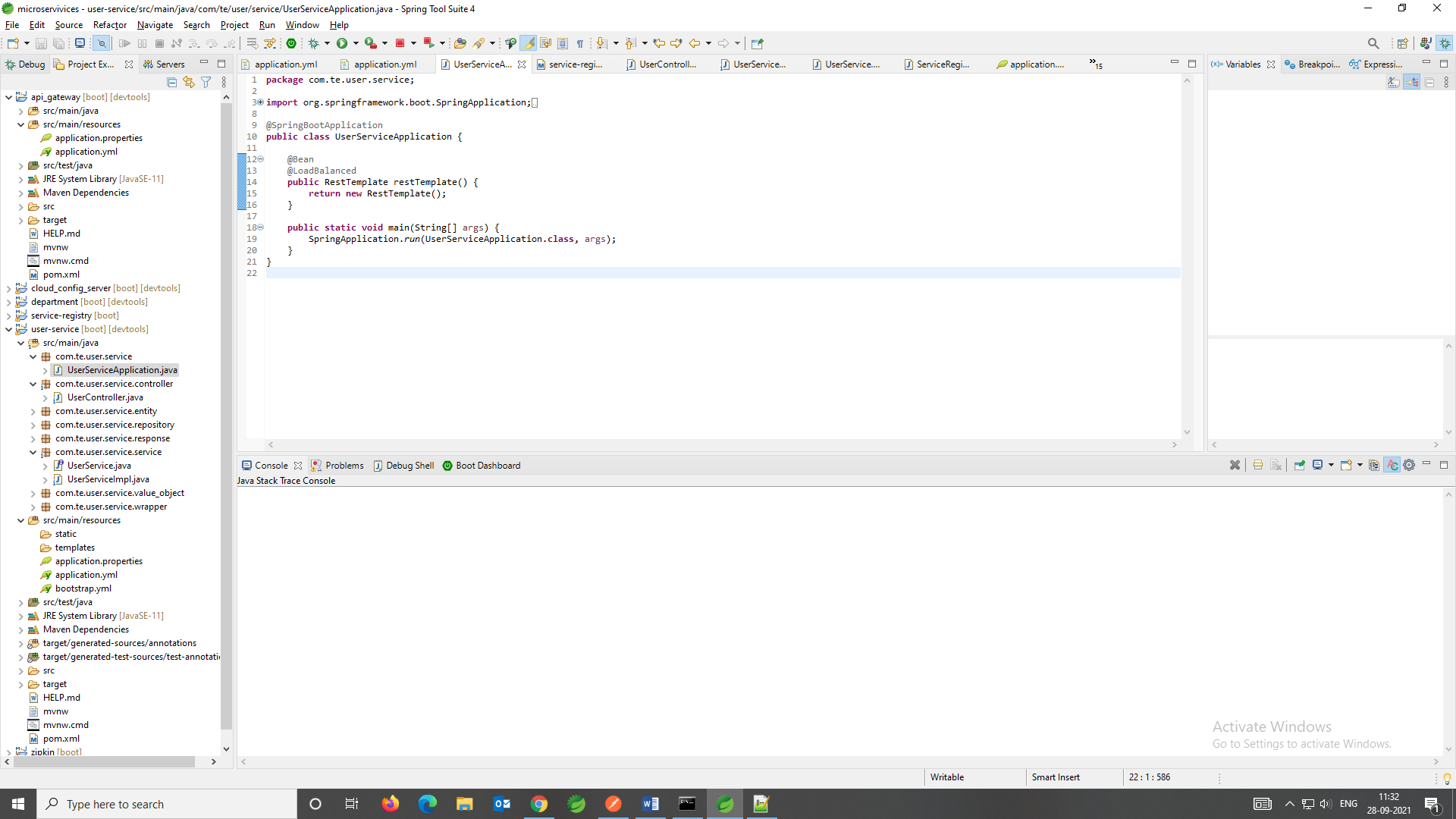


To resolve that one we need to tell the rest template as well “BOSS WE CONNECTED TO SERVICE-REGISTRY SO YOU NEED TO LOAD BALANCE YOUR REQUEST.”

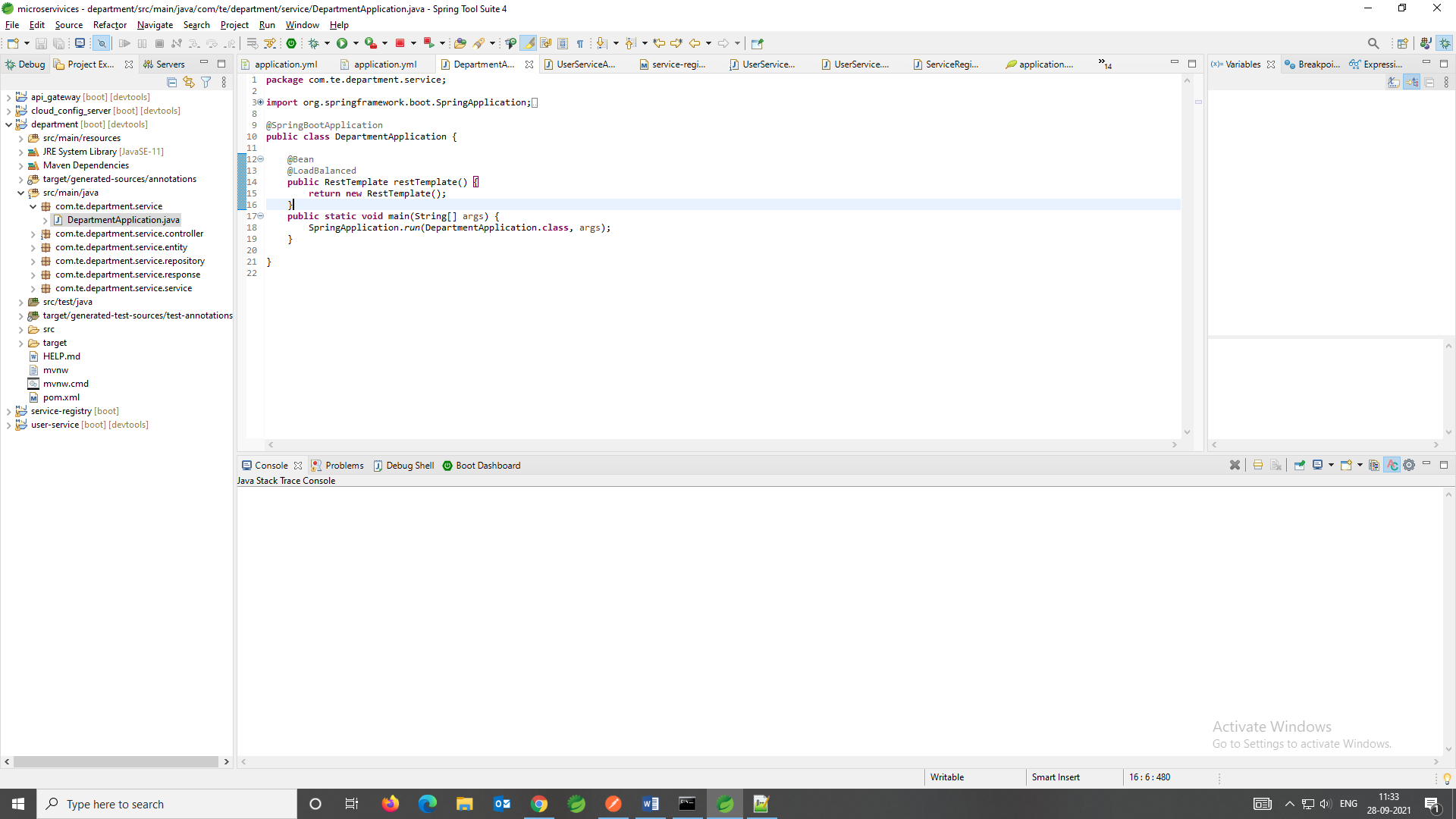
So to do that we need add an annotation named as **@LoadBalanced.** In the user main application.

Working principal: So if there are multiple services available connected to service-registry it will load balance the request for us.

**USER-SERVIE main class**

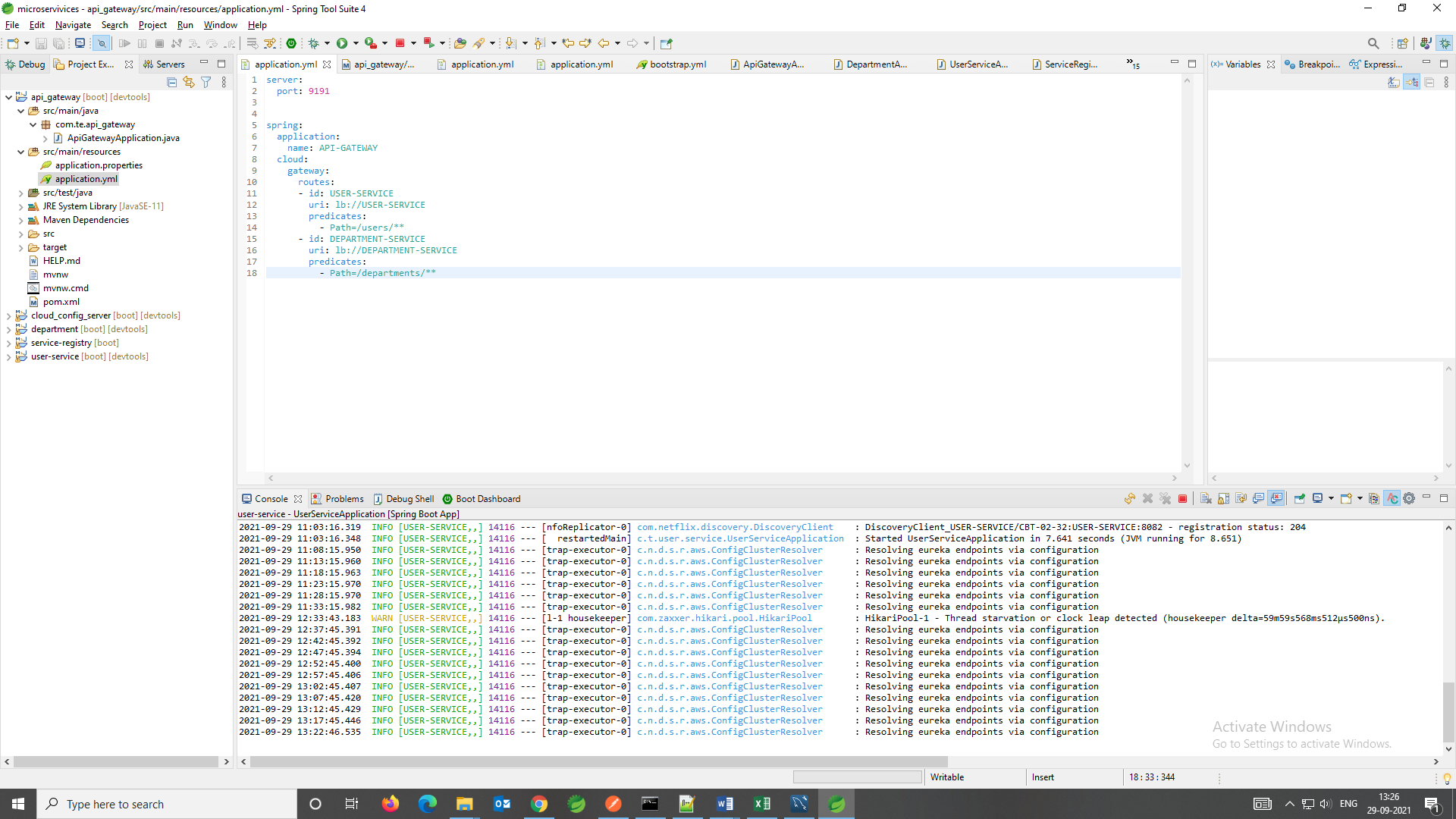


**DEPARTMENT-SERVICE main class**



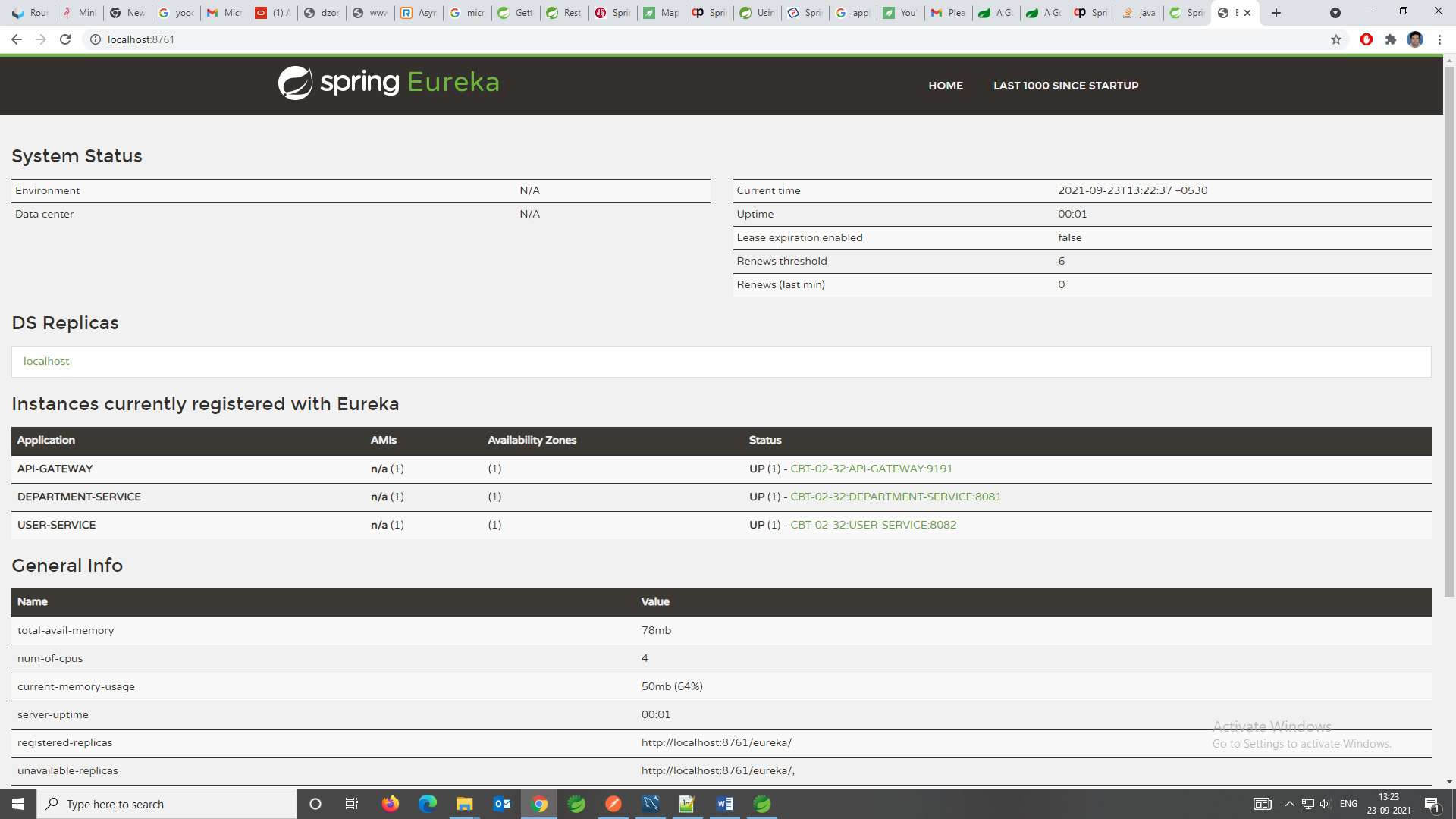
**NOW WE WILL CREATE THE API GATEWAY:**

* Create a new project called API-gateway.
* ADD the below dependencies,
  + Spring Boot Dev Tool
  + Spring Boot Actuator
  + Eureka Discovery Client
  + Gateway
* Now we have to do the routing configuration. So all the URLs which has a pattern “**/users**” should be redirected to the **user micro-service** and all the URLs which has a pattern “**/department**” should be redirect to the **department micro-service**.
* This routing configuration we have to do it in our API cloud Gateway



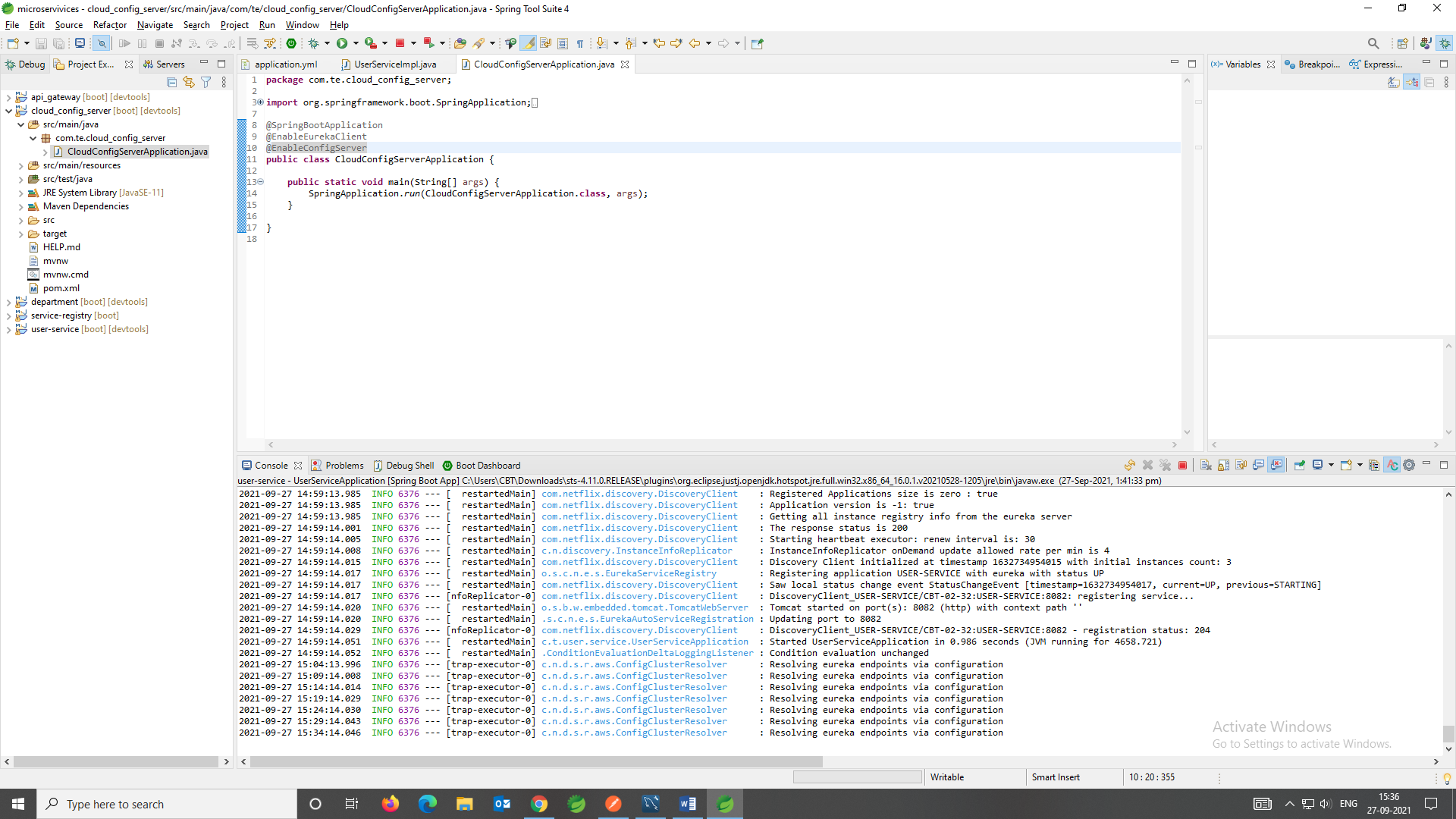
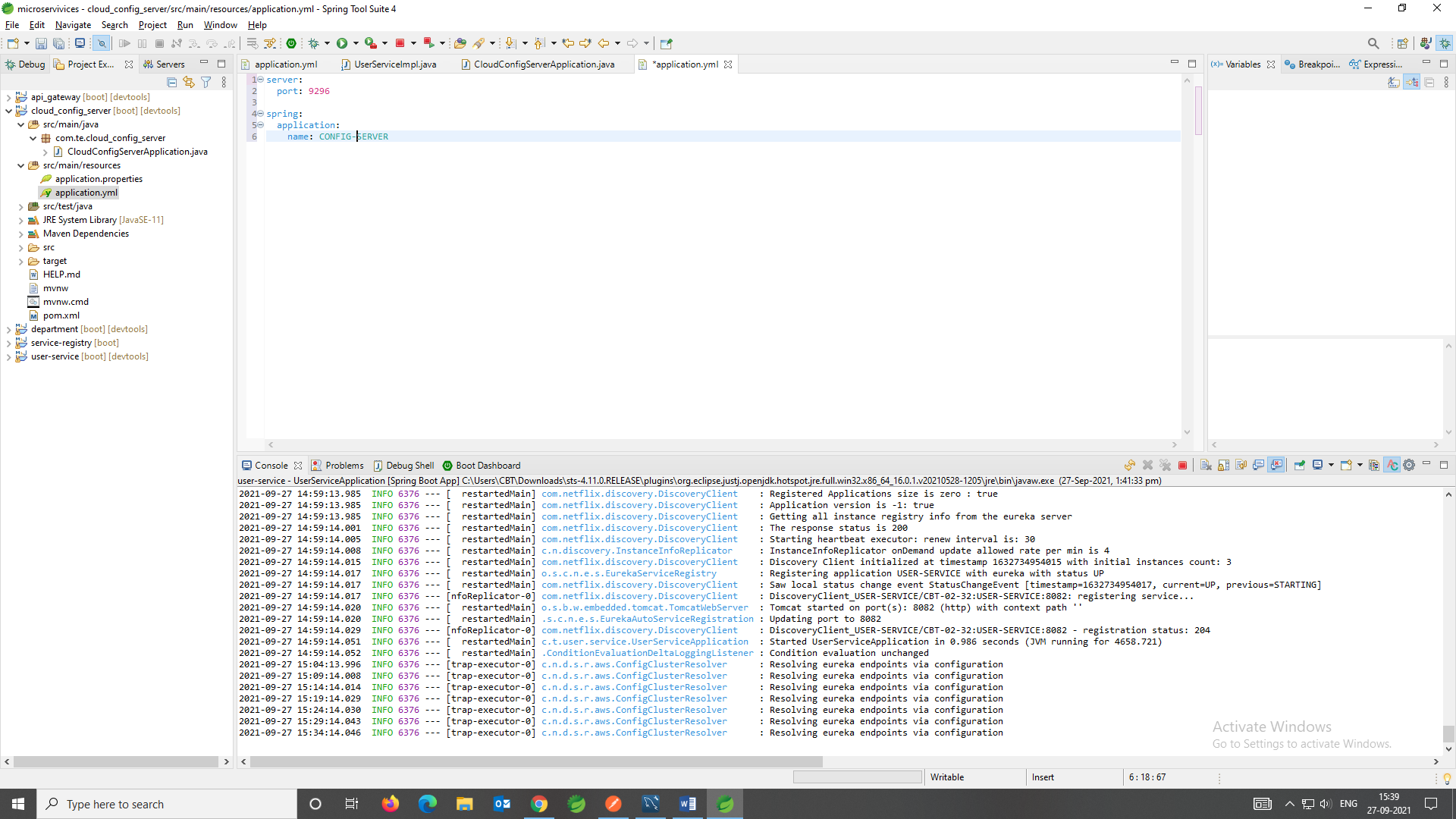
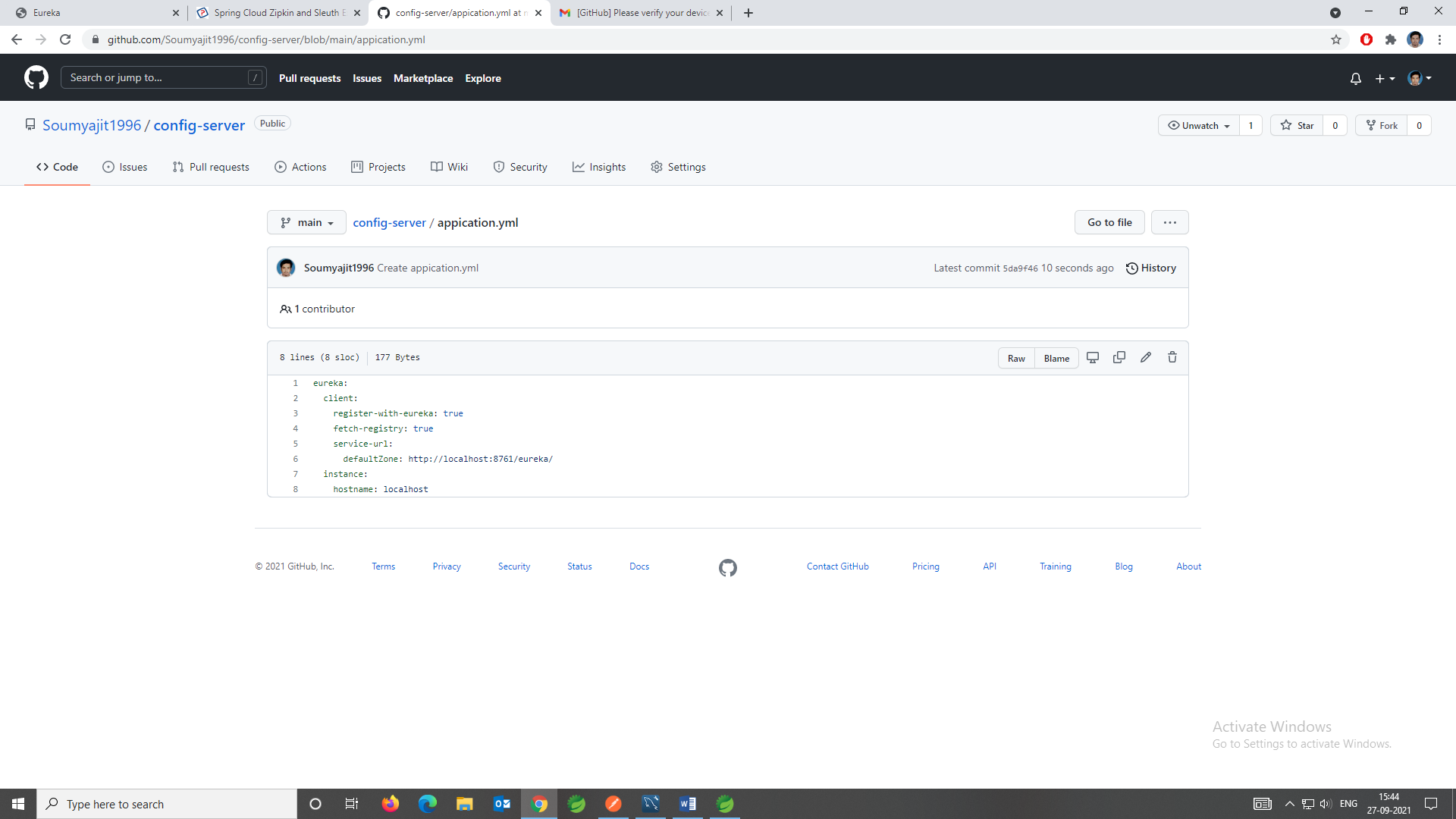
This is the routing configuration.

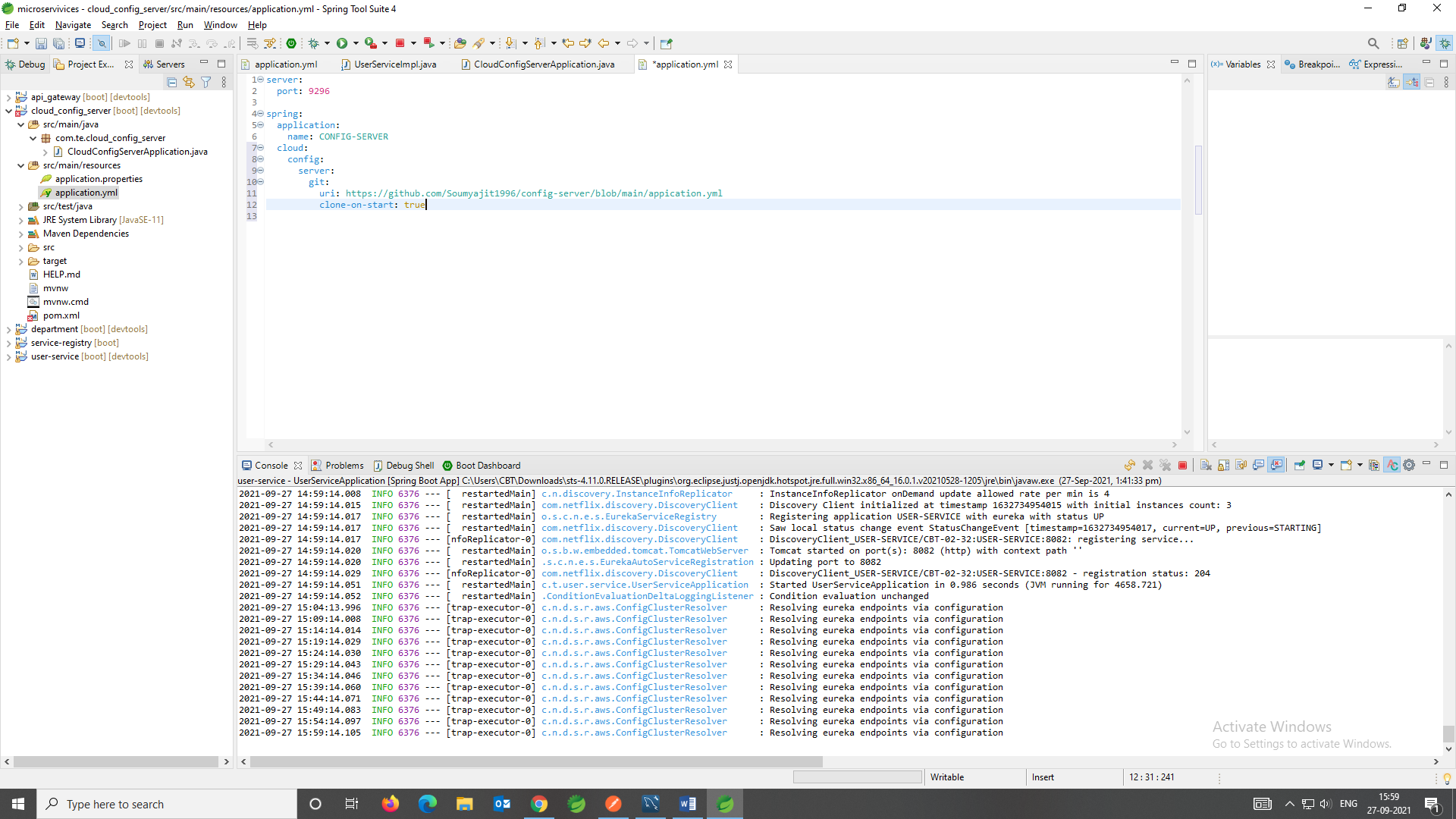
* Now after restarting the all microservices we will get like this on **localhost:8761**.



* Now we will check how the api gateway is behaving, earlier we are requesting the department service with the port 8081 and for the user service is 8082 now what we will do, we will hit all our request through the port 9191 which is our api gateway and this api gateway is responsible to traverse those request to the particular microservices which we have.
* Now go to the post man and test it out.

**NOW WE WILL CREATE THE CONFIGURATION-SERVER:**

* Create a project called “**Cloud-config-server**”.
* ADD dependencies
  + Eureka client
  + Config Server
* So the main class will looks like
* 
* And the application.yml file will looks like,
* 
* After this we will create an GIT-hub repository and there will create a **application.yml** and there we will store our all default configuration. The below picture is an application.yml file where we put the common configuration for the microservices.
* 
* Now we have to do the configuration for the particular GIT hub repository.



* Now we have to do the configuration from the other microservices for this config server. So to do that we need to enable the **Config client** and we need to point to this particular url. So let’s go ahead and add the dependencies in all the microservices.  
  We have to add this dependency

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

* After adding the dependencies, we have to add the configuration for each and every microservices. So that they can directly talk to our **config server**.
* **The MOST IMPORTANT POINT-------🡪**
  + Now the application.yml file is used by the application context. Means whenever we starting the application context spring boot will be using the application.yml file. But for getting all the cloud configuration there is a file called **bootstrap.yml** file so that is used to bootstrap the configuration and then it will start the application from the application.yml file.
  + So what we will do is we will create the bootstrap.yml file, then we will add the cloud configuration and then we will remove the bellow configuration from the application.

eureka:

client:

register-with-eureka: true

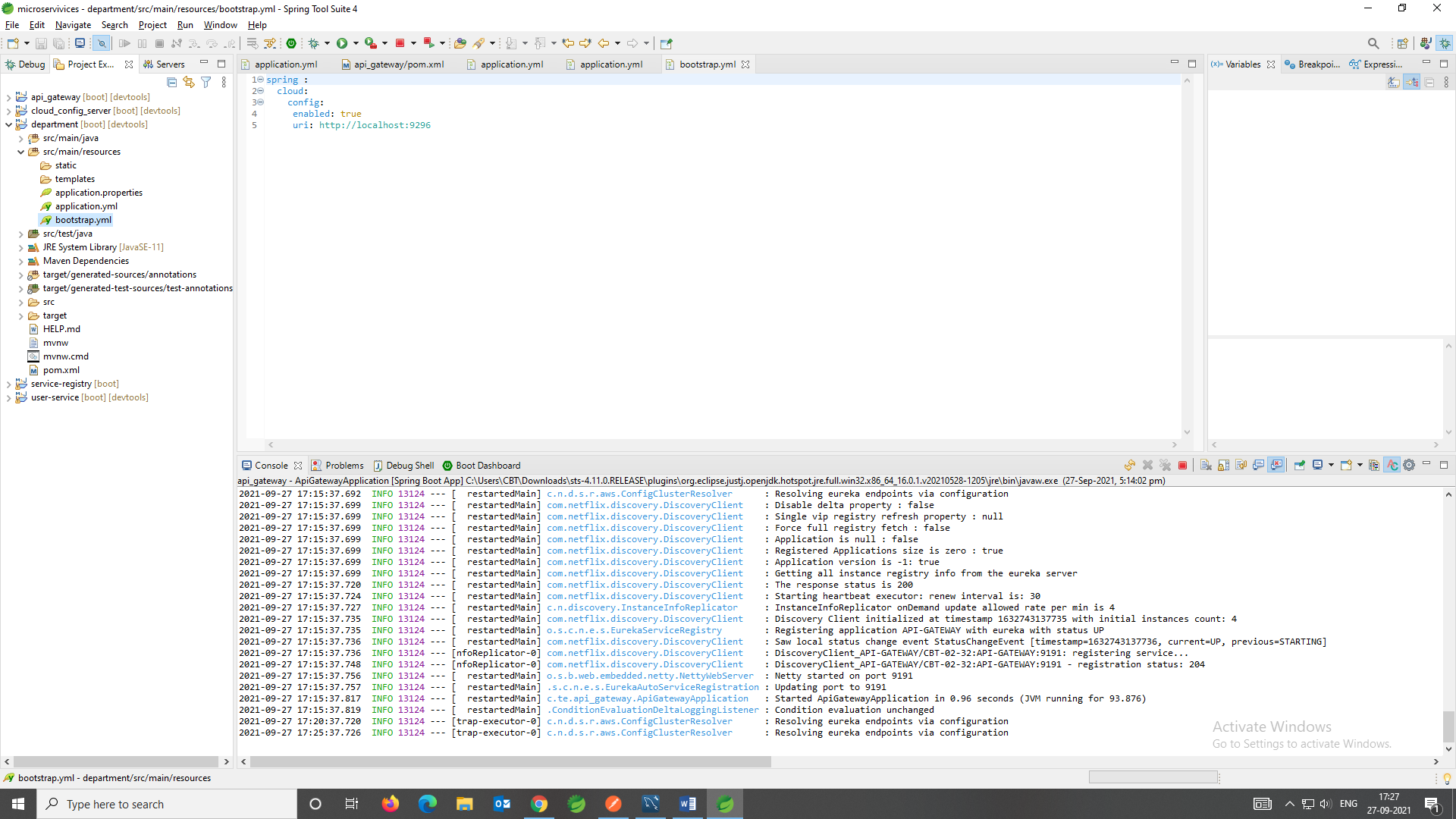
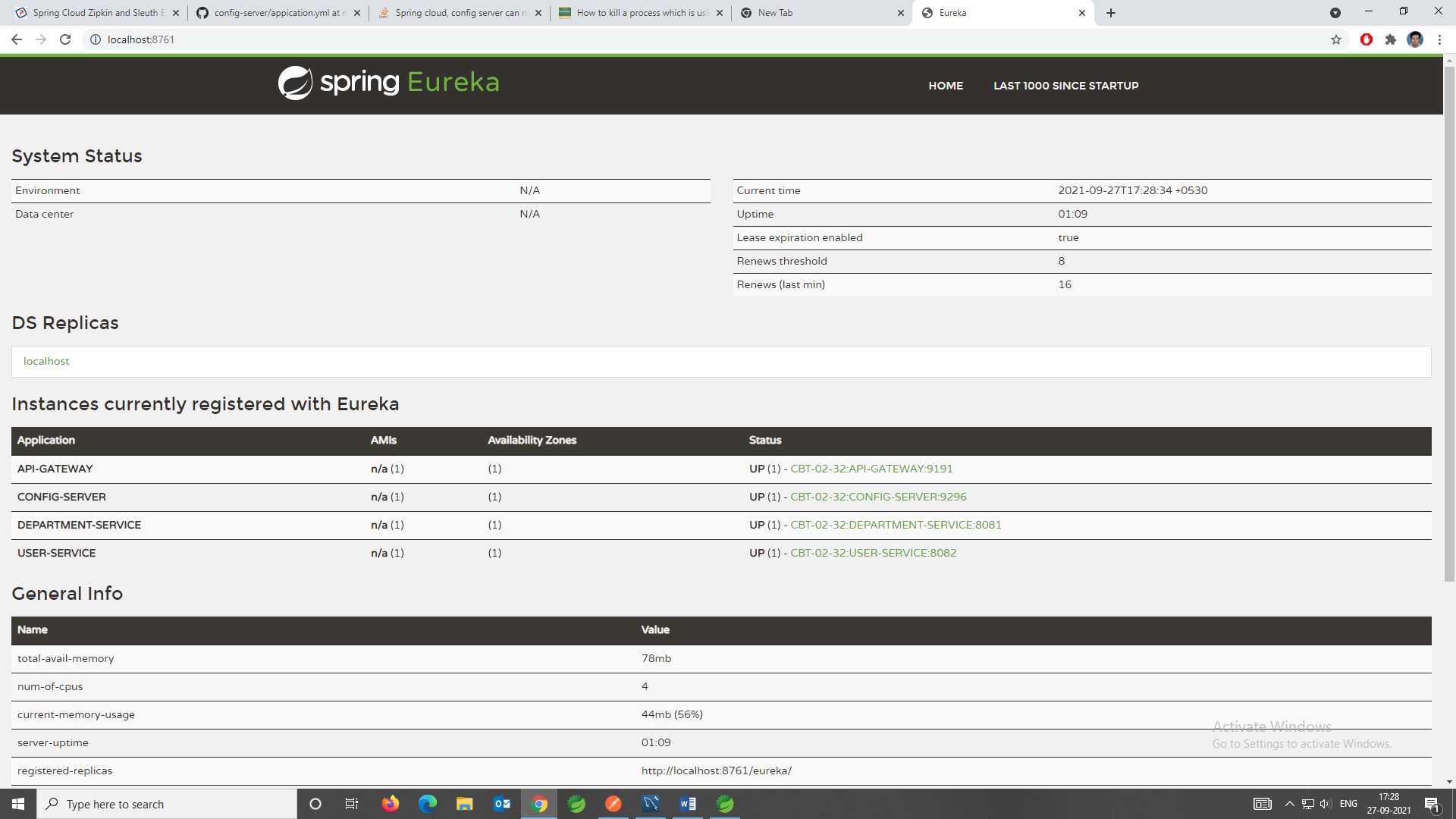
fetch-registry: true

service-url:

defaultZone: http://localhost:8761/eureka/

instance:

hostname: localhost

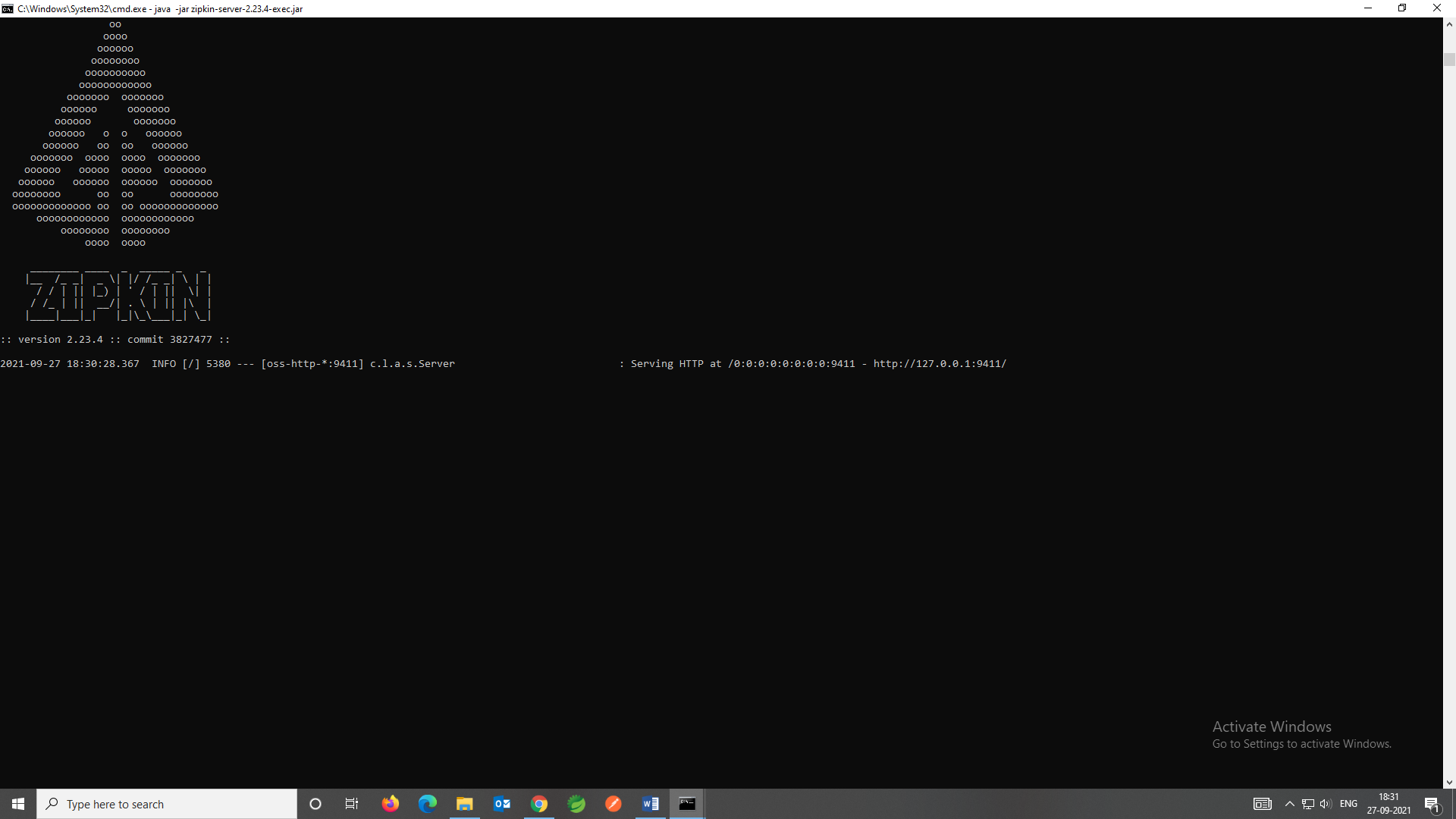
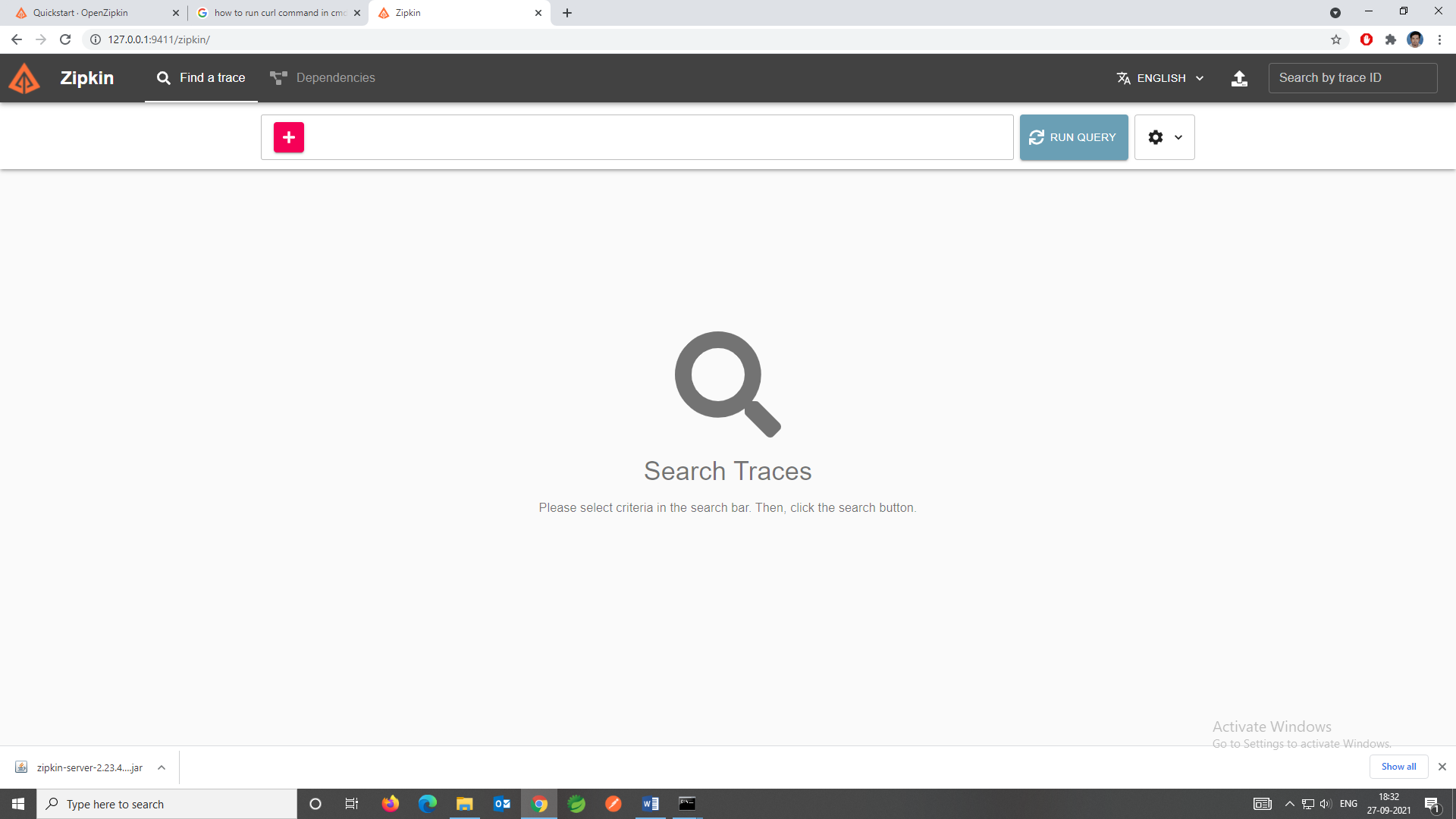
* + bootstrap.yml will contain
  + 
* So after running the server we will get this
* 

**NOW WE WILL IMPLEMENT THE ZIPKIN AND SLUETH:**

Now if there are lots of microservices then for tracing the status and error we will go for **zipkin and sleuth.** This two is responsible to give us the **log tracing** (**trace-id** (Which is unique across the microservices for particular one request) and **span-id** (this span id will change with respect to microservices, means if the request is moving from **user-service** to **department-service** then the span-id will change accordingly).

So the entire thig we will get to know with the help of this two which is nothing but **distributed-log-tracing**.

So to implement that we need to do the following things.

* We need the zipkin server and we need to implement the zipkin client and sleuth in all our microservices.
* 1st go to the zipkin server. Then go to the quick start, (docker is best) but for simplicity we will download the jar file and we will start the zipkin server using java-jar command. Steps are bellow.
  + CMD command-🡪 **java -jar zipkin-server-2.23.4-exec.jar**
  + 
  + **This is the output**
  + **And the browser will look like this**
* 
* Now in the server there is no services so to enable the services so there is no traces
* So now we have to implement the **zipkin client** and **sleuth** all over the microservices. So we will add to the **user-service** and **department-service**.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-sleuth-zipkin</artifactId>

</dependency>

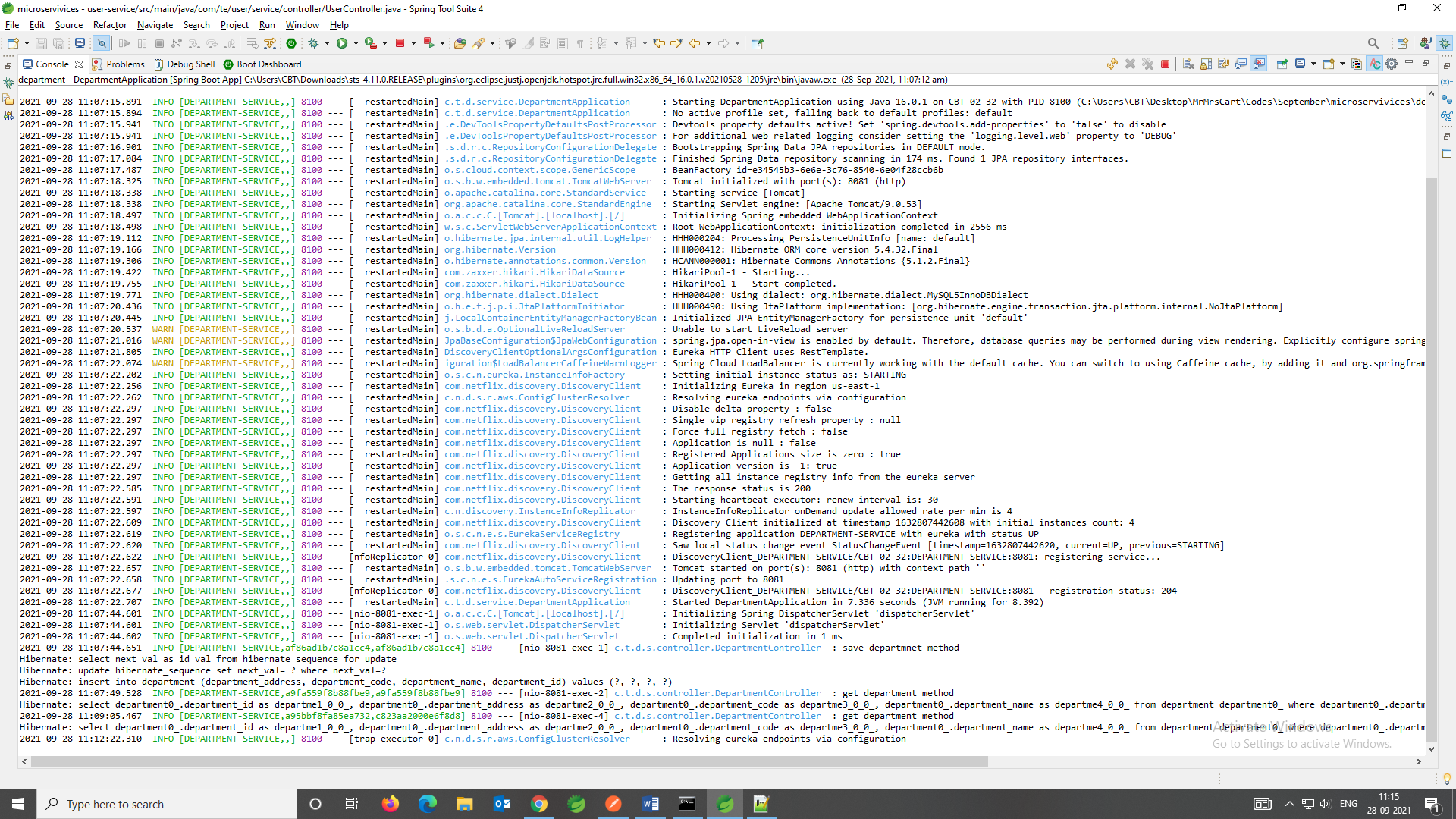
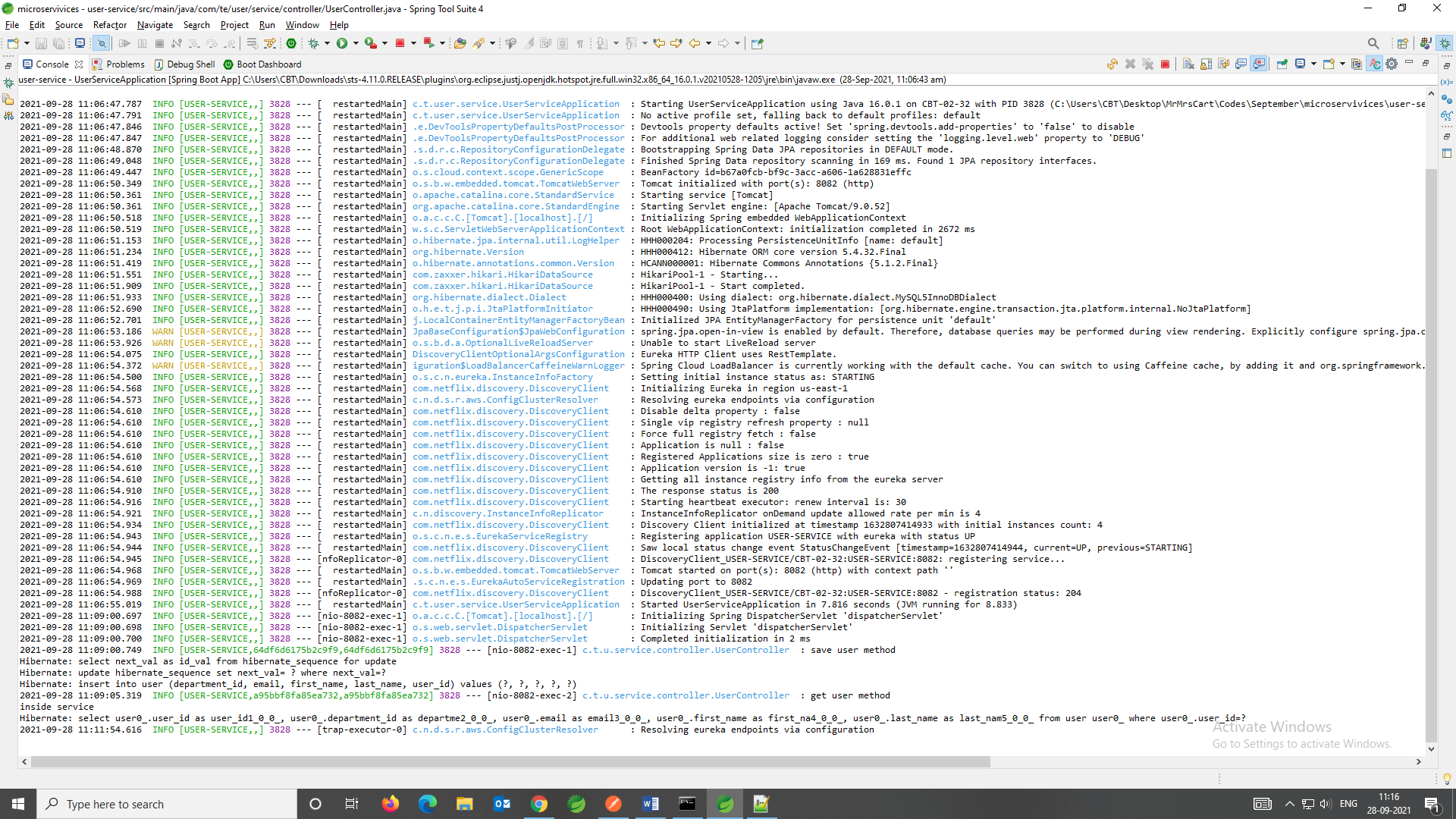
<dependency>

<groupId>org.springframework.cloud</groupId>

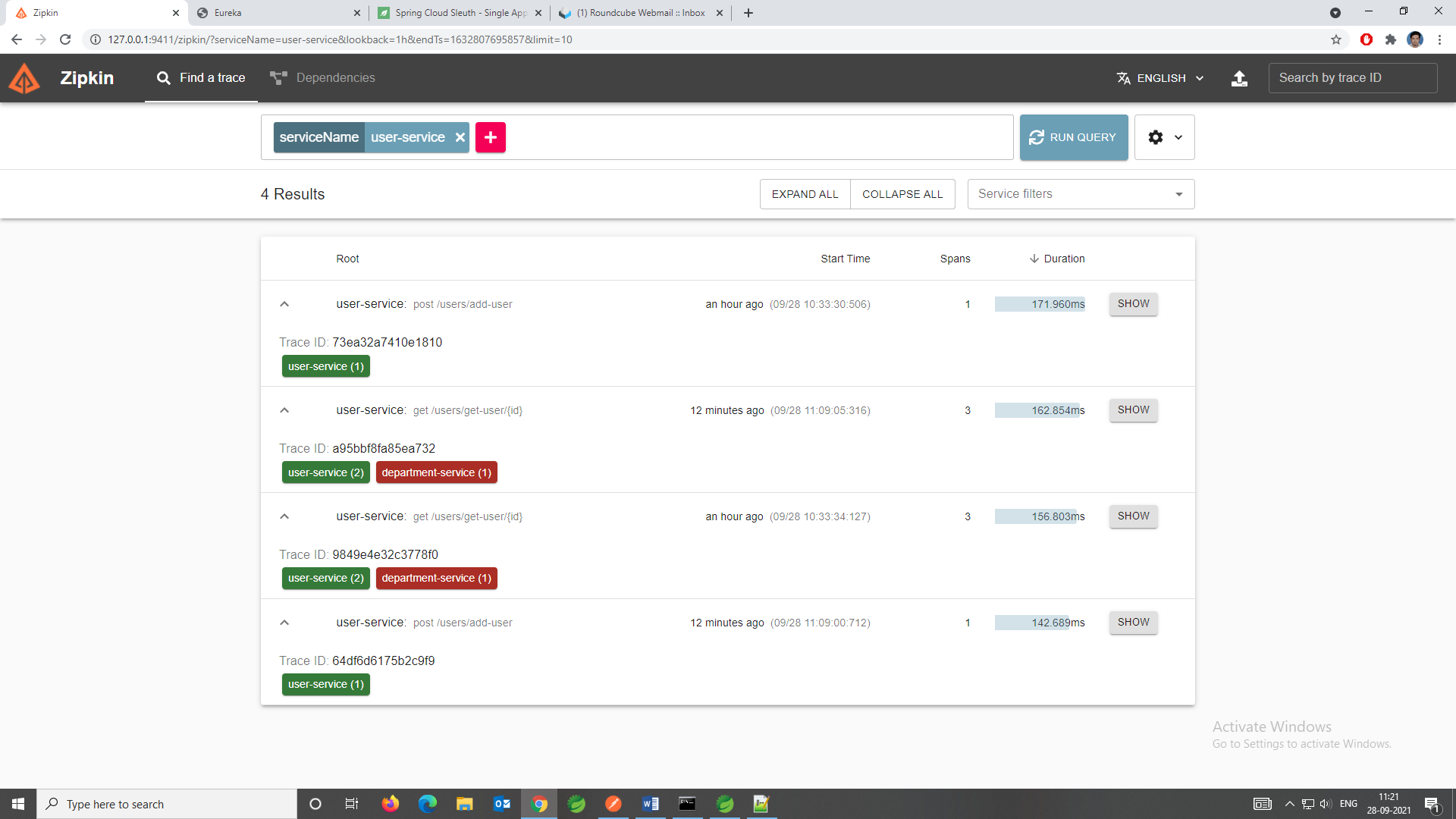
<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

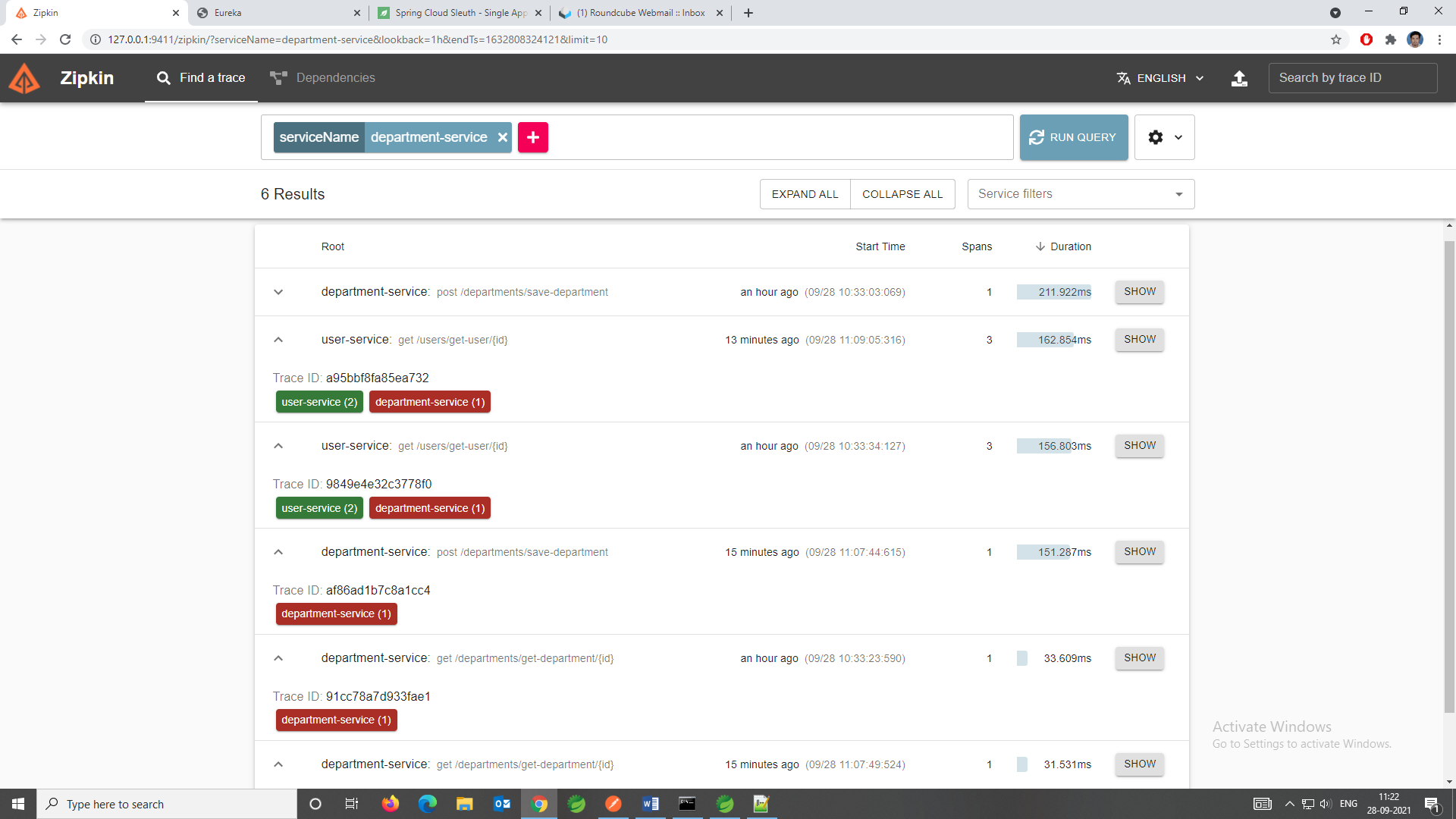
By adding this dependencies we can enable **zipkin-client** and **sleuth**.

* Now the department console will looks like
* 
* And the User console will looks like this
* 
* And in the zipkin server its looks like

**USER-SERVICE**



**DEPARTMENT-SERVICE**



**DEPENDENCIES**

