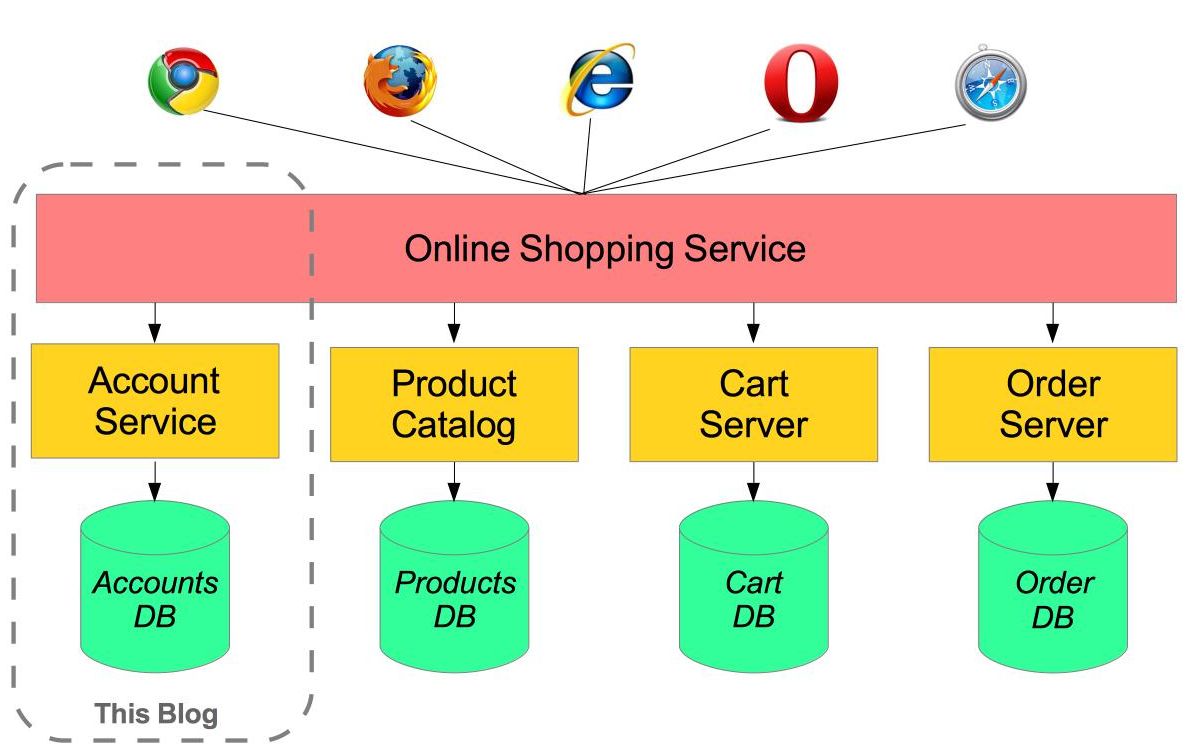
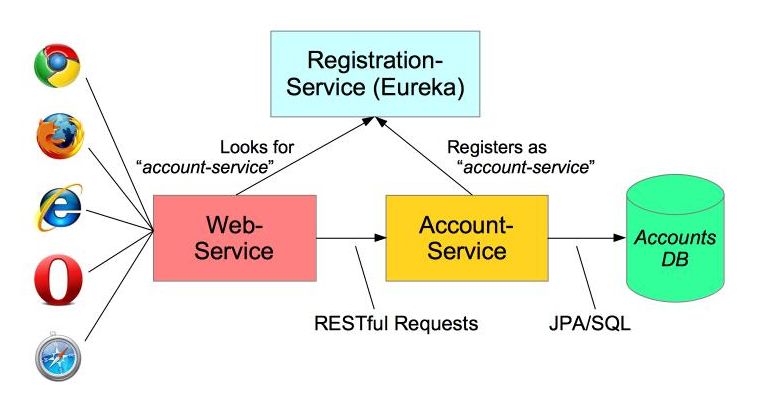
A simple example of setting up a microservices system using Spring, Spring Boot and Spring Cloud.

[Microservices](http://martinfowler.com/articles/microservices.html) 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.

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/shopping-system.jpg)

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

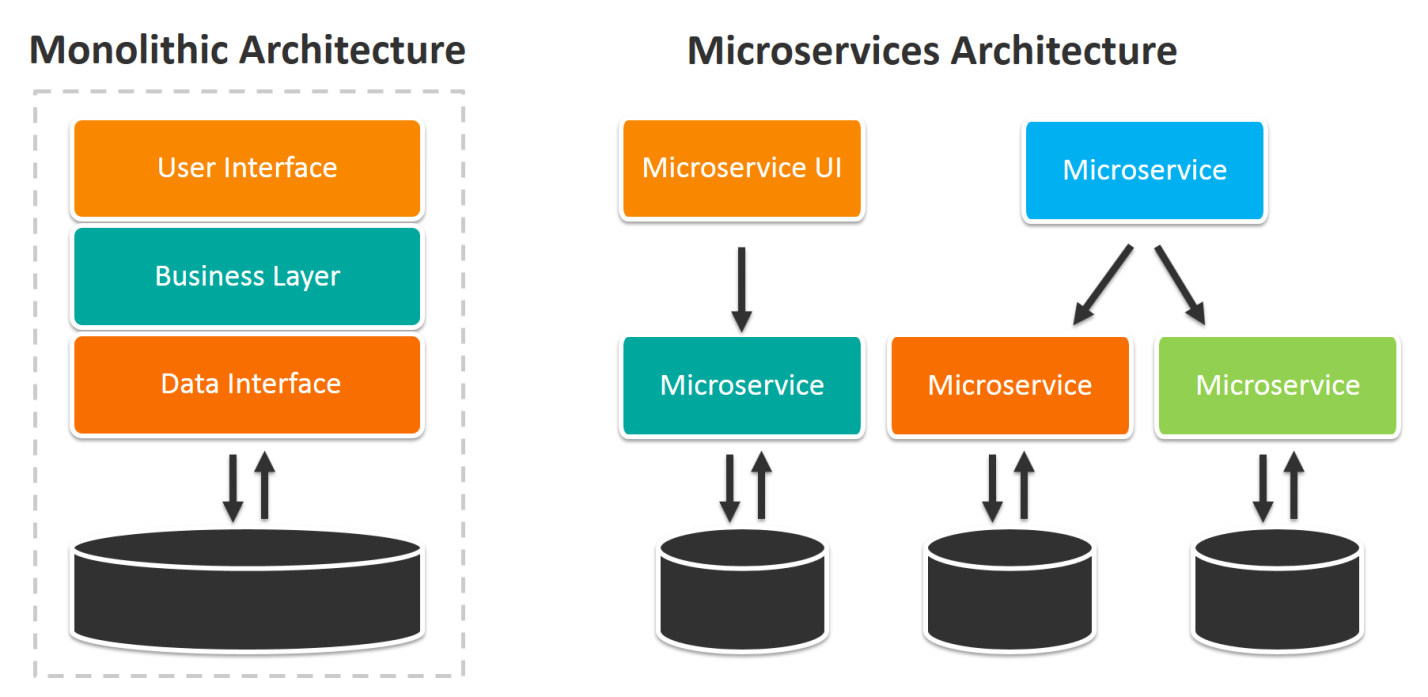
Inevitably there are a number of moving parts that you have to setup and configure to build such a system. How to get them working together is not obvious - you need to have good familiarity with Spring Boot since Spring Cloud leverages it heavily, several Netflix or other OSS projects are required and, of course, there is some Spring configuration “magic”!

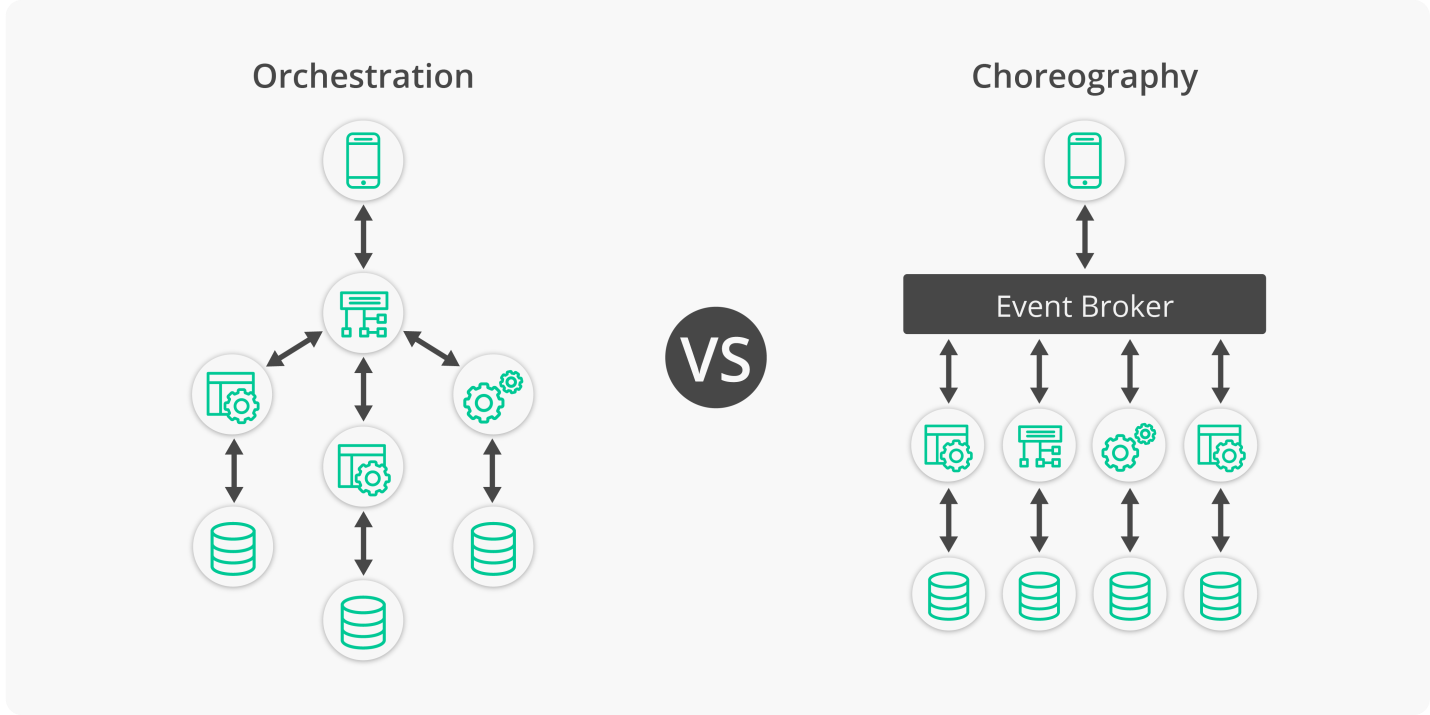
r[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/mini-system.jpg)

In this article I aim to clarify how things work by building the simplest possible system step-by-step. Therefore, I will only implement a small part of the big system - the user account service.

The Web-Application will make requests to the Account-Service microservice using a RESTful API. We will also need to add a discovery service – so the other processes can find each other.

Monolethic vs Microservices



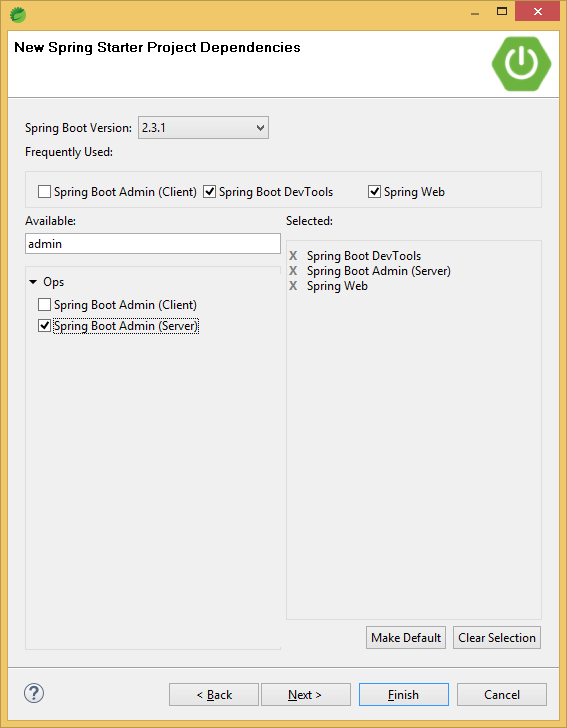


Demo1 : Admin Server-Client Application

1. Create 3 services as helloService(1111),WelcomeService(2222),GreetService(3333)



1. Create anew project as AdminServer with devTools, Springweb , Spring Admin Server and codecentric dependencies



<dependency>

<groupId>de.codecentric</groupId>

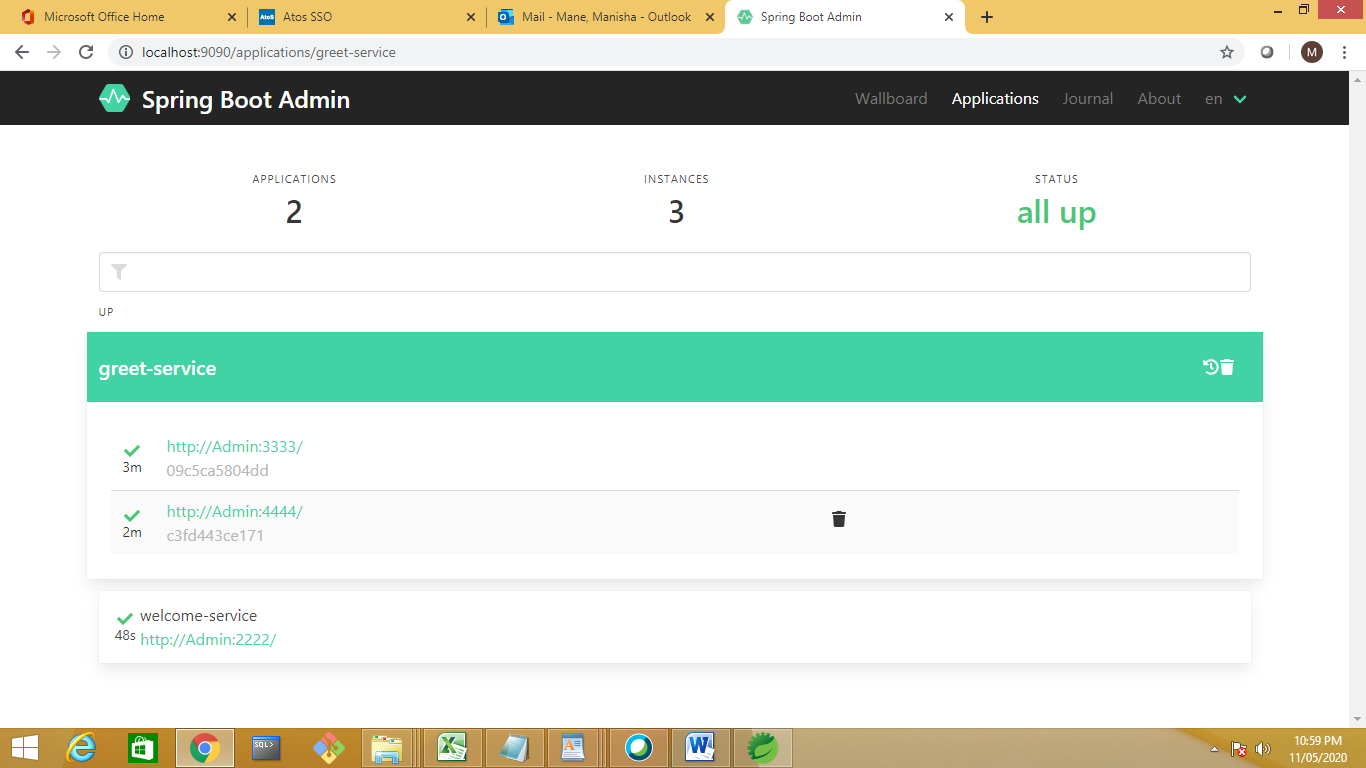
<artifactId>spring-boot-admin-server-ui</artifactId>

</dependency>

use @EnableAdminServer annotation.

Run Greet service on 2 diff ports

Comment the devtools dependencies for admin server and greet server

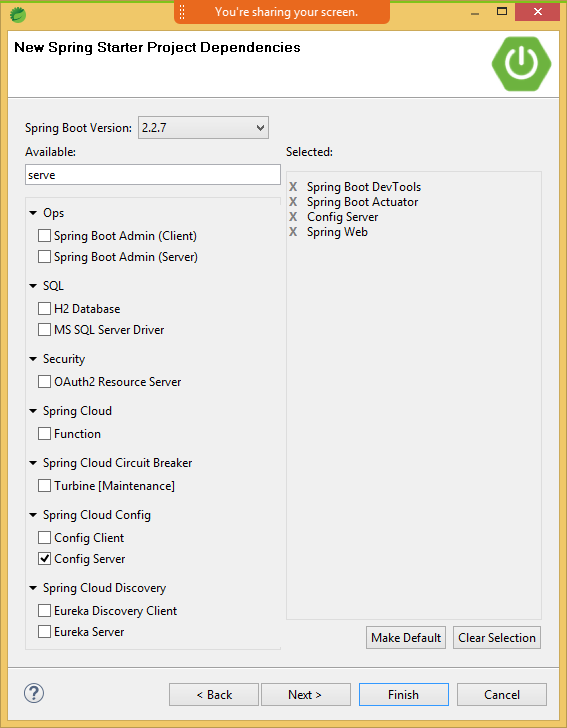


Day 2:

Spring CLoud Config Server with Local files

<https://www.youtube.com/watch?v=OytqzGq9hmE>

1. Create a spring application with spring cloud dependency



Add dependency for config server

Pom.xml

<dependencies>

<dependency>

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

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<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>

1. Enable the spring Cloud Server

**package** com.pradeep.springbootconfigserverfs;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.config.server.EnableConfigServer;

**import** org.springframework.context.annotation.Configuration;

@Configuration

@EnableConfigServer

@SpringBootApplication

**public** **class** SpringBootConfigServerFsApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootConfigServerFsApplication.**class**, args);

}

1. }

Defailt cloud server will read the configiuration form config folder

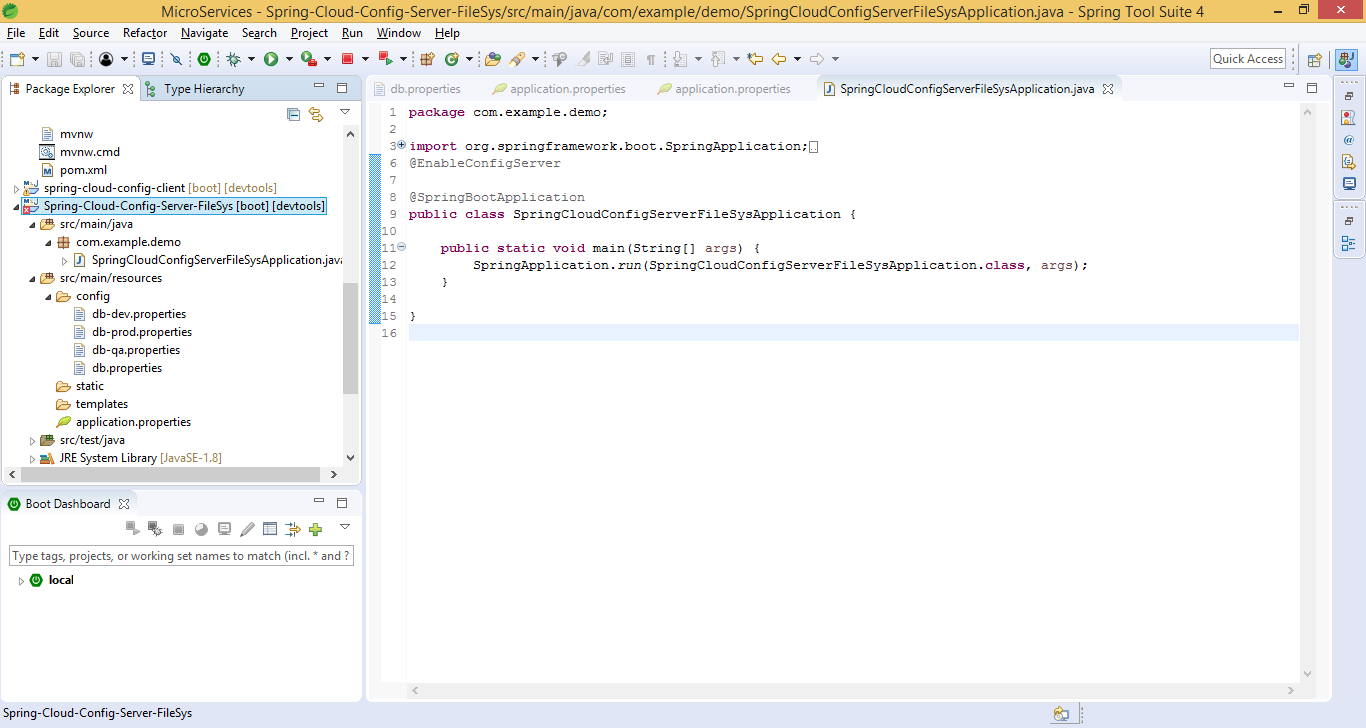
server.port=8181

server.servlet.context-path=/cloud-server

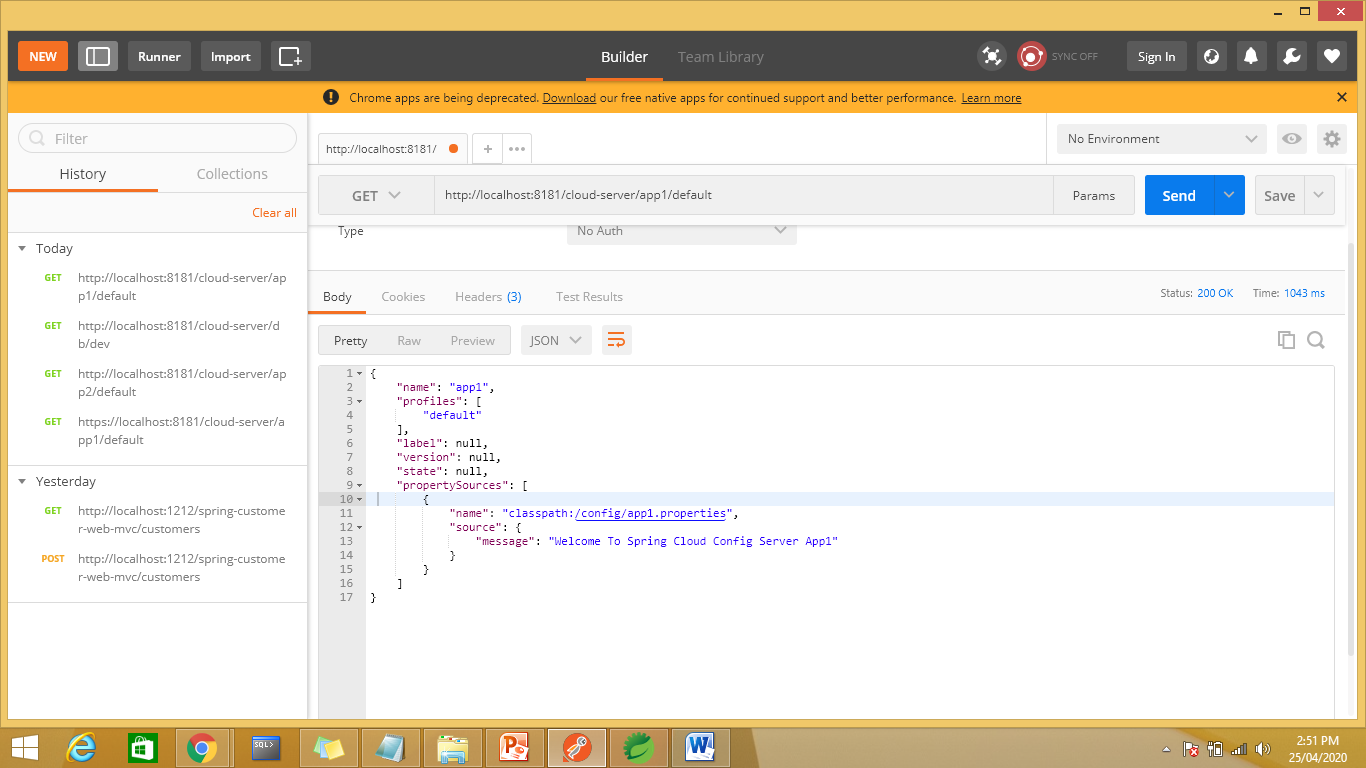
spring.profiles.active=native

spring.cloud.config.server.native.search-locations=classpath:/config,classpath:/app1

Create 2 new folders app1 and config



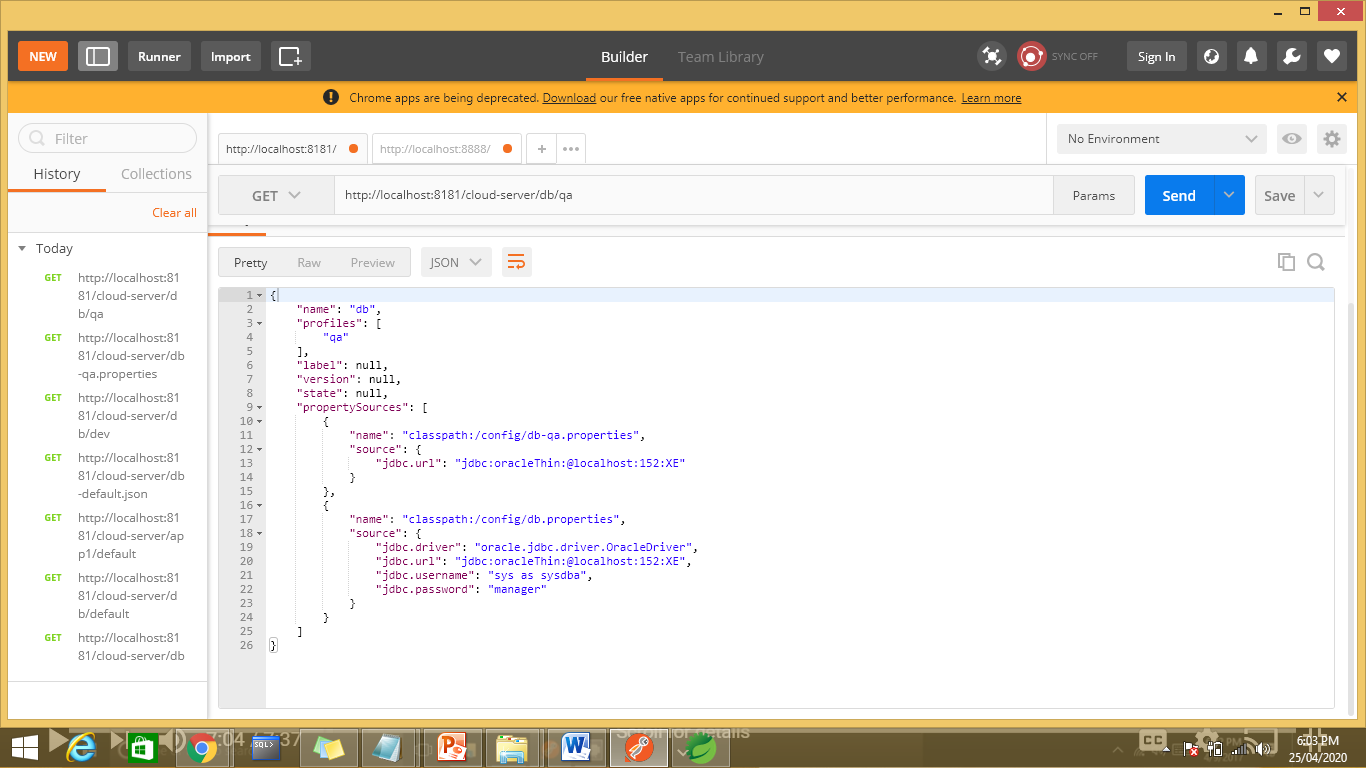
Run the application and the configuration from app1 and config folder



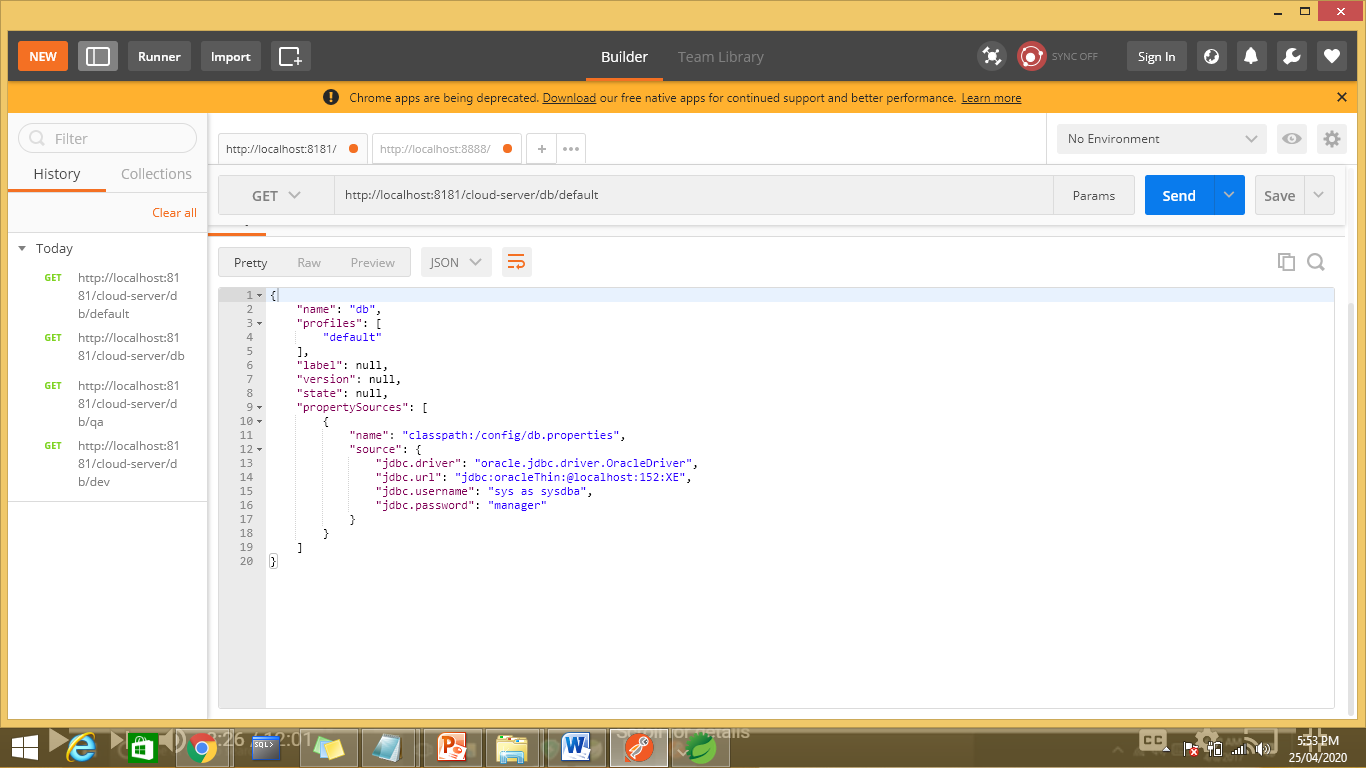
Db is the default profile add db-qa.properties with qa profile, db-dev.properties with dev profile and db-prod.properties with prod profile

In qa profile add only url

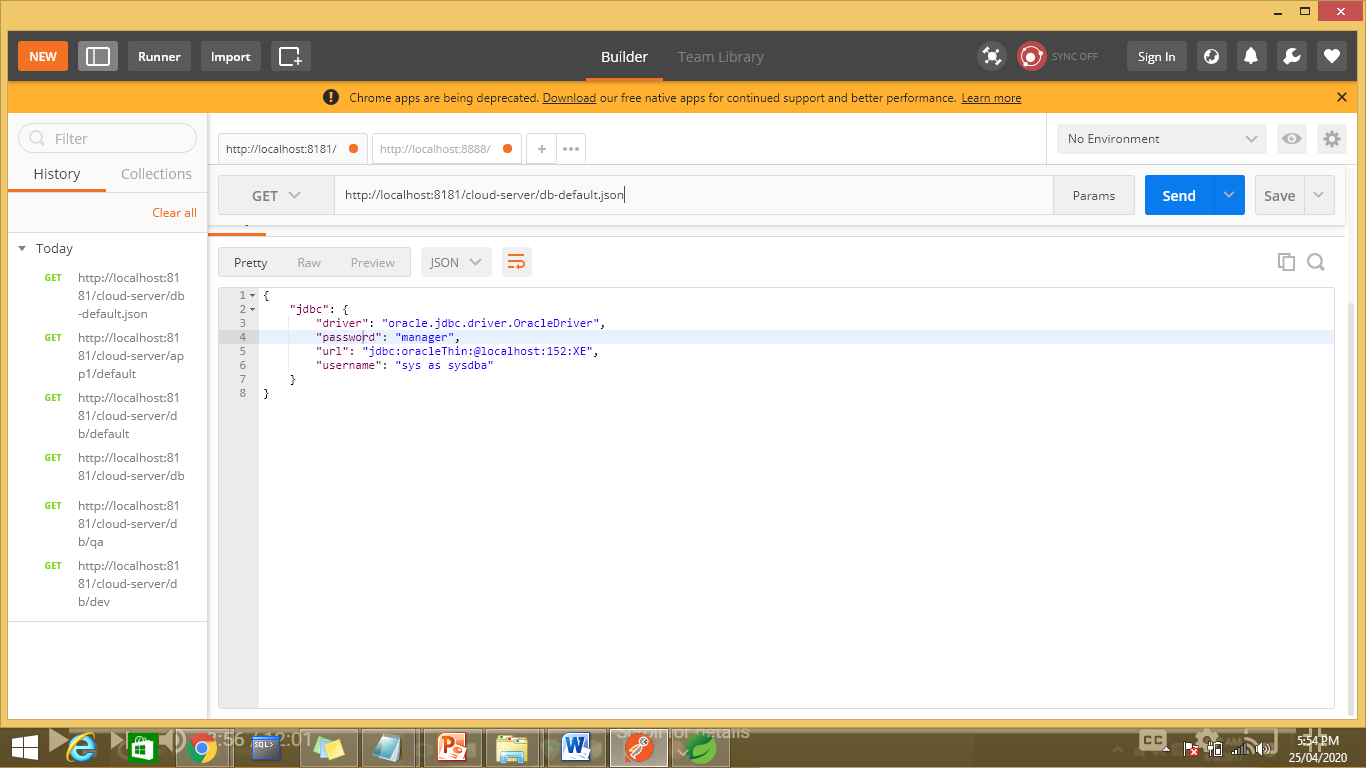
jdbc.url=jdbc:oracleThin:@localhost:152:XE

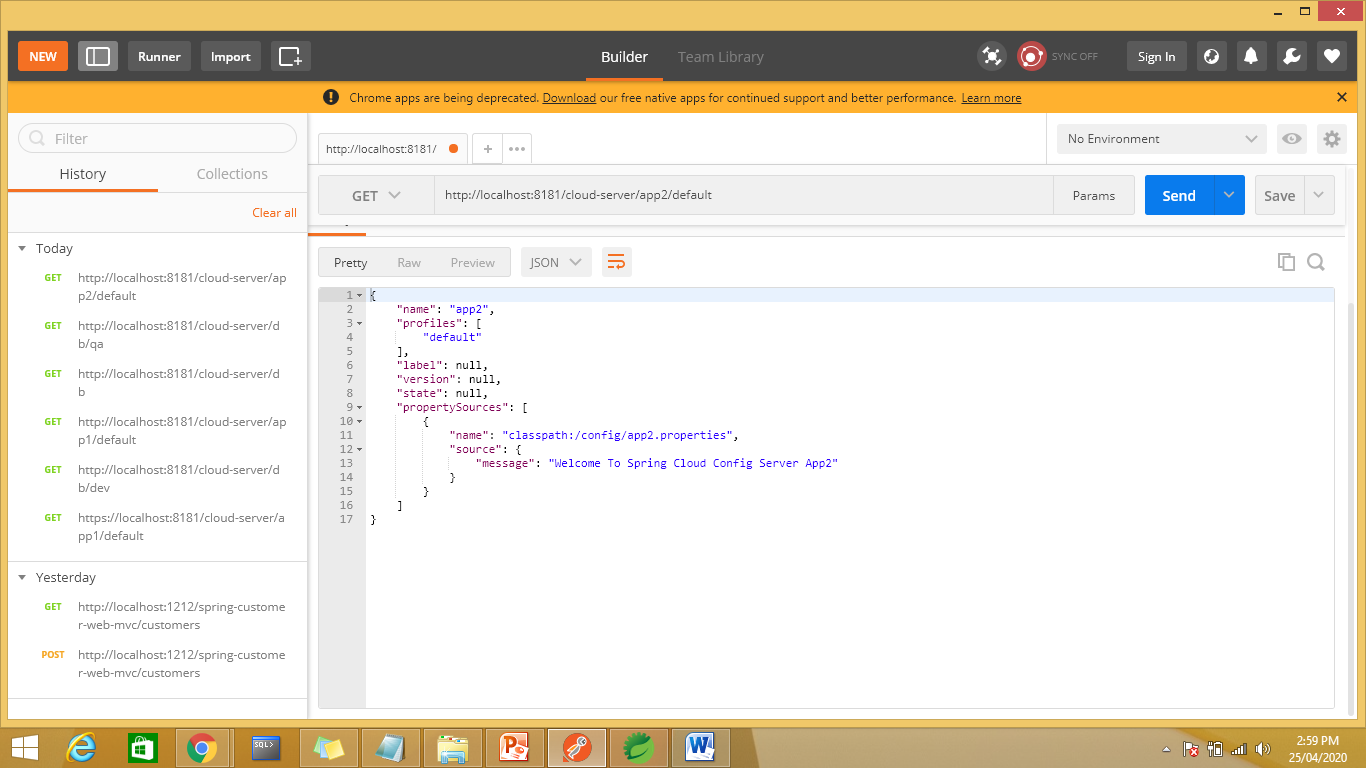




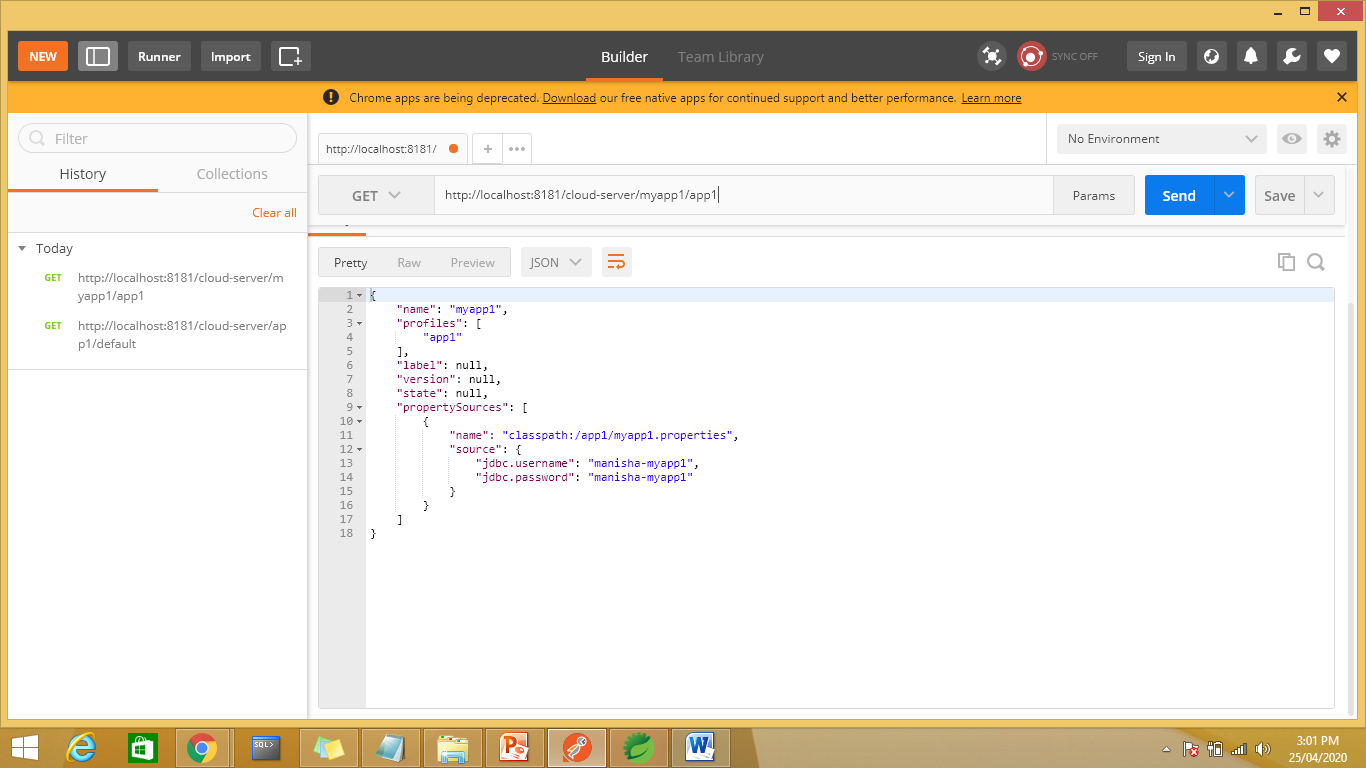


To read the json properties





To read the properties form app\myapp1.properties



To read data from external files

Create a folder d:\2019\config app

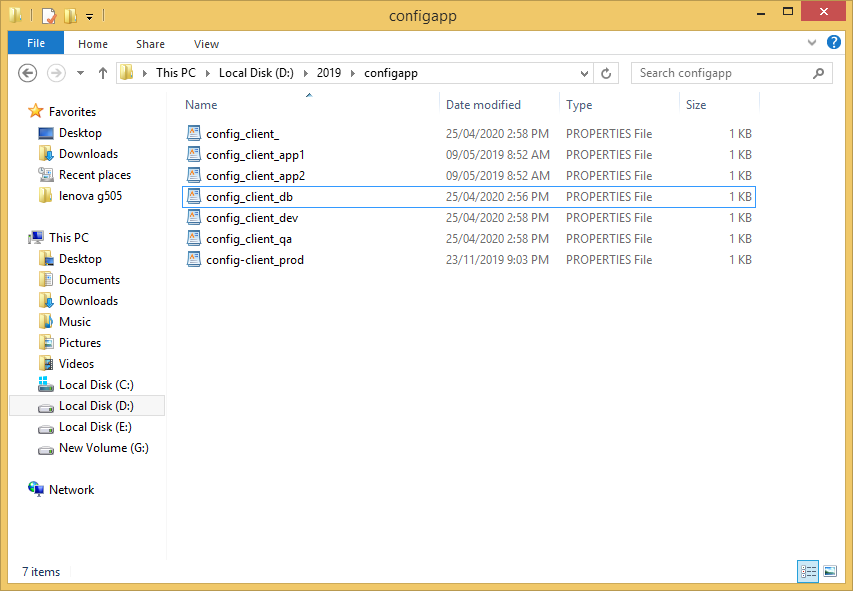
Add reference in application.properties file

server.port=8181

server.servlet.context-path=/cloud-server

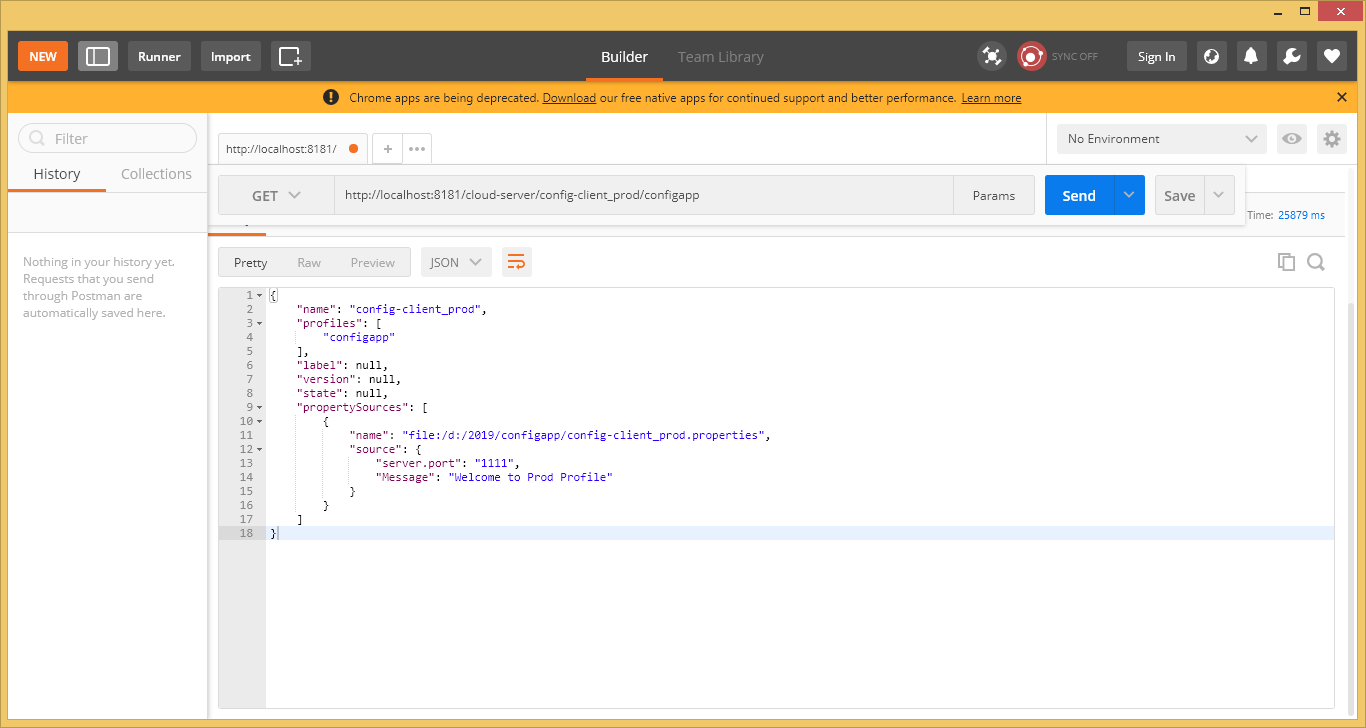
spring.profiles.active=native

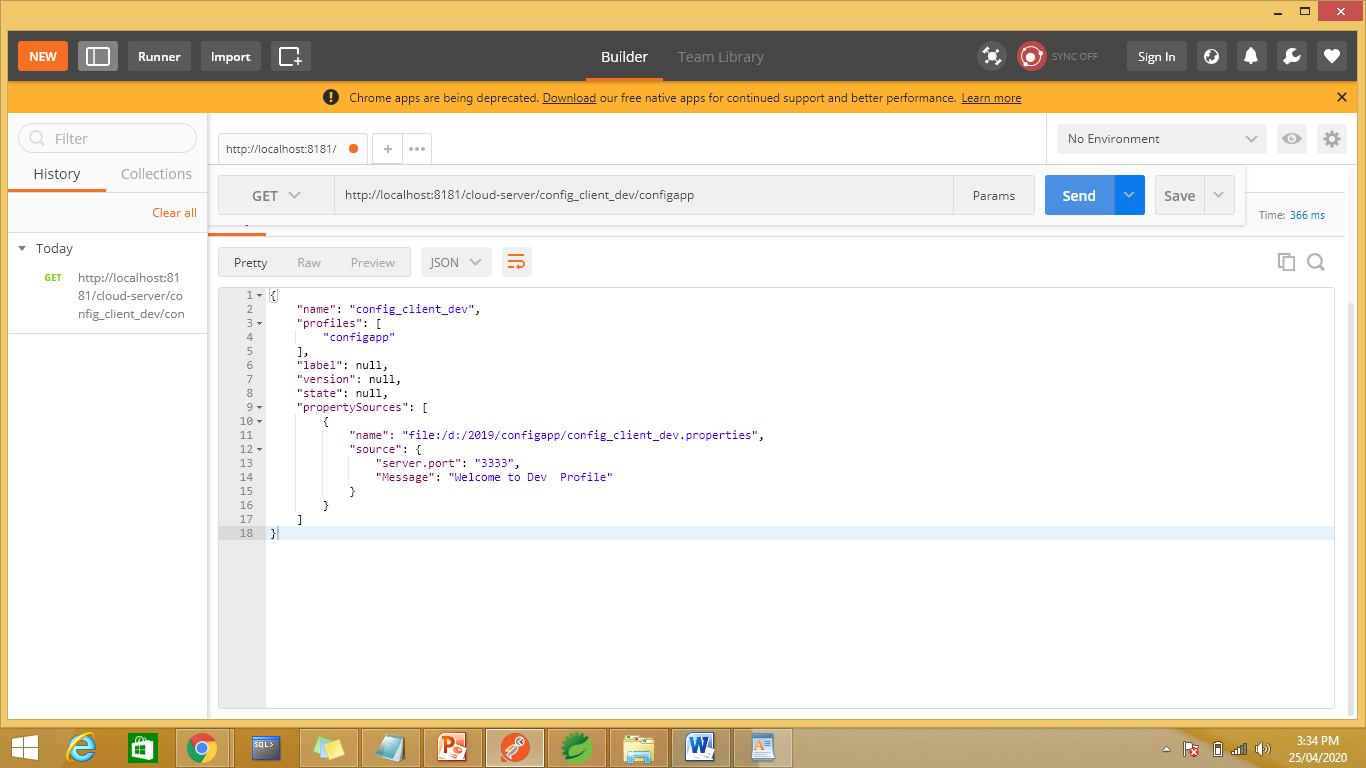
spring.cloud.config.server.native.search-locations=classpath:/config,classpath:/app1,file:/d:/2019/configapp



To run in portman

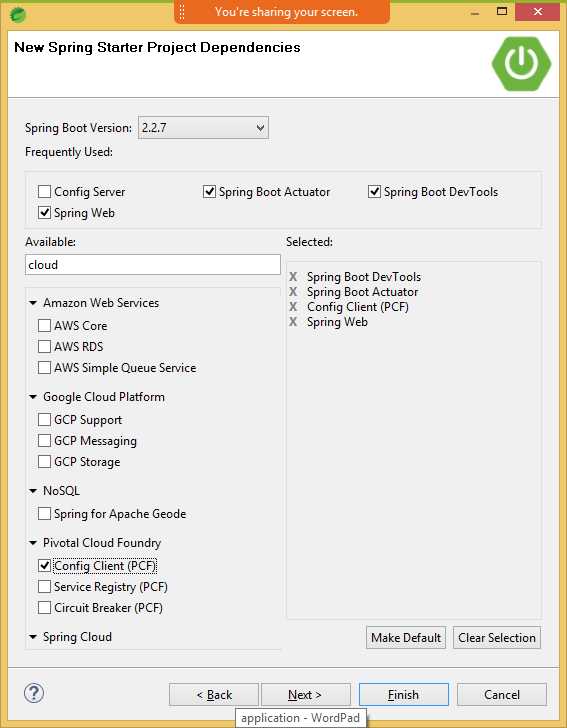






Spring cloud config client:

Config client and web dependencies



<https://www.youtube.com/watch?v=IOEHV-7gdpI>

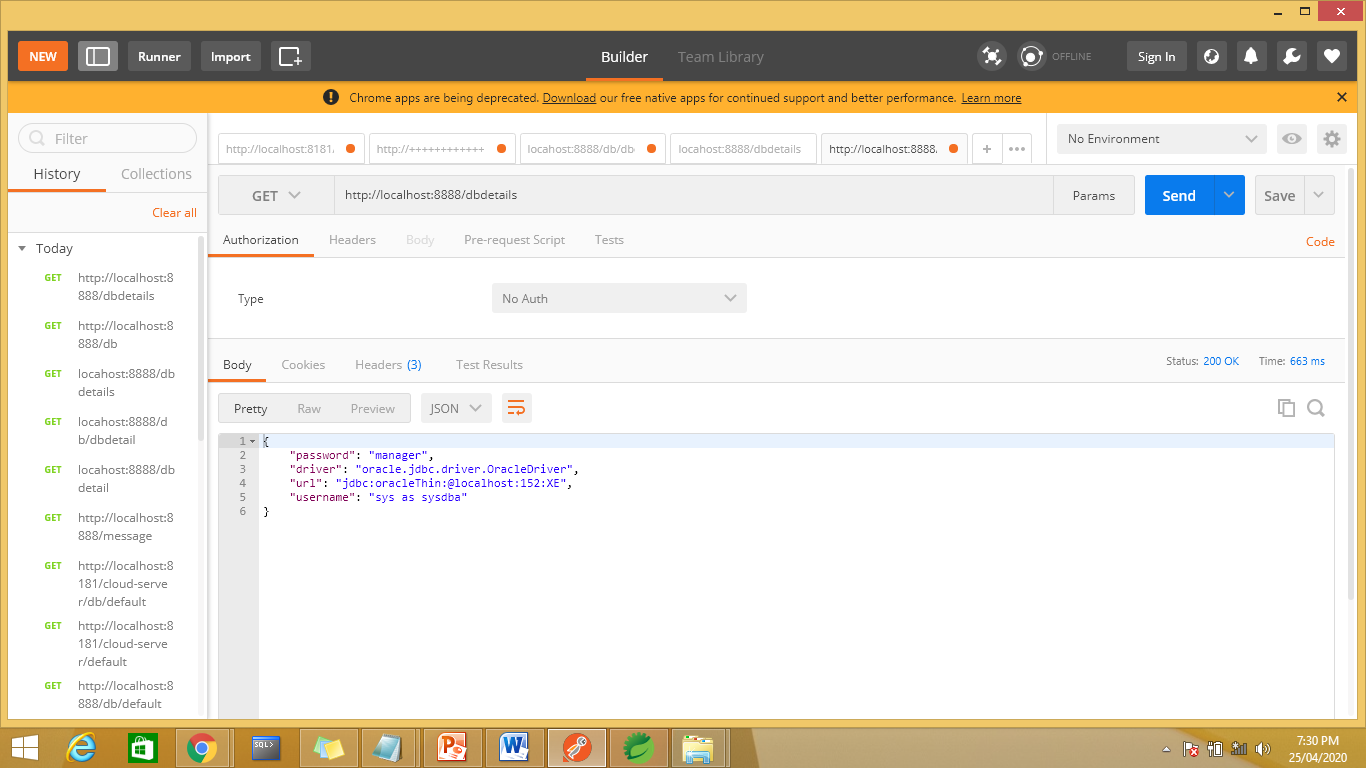
Add dependencies

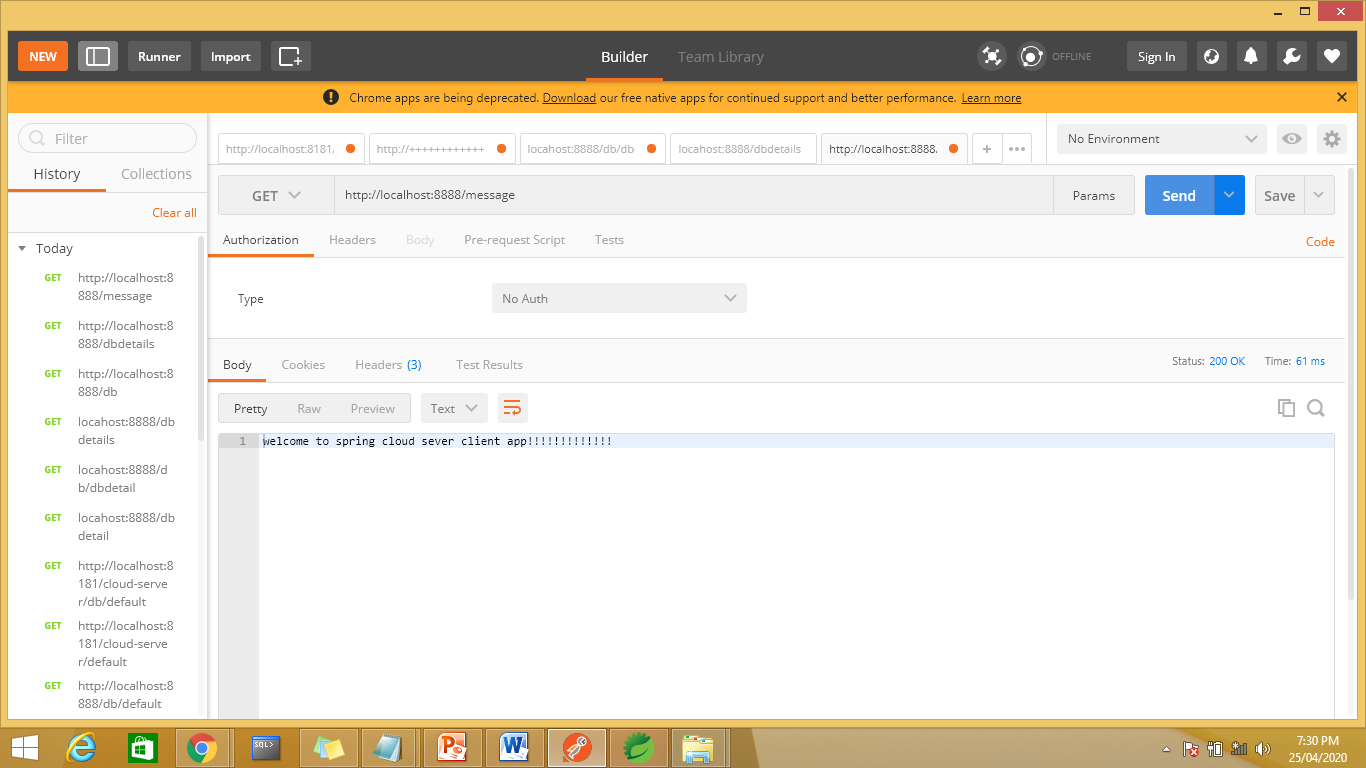
<dependency>

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

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

</dependency>





Spring Cloud server config with github

<https://www.youtube.com/watch?v=MoaF_G4qYW4>

server.port=8181

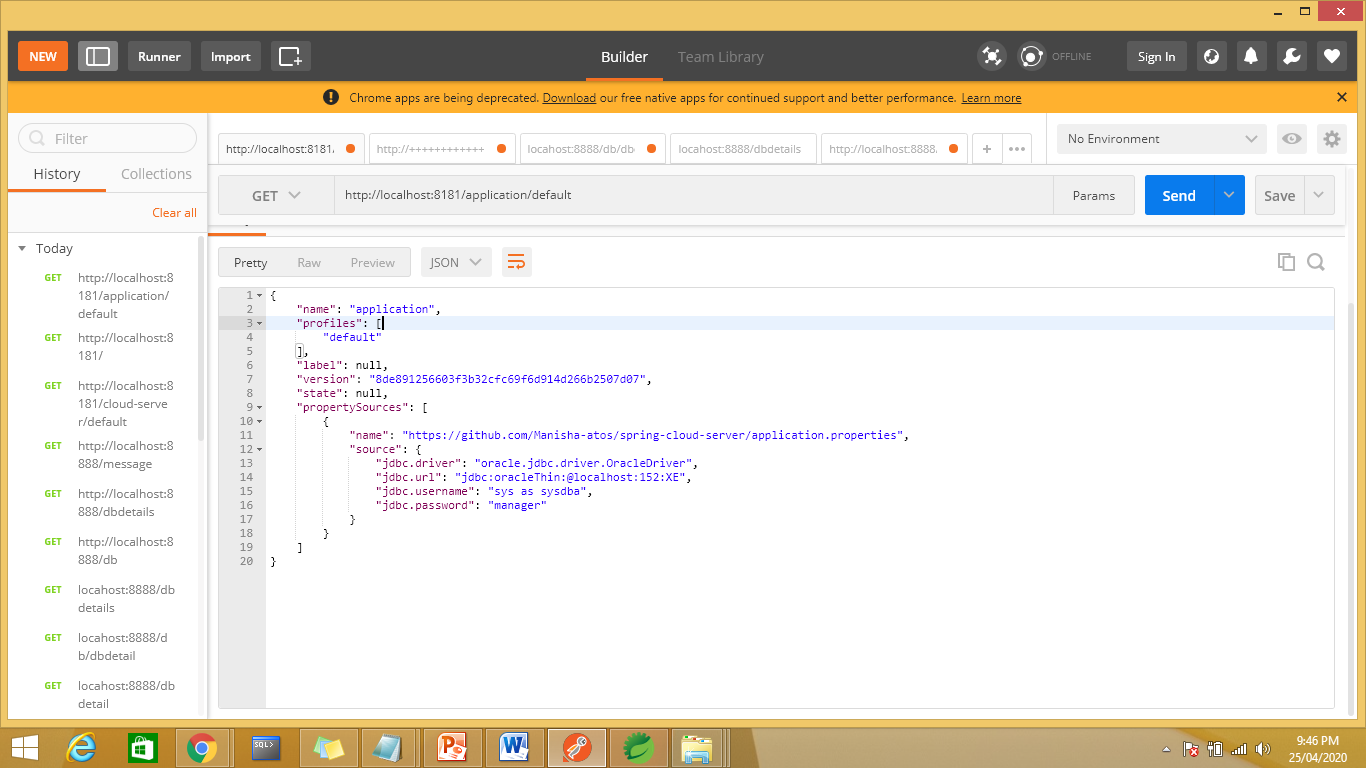
#server.servlet.context-path=/cloud-server

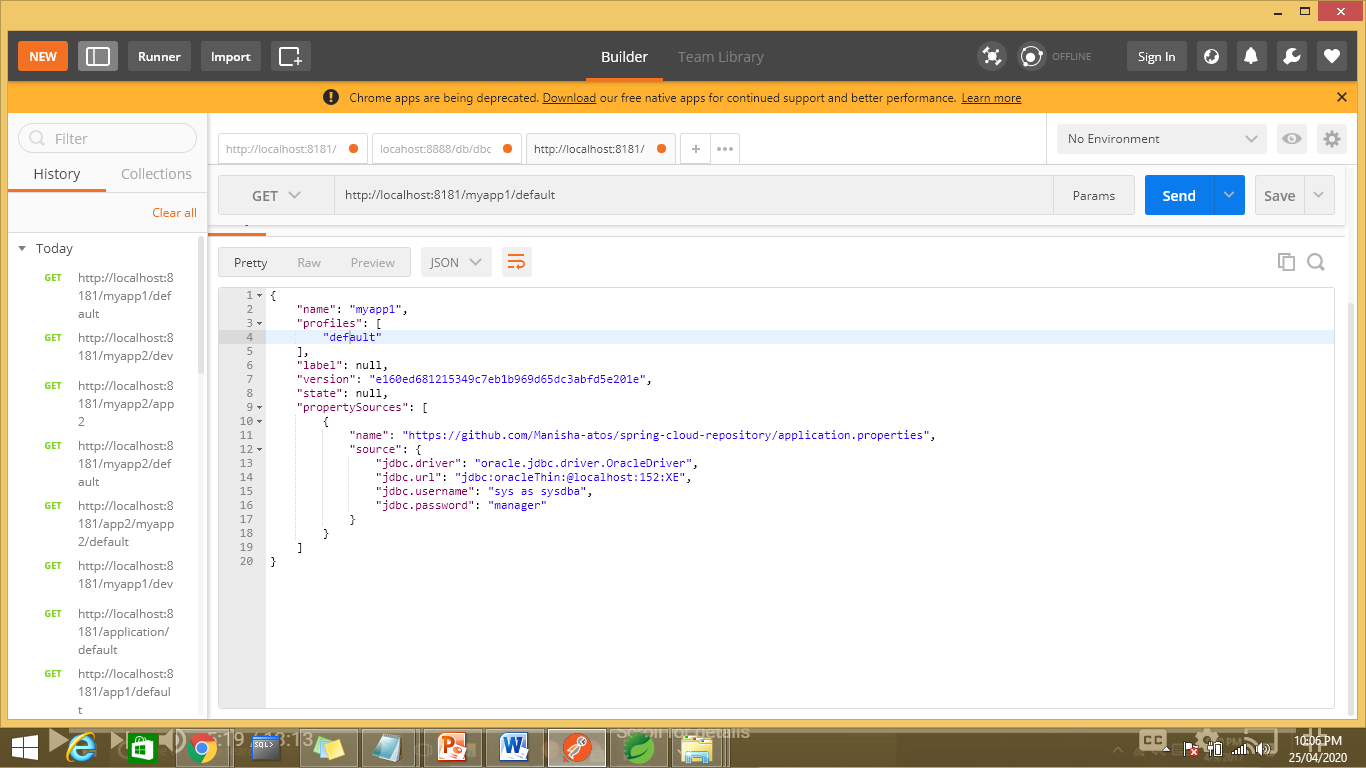
spring.cloud.config.server.git.uri=https://github.com/Manisha-atos/spring-cloud-server

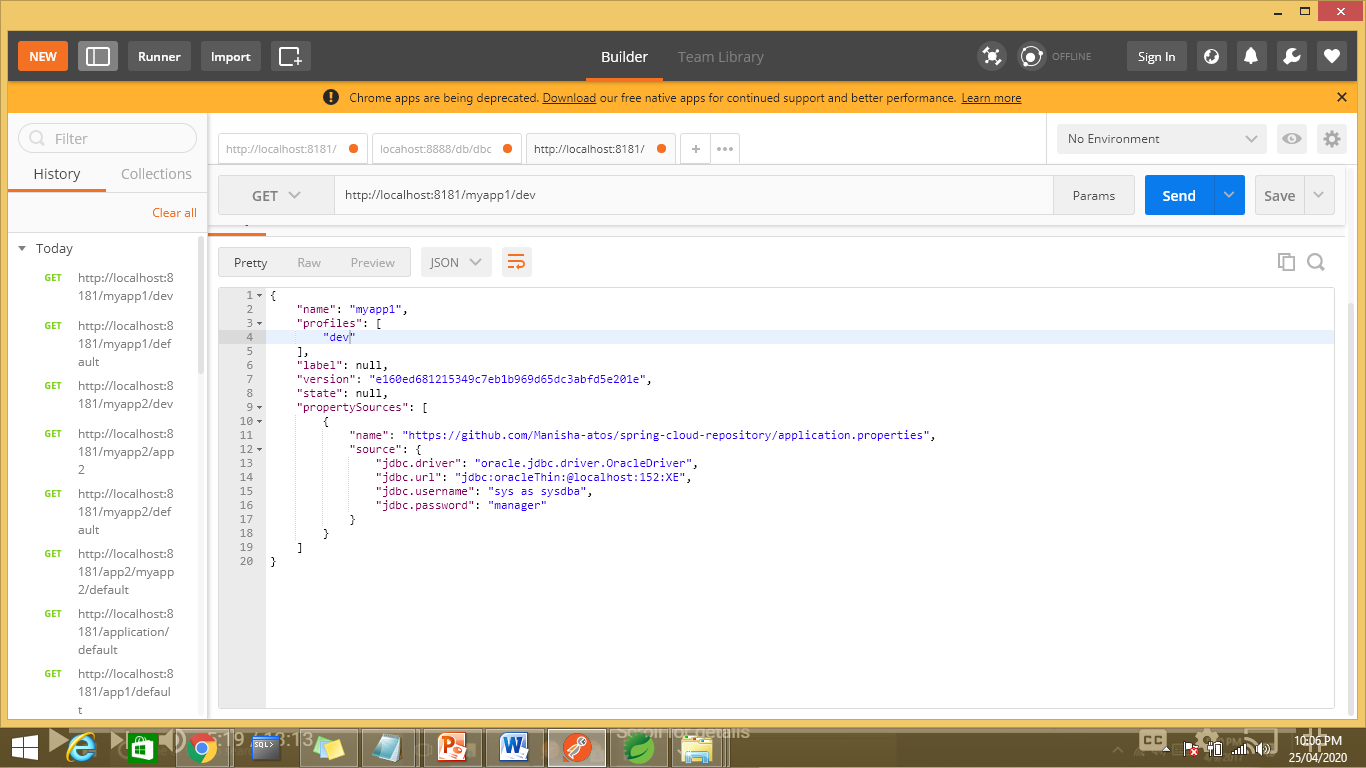
#provide the github credentials if repoitory is private or else no need

#spring.cloud.config.server.git.password=manisha

#spring.cloud.config.server.git.username=manisha







Not reading the .properties file from app1 and app2 folders from repository so add the search path

serverport:8181

#server.servlet.context-path=/cloud-server

spring.cloud.config.server.git.uri=https://github.com/Manisha-atos/spring-cloud-repository

#provide the github credentials if repoitory is private or else no need

#spring.cloud.config.server.git.password=manisha

#spring.cloud.config.server.git.username=manisha

search.paths=/app\*

#spring.cloud.config.server.git.search-paths=app\*

spring.cloud.config.server.git.search-paths=app1

spring.cloud.config.server.git.cloneOnStart=true

#Disable security of the Management endpoint

management.security.enabled=false

or application.yml

server:

port: 8181

spring:

cloud:

config:

server:

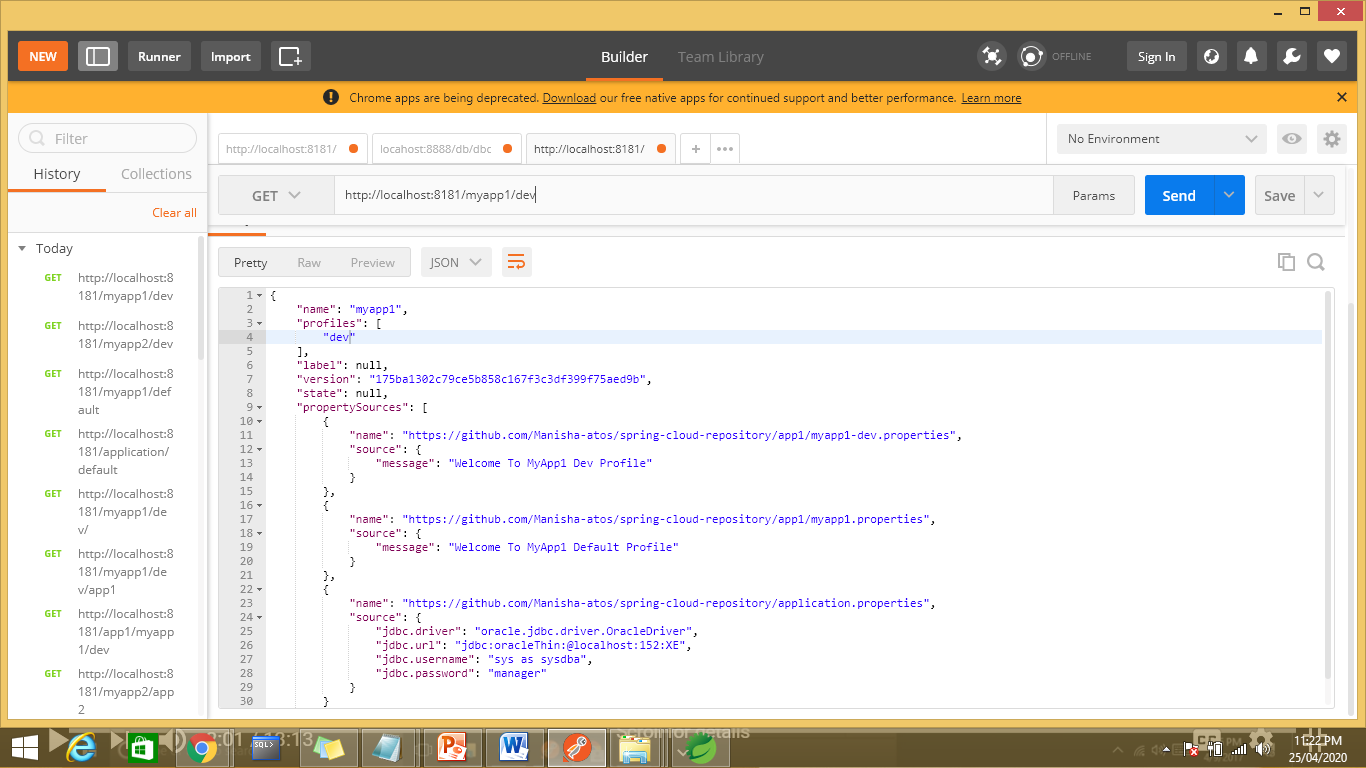
git:

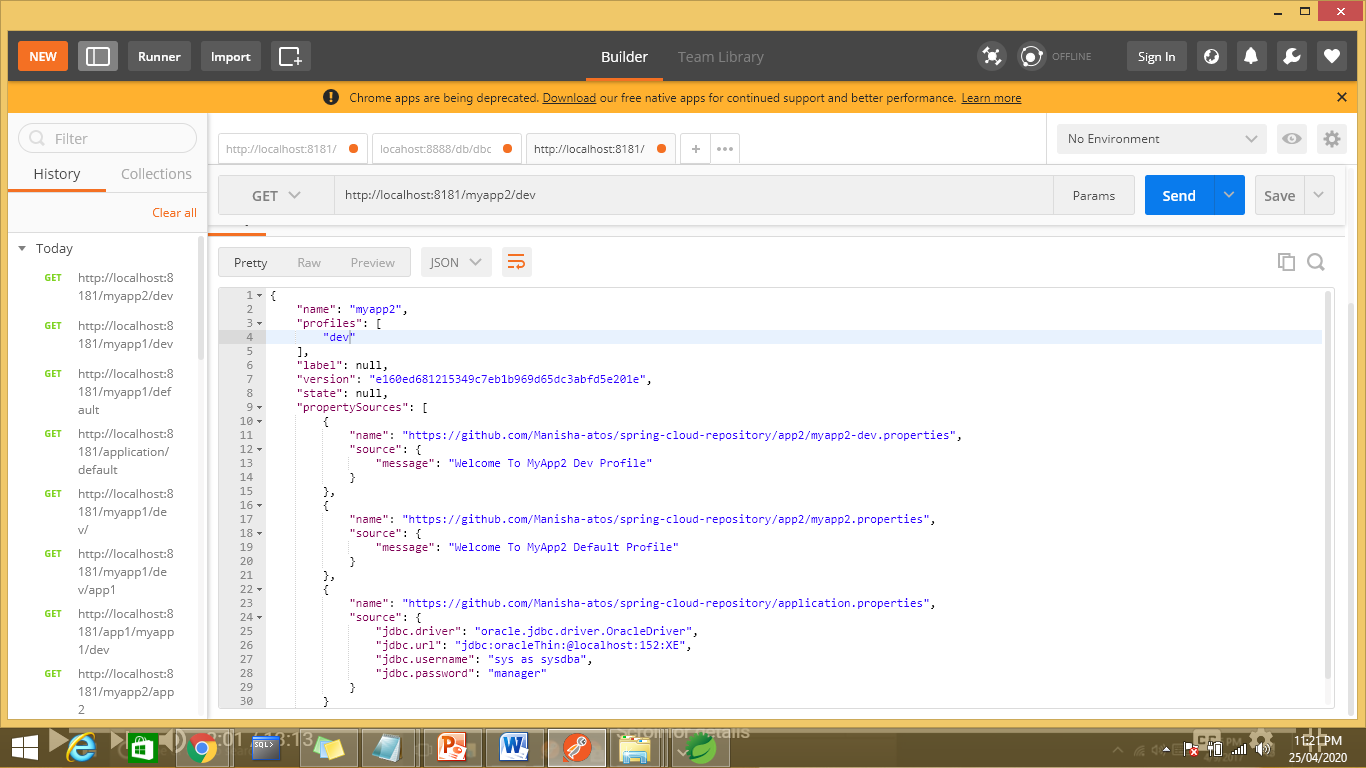
uri: https://github.com/Manisha-atos/spring-cloud-repository

search-paths:

- app\*

Output





Spring cloud config client

Application.properties

#spring.application.name=dbspring.application.name=config-client

#spring.application.name=db

#spring.profiles.active=qa

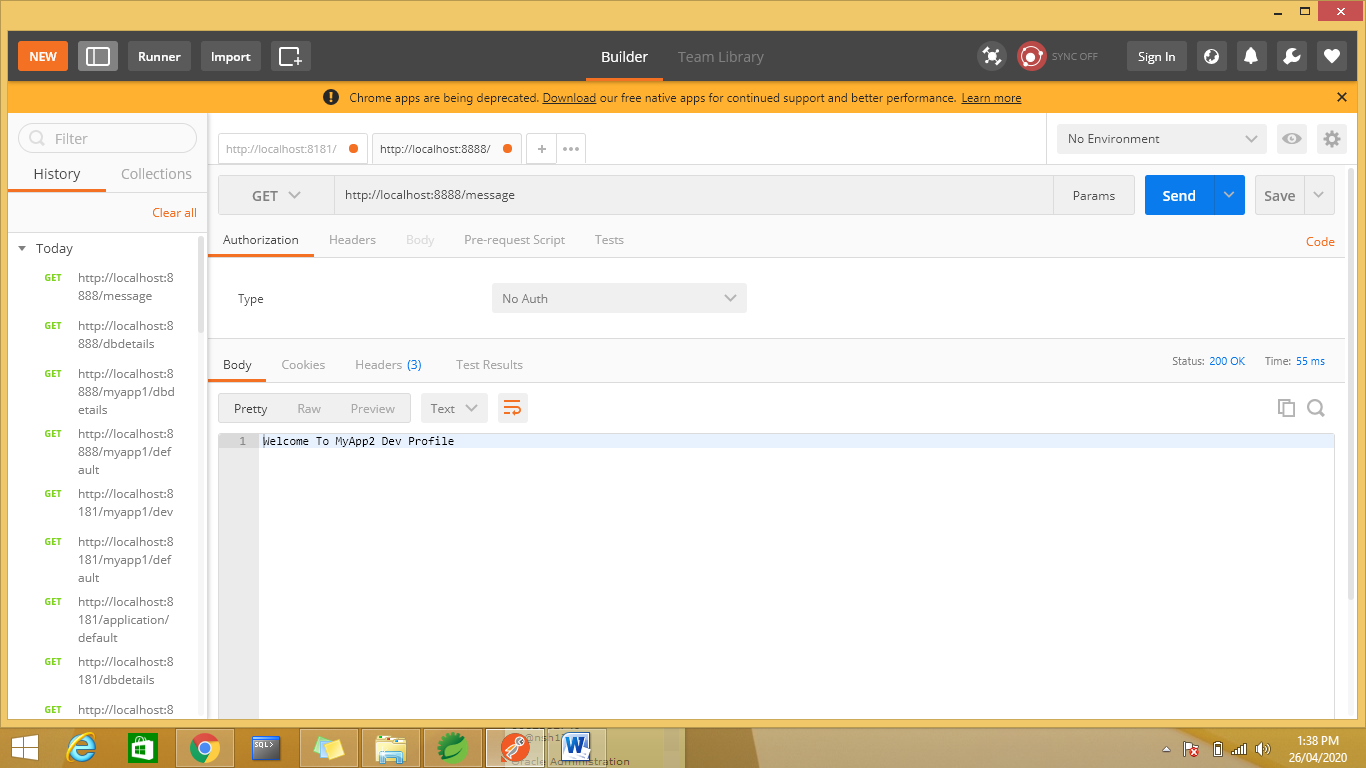
server.port=8888

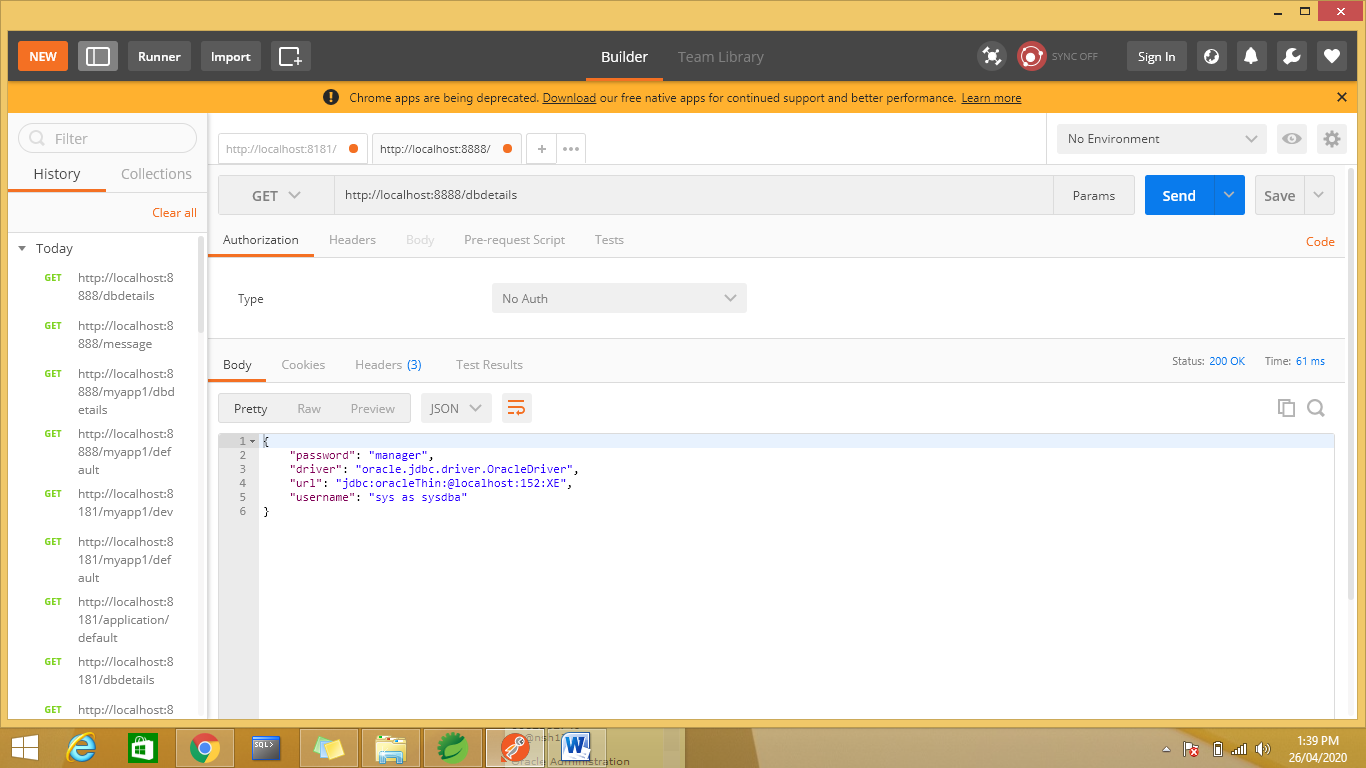
spring.application.name=myapp2

#spring.application.name=myapp2

spring.profiles.active=dev

spring.cloud.config.uri=http://localhost:8181





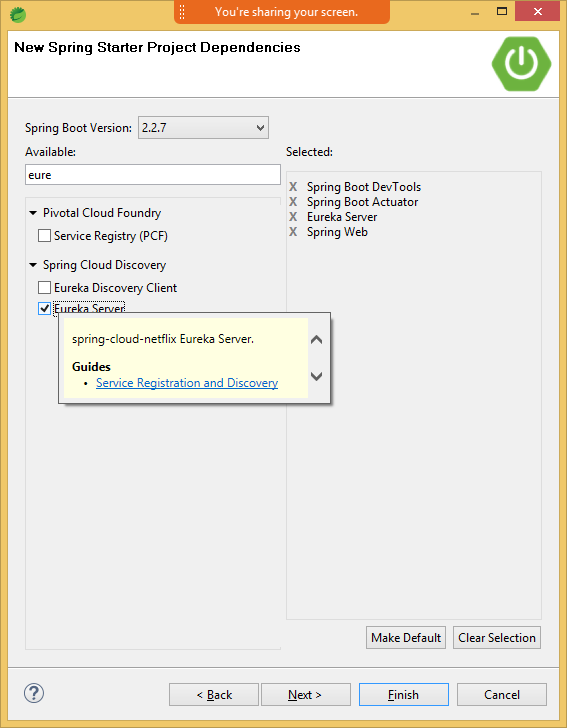
Day3:

Microservices with Eureka Server and Client

**Eureka** is a REST (Representational State Transfer) based service that is primarily **used** in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier **servers**

Eureka server runs on 8761 port number by default

Behavioral



Project—eureka discovery and eureka server

<dependency>

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

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

ES provides service registry and discobvery

@SpringBootApplication

@EnableEurekaServer

public class EurekaserverApplication {

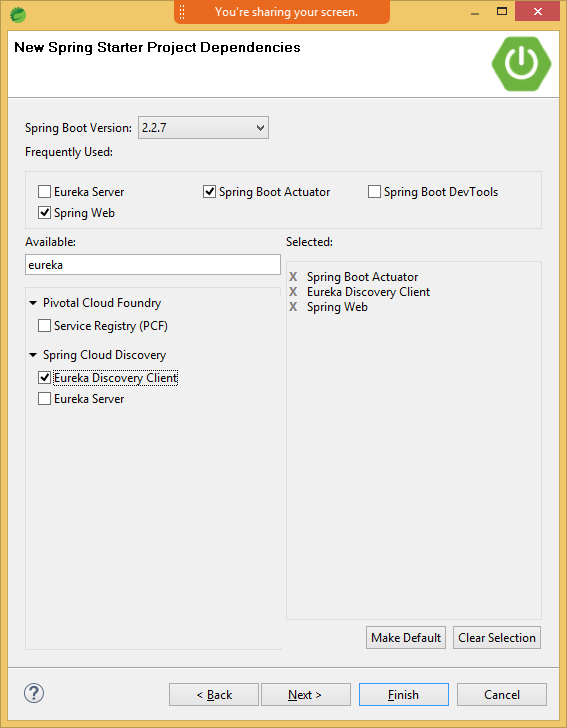
public static void main(String[] args) {

SpringApplication.run(EurekaserverApplication.class, args);

}

}

Create a Cart-Service and Order-Service



eureka.client.registerWithEureka=false // will not do any client registration

eureka.client.registerWithEureka=true// will do client registration

eureka.client.fetchRegistry=true

application will start at 8761 port

open th server in chrome will show the eureka server ui

Add a client App

Add @EnableEurekaCLinet or @EnbaleDiscoveyClinet annotation

**1. Where Zuul fits in microservices ecosystem?**

A common problem, when building microservices, is to provide a unique gateway to the client applications of your system. The fact that your services are split into small microservices apps that shouldn’t be visible to users otherwise it may result in substantial development/maintenance efforts. Also there are scenarios when whole ecosystem network traffic may be passing through a single point which could impact the performance of the cluster.

To solve this problem, Netflix (a major adopter of microservices) created and open-sourced its **Zuul proxy server** and later Spring under Pivotal has adapted this in its **spring cloud stack** and enabled us to use zuul easily and effectively with just few simple steps.

Interaction between Eureka , Service and Zuul

When we move to the cloud platform we can run our services as container in cloud

When we run services s container we can not predict the IP address of service all IP add are dynamic

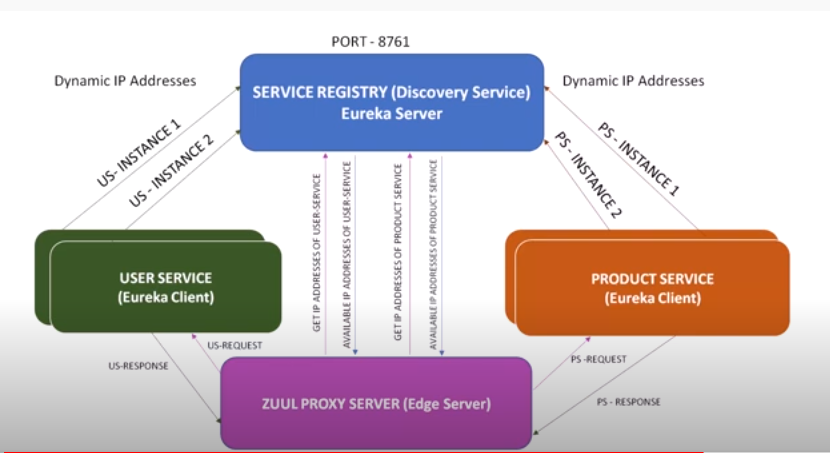
In this case eureka server helps to forward the request to the service.

Service registry is running on 8761 port no

2 services (eureka client ) user and product are registered with Eureka Server.

2 instances / services are running on dynamic ip address

To balance the load Zuul Server (Edge Server) come into picture.



Request🡪Zuul Proxy

Zuul Proxy🡪Eureka

Eureka 🡪send available ip address of the product service (user service) to zuul proxy server

Zuul proxy server does the load balancing and will send the request to the

s://howtodoinjava.com/spring-cloud/spring-cloud-api-gateway-zuul/

Demo:

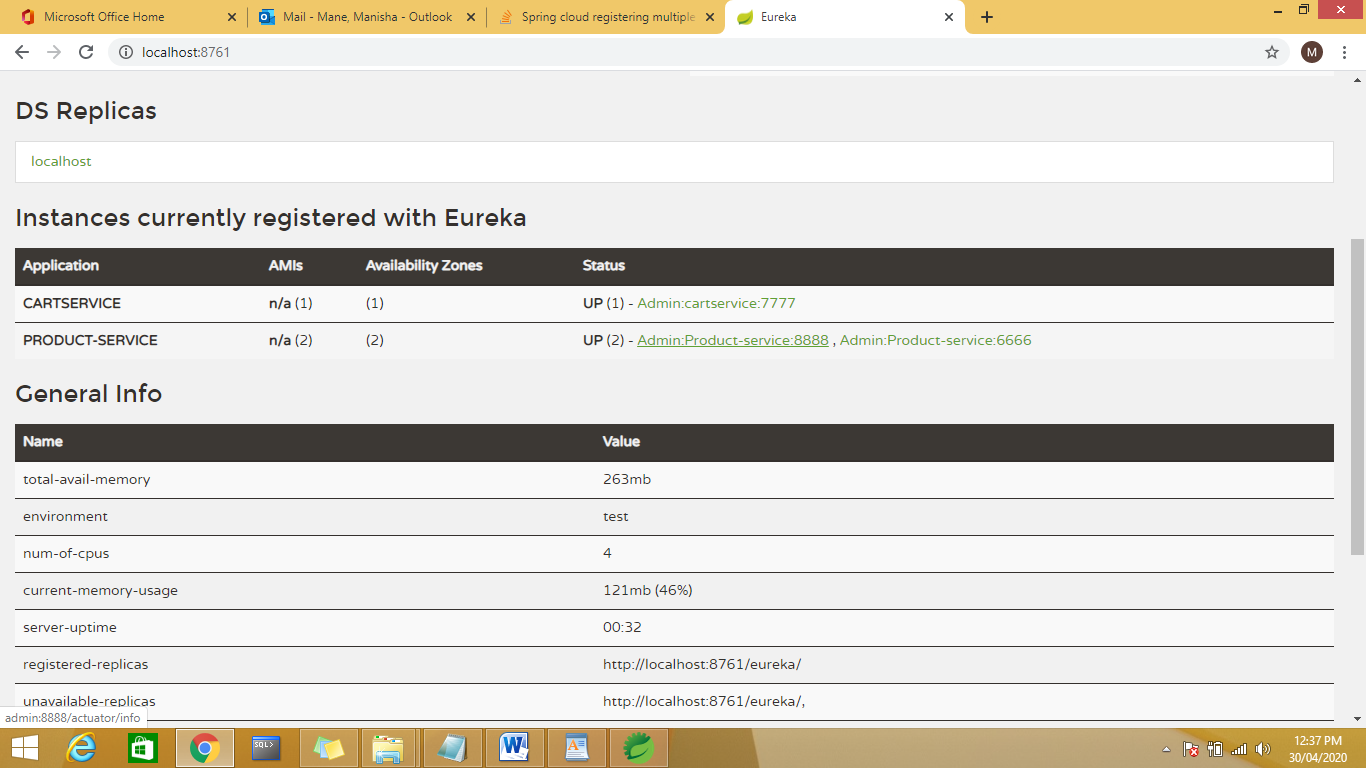
Create 2 services as product service and crat service .

Create Eureka server app

Create API Gateway zuul project

To create multiple instances Start the service by using diffetent ports , change the port in the application.xml file and run it again.

To create multiple instance don’t add devTools dependency



Zuul Proxy:--eureka client and zuul proxy dependencies

Routing

Service name:

2 ways to do routing:

url based or service based routing can be

when we are having multiple instances then use service name based routing

and for single instance use url based routing

Carts Service

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@EnableEurekaClient

@SpringBootApplication

public class CartServiceApplication {

public static void main(String[] args) {

SpringApplication.run(CartServiceApplication.class, args);

}

public CartServiceApplication()

{

System.out.println("CartServiceApplication..");

}

}

**package** com.example.demo;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** CartController {

@RequestMapping("/getCart/{name}")

**public** String getProducttName(@PathVariable(name = "name") String name)

{

**return** "Cart Details <strong style=\"color: red;\">" + name + " </strong>";

}

@RequestMapping(value = "/")

**public** String home() {

**return** "Eureka Cart application";

}

}

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

eureka.client.instance.preferIpAddress=true

eureka.client.register-with-eureka=true

eureka.server.peer-node-read-timeout-ms=300000

spring.application.name=Cart-Service

server.port=7777

------------------------Cart Service--------------------

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@EnableEurekaClient

@SpringBootApplication

public class CartServiceApplication {

public static void main(String[] args) {

SpringApplication.run(CartServiceApplication.class, args);

}

public CartServiceApplication()

{

System.out.println("CartServiceApplication..");

}

}

**package** com.example.demo;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** CartController {

@RequestMapping("/getCart/{name}")

**public** String getProducttName(@PathVariable(name = "name") String name)

{

**return** "Cart Details <strong style=\"color: red;\">" + name + " </strong>";

}

@RequestMapping(value = "/")

**public** String home() {

**return** "Eureka Cart application";

}

}

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

eureka.client.instance.preferIpAddress=true

eureka.client.register-with-eureka=true

eureka.server.peer-node-read-timeout-ms=300000

spring.application.name=Order-service

server.port=2222

Zuul server

package com.cassiomolin.example.gateway;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

/\*\*

\* Spring Boot application entry point.

\*

\* @author cassiomolin

\*/

//@EnableDiscoveryClient

@EnableZuulProxy

@SpringBootApplication

public class ZuulProxyApplication {

public static void main(String[] args) {

SpringApplication.run(ZuulProxyApplication.class, args);

}

}

server.port=8282

spring.application.name=onlineshopping

eureka.client.registerWithEureka=true

eureka.client.fetchRegistry=true

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

eureka.client.serviceUrl.healthcheck.enabled=true

eureka.instance.preferIpAddress=true

#zuul.prefix=/api

management.endpoints.web.exposure.include=\*

management.endpoints.jmx.exposure.include=\*

#zuul.ignored-services=\*

zuul.routes.Order-service.path=/orders/\*\*

#zuul.routes.Order-service.serviceId=Order-service

zuul.routes.Order-service.url=http://localhost:2222

zuul.routes.product-service.stripPrefix=true

zuul.routes.Cart-Service.path=/Carts-Api/\*\*

#zuul.routes.product-service.serviceId=product-service

zuul.routes.Cart-Service.url=http://localhost:7777

zuul.routes.cart.stripPrefix=true

router ampping

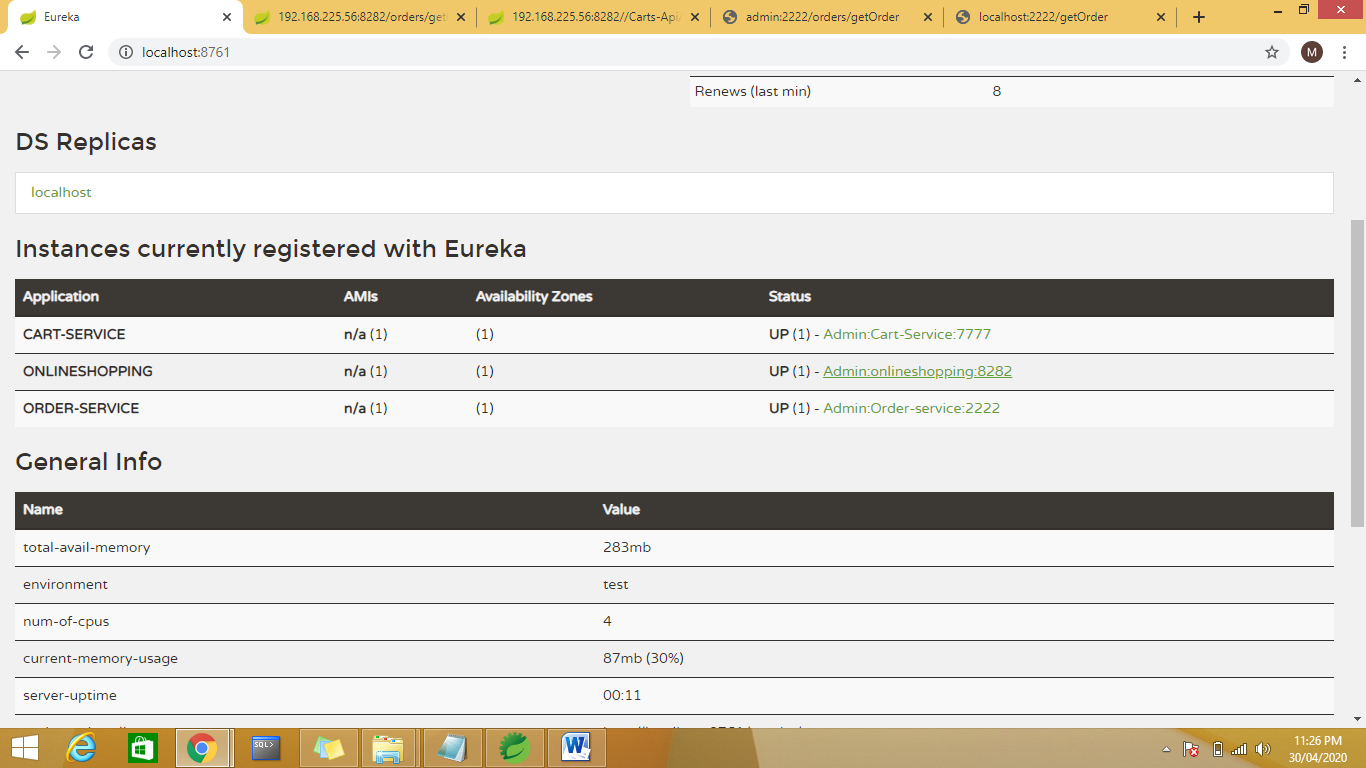
paths--

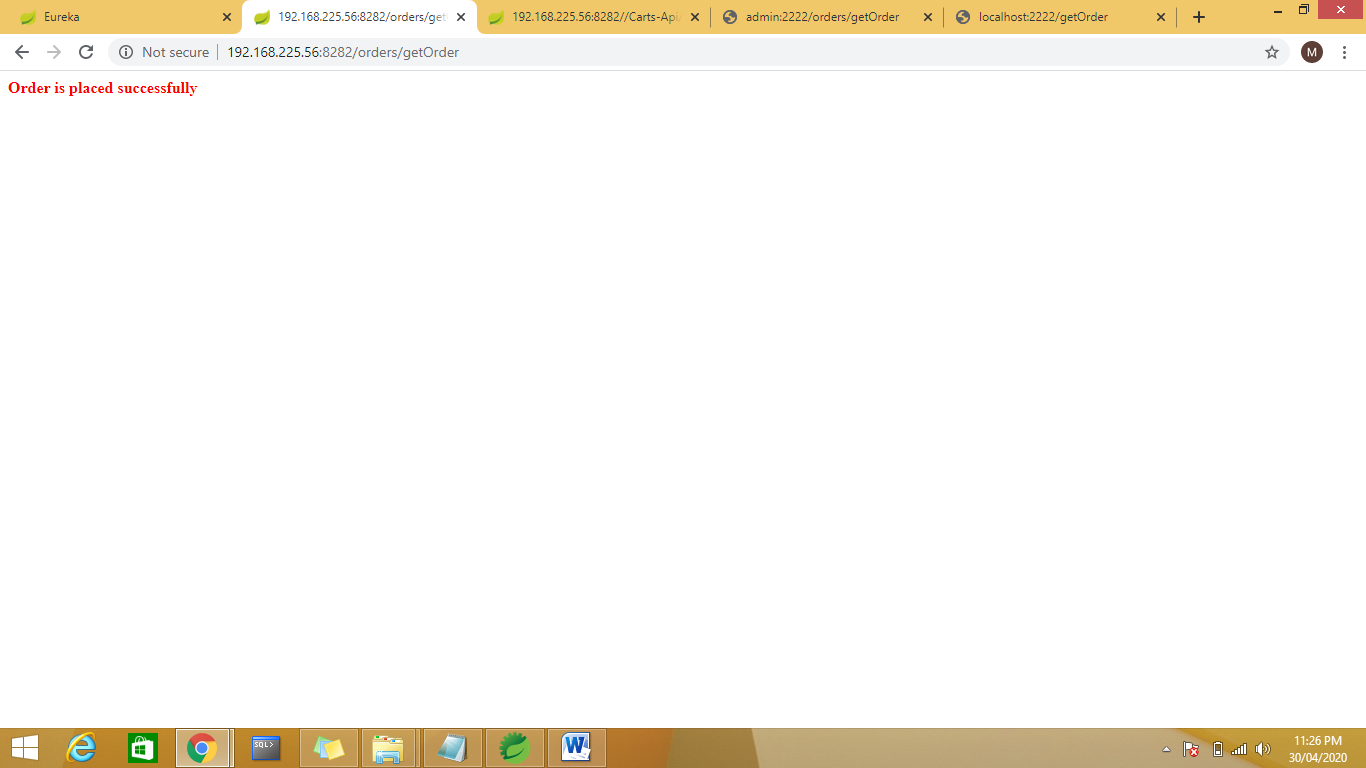
url based -- 1 instance

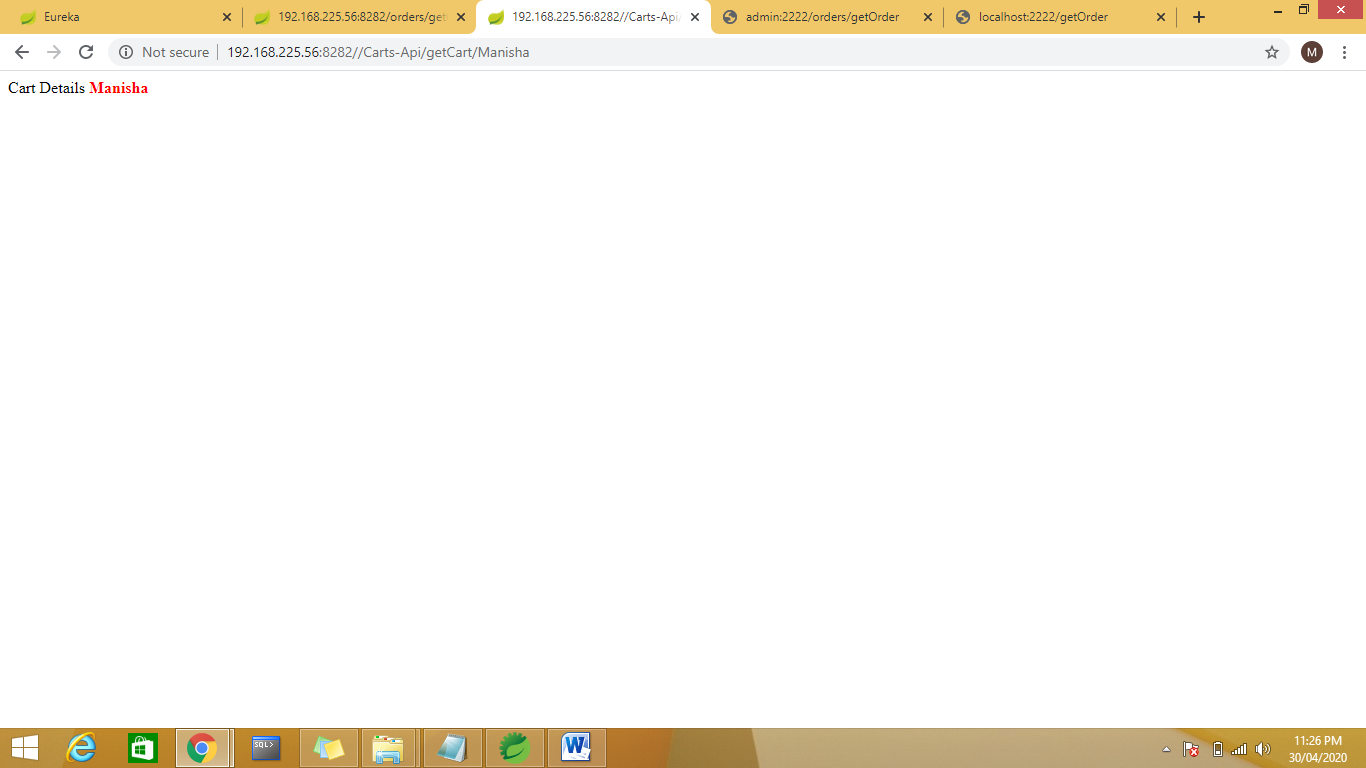
service name based –multiple instances

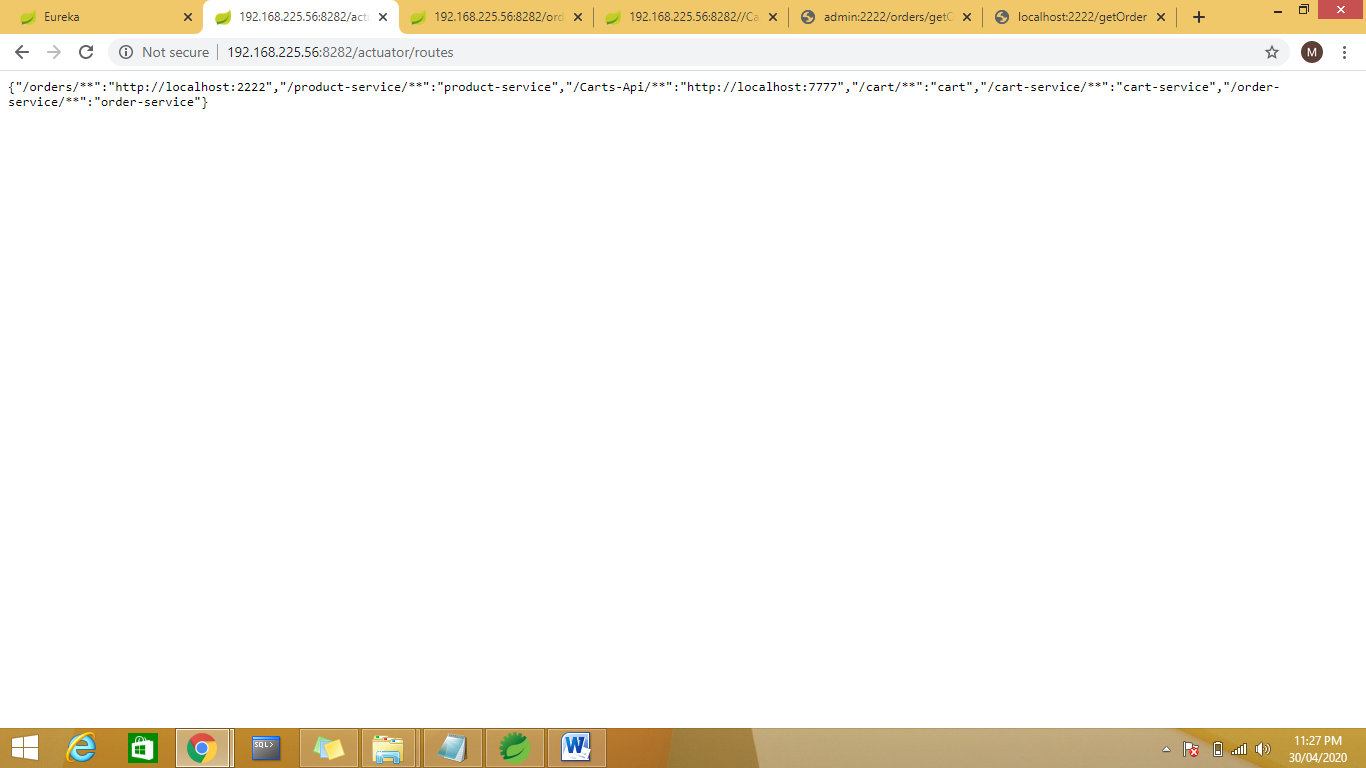
Execution:

Run eureka server,cart service,order service, zuul server.









Create 1 more instance for order:

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

eureka.client.instance.preferIpAddress=true

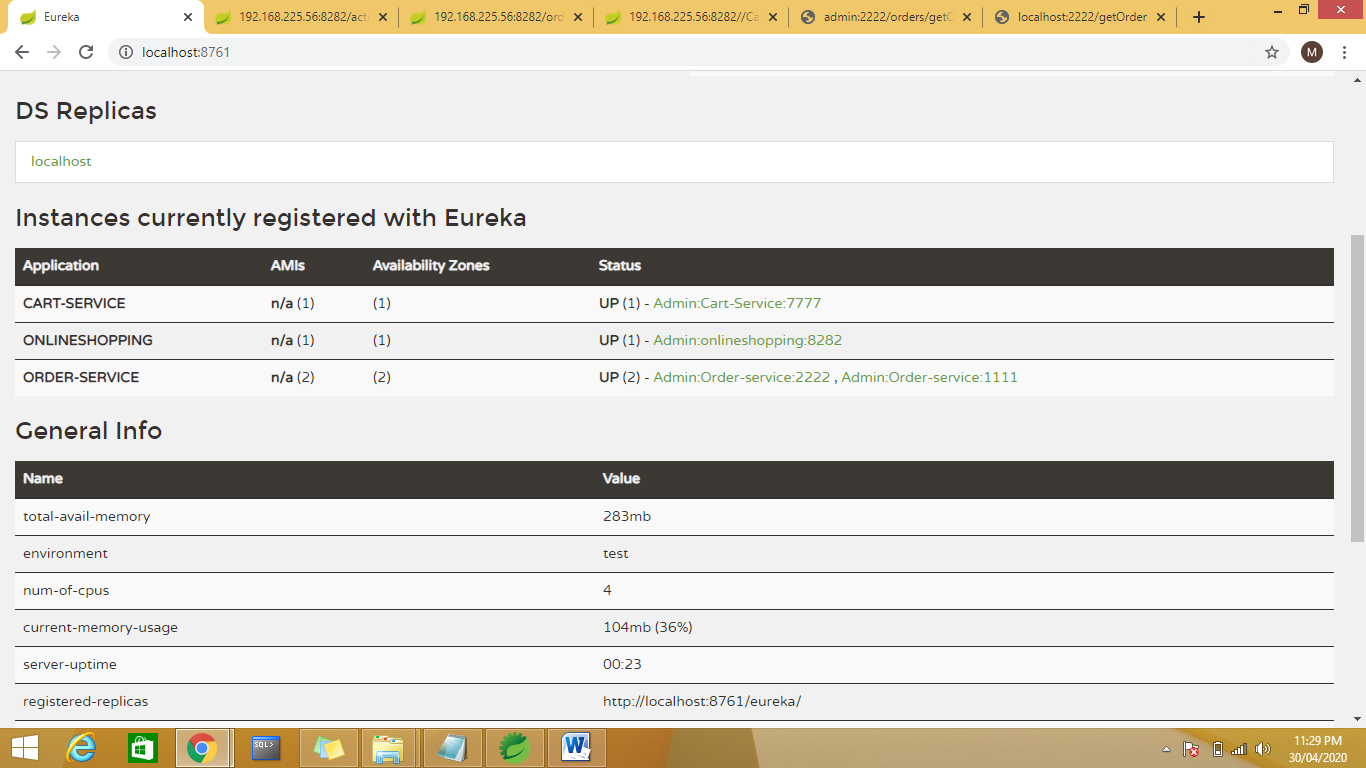
eureka.client.register-with-eureka=true

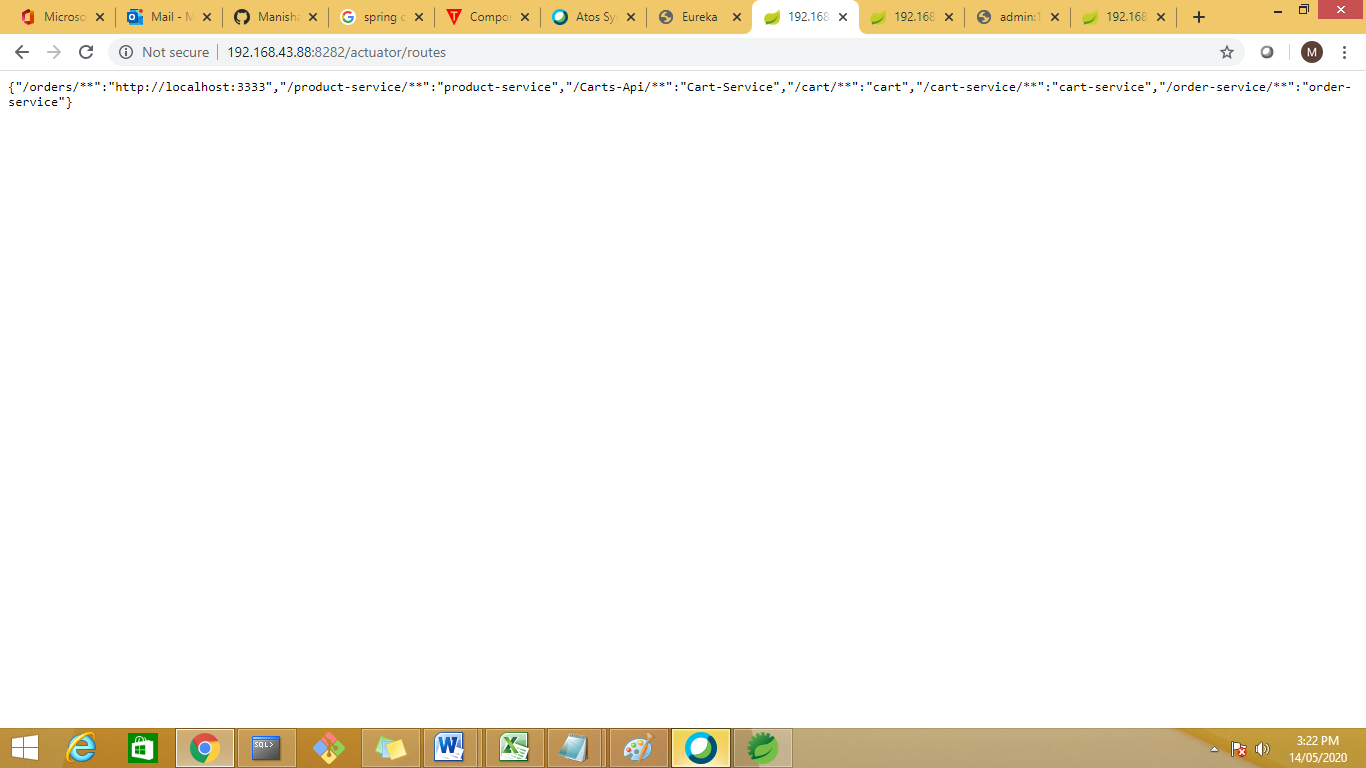
eureka.server.peer-node-read-timeout-ms=300000

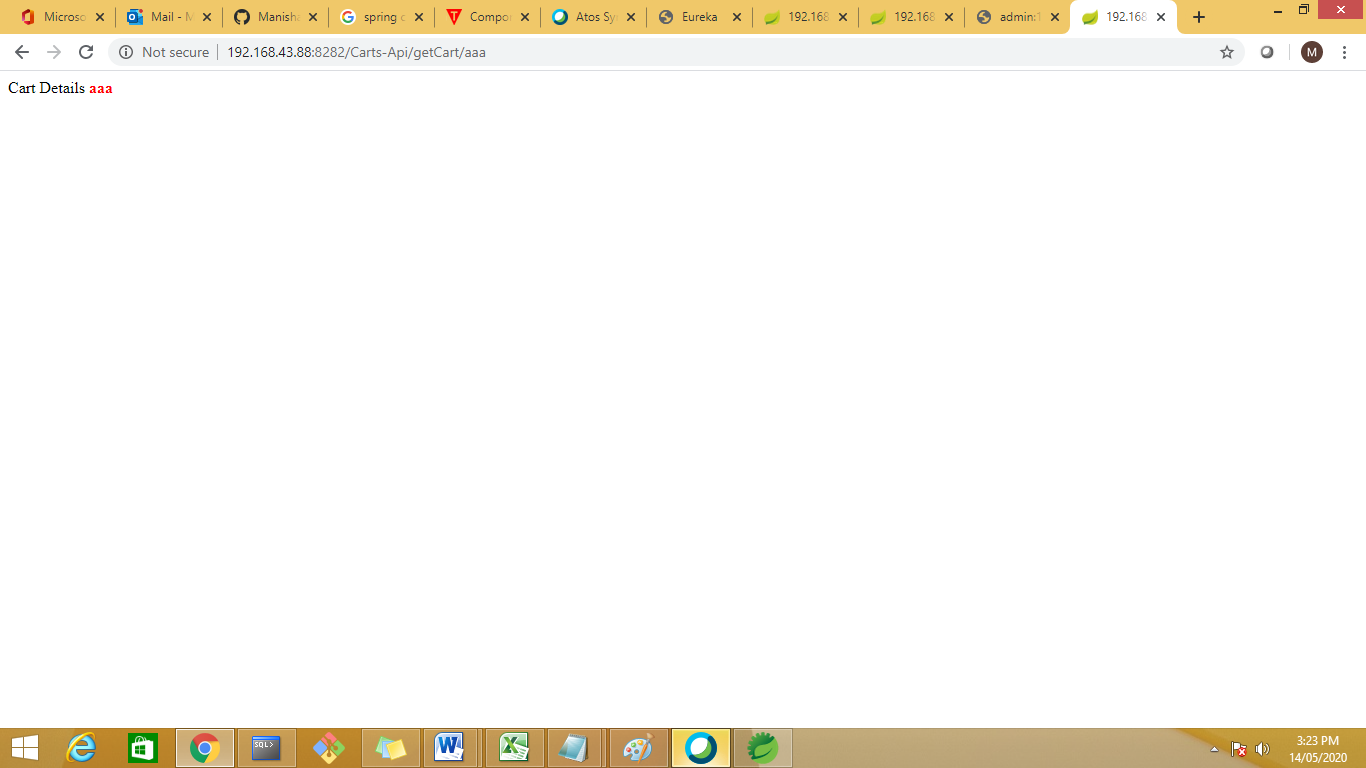
spring.application.name=Order-service

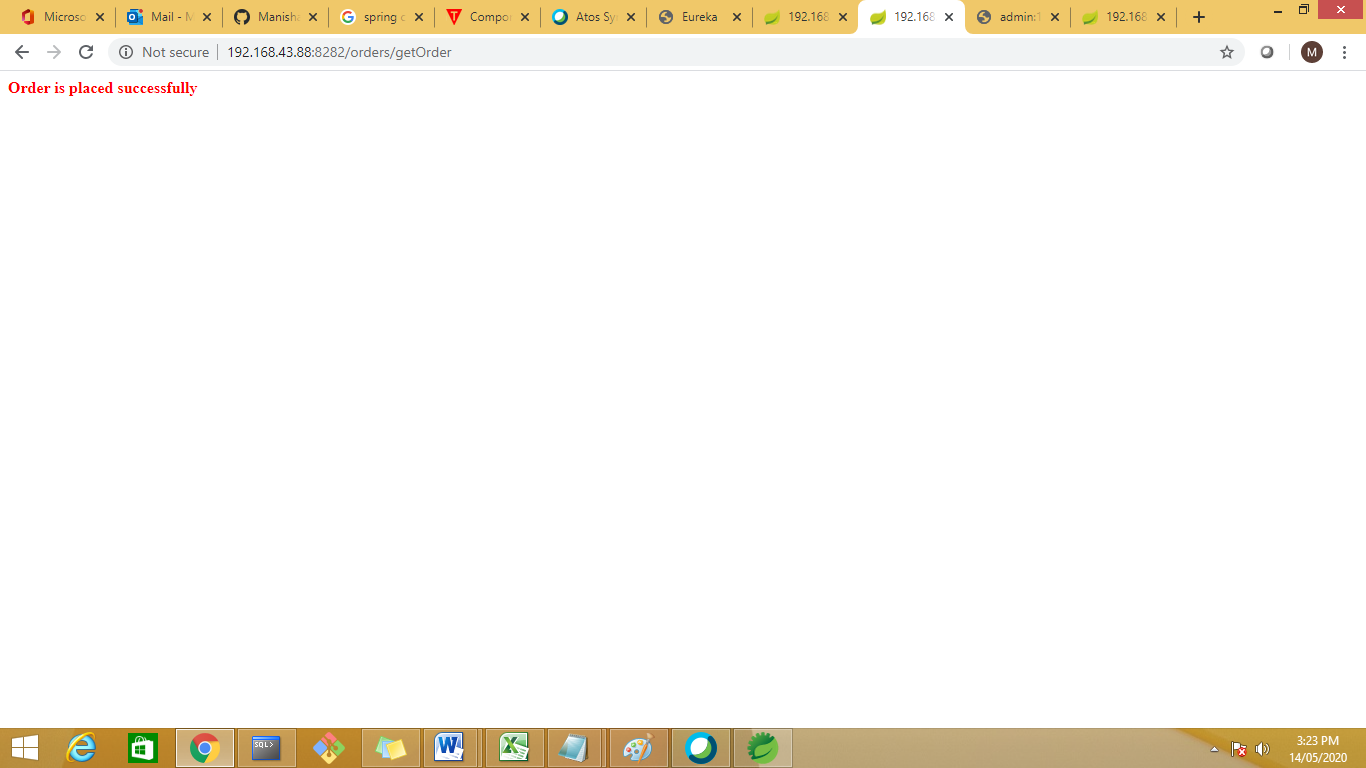
server.port=1111

Run the service on 1111









Spring Cloud:

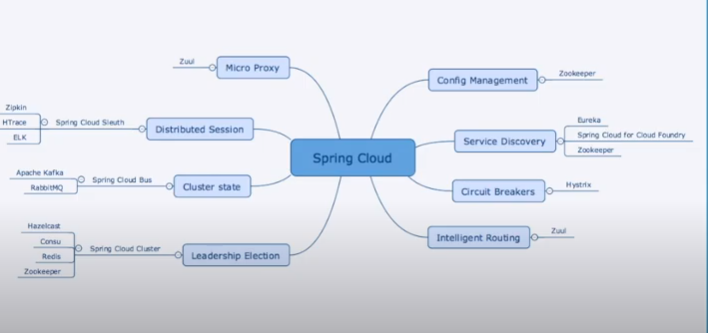
<https://www.youtube.com/watch?v=DduObGoRB9Y>

# What is Spring Cloud?

Spring Cloud is a framework for building robust cloud applications. Spring Cloud provides a solution to the commonly encountered patterns when developing a distributed system.

Spring Cloud provides tools for developers to build some of the **common patterns** in distributed systems quickly. For example, configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, a control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state.

Spring Cloud reduces the code with annotation we can avail the features



# <https://www.javatpoint.com/components-of-spring-cloud>

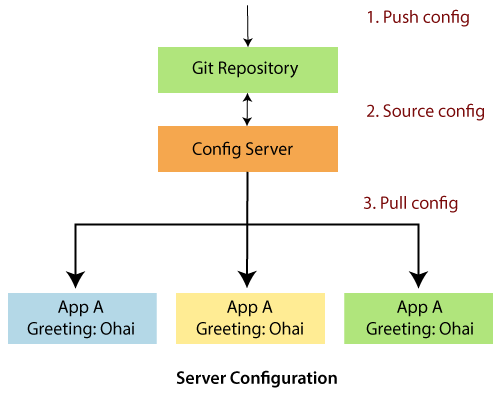
# Spring Cloud Components

There are the following components:

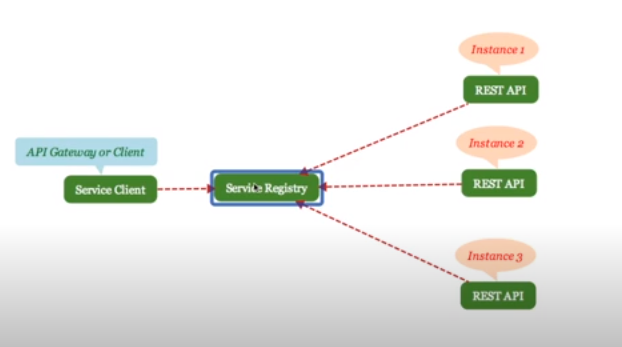
* Configuration(git repos)
* Service Discovery
* API Gateway(Zuul Proxy)
* Routing and Messaging
* Circuit Breakers(Hystrix)
* Tracing(Sleuth)
* CI Pipeline and Testing

## Configuration

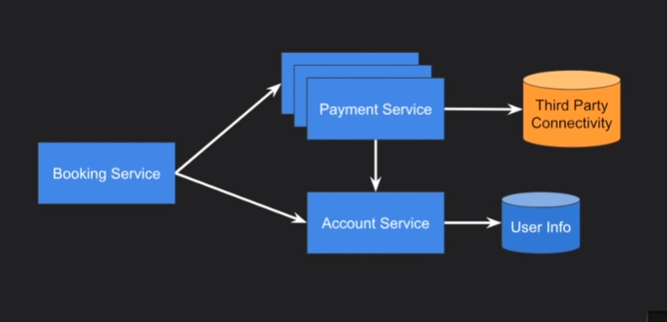
Spring Cloud configuration components provide server-side and client-side support for externalized configuration in a distributed system. We can manage the external properties with config server for applications across all environments. Spring Cloud config server can use Git, SVN (Apache Subversion), filesystem, and Vault to Store config. Config clients (microservice app) retrieve the configuration client from the server on startup.



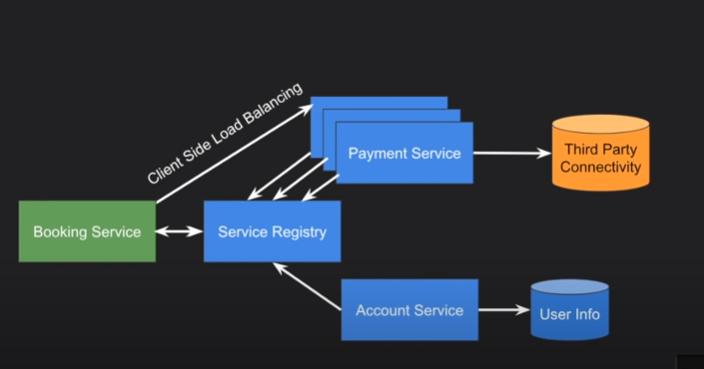
Service Registry:



For example 3 instances are running on 3 different machines with 3 different ip addresses. When client send a request then service registry check for the available instance and serve. For example cloud foundry . cloud foundry will hit the instance.check for the instance in round robbin fashion. Eureka



When we add the services in cloud environment or container , it is going to change over a period of time . to make the application let us add the service registry pattern



3 instances are running on 3 different machines with 3 different ip addresses.when we use service registry we nedd to do load balancing and redirect to a particular instance . for doing this we use either clinet side load balancing or server side load balancing

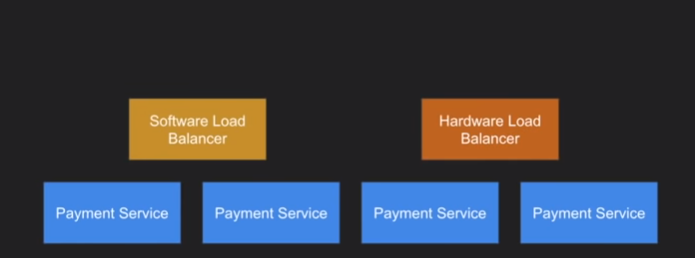
Client side load bal: booking service get the info from service reg saying that there are 3 instances running on 3 diff mc , on 3 diff ports the then bookingservice will choose directly one of the instance from 3 instances then , bs act as a client. Called as cliebt side load balancing . here we don’t need the central load balancer for load balancing. service can choose load balancing technique.

Example Ribbon Server created Netflix ,

Server Side Load balancing:

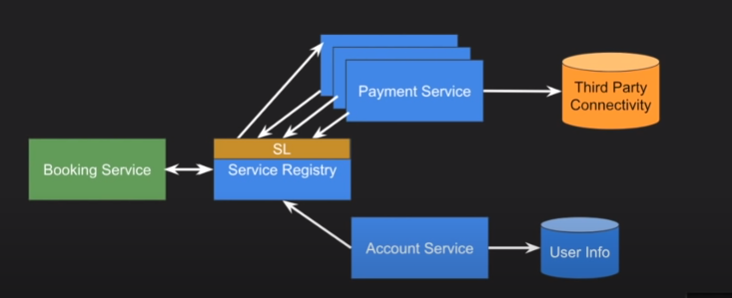
Software load balancer is a piece of code which will do load balancing by routing

Hardware load balancer is a server dedicating to do traffic routing which basically used in internet facing appliactions.



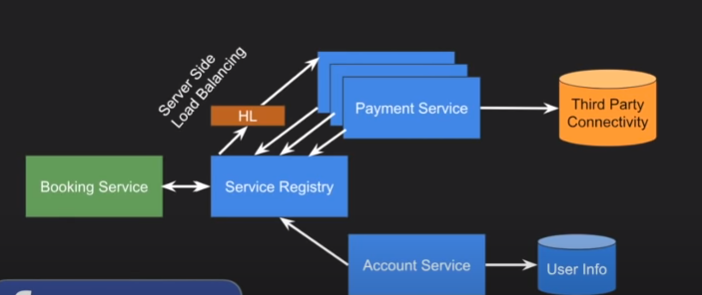
Client side load balancing is using software load balancer.

Server side balancing can be done using SLB or HLB.



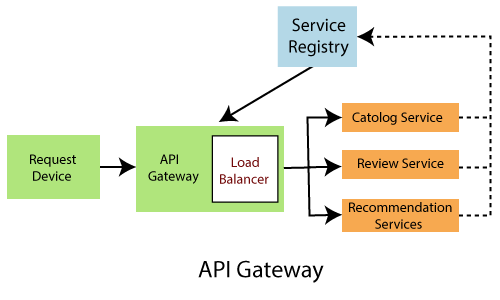
In this digram software load balancing is embedded in service registry.

Same can be done with HL balancing



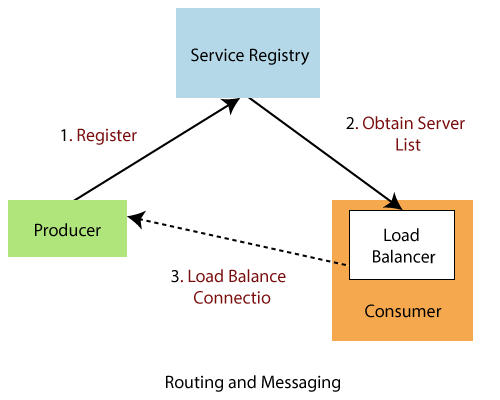
API Gateway

API Gateway allows us to route API request (external or internal) to connect services. It also provides a library for building an API gateway on the top of Spring MVC. Its aims to provide cross-cutting concerns to them, such as **security** and **monitoring**.



Routing and Messaging

The cloud application made up of many microservices so the communication will be critical. Spring Cloud supports communication via messaging or HTTP request. Routing uses **Netflix Ribbon** and **Open Feign while** messaging uses Kafka or Rabbit MQ.



## Microservices Monitoring Tool

There are three monitoring tools are as follows:

* Hystrix dashboard
* Eureka admin dashboard
* Spring boot admin dashboard

# Components of Microservices

There are the following components of microservices:

* Spring Cloud Config Server
* Netflix Eureka Naming Server
* Hystrix Server
* Netflix ZuulAPI Gateway Server
* Netflix Ribbon
* Zipkin Distributed Tracing Server

### Spring Cloud Config Server

Spring Cloud Config Server provides the HTTP resource-based API for external configuration in the distributed system. We can enable the Spring Cloud Config Server by using the annotation **@EnableConfigServer**.

### Netflix Eureka Naming Server

Netflix Eureka Server is a discovery server. It provides the REST interface to the outside for communicating with it. A microservice after coming up, register itself as a discovery client. The Eureka server also has another software module called **Eureka Client**. Eureka client interacts with the Eureka server for service discovery. The Eureka client also balances the client requests.

### Hystrix Server

Hystrix server acts as a fault-tolerance robust system. It is used to avoid complete failure of an application. It does this by using the **Circuit Breaker mechanism**. If the application is running without any issue, the circuit remains closed. If there is an error encountered in the application, the Hystrix Server opens the circuit. The Hystrix server stops the further request to calling service. It provides a highly robust system.

### Netflix Zuul API Gateway Server

Netflix Zuul Server is a gateway server from where all the client request has passed through. It acts as a unified interface to a client. It also has an inbuilt load balancer to load the balance of all incoming request from the client.

### Netflix Ribbon

Netflix Ribbon is the client-side Inter-Process Communication (IPC) library. It provides the client-side balancing algorithm. It uses a Round Robin Load Balancing:

* Load balancing
* Fault tolerance
* Multiple protocols(HTTP, TCP, UDP)
* Caching and Batching

### Zipkin Distributed Server

Zipkin is an open-source project m project. That provides a mechanism for sending, receiving, and visualization traces.

One thing you need to be focused on that is port number.

|  |  |
| --- | --- |
| **Application** | **Port** |
| Spring Cloud Config Server | 8888 |
| Netflix Eureka Naming Server | 8761 |
| Netflix Zuul API gateway Server | 8765 |
| Zipkin distributed Tracing Server | 9411 |

<https://www.javatpoint.com/spring-cloud-vs-spring-boot>

<https://www.javatpoint.com/challenges-of-microservices-architecture>

<https://www.baeldung.com/zuul-load-balancing>

<https://www.youtube.com/watch?v=yK3Qt8eCrmQ>

NOTE : add actuator dependencies and not the devTools

Zuul is used for routing and provides the unified user interface to communicate with themicroservices.

Execution:

Run Eureka , serevices and zuul proxy server

Day 4:

Hystrix Server and Zipkin Proxy

Zipkin 🡪Error handling, Tracing the logs:

Challenges Availability

Scalability

Monitoring

Integration: A🡪B🡪C🡪D🡪E

If any service is down it is bit difficult to trace which service is down.

**Zipkin Server**

**Zipkin** is an application that monitors and manages the **Spring** Cloud Sleuth logs of your **Spring Boot** application. To build a **Zipkin server**, we need to add the **Zipkin** UI and **Zipkin Server** dependencies in our build configuration file. Now, configure the **server**. port = 9411 in application properties file.

Most developers face difficulty of tracing logs if any issue occurred. This can be solved by Spring Cloud Sleuth

and ZipKin server for Spring Boot application.

Spring Cloud Sleuth

Spring cloud Sleuth logs are printed in the following format -

[application-name,traceid,spanid,zipkin-export]

Where,

Application-name = Name of the application

Traceid = each request and response traceid is same when calling same service or one service to another service.

Spanid = Span Id is printed along with Trace Id. Span Id is different every request and response calling one service to another service.

Zipkin-export = By default it is false. If it is true, logs will be exported to the Zipkin server.

Demo

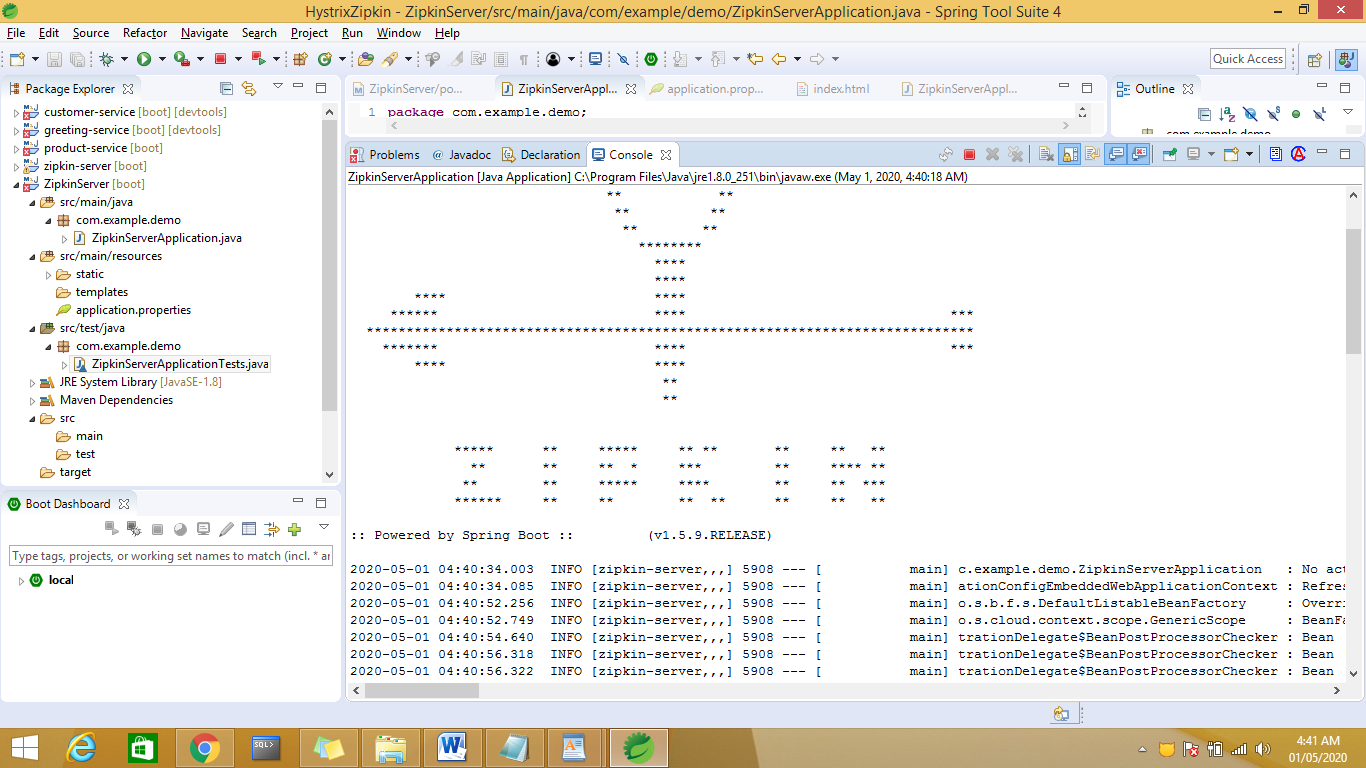
1. Create a spring boot application as zipkin server which will trace the logs.

Add the Zipkin UI and Zipkin Server dependencies in our build configuration file. This dependencies are not form spring they are open source third party services.

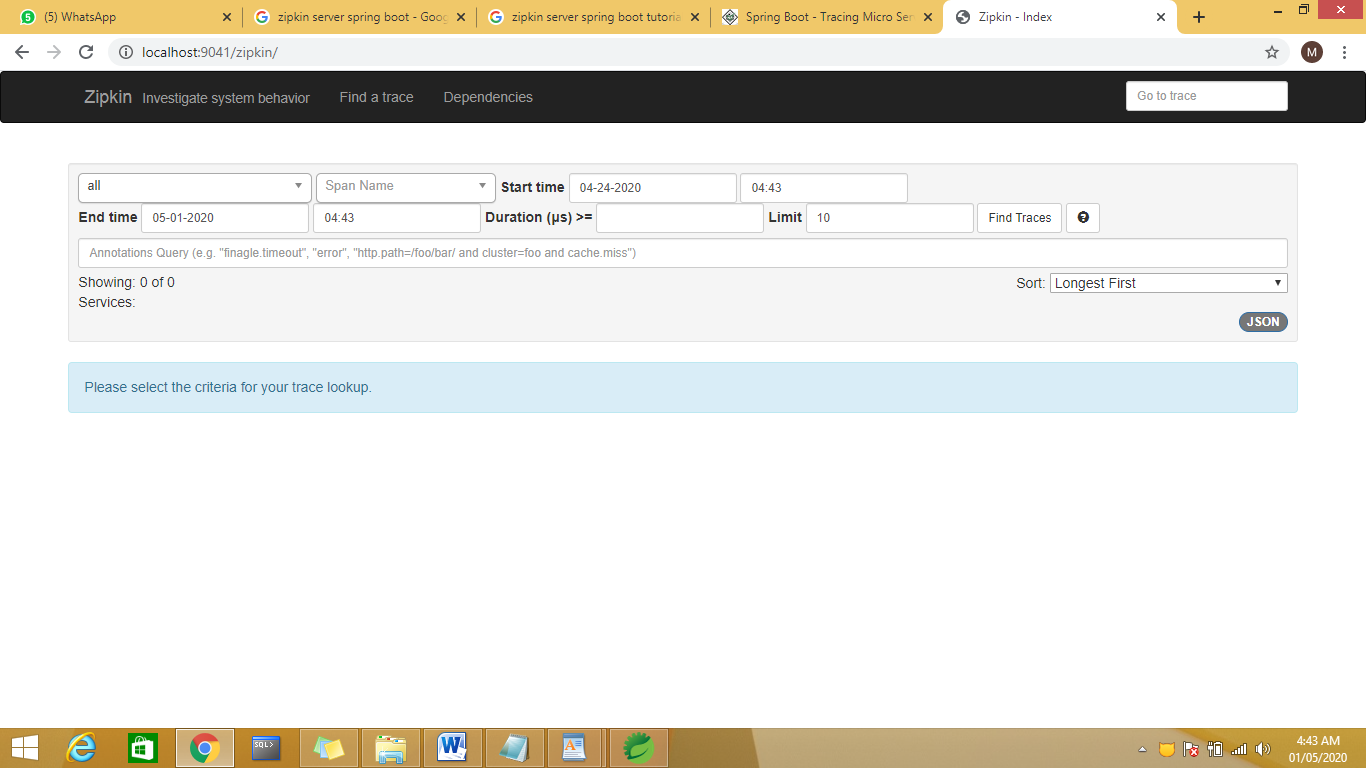
Use @EnableZipkinServer

Application.port

Server.port=9041 default port for zipkin server

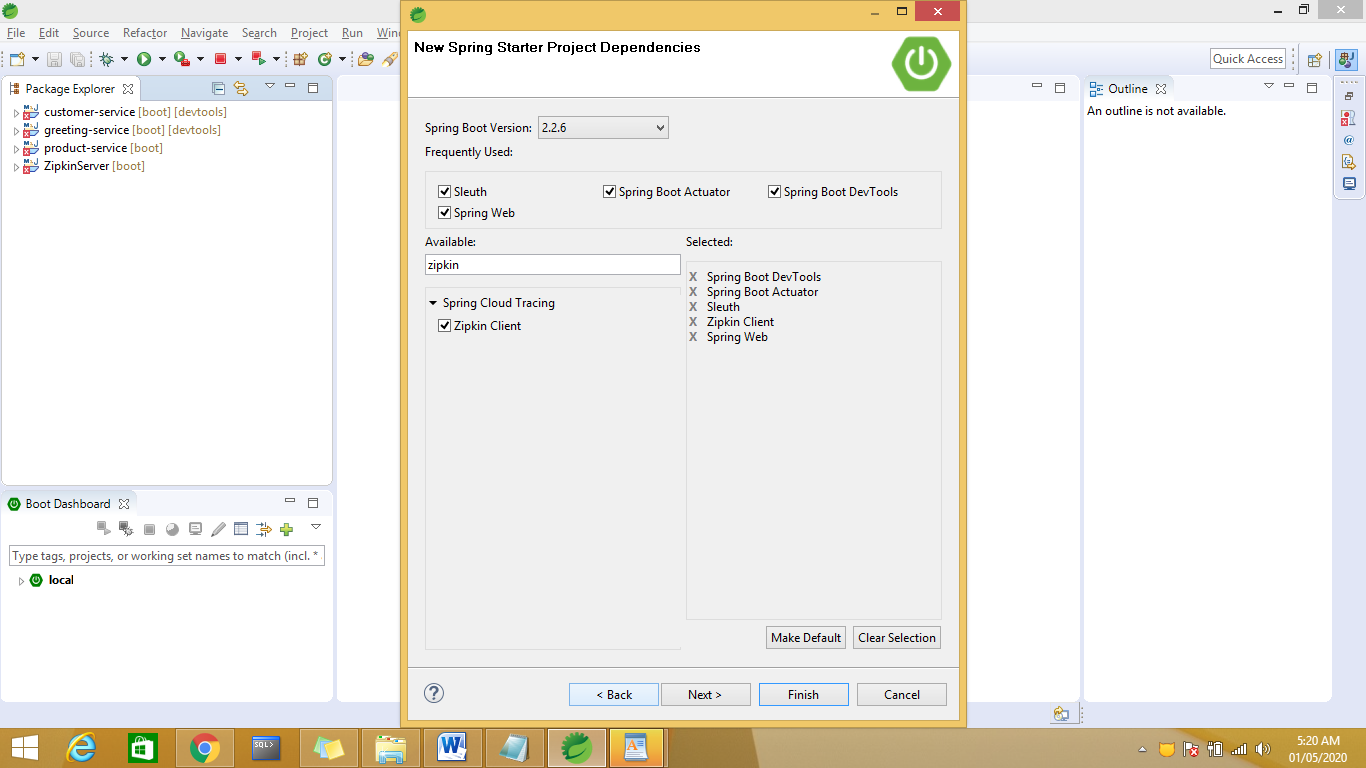


Run the application :



spring-cloud-starter-zipkin will automatically trace the logs.

1. Create 2 micro services with web ,actuator, devTools, zipkin client,sleuth dependencies



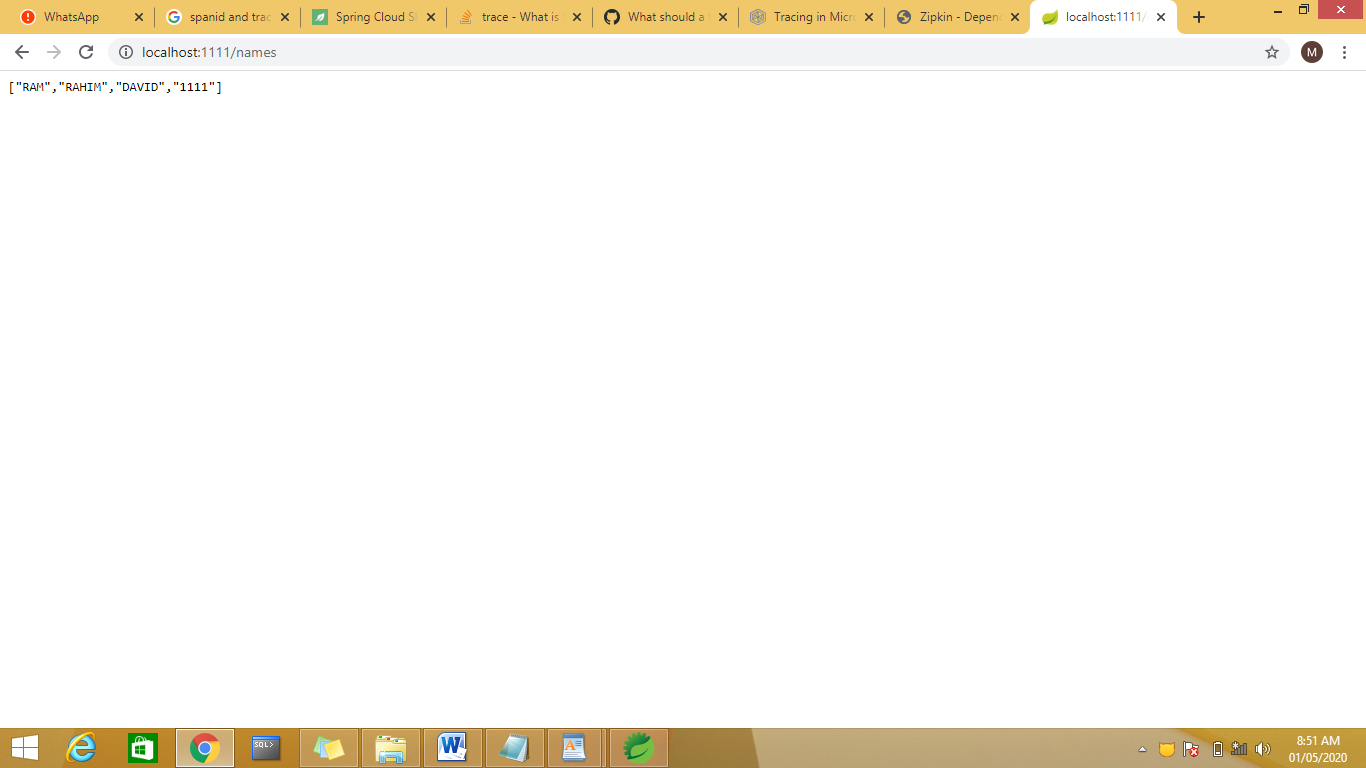
Sleuth is used to register app with zipkin , application logs will be exported to zipkin server.

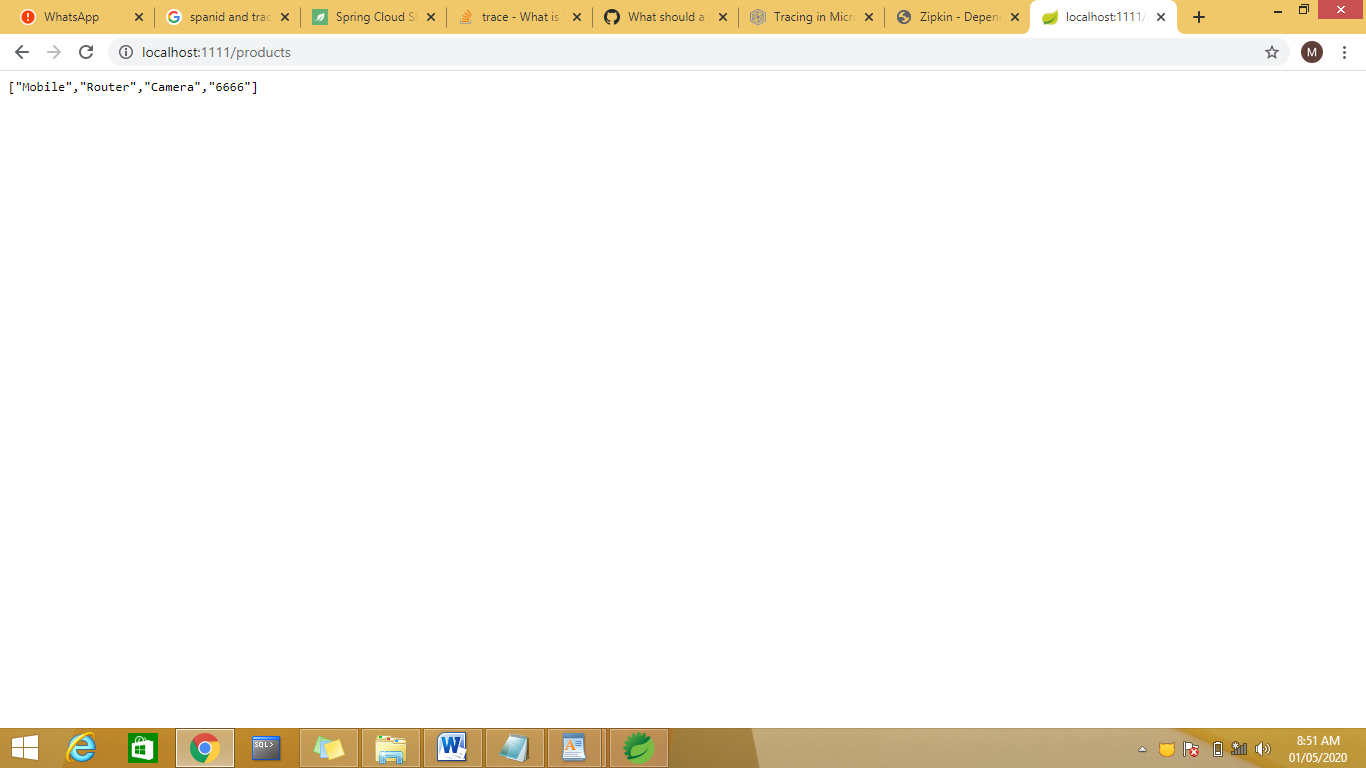
RestTemplate to make request to another service

Cutomet-Service-1111

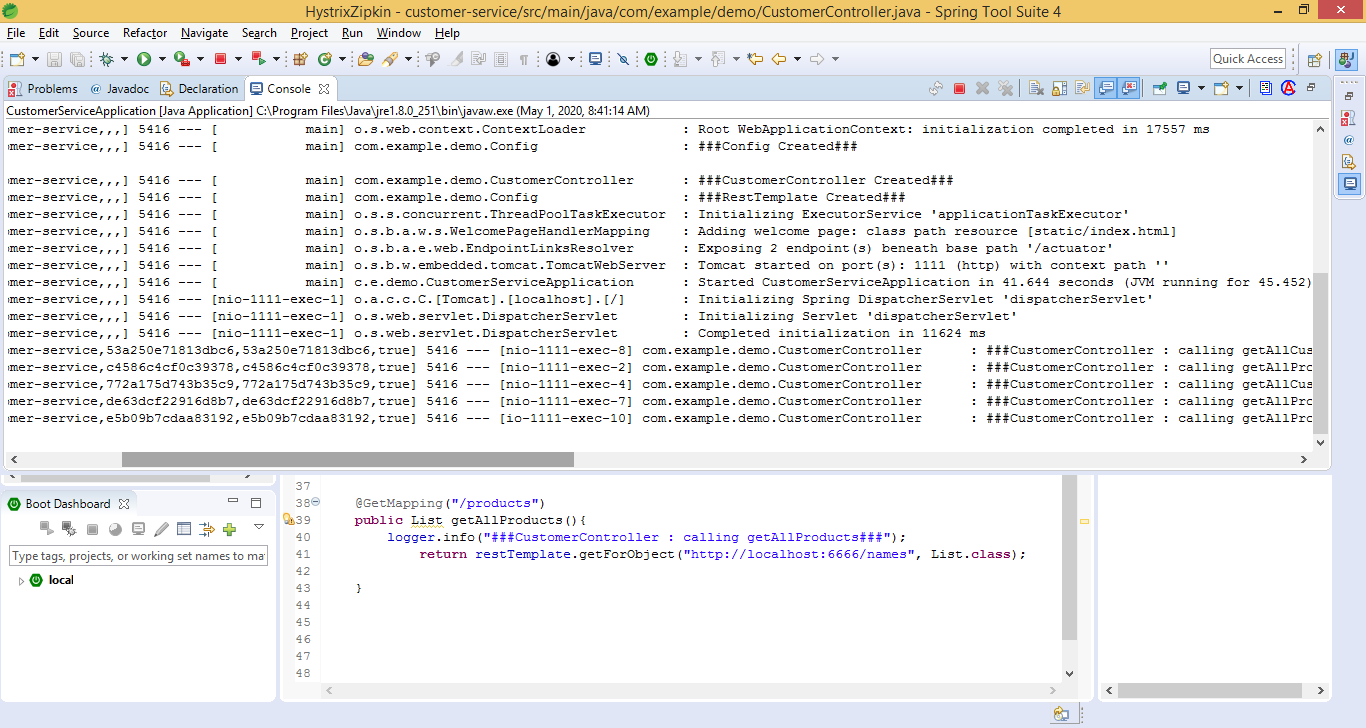
Product-Service-6666

Customer-Service🡪Product-Service

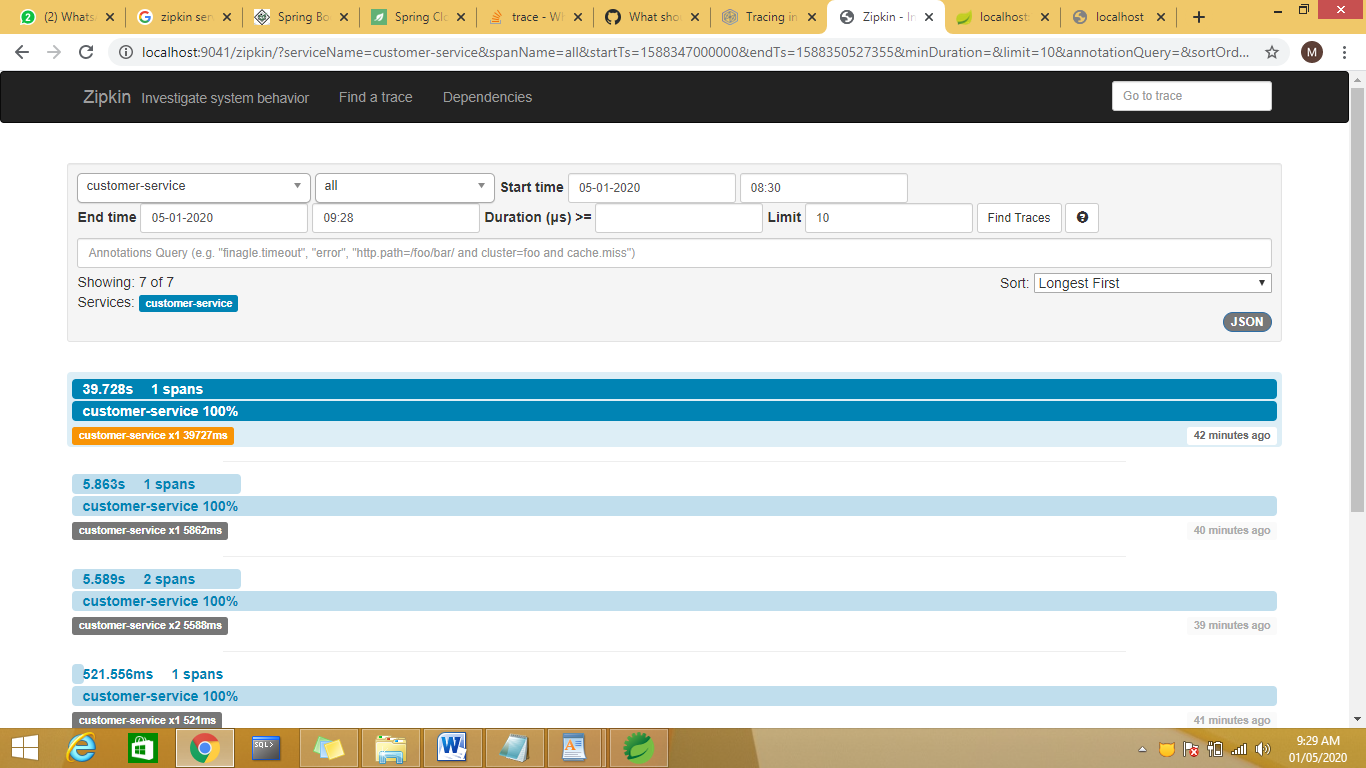




Customer is calleing Product Service

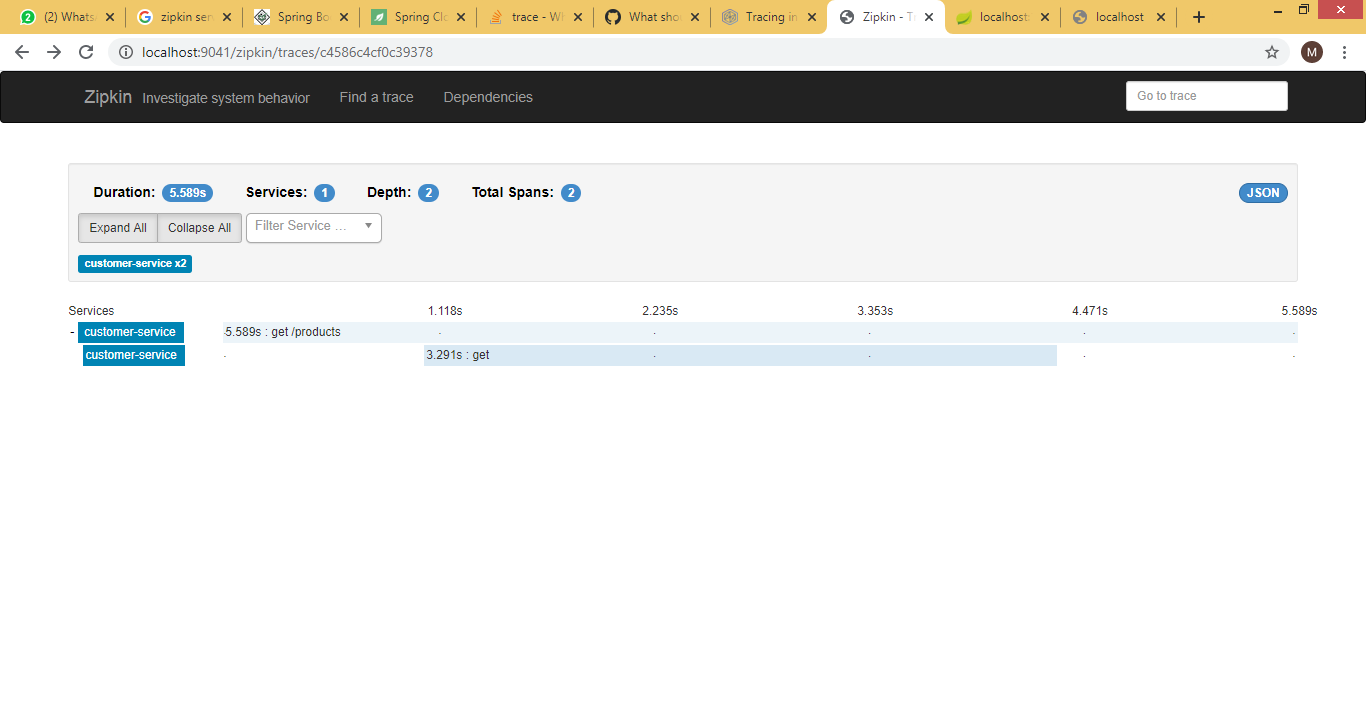


As the request is with in the same service same traced and Spanid.

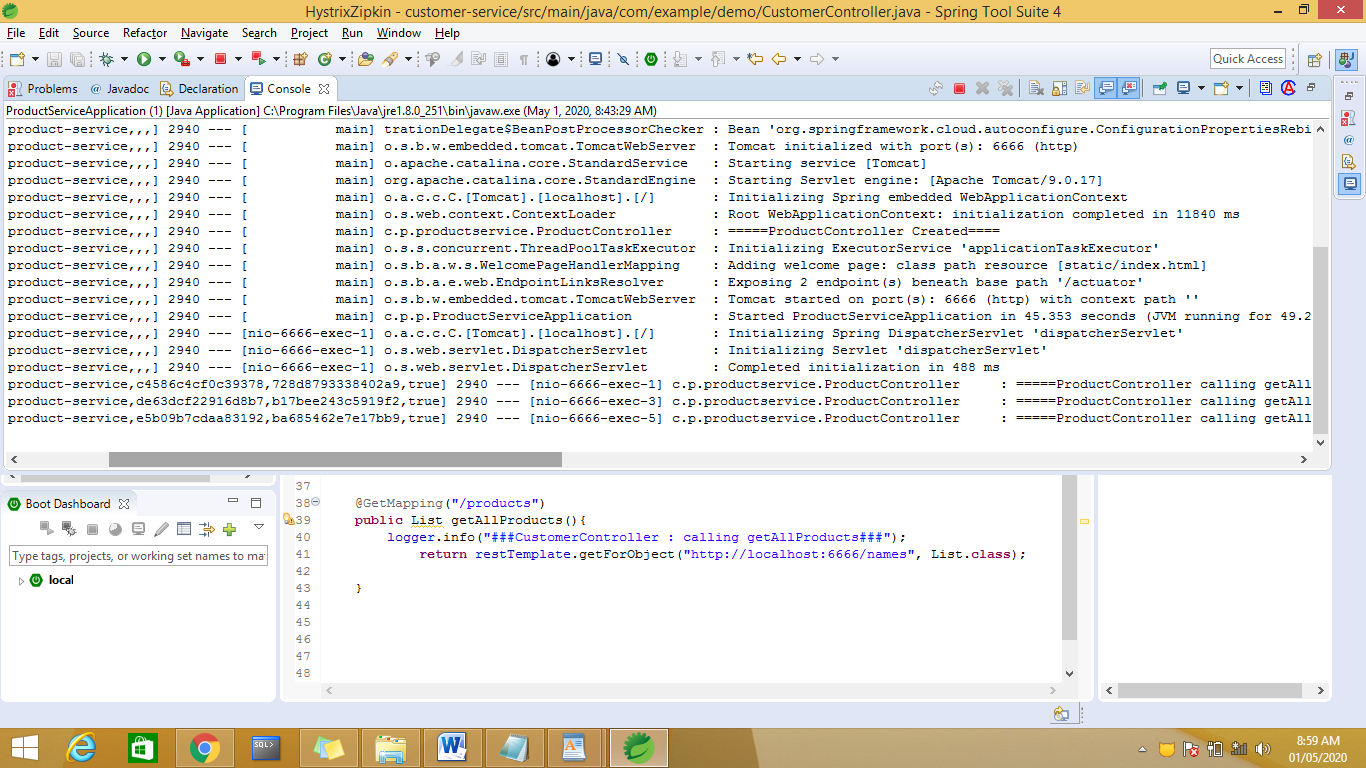


In Zipkin Server checkthe traces by usring url <http://localhost:9041/zipkin/traces/c4586c4cf0c39378>

c4586c4cf0c39378 is the trace id



Click on Json Button and check the Json details



Observe the traceid for CustomerService is same for ProductService but span id is different as two different services are communicating.

Hystrix🡪Circuit Broker

called [Hystrix](https://github.com/Netflix/Hystrix" \t "_blank) to implement **circuit breaker** while invoking underlying [microservice](https://howtodoinjava.com/microservices/microservices-definition-principles-benefits/). It is generally required to enable fault tolerance in the application where some underlying service is down/throwing error permanently, we need to fall back to different path of program execution automatically. This is related to distributed computing style of Eco system using lots of underlying Microservices. This is where circuit breaker pattern helps and Hystrix is an tool to build this circuit breaker.

<https://www.javainuse.com/spring/springcloud>

When we doing service integration

A->B->C->D->E->F dependencies amg n number of services

If any service is down then how the error will get propagated??? If F is having any error then error will be propagated in sequential order in reverse order.

F->E->D->C->B->A

Propagation of error is time consuming to handle this we have to give some time out value if within that time out if no error occurs then we have to give some default msg , we need to break the flow using circuit.

If a service is trying to connect the resource then if it is unable to get the resource within the specified time out time then hystrix will come into picture.

<https://www.todaysoftmag.com/article/1531/fault-tolerant-microservices-with-netflix-hystrix>

Maven users can add the following dependency in the pom.xml file −

<dependency>

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

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

</dependency>

add the @EnableHystrix annotation into your main Spring Boot application class file.

<dependency>

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

<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>

</dependency>

<https://howtodoinjava.com/spring-cloud/spring-hystrix-circuit-breaker-tutorial/>

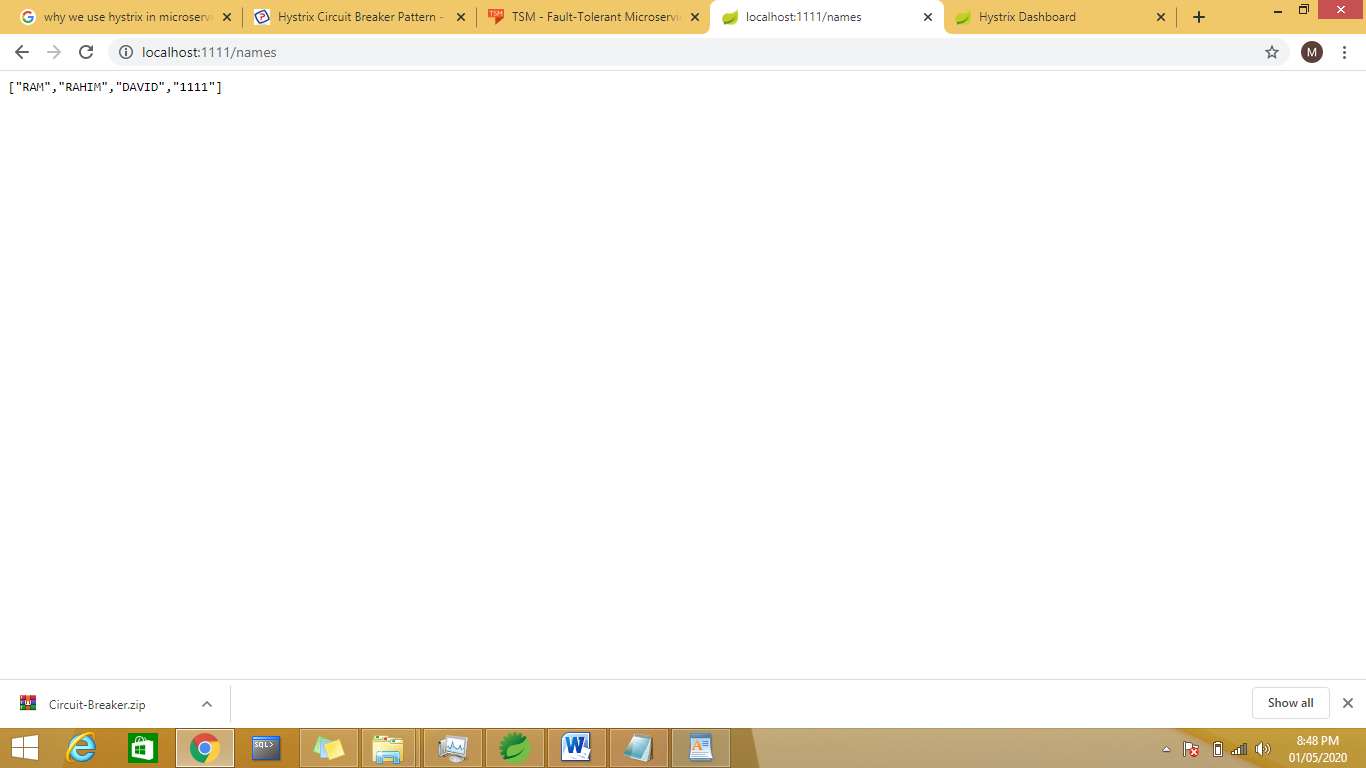
execution run the customet and product services

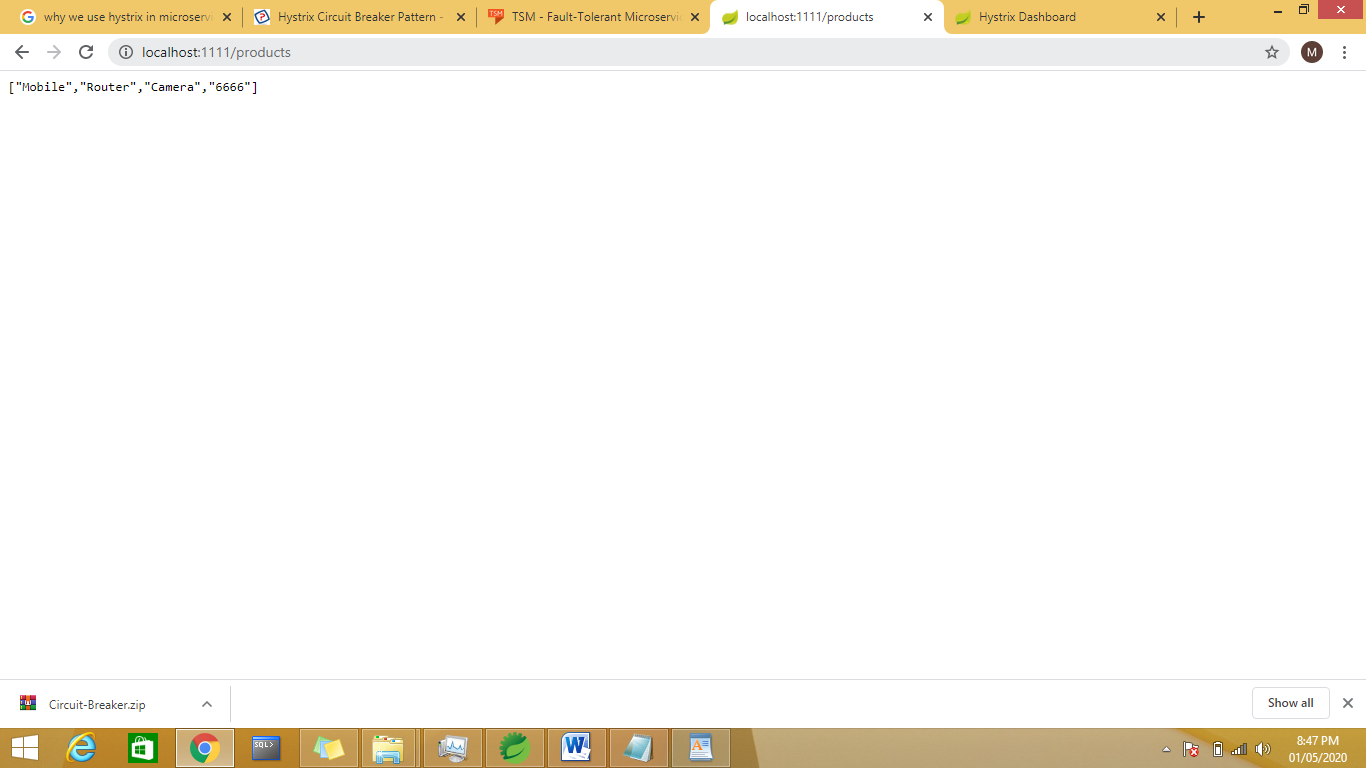
run the hytrix example

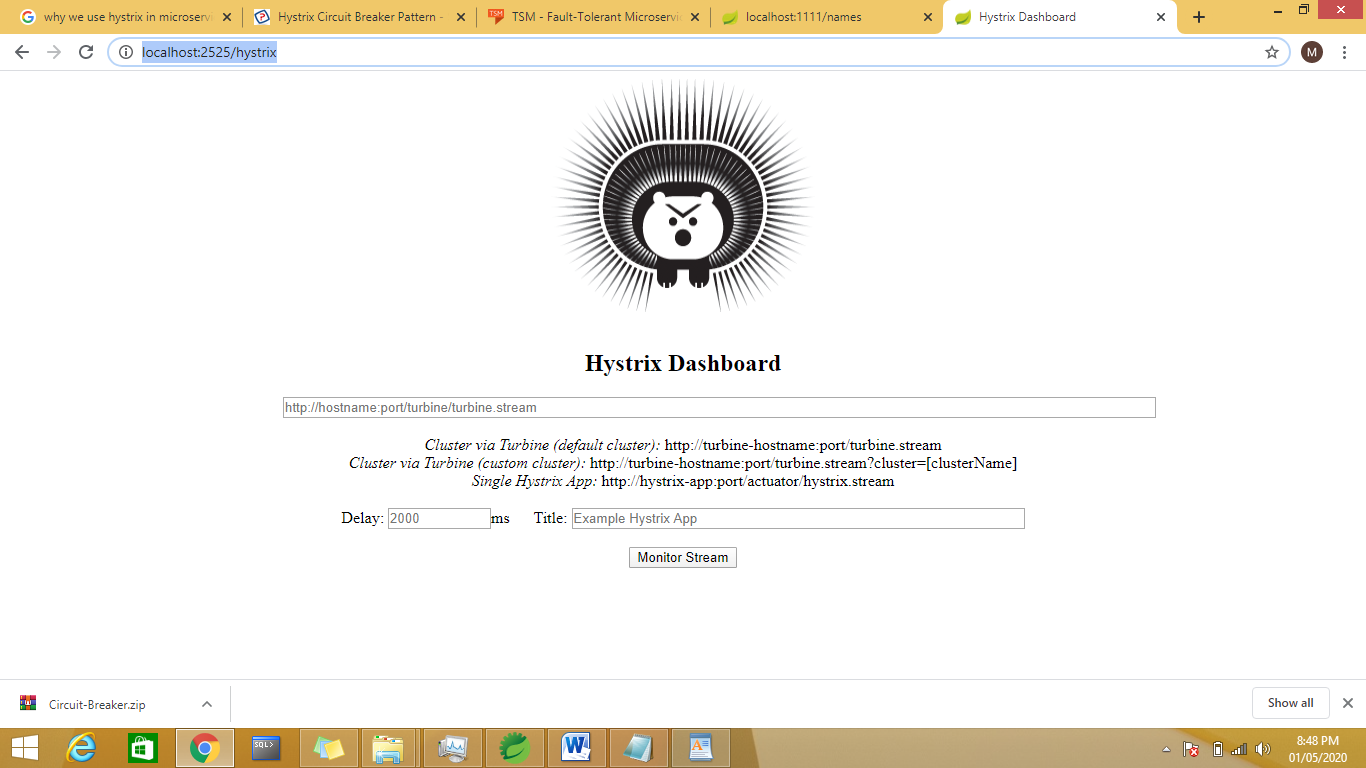
customer –1111

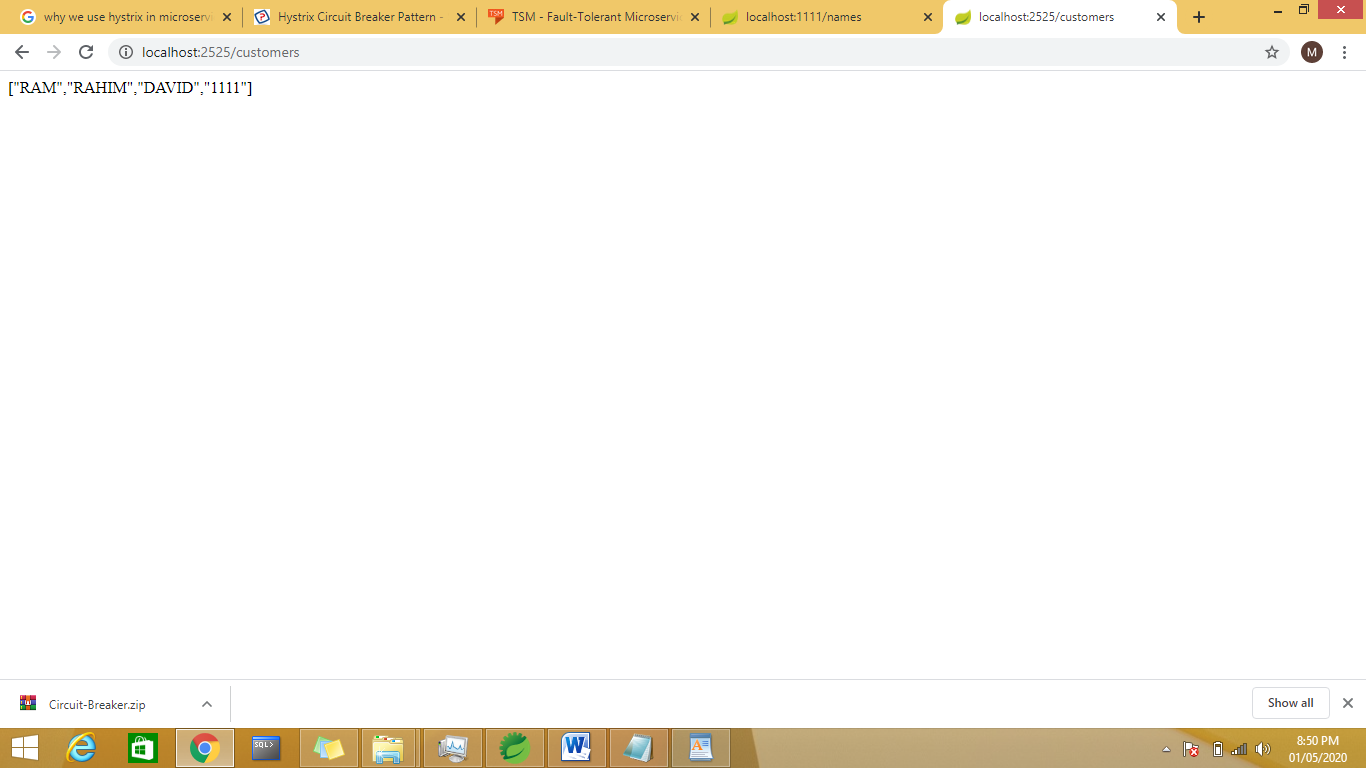
produc—6666

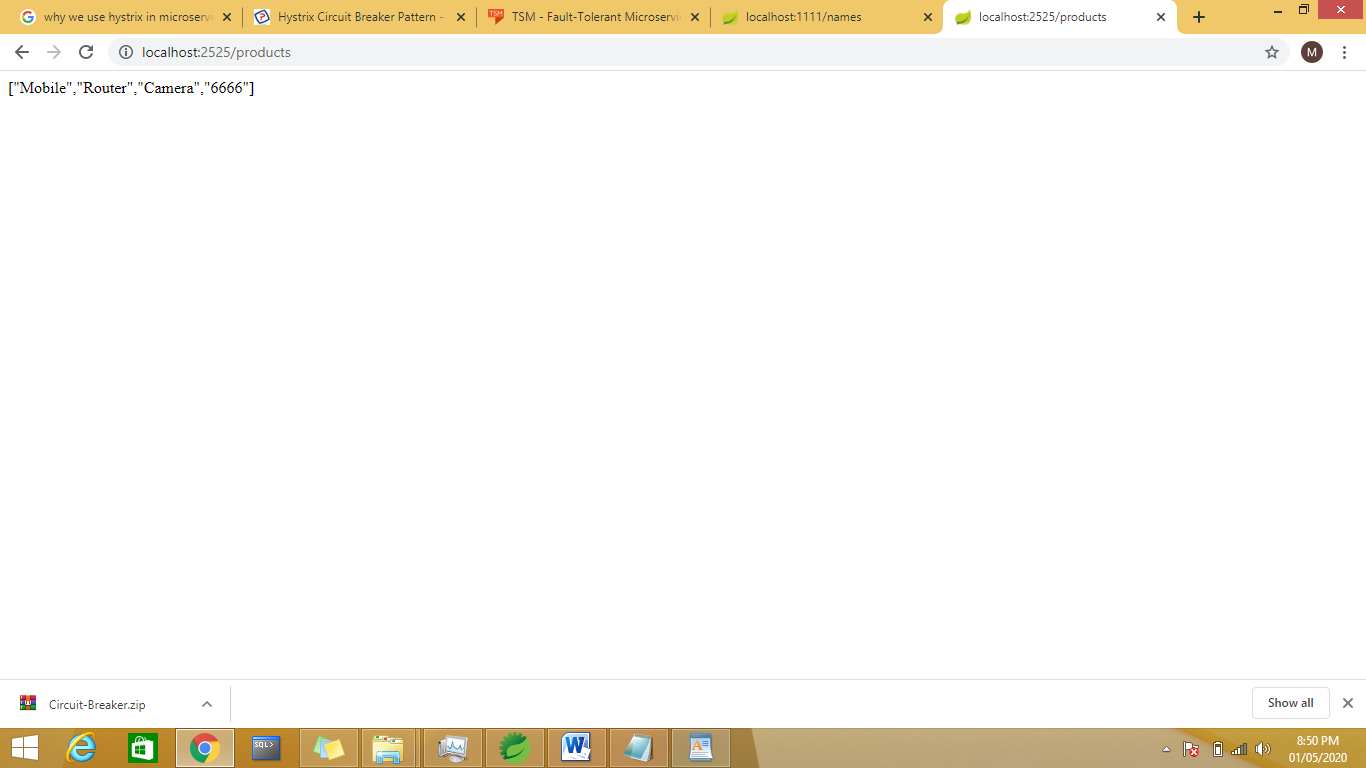
hystrix—2525











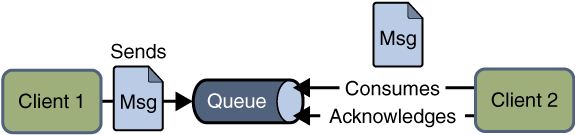
Swagger🡪Swagger with MicroServices

JMS using Active MQ and case study

#### 3.1. Point-to-Point Messaging Domain

In the point-to-point messaging domain the application is built on the basis of message queues, senders and receivers. Each and every message is addressed to a particular [**queue**](https://docs.oracle.com/javaee/6/api/javax/jms/Queue.html). Queues retain all messages sent to them until the messages are consumed or expired. There are some characteristics of PTP messaging:

1. There is only one client for each message.
2. There is no timing dependency for sender and receiver of a message.
3. The receiver can fetch message whether it is running or not when the sender sends the message.
4. The receiver sends the acknowledgement after receiving the message.

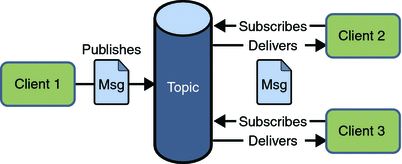
Point to point JMS Messaging

#### 3.2. Publish/Subscribe Messaging Domain

In publish/subscribe messaging domain, only one message is published which is delivered to all clients through [**Topic**](https://docs.oracle.com/javaee/6/api/javax/jms/Topic.html) which acts as a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the topic. The Topic is responsible to hold and deliver messages. The topic retains messages as long as it takes to distribute to the present clients.

Some of the characteristics are:

1. There can be multiple subscribers for a message.
2. The publisher and subscribe have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

Publish Subscribe JMS Messaging

Read More: [HornetQ Basic Example](https://howtodoinjava.com/hornetq/basic-jms-messaging-example-using-hornetq-stand-alone-server/" \t "_blank)

ActiveMQ is MOM

Sender and receiver

Demo1 :

Activemq-inmemory

Demo2:external active mq broker

Application.propeties

Default port number is 8081

12 factor applications

<https://www.springboottutorial.com/12-factor-app-cloud-native-microservices-best-practices>

<https://sathishpk.wordpress.com/2019/09/15/shoppingcart-example-using-java-microservicesjms-activemq-springboot/>

https://dzone.com/articles/event-driven-microservices-with-spring-boot-and-ac

https://medium.com/startlovingyourself/need-of-messaging-queues-in-microservices-architecture-91de0db89120

Ribbon server

