**MICROSERVICES**

**Monolithic Application: -** A Project which holds all modules together and converted as one Service (one .war file).

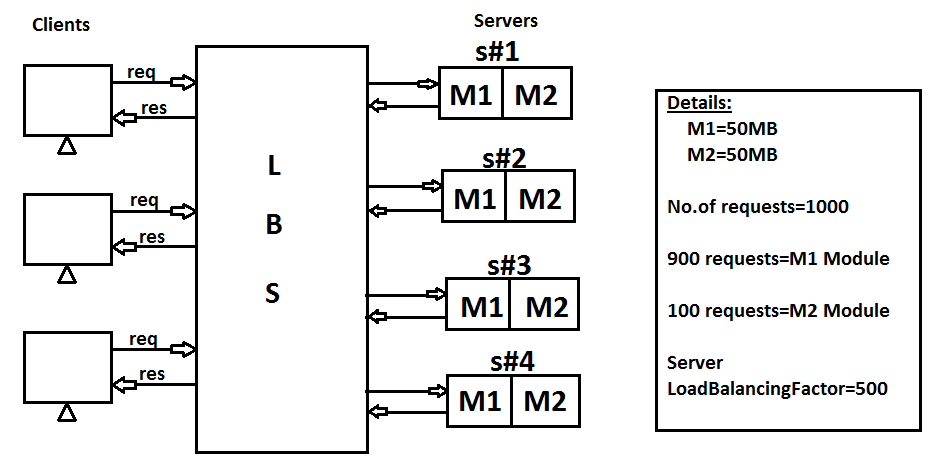
🡺In this case, if no. of users is getting increased, then to handle multiple request (load) use LBS (Load Balancing Server).

🡺But few modules need Extra load, not all. In this case other modules get memory which is waste (no use). Hence reduces performance of server (application).

🡺Consider Project P1 is having 2 modules M1, M2 and their runtime memories are M1=50 MB, M2=50 MB.

🡺Load Balancing is done using Servers looks like.

**Diagram:--**

****

🡺In above example, M2 Module is getting less requests from Client. So, max 2 instances are may be enough. Other 2 instances (memories) are no use. It means near 100MB memory is not used, which impacts server performance.

**Microservices:** It is an independent deployment component.

🡺It is a combination of one (or more) modules of a Projects runs as one Service.

**Nature of Microservices:**

1.Every Service must be implemented using Webservices concept.

2.Each service should be independent.

3.Services should able to communicate with each other. It is also called as **“Intra Communication**”.

4.Required Services must be supported for **Load Balancing** (i.e. one service runs in multiple instances).

5.Every service should able to read input data (\_\_\_\_.properties/\_\_\_\_\_.yml) from External **Configuration Server** [Config Server].

6.Service communication (Chain of execution) problems should be able to solve using **CircuitBreaker** [Find other possible…].

7.All Servers must be accessed to Single Entry known as **Gateway Service** [ Proxy Gateway or API Gateway], It supports **securing**, **metering** and **Routing**.

**Netflix Component Names:**

1.Service Registry and Discovery = Eureka

2.Load Balancing Server = Ribbon

3.Circuite Breaker = Hystrix

4.API Gateway = Zuul

5.Config Server = Github

6.Secure Server = OAuth2

7.Log and Trace = Zipkin + Sleuth

8.Message Queues = Kafka

9.Integration Service = Camel

10.Metrics UI = Admin (Server/Client)

11.Cloud Platform with Deploy services = PCF, Docker

**SOA (Service Oriented Architecture):**

🡺It is a Design Pattern used to create communication links between multiple services providers and users.

**Components of SOA:**

a. Service Registry and Discovery [Eureka]

b. Service Provider [Webservice Provider]

c. Service Consumer [Webservice Client]

**Operations:**

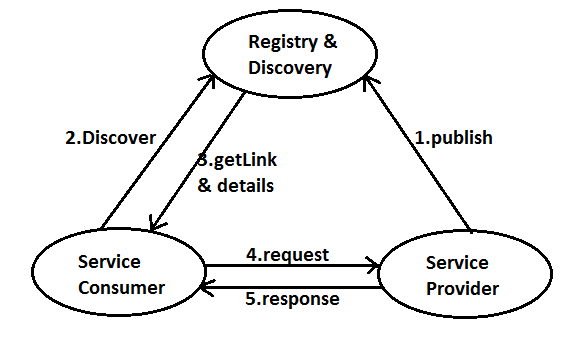
1.Publish

2.Discover

3.Link Details of Provider

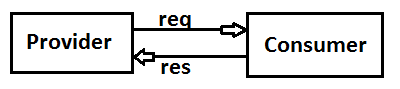
4.Query Description (Make Http Request).

5.Access Service (Http Response).



**Implementing MicroService Application Using Spring Cloud:**

**Design #1** A Simple Rest WebService using Spring Boot.



🡺This application is implemented using Spring Boot Restful webservices which provides Tightly coupled design. It means any changes in Provider application effects Consumer application, specially server changes, port number changes, context changes etc…,

🡺This design will not support LoadBalancing.

🡺It is implemented using RestController and RestTemplate.

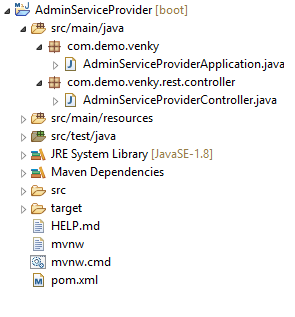
**Step #1 Create Provider Application (Dependencies: web only)**

Group Id: com.demo.venky

ArtifactId: AdminServiceProvider

Version: 1.0

**Folder Structure of AdminServiceProvider:**



**StarterClass(AdminServiceProviderApplication.java)**

**package** com.demo.venky;

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

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

@SpringBootApplication

**public** **class** AdminServiceProviderApplication {

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

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

System.***out***.println("=-=-=-Hello From AdminServiceProvider=-=-=-");

}

}

**Step #2 Define one RestController**

**package** com.demo.venky.rest.controller;

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

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

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

@RestController

@RequestMapping("/provider")

**public** **class** AdminServiceProviderController {

@GetMapping("/show")

**public** String showMsg() {

**return**"Hello Venky";

}

}

=--=-=application.properties=-=-=-=-=-

server.port=8900

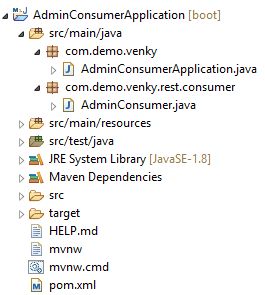
**Step #3 Create Consumer Application (Dependencies: web only)**

GroupId: com.demo.venky

ArtifactId: AdminServiceConsumer

Version: 1.0

**Folder Structure of AdminConsumerApplication:**



**Starterclass(AdminConsumerApplication.java)**

**package** com.demo.venky;

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

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

@SpringBootApplication

**public** **class** AdminConsumerApplication {

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

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

}

}

**Step #4 Define Consumer (call) code (AdminConsumer.java)**

**package** com.demo.venky.rest.consumer;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.stereotype.Component;

**import** org.springframework.web.client.RestTemplate;

@Component

**public** **class** AdminConsumer **implements** CommandLineRunner {

@Override

**public** **void** run(String... args) **throws** Exception {

RestTemplate rt=**new** RestTemplate();

ResponseEntity<String>resp=rt.getForEntity("http://localhost:8900/provider/show",String.**class**);

System.***out***.println(resp.getBody());

System.***out***.println("=-=-=-=-=-Message From Admin Consumer=-=-=-=-");

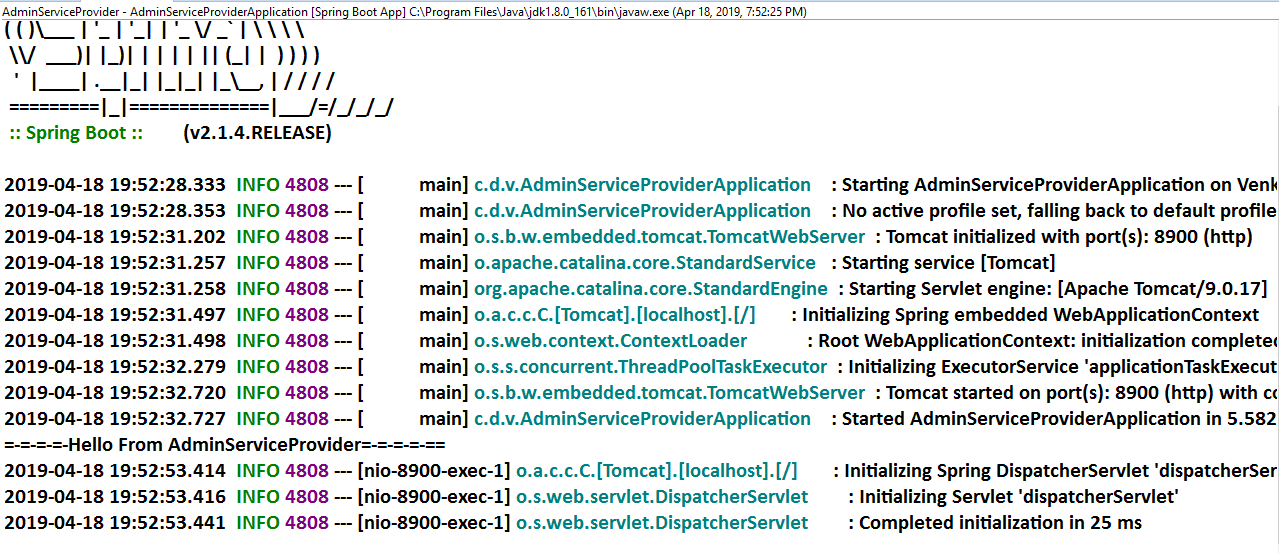
System.*exit*(0);

}

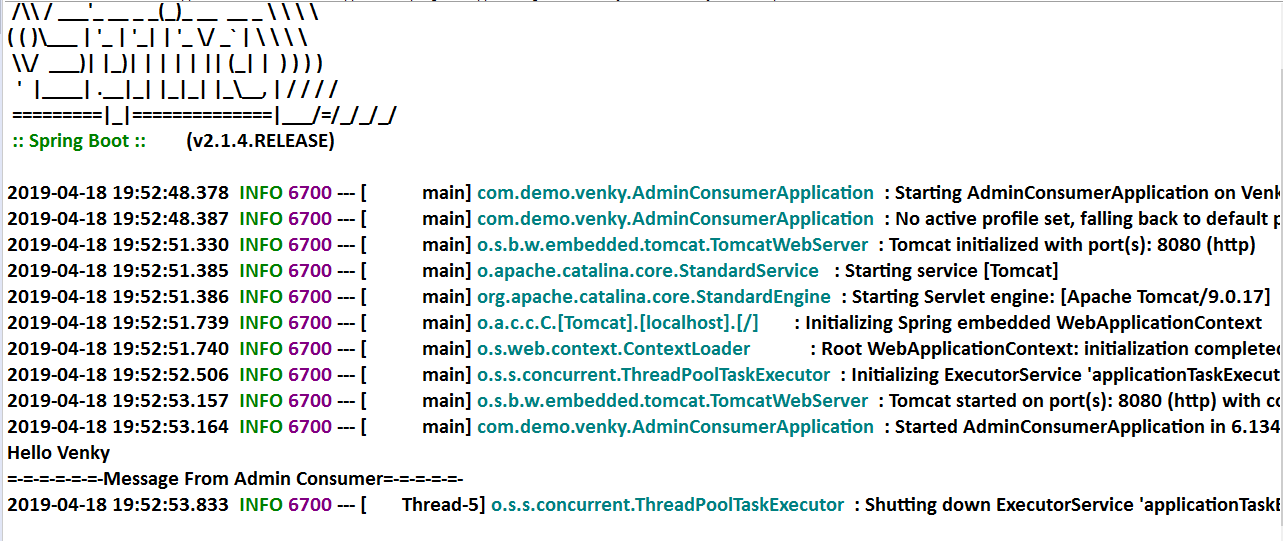
}

**Execution flow Screen:**

🡺First Run ProviderApplication (Starter), then ConsumerApplication(Starter)

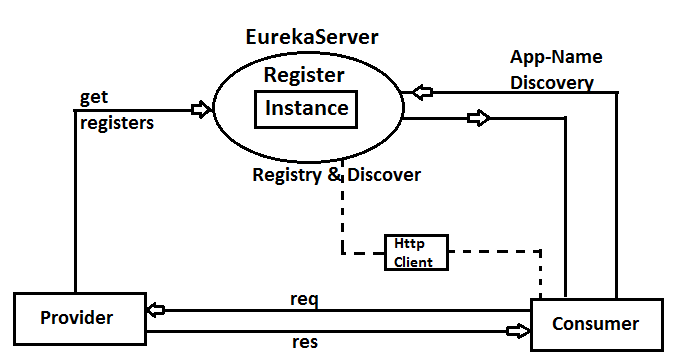
**PROVIDER SCREEEN:-**

**CONSUMER SCREEN:**



**MicroService Design and Implementation using Spring Cloud (Netflix Eureka Registry & Discovery)**

**Design #1# [Basic – No Load Balancing]**



**Step #1: Create Eureka Server:** Create one Spring Boot Starter Project with **Dependencies:** Eureka Server

**Eureka Server Dependencies:--**

<dependency>

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

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

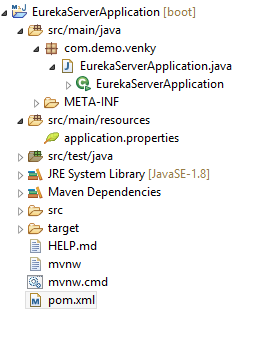
</dependency>

GroupId: com.demo.venky

ArtifactId: EurekaServerApp

Version: 1.0

**Folder Structure of Eureka Server:**



**Step #2:-** At **Starter class** level add Annotation @EnableEurekaServer Annotation

**package** com.demo.venky;

**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);

}

}

**Step #3:-** In **application.properties** add keys

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

**Step #4:-** Run starter class and Enter URL <http://localhost:8761> in browser

**NOTE:-** Default port no of Eureka Server is 8761.

**Screen Short of Eureka Server Dashboard:--**



**#2# Provider Application:**

**Step #1:** Create one Spring starter App with web and Eureka Discovery dependencies

<!-- web Dependencies -->

<dependency>

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

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

</dependency>

<!-- Eureka Discovery Dependencies -->

<dependency>

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

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

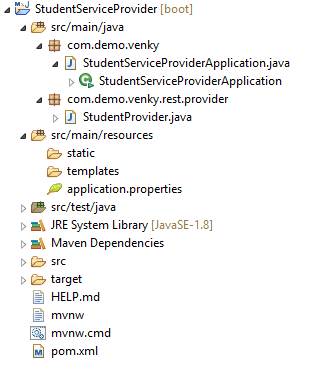
</dependency>

GroupId : com.demo.venky;

ArtifactId : StudentServiceProvider

Version : 1.0

**Folder Structure of Eureka Discovery for Provider Application:--**



**Step #2:-** Add below annotation at Starter class level @EnableEurekaClient (given by Netflix) or @EnableDiscoveryClient (given by Spring Cloud) both are optional.

**Spring Starter class:--**

**package** com.demo.venky;

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

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

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

@SpringBootApplication

@EnableEurekaClient

**public** **class** StudentServiceProviderApplication {

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

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

System.***out***.println("StudentServiceProvider");

}

}

**Step #3:-** In application.properties file

server.port=9800

spring.application.name=STUDENT-PROVIDER

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

**Step #4 Define one Provider Controller**

**package** com.demo.venky.rest.provider;

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

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

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

@RestController

@RequestMapping("/provider")

**public** **class** StudentProvider {

@GetMapping("/show")

**public** String showMsg() {

**return**"Hello From Provider";

}

}

**Execution Order: (RUN Starter Classes)**

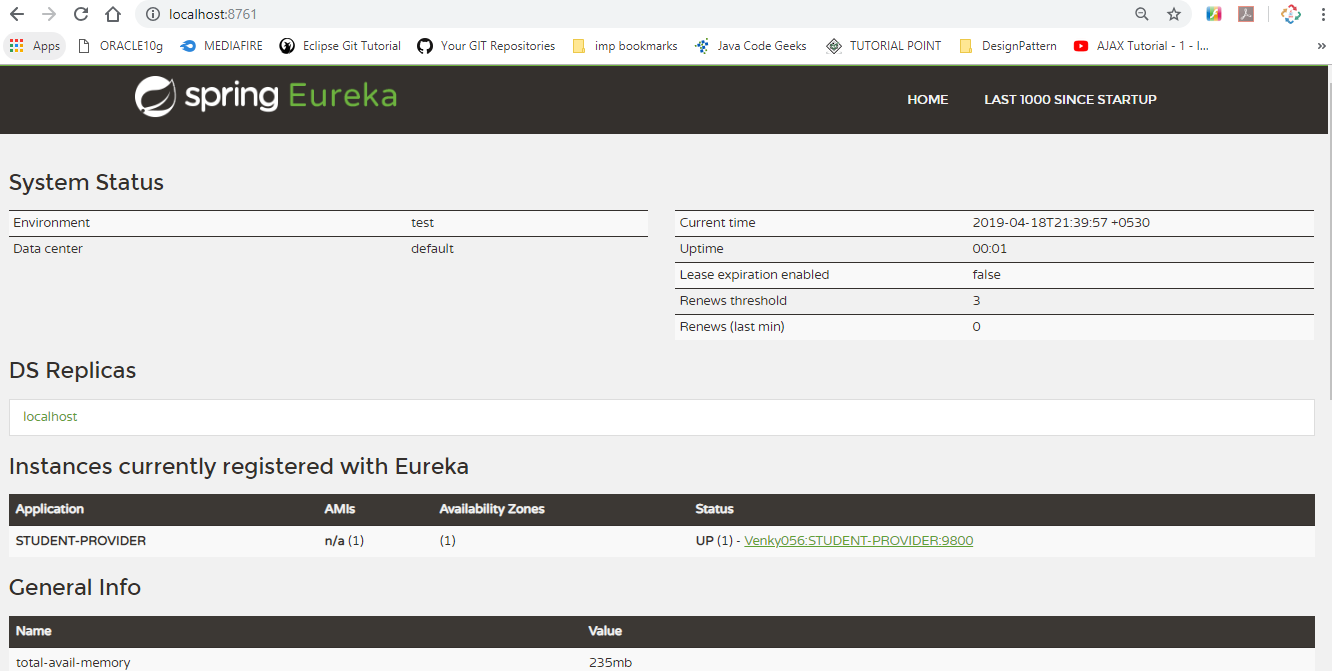
1. EUREKA SEVER

2. PROVIDER APPLICATION

🡺GOTO EUREKA Dashboard and CHECK FOR APPLICATION

🡺CLICK ON URL and ADD /provider/show🡺PATH

**#1 Screen Short of Eureka Server Dashboard: --**



**NOTE:--** Click on URL ([Venky056:STUDENT-PROVIDER:9800](http://venky056:9800/actuator/info)) then add provider path after URI (<http://venky056:9800/provider/show>)

**#2 Screen Short of Provider Application:--**



**#3# Consumer Application:**

In general Spring Boot application, by using any HTTP Client code, Consumer makes request based on URL (static/hard coded) given in code.

**\*\*\* Hard coding:** Providing a direct value to a variable in .java file or fixed for multiple runs.

🡺By using RestTemplate with URL (hard coded) we can make request. But it will not support.

a. If provider IP/PORT number gets changed

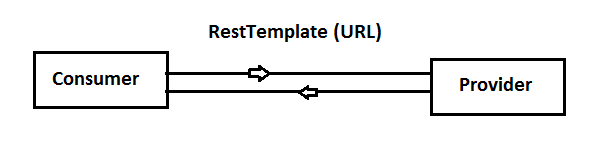
b. Load Balancing Implementation

🡺so, we should use Dynamic Client that gets URL at runtime based on Application name registered in “Registry and Discovery Server (Eureka)”.

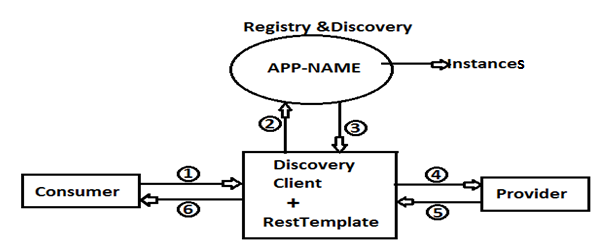
🡺DiscoveryClient is used to fetch Instance based on Application Name and we can read URI of provider at runtime.

🡺RestTemplate uses URI (+path = URL) and makes Request to Provider and gets ResponseEntity which is given back to the Consumer.

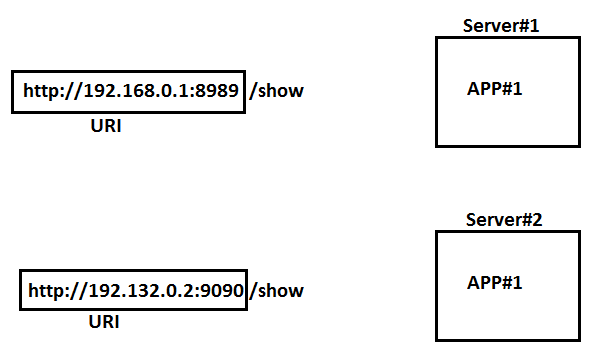
**Simple Web Service Example:--**

****

**Microservices Example:--**



🡺If one Application is moved from one Server to another server then URI gets changed (Paths remain same).



**Consumer code:--**

**Step #1: --** Create one Spring Starter Project using Dependencies web, Eureka Discovery

GroupId: com.demo.venky

ArtifactId: StudentServiceConsumer

Version: 1.0

**# Folder Structure of Eureka Discovery Consumer Application:--**



**Step #2:--** At Starter class level add Annotation either @EnableEurekaClient or @EnableDiscoveryClient (both are optional)

**package** com.demo.venky;

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

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

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

@SpringBootApplication

//@EnableEurekaClient

//(both are optional Annotations)

@EnableDiscoveryClient

**public** **class** StudentConsumerApplication {

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

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

}

}

**Step #3:**  in application. Properties file

server.port=9852

spring.application.name=STUDENT-CONSUMER

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

**Step #4: Define Consumer code with RestTemplate and DiscoveryClient**

**package** com.demo.venky.consumer;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.cloud.client.ServiceInstance;

**import** org.springframework.cloud.client.discovery.DiscoveryClient;

**import** org.springframework.http.ResponseEntity;

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

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

**import** org.springframework.web.client.RestTemplate;

@RestController

**public** **class** StudentConsumer {

@Autowired

**private** DiscoveryClient client;

@GetMapping("/consume")

**public** String consumeData() {

RestTemplate rt = **new** RestTemplate();

List<ServiceInstance> list=client.getInstances("STUDENT-PROVIDER");

ResponseEntity<String> resp =rt.getForEntity(list.get(0).getUri()+"/provider/show", String.**class**);

**return** "FROM CONSUMER=>" +resp.getBody();

}

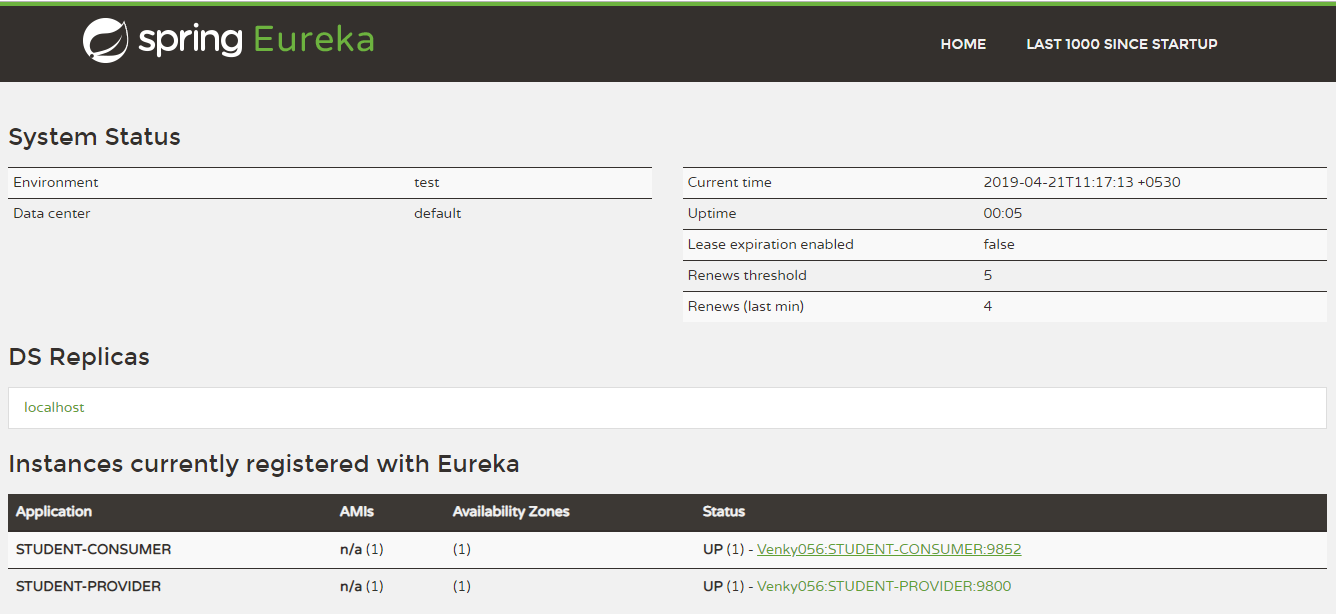
}

**Execution order: -**

#1 Run Starter classes in order

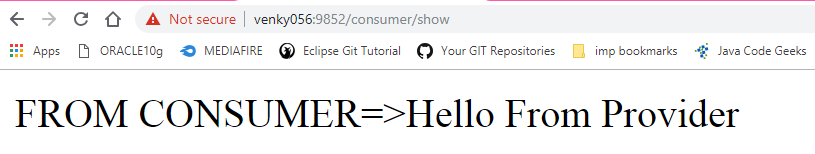
Eureka server, Provider Application, Consumer Application

#2 Go to Eureka and client Consumer URL enter /consumer path after PORT number.



**NOTE:--** Click on URL ([Venky056:STUDENT-CONSUMER:9852](http://venky056:9800/actuator/info)) then add provider path after URI (<http://venky056:9800/consumer/show>)

**Output:**



**Declarative ReST Client : [Feign Client]**

🡺Spring cloud supports any HTTP Client to make communication between (Microservices) Provider and Consumer.

🡺RestTemplate is a **legacy style** which is used to make HTTP calls with URL and Extra inputs.

🡺RestTemplate with DiscoveryClient makes mask to Provider URL. It means works based on Application Name (Service ID). Even URI gets changed it works without any modification at consumer side.

🡺RestTemplate combination always makes Programmer to write manual coding for HTTP calls.

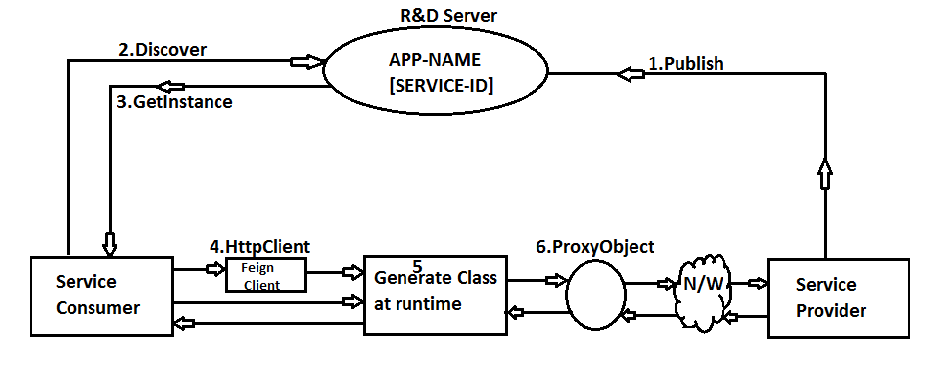
🡺Spring Cloud has Provided one ActingClient [which behaves as Client, but not].

It means, Provide **Abstraction** at code level by programmer and Implementation is done at **runtime** by Spring cloud.

🡺Feign is **Declarative Client**, which supports generating code at runtime and **proxy Object.** By using **Proxy HTTP Request** calls can be made.

🡺It supports even Parameters (Path/Query…)and Global Data Conversion (XML/JSON).

**Diagram:**

****

FeignClient is an **Interface** and contains abstraction details, like:

a. Path (Provider paths at class and method).

b. Http Method Type (GET, POST…).

c. ServiceId (Application Name).

d. Input Details and Output Type.

🡺We need to apply Annotation at starter class level **@EnableFeignClients.**

🡺At interface level apply **@FeignClient(name=”serviceId”).**

**Syntax:** Feign Client

@FeignClient(name=”serviceId”)

public interface <ClientName> {

@GetMapping(“/path”)

//or @RequestMapping(“path”)

public <return> <method>(<params>);

……..

}

**Example: (**Provider Code (SID: EMP-PROV))

@RestController

@RequestMapping(“/emp”)

public class EmpProvider {

@GetMapping(“/show”)

Public String findMsg () {

……

}

}

Consumer Code: Feign Client

@FeignClient (name=” EMP\_PROV”)

public interface EmpConsumer {

@GetMapping (“/emp/show”)

public String getMsg (); //return type and path must be same as Provider

}

-----------------------------------------------------------------------------------------------------------------

(\*\*\*)🡺Consider last Example **Eureka Server** and **Provider Application** (STUDENT PROVIDER)

**Step #1** Create one Spring Boot Starter Project for Consumer (using Feign, web, Eureka Discovery)

GroupId: com.demo.venky

ArtifactId: StudentServiceConsumerFeign

Version: 1.0

**Step #2** Define one public interface as

package com.app.client;

@FeignClient(name=”STUDENT-PROVIDER”)

Public interface StudentFeignClient {

@GetMapping(“/show”)

Public String getMsg();

}

**Step #3** Use in any consumer class (HAS-A) and make method call (HTTP CALL)

package com.app.consumer;

@RestController

public class StudentConsumer {

@Autowired

private StudentFeignClient client;

@GetMapping(“/consumer”)

Public String showData () {

System.out.pritln(client.getClass().getName());

Return “CONSUMER=>”+client.getMsg();

}}

**NOTE:** Here client.getMsg() method is nothing but HTTP Request call.

**LoadBalancing in SpringCloud(MicroServices)**

🡺To handle Multiple Requests made by any HTTPClient (or consumer) in lesstime, one Provider should **run as multiple instances** and **handle request in parallel** such concept is called as “**LoadBalancing”.**

🡺LoadBalancing is to make request handling faster (reduce waiting time in queue).

**Steps to implement LoadBalancing: -**

1. Create one Provider Application
2. Register one Provider as multiple instances in Registry and Discovery [Eureka] server. Every instances with uniqueId.

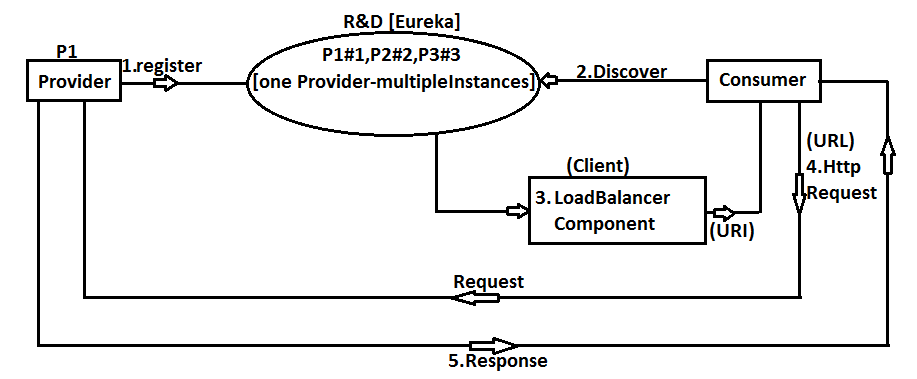
Ex: P1-58266, P2-23345, P3-64785 etc...,

1. Define Consumer with anyone LoadBalancerComponent (Ex: Ribbon, Feign)
2. Ribbon chooses one Provider URI, based on InstanceId with help of **LBSRegister** which maintains request count.
3. Consumer will add Paths to URI and makes Request using **“RequestClient”.**

[Ex: LoadBalancerClient(I) or @FeignClient]

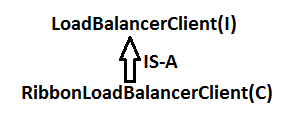
1. ResponseEntity is returned by Provider to Consumer.

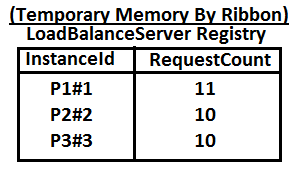
**Diagram:**



**🡺**LoadBalancerComponent: Ribbon, Feign

🡺Request Component:





**Ribbon:**

🡺It is a Netflix component provided for SpringBootCloudLoadBalancing.

🡺It is also called as **“ClientSideLoadBalancing”.** It means ConsumerApp,reads URI (which one is free) using LBS Register.

🡺Ribbon should be enabled by every consumer.

🡺spring cloud uses LoadBalancerClient(I) for **“choose and invoke”** process.

🡺Its implementation is provided by Ribbon.

**=-=-CodingSteps=-=-=**

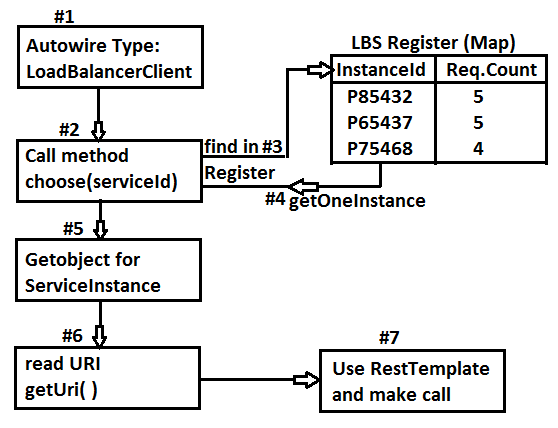
**#1** In Provider, define InstanceId for PROVIDER APPLICATION using key “eureka.instance.instance-id”**.** If not provided default is taken as SpringApp name.

**eureka.instance.instance-id=${spring.application.name}:${random.value}**

[add in application.properties]

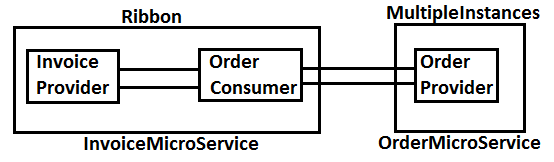
**#2** In Consumer, add dependency for Ribbon and use LoadBalancerClient (Autowired) and call choose(“APP-NAME”) method to get ServiceInstance (for URI)

**Consumer Execution flow for Request:**



**Consider below Example for Ribbon:**





**Step #1** Configure Eureka Server Dependencies : Eureka Server only

**FolderStructure:**



**EurekaServerCloudApplication.java**

**package** com.app;

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

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

**import** org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

**public** **class** EurekaServerCloudApplication {

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

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

}

}

**application.properties**

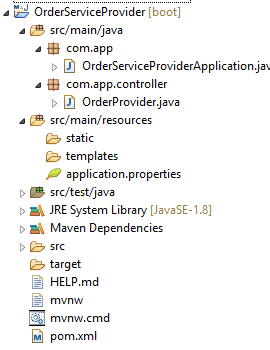
server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

**Step #2** Create Order Service Provider Application Dependencies : Eureka Discovery, Web

**FolderStructure:**



**OrderServiceProviderApplication.java:**

**package** com.app;

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

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

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

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** OrderServiceProviderApplication {

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

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

}

}

**OrderProvider.java:**

**package** com.app.controller;

**import** org.springframework.beans.factory.annotation.Value;

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

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

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

@RestController

@RequestMapping("/order")

**public** **class** OrderProvider {

@Value("${server.port}")

**private** String port;

@GetMapping("/status")

**public** String getOrderStatus() {

**return** "FINISHED::"+port;

} }

**application.properties:**

server.port=9803

spring.application.name=ORDER-PROVIDER

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

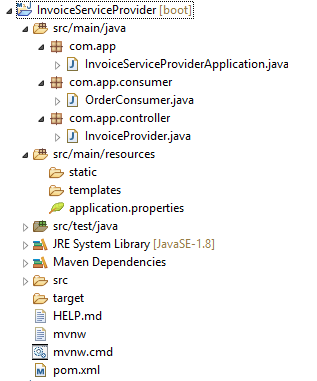
eureka.instance.instance-id=${spring.application.name}:${random.value}

🡺If no instance-id is provided then application name (service Id) behaves as instance -Id

**Step #3** Define Invoice Service Provider

Dependencies : Eureka Discovery, Web, Ribbon

**FolderStructure:**



**InvoiceServiceProviderApplication.java:**

**package** com.app;

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

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

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

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** InvoiceServiceProviderApplication {

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

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

}

}

**application.properties:**

server.port=8800

spring.application.name=INVOICE-PROVIDER

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

**OrderConsumer.java:**

**package** com.app.consumer;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.cloud.client.ServiceInstance;

**import** org.springframework.cloud.client.loadbalancer.LoadBalancerClient;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.stereotype.Service;

**import** org.springframework.web.client.RestTemplate;

@Service

**public** **class** OrderConsumer {

@Autowired

**private** LoadBalancerClient client;

**public** String getStatus() {

String path="/order/status";

ServiceInstance instance=client.choose("ORDER-PROVIDER");

String uri=instance.getUri().toString();

RestTemplate rt=**new** RestTemplate();

ResponseEntity<String> resp=rt.getForEntity(uri+path, String.**class**);

**return** "CONSUMER=>"+resp.getBody();

}

}

**InvoiceProvider.java:**

**package** com.app.controller;

**import** org.springframework.beans.factory.annotation.Autowired;

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

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

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

**import** com.app.consumer.OrderConsumer;

@RestController

@RequestMapping("/invoice")

**public** **class** InvoiceProvider {

@Autowired

**private** OrderConsumer consumer;

@GetMapping("/info")

**public** String getOrderSatus() {

**return** consumer.getStatus();

} }

**Execution:**

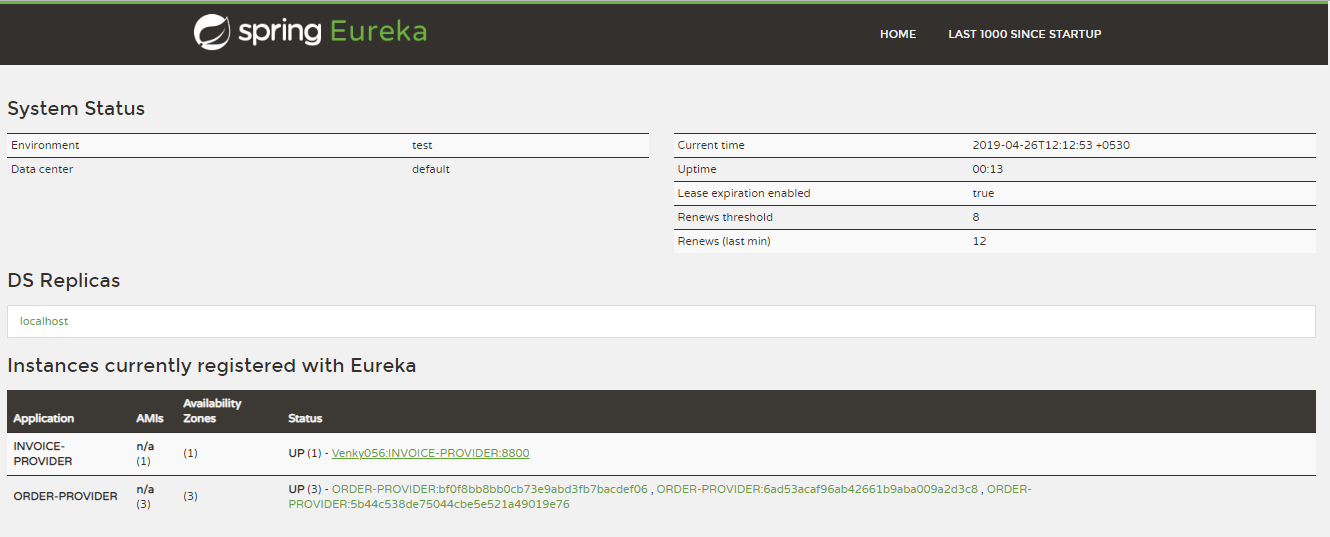
1. start Eureka Server
2. Run OrderProvider starter class 3 times

\*\*\* Change every time port number like: 9800,9801, 9802

1. Run InvoiceProvider Starter class
2. Goto Eureka and Execute Invoice Provider Instance type full URL

<http://localhost:8080/invoice/info>

**outputs:**





**Load Balancing using Feign Client**

In case of Manual Coding for load Balancing Ribbon Component is used with Type “LoadBalancingClient” (I).

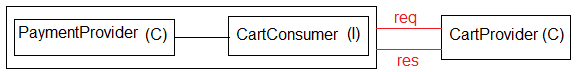
🡺Here, using this programmer has to define logic of consumer method.

🡺Feign Client reduces coding lines by Programmer, by generating logic/code at runtime.

🡺Feign Client uses abstraction Process, means Programmer has to provide path with Http Method Type and also input, output details.

🡺At Runtime RibbonLoadBalancerClient instance is used to choose serviceInstance and make HTTP call.

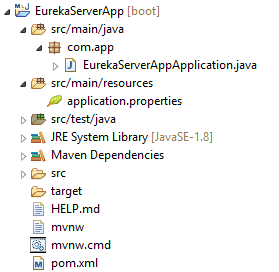




**Example:**

**Step #1:** Create Spring Starter Project for Eureka Server with port 8761 and dependency “EurekaServer”.

**FolderStructure:**



**EurekaServerAppApplication.java:**

**package** com.app;

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

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

**import** org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

**public** **class** EurekaServerAppApplication {

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

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

}

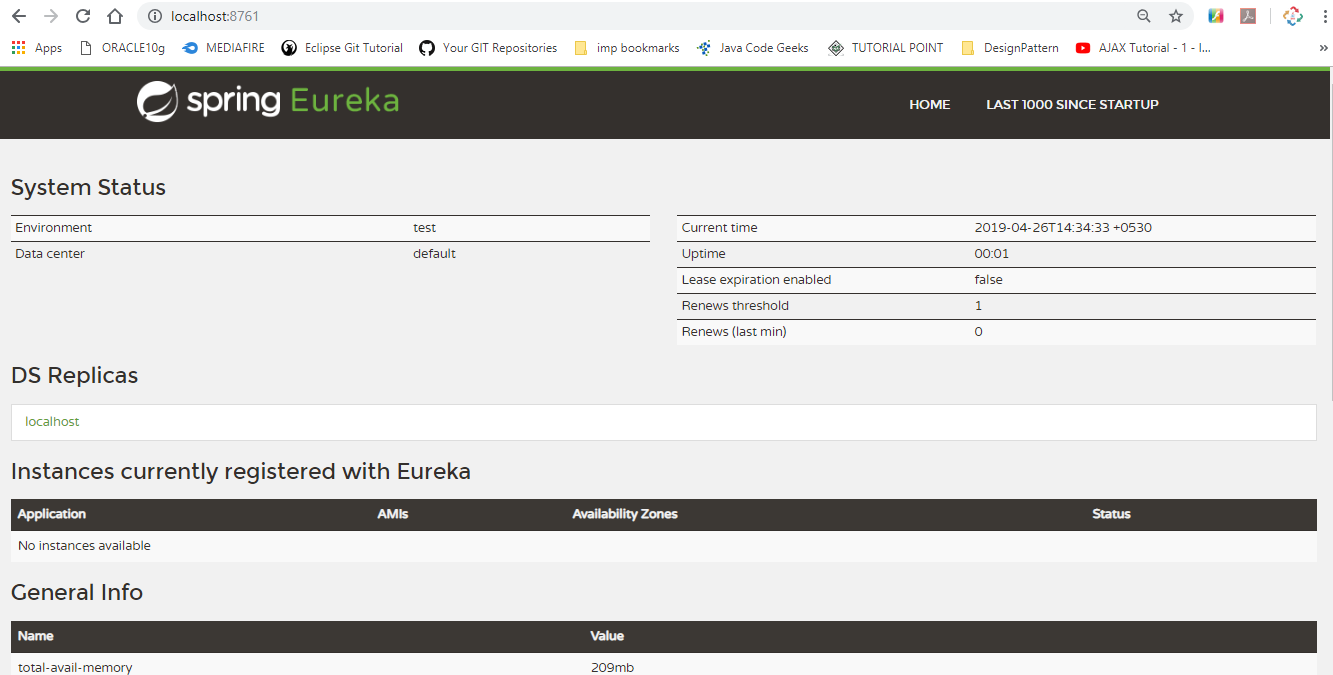
}

**application.properties:**

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

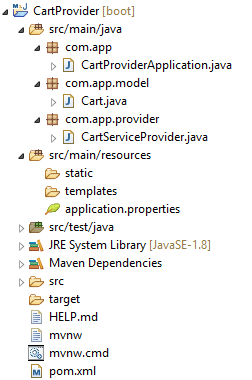


**Step #2:** Cart provider Application

Dependencies : web, EurekaDiscovery

🡺At starter class level : @EnableDiscoveryClient

**FolderStructure:**



**CartProviderApplication.java:**

**package** com.app;

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

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

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

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** CartProviderApplication {

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

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

}

}

**Cart.java:**

**package** com.app.model;

**public** **class** Cart {

**private** Integer cartId;

**private** String cartCode;

**private** Double cartFinalCost;

**public** Cart() {

**super**();

}

**public** Cart(Integer cartId, String cartCode, Double cartFinalCost) {

**super**();

**this**.cartId = cartId;

**this**.cartCode = cartCode;

**this**.cartFinalCost = cartFinalCost;

}

**public** Integer getCartId() {

**return** cartId;

}

**public** **void** setCartId(Integer cartId) {

**this**.cartId = cartId;

}

**public** String getCartCode() {

**return** cartCode;

}

**public** **void** setCartCode(String cartCode) {

**this**.cartCode = cartCode;

}

**public** Double getCartFinalCost() {

**return** cartFinalCost;

}

**public** **void** setCartFinalCost(Double cartFinalCost) {

**this**.cartFinalCost = cartFinalCost;

}

@Override

**public** String toString() {

**return** "Cart [cartId=" + cartId + ", cartCode=" + cartCode + ", cartFinalCost=" + cartFinalCost + "]";

}

}

**CartServiceProvider.java:**

**package** com.app.provider;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Value;

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

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

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

**import** com.app.model.Cart;

@RestController

@RequestMapping("/cart")

**public** **class** CartServiceProvider {

@Value("${server.port}")

**private** String port;

@GetMapping("/info")

**public** String getMsg() {

**return** "CONSUMER:"+port;

}

@GetMapping("/data")

**public** Cart getObj() {

**return** **new** Cart(109, "ABC:"+port, 636.36);

}

@GetMapping("/list")

**public** List<Cart> getBulk() {

**return** Arrays.*asList*(

**new** Cart(101, "A:"+port, 636.36),

**new** Cart(102, "B:"+port, 526.46),

**new** Cart(103, "C:"+port, 839.38)

);

}

}

**application.properties:**

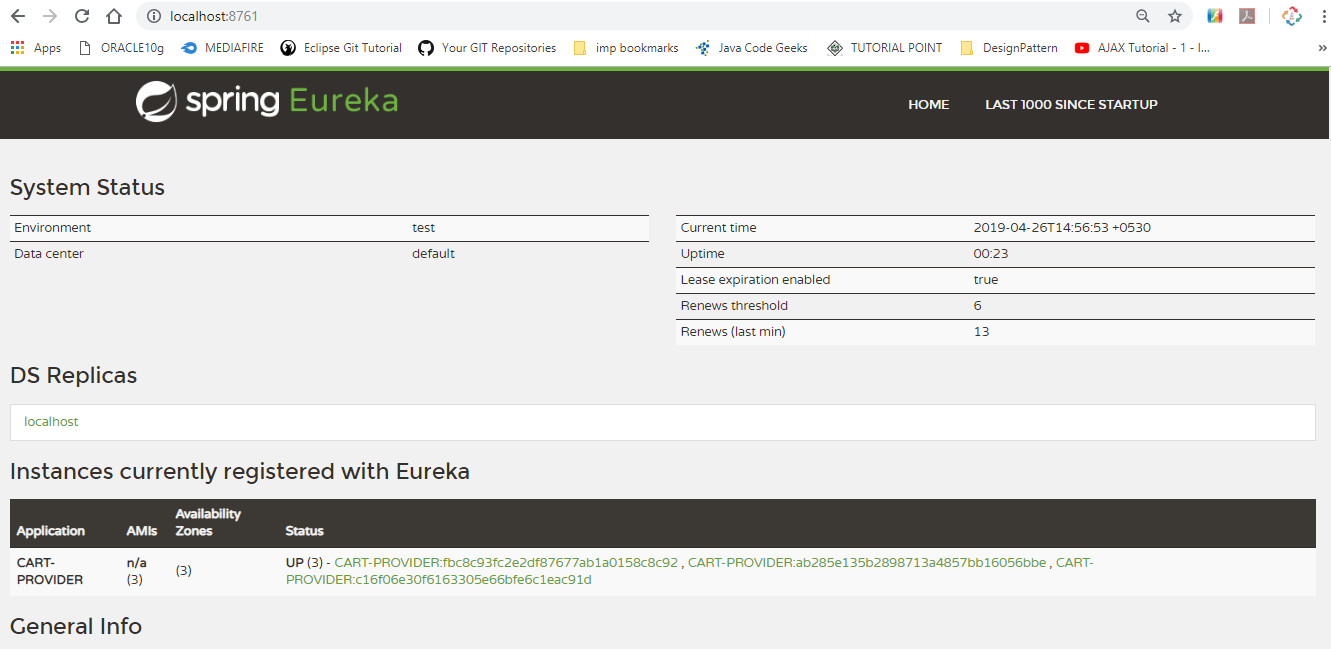
server.port=8603

spring.application.name=CART-PROVIDER

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

eureka.instance.instance-id=${spring.application.name}:${random.value}

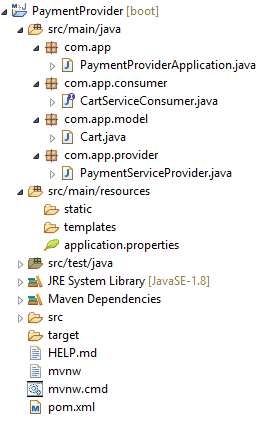
**output:**



**Step #3:** Payment Provider App with Cart Consumer code

Dependencies : web, EurekaDiscovery, Feign

**FolderStructure:**



\*\* At Starter class level : @EnableFeignClients

**PaymentProviderApplication.java:**

**package** com.app;

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

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

**import** org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableFeignClients

**public** **class** PaymentProviderApplication {

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

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

}

}

**CartServiceConsumer.java:**

**package** com.app.consumer;

**import** java.util.List;

**import** org.springframework.cloud.openfeign.FeignClient;

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

**import** com.app.model.Cart;

@FeignClient(name="CART-PROVIDER")

**public** **interface** CartServiceConsumer {

@GetMapping("/cart/info")

**public** String getMsg() ;

@GetMapping("/cart/data")

**public** Cart getObj() ;

@GetMapping("/cart/list")

**public** List<Cart> getBulk();

}

**Cart.java:**

**package** com.app.model;

**public** **class** Cart {

**private** Integer cartId;

**private** String cartCode;

**private** Double cartFinalCost;

**public** Cart() {

**super**();

}

**public** Cart(Integer cartId, String cartCode, Double cartFinalCost) {

**super**();

**this**.cartId = cartId;

**this**.cartCode = cartCode;

**this**.cartFinalCost = cartFinalCost;

}

**public** Integer getCartId() {

**return** cartId;

}

**public** **void** setCartId(Integer cartId) {

**this**.cartId = cartId;

}

**public** String getCartCode() {

**return** cartCode;

}

**public** **void** setCartCode(String cartCode) {

**this**.cartCode = cartCode;

}

**public** Double getCartFinalCost() {

**return** cartFinalCost;

}

**public** **void** setCartFinalCost(Double cartFinalCost) {

**this**.cartFinalCost = cartFinalCost;

}

@Override

**public** String toString() {

**return** "Cart [cartId=" + cartId + ", cartCode=" + cartCode + ", cartFinalCost=" + cartFinalCost + "]";

}

}

**PaymentServiceProvider.java:**

**package** com.app.provider;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

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

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

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

**import** com.app.consumer.CartServiceConsumer;

**import** com.app.model.Cart;

@RestController

@RequestMapping("/payment")

**public** **class** PaymentServiceProvider {

@Autowired

**private** CartServiceConsumer consumer;

@GetMapping("/message")

**public** String getMsg() {

**return** consumer.getMsg();

}

@GetMapping("/one")

**public** Cart getOneRow() {

**return** consumer.getObj();

}

@GetMapping("/all")

**public** List<Cart> getAllRows(){

**return** consumer.getBulk();

}

}

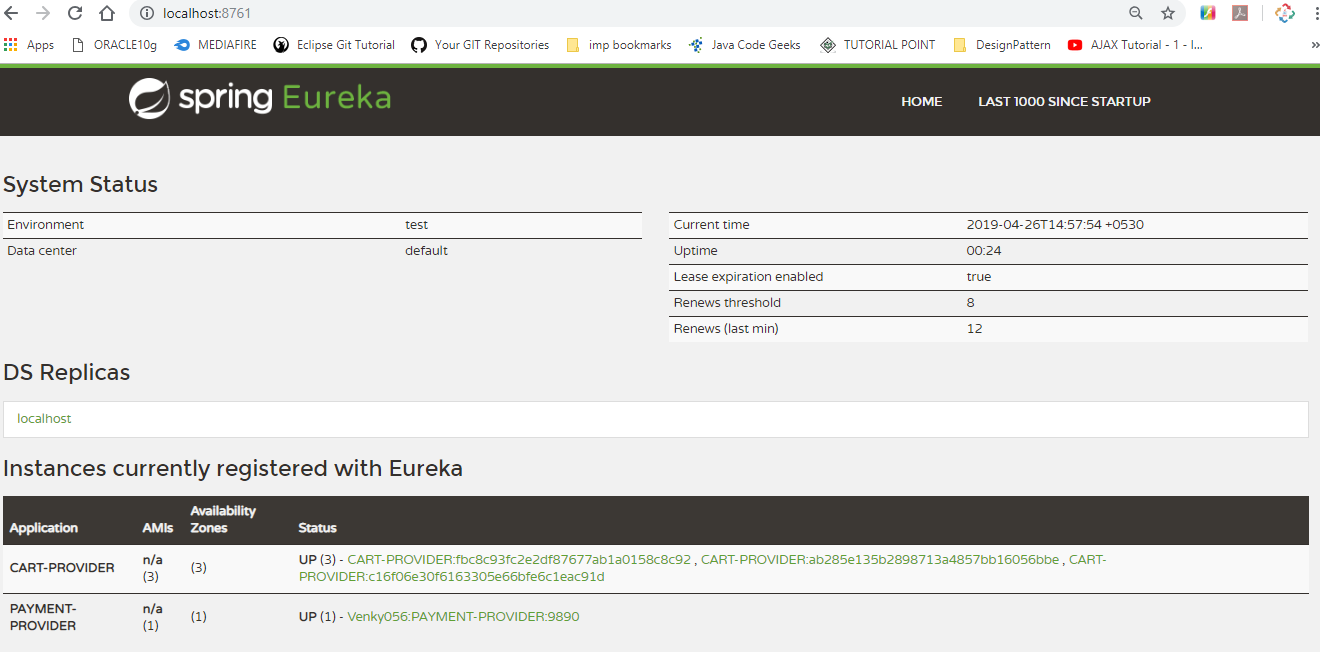
**application.properties:**

server.port=9890

spring.application.name=PAYMENT-PROVIDER

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

**output:**



**Step #4** Execution Order:

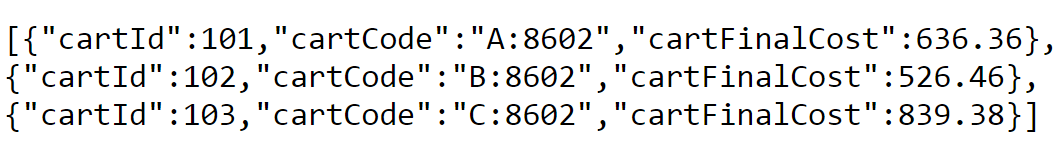
🡺Run Eureka Server

🡺Run Cart Provider 3 times (with different port)

🡺run Payment Provider 1 time

🡺Go to Eureka server and Run Payment service

**Output:**



**Spring Cloud Confgi Server**

🡺It is also called as Configuration Server. In Spring Boot/Cloud Projects, it contains files like : **properties files [application.properties]**

🡺In some cases Key=Values need to be changed (or) new key=value need to be added. At this time, we should

--> Stop the Server (Application)

--> Open/find application.properties file

--> Add External key=value pairs or do modifications.

--> Save changes [save file]

--> Re-build file [re-create jar/war]

--> Re-Deploy [re-start server]

🡺And this is done in All related Project [ multiple microservices ] which is repeated task for multiple applications.

**\*#\***To avoid this repeated (or lengthy) process use **“application.properties”** which is placed outside of your Project. i.e. known as **“ConfigServer”**.

🡺Config server process maintains three (3) properties file. Those are:

1. One in = Under Project (Microservice)
2. One in = Config Server (link file)
3. One in = Config server(native) (or) outside ConfigServer(External)also called

as Source file.

🡺Spring cloud Config server can be handled in two ways. Those are

1. Native Config Server
2. External Config Server

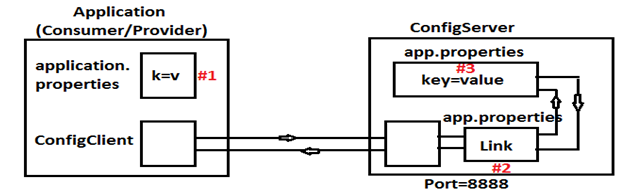
**1.Native Config Server:** It is also called as local file placed in Config server only.

**2.External Config Server:** In this case properties file is placed outside the Config server. Ex: GIT (github)

🡺In Consumer/Producer Project we should add ConfigClient dependency which gets config server details at runtime.

**#1(NativeConfigServer )**

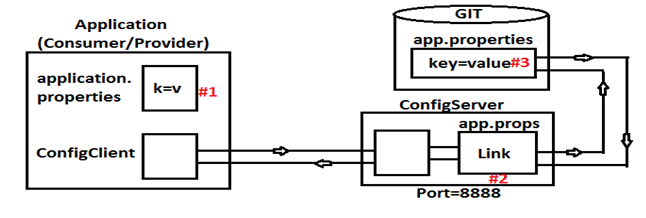
**LocalFileConfigServer**



🡺Config server runs at default port=8888.

**2# (GIT ConfigServer)**

**ExternalConfigServer**



**Steps to implement Spring Cloud Config Server (-Native & External-) :-**

**Step#1** Create SpringBoot Starter Project for **“ConfigServer”.**

<dependency>

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

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

</dependency>

**Step#2** Provide key value in properties files

**Case#a) native**

**application.properties**

##link file##

server.port=8888

spring.profiles.active=native

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

**case#b) External**

**application.properties**

server.port=8888

spring.cloud.config.server.git.uri=https://github.com/venkatadri053/config-server-ex

**Step#3** Create sub folder “myapp-config” in src/main/resources folder.

Create file application.properties inside this folder. It behaves like source

Having ex key:

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

(or)

Create Git Project “configserverex” and create application.properties inside this. place above key and save file.

**Step#4** Provide include resource code in pom.xml

To Consider properties files to be loaded into memory.

<resources>

<resource>

<filtering>true</filtering>

<directory>src/main/resources</directory>

<includes>

<include>\*.properties</include>

<include>myapp-config</include>

</includes>

</resource>

</resources>

**Step#5** At Starter class of ConfigServer App,add Annotation: **@EnableConfigServer**

**Step#6** InConsumer/Provider Projects pom.xml file add dependency: **Config Client** (or copy below dependency)

<dependency>

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

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

</dependency>

**---ExecutionOrder----**

🡺Eureka Server 🡺Config Server 🡺Producer(Provider) 🡺Consumer

**Steps for Coding**

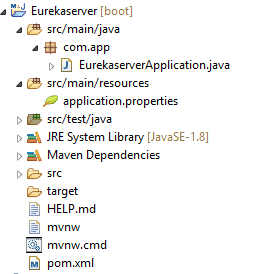
**Step#1 Eureka Server Project**

🡺Dependency: Eureka Server

🡺Write application.properties

🡺Add @EnableEurekaServer annotation

**FolderStructure:**



**EurekaserverApplication.java:**

**package** com.app;

**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);

}

}

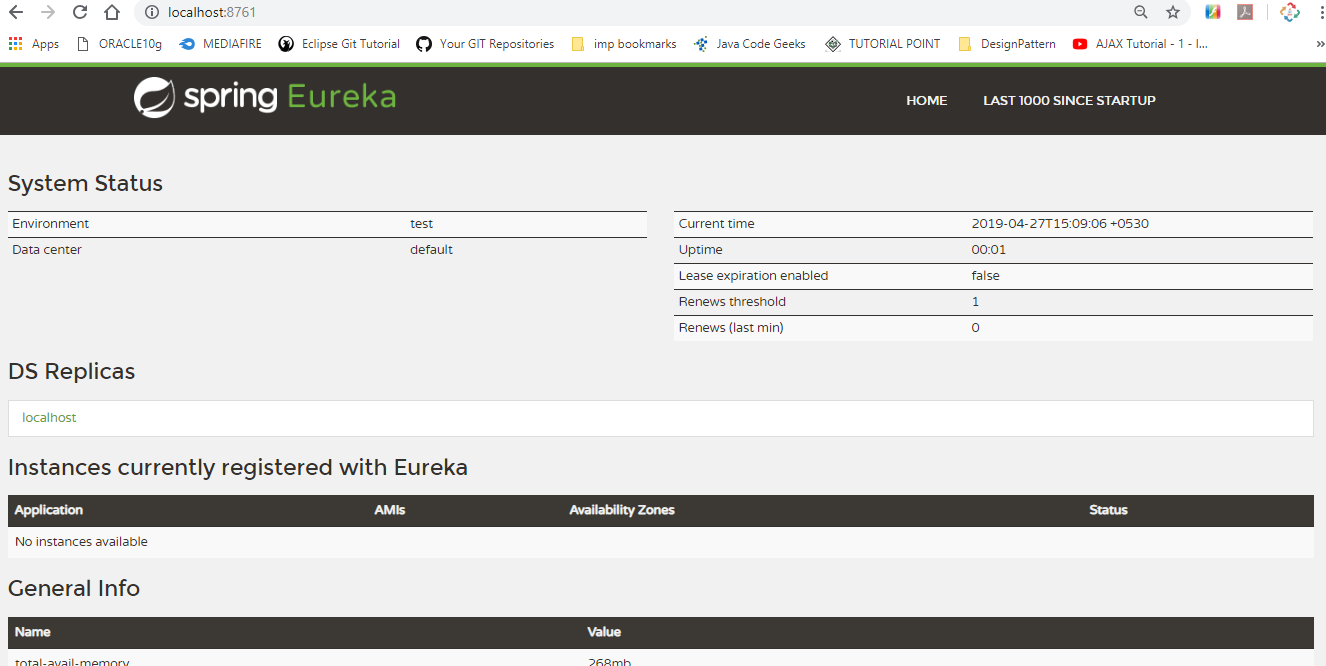
**application.properties:**

server.port=8761

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

**output:**



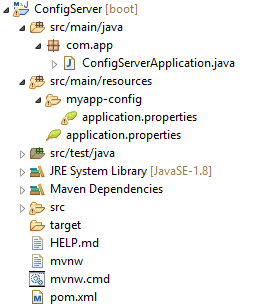
**Step#2 Define Config server Project**

🡺Dependency:Config Server

🡺Write application.properties

🡺Add @EnableConfigServer annotation

**FolderStructure:**



**ConfigServerApplication.java:**

**package** com.app;

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

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

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

@SpringBootApplication

@EnableConfigServer

**public** **class** ConfigServerApplication {

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

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

}

}

**myapp.properties:**

##Source Fie##

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

**application.properties:**

##link file##

server.port=8888

spring.profiles.active=native

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

**Step#3 Create Provider Project (Order)**

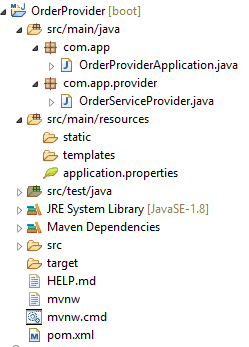
🡺Dependency : web,eureka discovery,Config client

🡺Write application.properties

🡺Add @EnableDiscoveryClient annotation

🡺Writer Provider (RestController)

**FolderStructure:**



**InvoiceProviderApplication.java:**

**package** com.app;

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

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

**import** org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableFeignClients

**public** **class** InvoiceProviderApplication {

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

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

}

}

**OrderConsumer.java:**

**package** com.app.consumer;

**import** org.springframework.cloud.openfeign.FeignClient;

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

@FeignClient(name="ORDER-PROVIDER")

**public** **interface** OrderConsumer {

@GetMapping("/order/show")

**public** String showMsg();

}

**InvoiceServiceProvider.java:**

**package** com.app.provider;

**import** org.springframework.beans.factory.annotation.Autowired;

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

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

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

**import** com.app.consumer.OrderConsumer;

@RestController

@RequestMapping("/invoce")

**public** **class** InvoiceServiceProvider {

@Autowired

**private** OrderConsumer consumer;

@GetMapping("/info")

**public** String getConsumerMsg() {

**return** "Consumer=>"+consumer.showMsg();

}

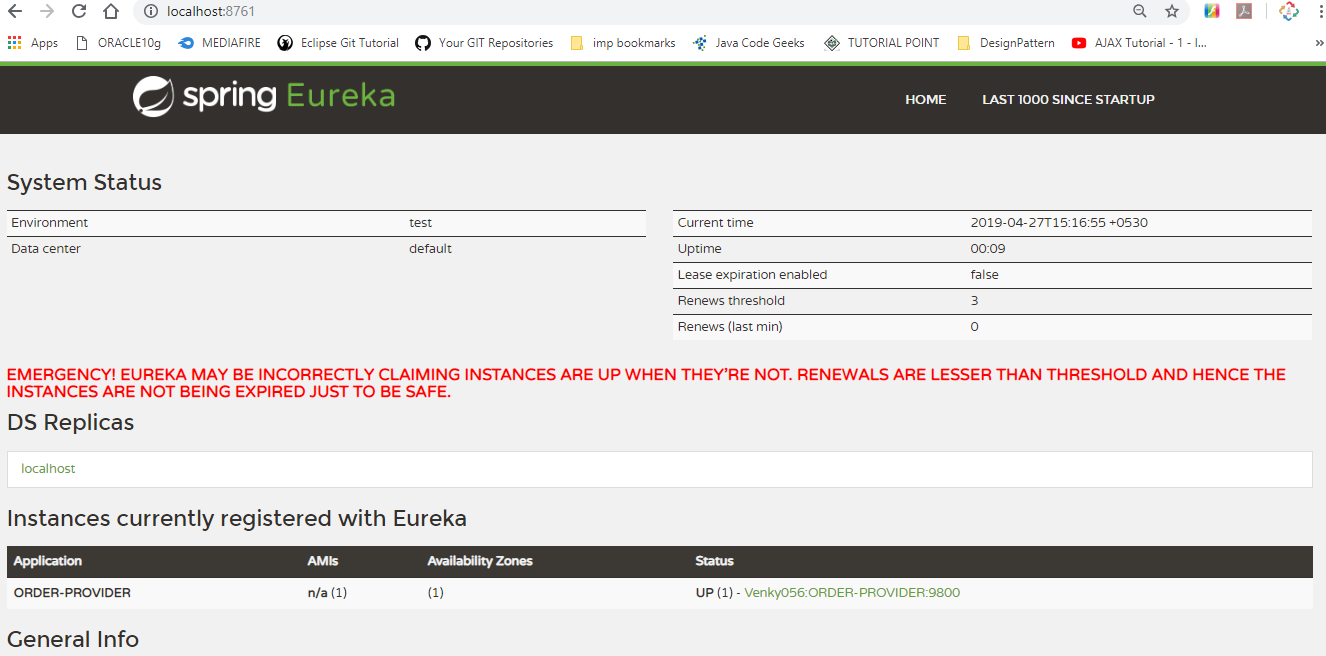
}

**application.properties:**

server.port=9500

spring.application.name=INVOICE-PROVIDER

**Output:**



**Step#4 Create Consumer Project (Invoice)**

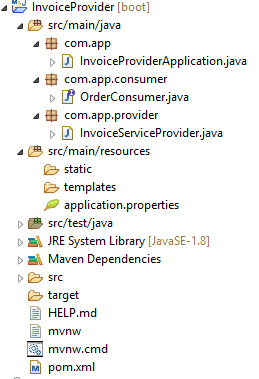
🡺Dependency : web,Eureka Discovery,Config client,Feign

🡺Write application.properties

🡺Add @EnableFeignClient annotation

🡺Writer Consumer [Feign Client] and Invoice provider (RestController)

**FolderStructure:**



**InvoiceProviderApplication.java:**

**package** com.app;

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

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

**import** org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableFeignClients

**public** **class** InvoiceProviderApplication {

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

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

}

}

**OrderConsumer.java:**

**package** com.app.consumer;

**import** org.springframework.cloud.openfeign.FeignClient;

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

@FeignClient(name="ORDER-PROVIDER")

**public** **interface** OrderConsumer {

@GetMapping("/order/show")

**public** String showMsg();

}

**InvoiceServiceProvider.java:**

**package** com.app.provider;

**import** org.springframework.beans.factory.annotation.Autowired;

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

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

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

**import** com.app.consumer.OrderConsumer;

@RestController

@RequestMapping("/invoce")

**public** **class** InvoiceServiceProvider {

@Autowired

**private** OrderConsumer consumer;

@GetMapping("/info")

**public** String getConsumerMsg() {

**return** "Consumer=>"+consumer.showMsg();

}

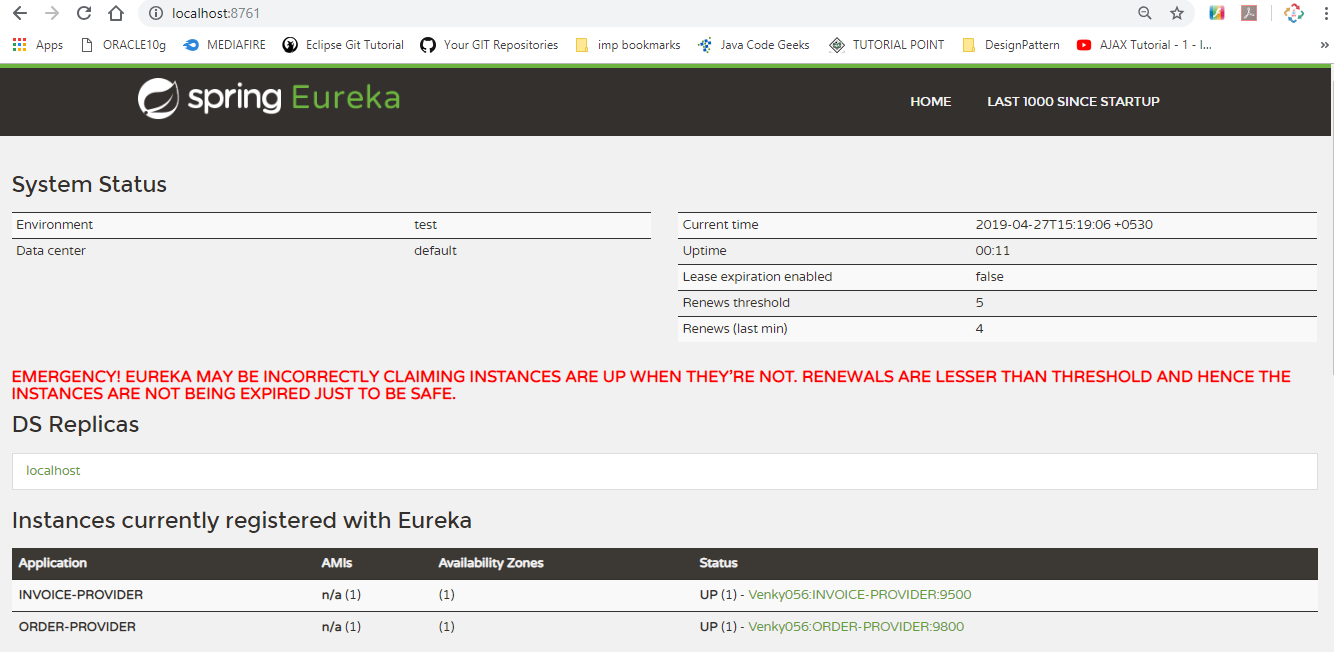
}

**application.properties:**

server.port=9500

spring.application.name=INVOICE-PROVIDER

**Output:**





**Steps to configure External ConfigServer:(only modifications)**

**Step#1** in config server project,delete folder myapp-config

**Step#2** in config Server Project modify application.properties with Git URI

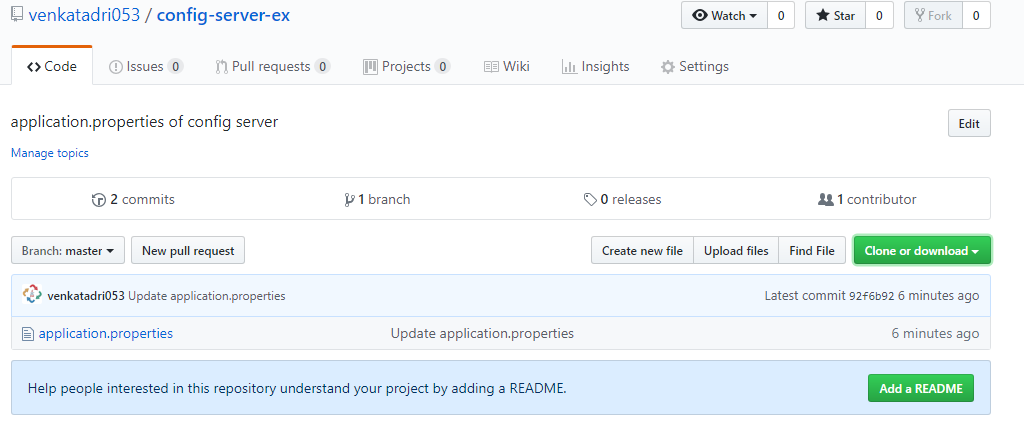
server.port=8888

spring.cloud.config.server.git.uri=https://github.com/venkatadri053/config-server-ex

**Step#3** in Config Server Project,in pom.xml delete line

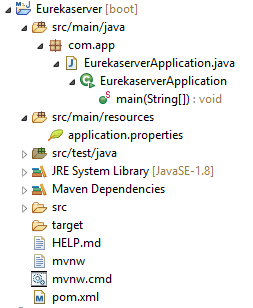
<include>myapp-config</include>

**Step#4** Create git account and create config-server-ex repository. Under this create file application.properties



**EurekaServer:**

**FolderStructure:**



**EurekaserverApplication.java:**

**package** com.app;

**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);

}

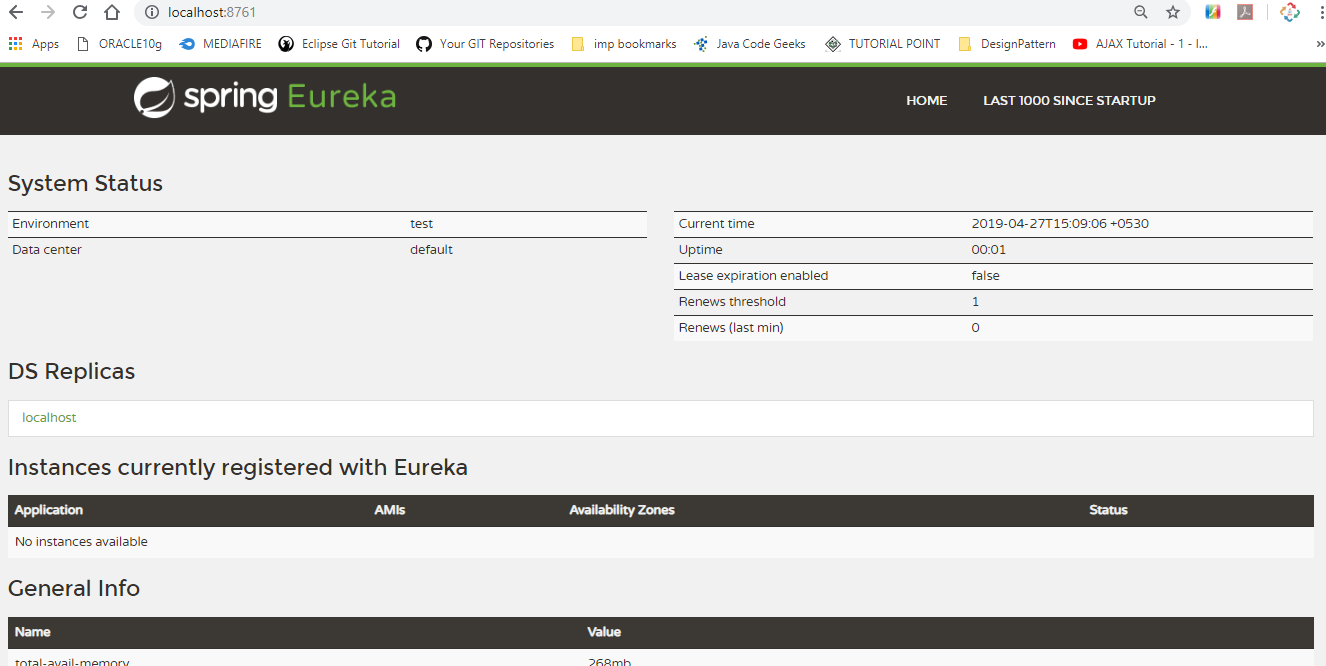
}

**application.properties:**

server.port=8761

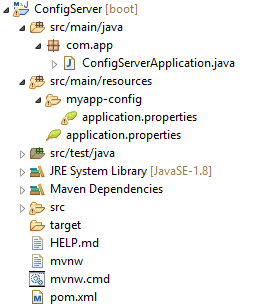
eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false



**ConfigServer:**

**FolderStructure:**



**ConfigServerApplication.java:**

**package** com.app;

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

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

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

@SpringBootApplication

@EnableConfigServer

**public** **class** ConfigServerApplication {

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

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

}

}

**application.properties:**

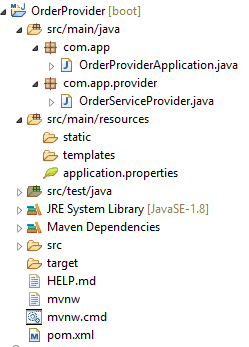
##link file##

server.port=8888

spring.cloud.config.server.git.uri=https://github.com/venkatadri053/config-server-ex

**OrderProvider:**

**FolderStructure:**



**OrderProviderApplication.java:**

**package** com.app;

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

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

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

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** OrderProviderApplication {

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

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

}

}

**OrderServiceProvider.java:**

**package** com.app.provider;

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

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

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

@RestController

@RequestMapping("/order")

**public** **class** OrderServiceProvider {

@GetMapping("/show")

**public** String showMsg() {

**return** "From Provider";

}

}

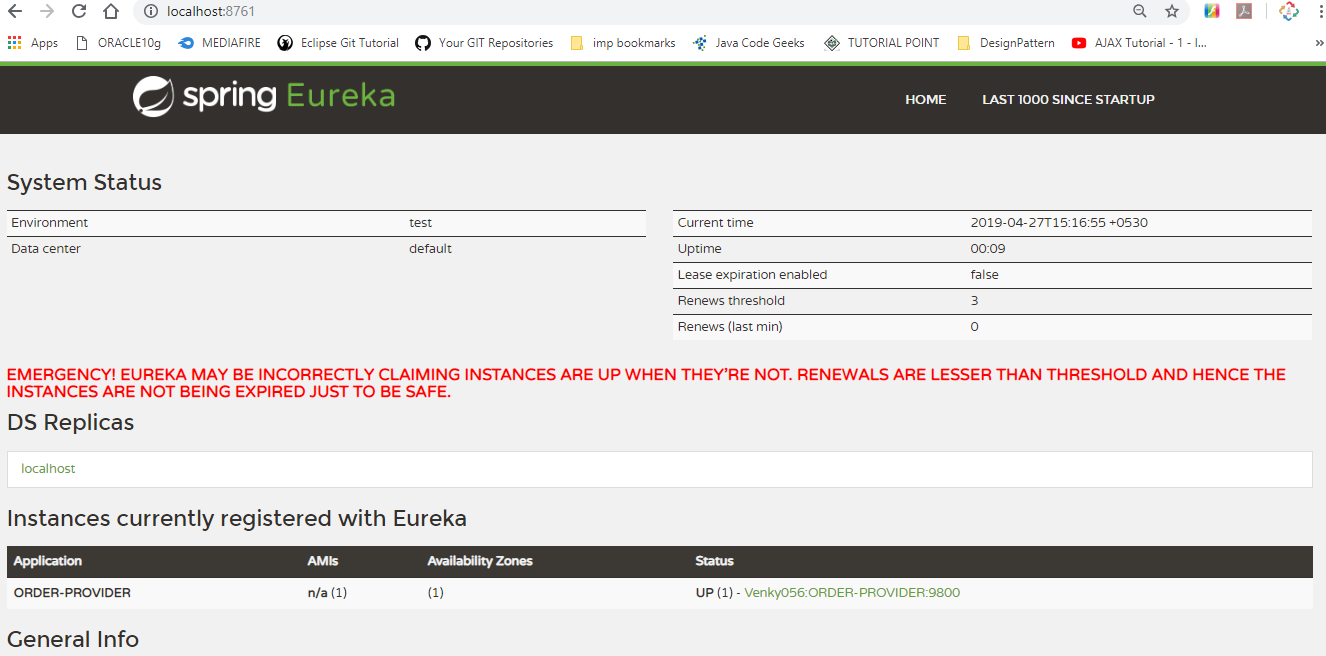
**application.properties:**

server.port=9800

spring.application.name=ORDER-PROVIDER

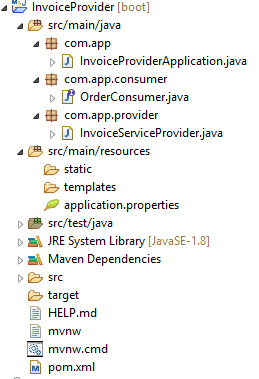
eureka.instance.instance-id=${spring.application.name}:${random.value}

**Output:**



**InvoiceProvider:**

**FolderStructure:**



**InvoiceProviderApplication.java:**

**package** com.app;

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

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

**import** org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableFeignClients

**public** **class** InvoiceProviderApplication {

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

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

}

}

**OrderConsumer.java:**

**package** com.app.consumer;

**import** org.springframework.cloud.openfeign.FeignClient;

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

@FeignClient(name="ORDER-PROVIDER")

**public** **interface** OrderConsumer {

@GetMapping("/order/show")

**public** String showMsg();

}

**InvoiceServiceProvider.java:**

**package** com.app.provider;

**import** org.springframework.beans.factory.annotation.Autowired;

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

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

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

**import** com.app.consumer.OrderConsumer;

@RestController

@RequestMapping("/invoce")

**public** **class** InvoiceServiceProvider {

@Autowired

**private** OrderConsumer consumer;

@GetMapping("/info")

**public** String getConsumerMsg() {

**return** "Consumer=>"+consumer.showMsg();

}

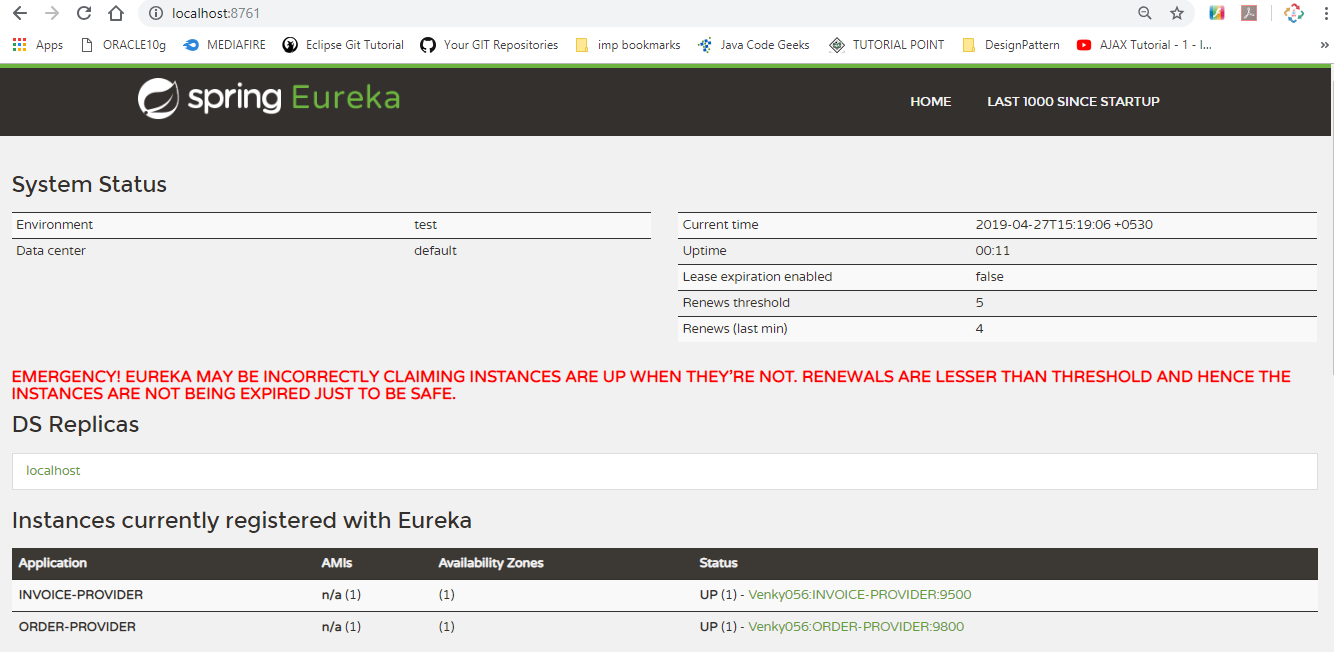
}

**application.properties:**

server.port=9500

spring.application.name=INVOICE-PROVIDER

**Output:**





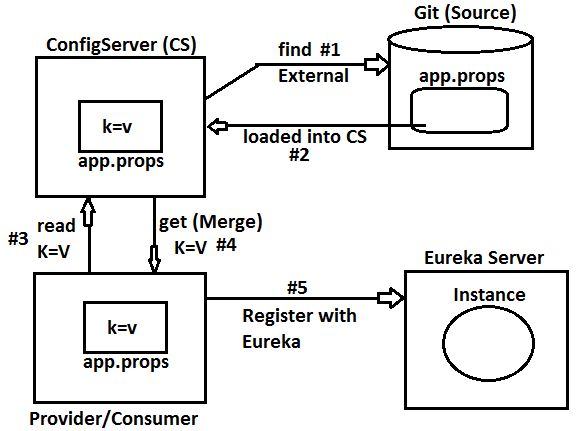
**Config Server With Provider/Consumer Execution flow[External Source]**

🡺On startup ConfigServer(CS) Application it will goto External Source (Git) and read \_\_\_\_.properties (or) \_\_\_\_.yml file having key=value pairs.

🡺Then Provider/Consumer Application (on startup) will try to read k=v from config server and merge with our application properties.

🡺If same key is found in both Config Server and Provider/Consumer App,then 1st priority is : **ConfigServer.**

**🡺**After **fetching** all required properties then Provider gets Registered with **EurekaServer**.



\*\*\*) What is bootstrap.properties in SpringBoot (Cloud) Programming?

Ans) Our Project (child Project) contains input keys in application.properties,in same way ParentProject also maintains one Properties file named as:bootstrap.properties which will be loaded before our application.properties.

**Execution Order:**

Parent Project-loads-bootstrap.properties / bootstrap.yml

Our Project-loads-application.properties / application.yml

🡺We can override this file in our project to provide key-values to be loaded by Parent Project.

\*\*) by default, Config Server runs on <http://localhost:8888> even config client also takes this as default location.

\*\*) To modify this IP/PORT use bootstrap.properties file in Our Project.

\*\*) In this bootstrap.properties file override key: **spring.cloud.config.uri** to any other location where Config server is running.

**DIAGRAM:**



**Coding changes for Config Server and provider/consumer App:**

**Step#1** Open application.properties file in ConfigServer project and modify port number.

**server.port=9999**

**Step#2** In Provider /Consumer Project, create file bootstrap.properties under src/main/resources

**Step#3** Add key=value in bootstrap.properties file

**spring.cloud.config.uri=http://localhost:9999**

**Q)** which class will load bootstrap.properties file for Config Server URI fetching?

**A)** ConfigServicePropertySourceLocator will read config server input by default from <http://localhost:8888>.

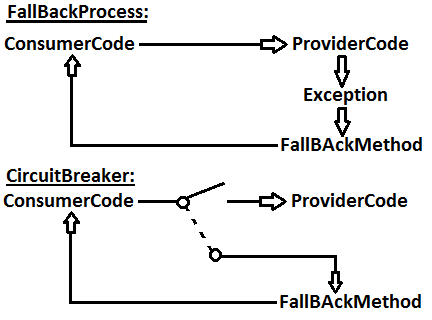
It is only first execution step in Provider/Consumer Project.

**Fault Tolerance API**

If any Microservice is continuously throwing Exceptions, then logic must not be executed every time also must be finished with smooth termination. such Process is called as “**Fault Tolerance”**.

🡺Fault Tolerance is achieved using **FallBackMethod** and **CircuitBreaker**.

1. FallBackMethod = If Microservice is throwing exception then service method execution is redirected to another supportive method (**FallBackMethod**) which gives dummy output and **“alerts to DashBoard”** (to Dev, MS, Admin teams).
2. CircuitBreaker = If Service is throwing exceptions continuously then Exception flow directly linked to fallback method. After some time gap (or) no. of request again re-checks once, still same continue to FallBackMethod else execute Microservice.



**Hystrix:** It is a API (set of classes and interfaces) given by Netflix to handle Proper Execution and Avoid repeated exception logic (Fault Tolerance API) in Microservice Programming.

🡺It is mainly used in production Environment not in Dev Environment.

🡺Hystrix supports FallBack and CircuitBreaker process.

🡺It provides Dashboard for UI. (View problems and other details in Services).

**=-=-=Working with Hystrix=-=-=**

**Step#1:--** In pom.xml add Netflix Hystrix dependency

<dependency>

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

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

</dependency>

**Step#2:--** At starter class level apply annotation @EnableCircuitBreaker (or) @EnableHystrix.

@EnableCIrcuitBreaker will find concept at runtime using pom.xml dependency

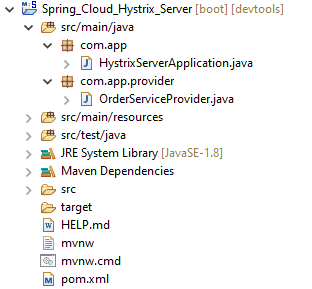
ex: Hystrix, Turbine etc…

where as @EnableHystrix will execute only Hystrix CIrcuitBreaker.

**Step#3:--** Define one Service method and apply Annotation : **@HystrixCommand** with Details like fallBackMethod, commandKey…

**Example:**

**Folder Structure :**

****

**Step#1:--** Create Spring Starter Project with Dependencies : web, eureka discovery, Hystrix.

**Hystrix Dependency:--**

<dependency>

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

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

</dependency>

GroupId : org.sathyatech

ArtifactId : Spring\_Cloud\_Hystrix\_Server

Version : 1.0

=>Finish

**Step#2:--** Apply below annotations at Starter class (both)

@EnableDiscoveryClient

@EnableHystrix

**HystrixServerApplication.java**

**package** com.app;

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

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

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

**import** org.springframework.cloud.netflix.hystrix.EnableHystrix;

@SpringBootApplication

@EnableDiscoveryClient

@EnableHystrix

**public** **class** HystrixServerApplication {

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

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

}

}

**Step#3:--** application.properties file:--

server.port=9800

spring.application.name=ORDER-PROVIDER

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

**Step#4:--** Define RestController with FallbackMethod (**OrderServiceProvider.java**).

**package** com.app.provider;

**import** java.util.Random;

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

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

**import** com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

@RestController

**public** **class** OrderServiceProvider {

@GetMapping("/show")

@HystrixCommand(fallbackMethod="showFallBack")

//parameter must be same as fallBackMethod name

**public** String showMsg() {

System.***out***.println("From service");

**if** (**new** Random().nextInt(10)<=10)

{

**throw** **new** RuntimeException("DUMMY");

}

**return** "Hello From Provider";

}

//fallBack method

**public** String showFallBack() {

System.***out***.println("From ballback");

**return** "From FallBack method";

}

}

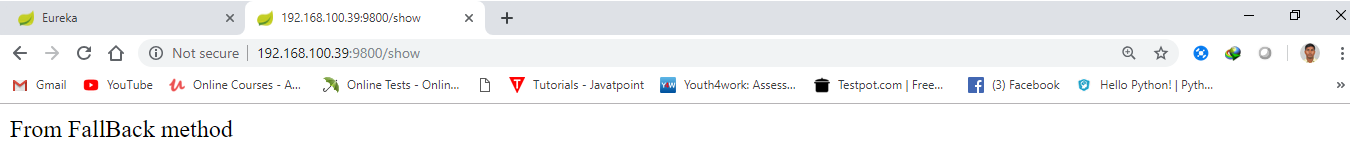
**NOTE: Fallback method ReturnType must be same as service method return type.**

**Execution Process:--**

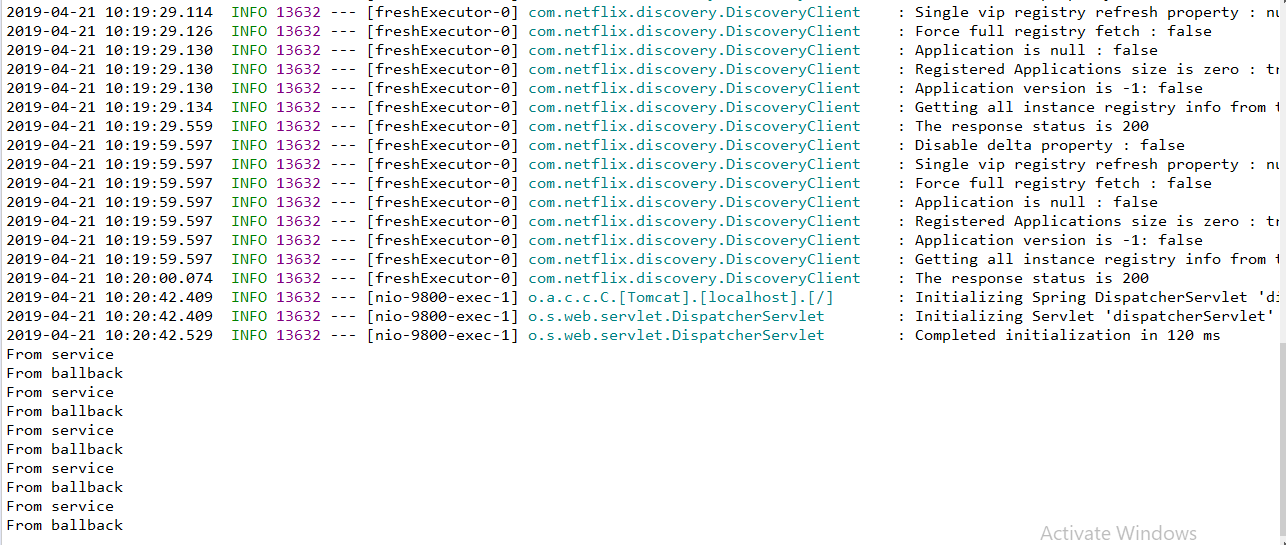
**Step#5:--** Run Eureka Server and Order Provider.

**Step#6:--** Goto Eureka and Execute **ORDER PROVIDER** and enter URL path **:/show**

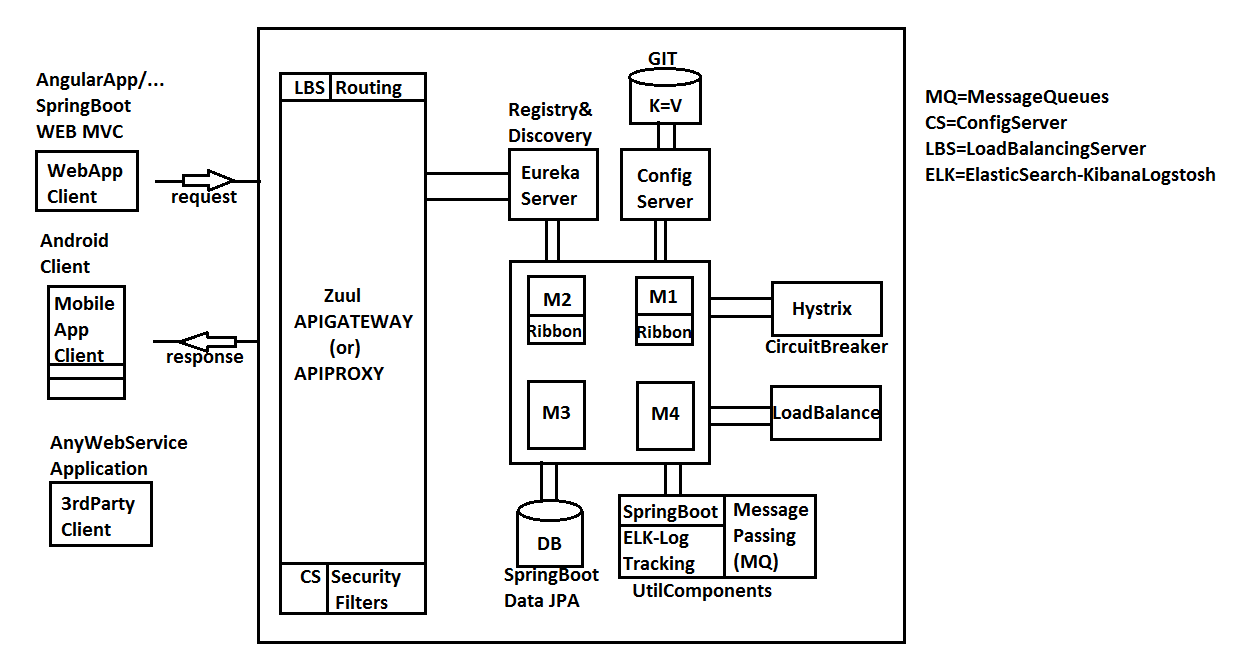
(<http://192.168.100.39:9800/show>)

****

**OUTPUT SCREEN OF “HYSTRIX-SERVICE-APP”:--**

****

**SpringCloud Netflix MicroService Design**



**API PROXY /API GATEWAY**

­­🡺In one application there will be Multiple MicroServices running in different servers and ports.

🡺Accessing these by client will be complicated.so, all are accessed through one entrypoint which makes

🡺Single Entry and Exit

🡺One Time Authentication(SSO=SingleSignOn)

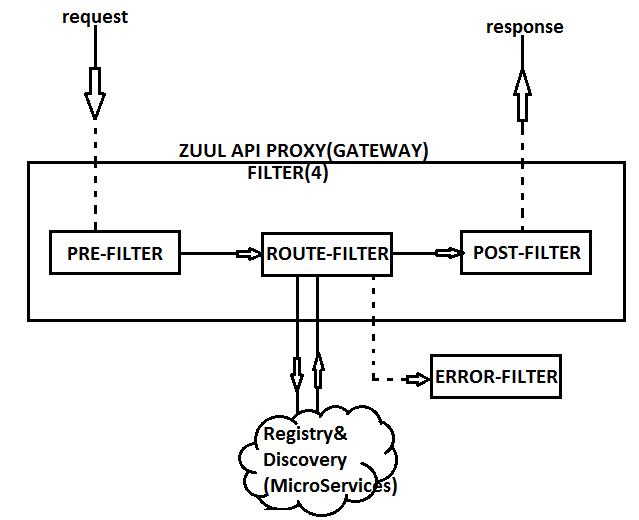
🡺Executing of Routing(Find Execution Path b/n multiple microservices)

🡺Avoid Direct URL to Client (Avoid CROS origin req/res format)

🡺Supports Filtering

\*­­Spring Cloud Netflix ZUUL behaves as API PROXY for Microservices Application which supports “Integration with any type of client component”(web,mobile,3rd party webservices..etc)

**ZUUL (APIPROXY) Working flow:**



**ZUUL API (PROXY-SERVER) GATEWAY:**

Zuul Server is a netflix Component used to Configure **“Routing for MicroServices”**

Using keys like:

zuul.routes.<modules>.path= . . . . . . . .

zuul.routes.<modules>.serviceId= . . . . . . . .

**\*#\***Before Creating ZuulServer,we should have already created:

a) Eurekaserver project

b) MicroServices (Ex:PRODUCT-SERVICE,CUSTOMER-SERVICE,STUDENT-SERVICE…) (with load balance implemented)

**Step#1** Create Spring Starter Project as: **ZUUL-SERVER**

with dependencies: web,zuul,eureka discovery.

**Step#2** application.properties should have below details like:

server.port=8558

eureka.client.service-url.default-zone=http://localhost:8761/eureka

spring.application.name=ZUUL-PROXY

zuul.routes.<module>.path=/<module>-api/\*\*

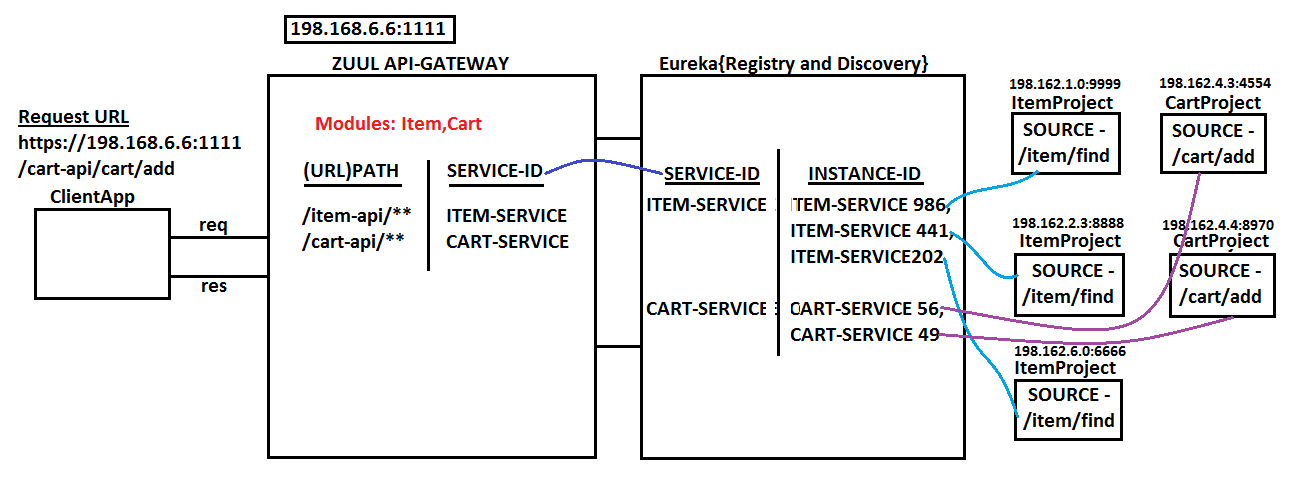
zuul.routes.<module>.service-id=[SERVICE-ID]

**Step#3** In ZuulServer project, at starter class level add annotation:**@EnableZuulProxy**

**Step#4** Implement Filters like **: PRE,ROUTE,POST,ERROR**

Using one abstractclass : **ZuulFilter(AC)** and use **FilterConstants(C)**.

**ZUUL ExampleService:**

****

**🡺**Here, ZuulServer behaves as Entry and Exit Point.

🡺It hides all details like Eureka Server and Service-Instances from Client.

🡺Zuul Provides and only ZUUL-URL and Paths of Services only.

🡺Zuul takes care of Client (Request) Loadbalancing even.

🡺Provides Routing based on API (PATH/URL)

🡺Zuul must be registered with EurekaServer.

🡺In Zuul Server Project,we should provide module details like Path, Service-Id using

application.properties (or) .yml

🡺Example application.properties

zuul.routes.item.path=/item-api/\*\*

zuul.routes.item.service-id=ITEM-SERVICE

zuul.routes.cart.path=/ cart -api/\*\*

zuul.routes. cart.service-id=CART-SERVICE

🡺If two modules are provided in Zuul then, only module name gets changed in keys. Consider below example:

**MODULE PATH SERVICE-ID**

Product /prod-api/\*\* PROD-SERVICE

Student /std-api/\*\* STD-SERVICE

**application.properties:**

zuul.routes.product.path=/prod-api/\*\*

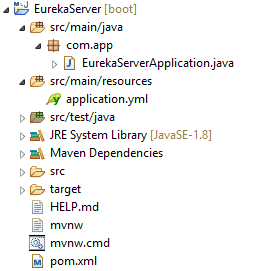
zuul.routes.product.service-id=PROD-SERVICE

zuul.routes.student.path=/ std-api/\*\*

zuul.routes.student.service-id=STD-SERVICE

**EUREKA SERVER:**

**FolderStructure:**



**EurekaServerApplication.java:**

**package** com.app;

**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);

}

}

**application.yml:**

server:

port: 8761

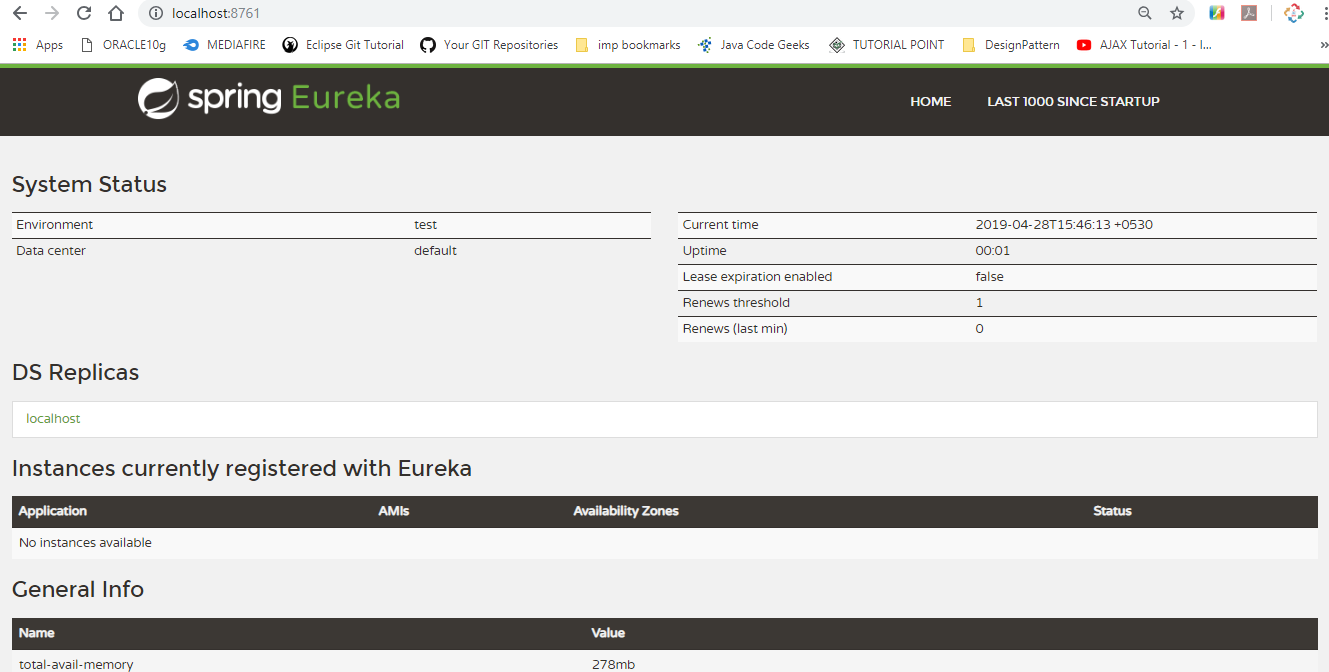
eureka:

client:

register-with-eureka: false

fetch-registry: false

**Output:**



**ZUULSERVER:**

**FolderStructure:**



**ZuulServerApplication.java:**

**package** com.app;

**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;

@SpringBootApplication

@EnableDiscoveryClient

@EnableZuulProxy

**public** **class** ZuulServerApplication {

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

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

}

}

**application.yml:**

server:

port: 9999

spring:

application:

name: ZUUL-PROXY

eureka:

client:

service-url:

default-zone: http://localhost:8761/eureka

zuul:

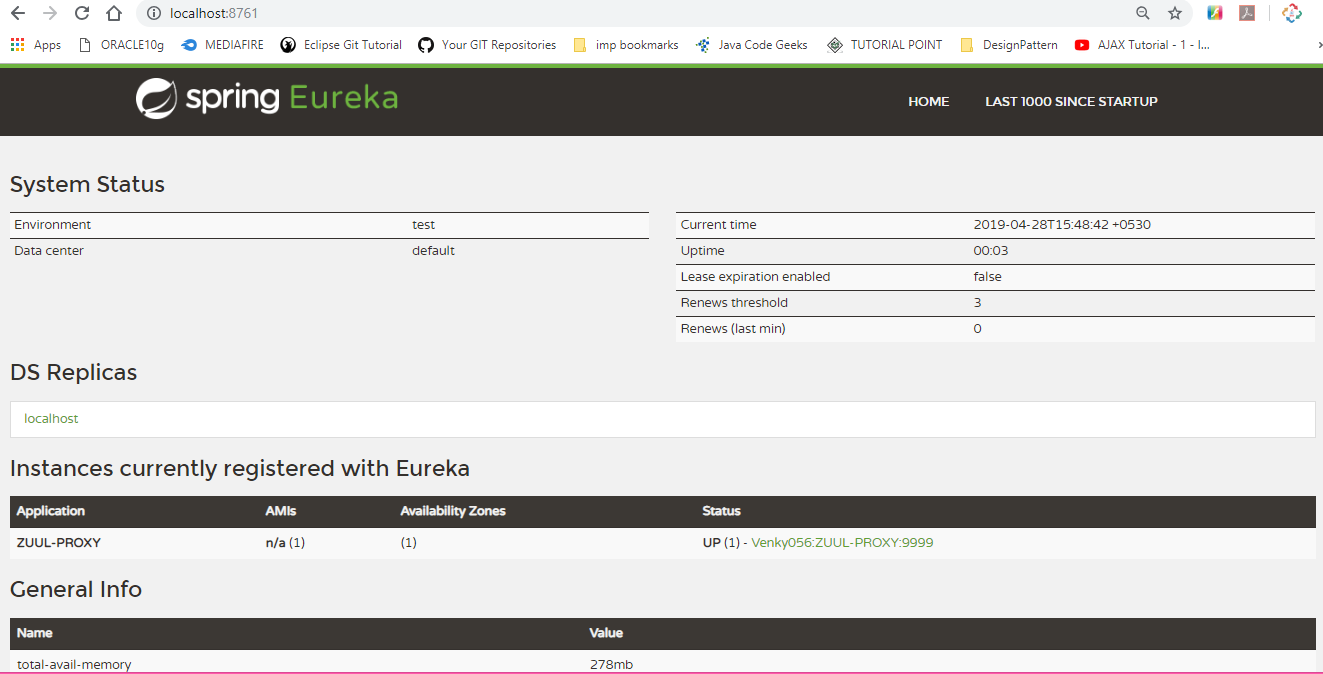
routes:

item:

path: /item-api/\*\*

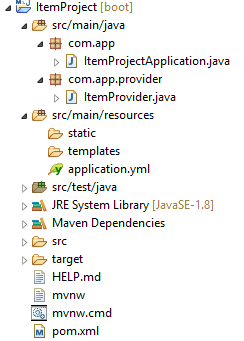
service-id: ITEM-SERVICE

**Output:**



**ITEMPROJECT:**

**FolderStructure:**



**ItemProjectApplication.java:**

**package** com.app;

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

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

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

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** ItemProjectApplication {

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

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

}

}

**ItemProvider.java:**

**package** com.app.provider;

**import** org.springframework.beans.factory.annotation.Value;

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

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

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

@RestController

@RequestMapping("/item")

**public** **class** ItemProvider {

@Value("${server.port}")

**private** String port;

@GetMapping("/find")

**public** String findItem() {

**return** "ITEM FOUND:"+port;

}

}

**application.yml:**

server:

port: 9966

spring:

application:

name: ITEM-SERVICE

eureka:

client:

service-url:

default-zone: http://localhost:8761/eureka

instance:

instance-id: ${spring.application.name}:${random.value}

**output:**



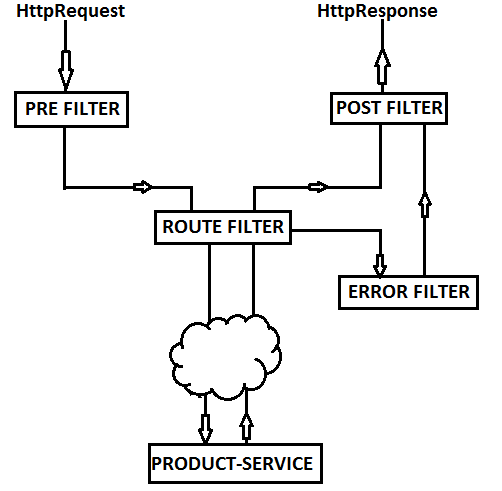


**Working with ZUUL Filter:**

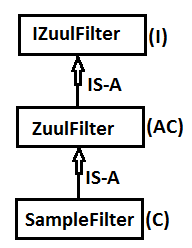
Filters are used to validate request construct valid response. In simple , we also call it as “PRE-POST” processing logic.

ZUUL Filter provides even extra types like ROUTE FILTERS and ERROR FILTERS.

* When client made request to ZUUL then Pre Filter gets called automatically.
* After validating, request is dispatched to Route Filter.
* Route Filter is like 2nd level validation at required SERVICE level.
* Route Filter will dispatch request to one Microservice based on Service-Id
* If microservice is not executed properly (i.e. throwing exception) then Error filter is called.
* Finally Post Filter works on Http Response (adds Headers, Encode data, Provide info to client..etc.) in case of either success or failure.



* Here, filter is a class must extends one abstract class **“ZuulFilter”** provided by Netflix API.
* We can define multiple filters in one application. Writing Filters are optional.
* While creating filter class we must provide Filter Order (0, 1, 2,3…) and Filter type (“pre”, “route”, “error”, “post”)
* Two filters of same type can have same order which indicates any execution order is valid.
* We can enable and disable filters using its flags (True/False).



* To indicate Filter types, we use its equal constants (public static final string variables), provides as

|  |  |
| --- | --- |
| * Type | * Constant |
| * pre | * PRE\_TYPE |
| * route | * ROUTE\_TYPE |
| * error | * ERROR\_TYPE |
| * post | * POST\_TYPE |

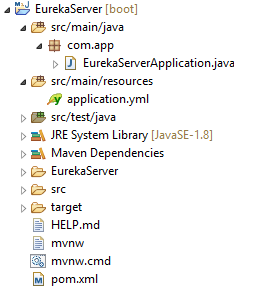
* All above constants are defined in FilterConstants (C) as global variables (public static final string)
* Write one class in ZUUL server and extend ZuulFilter Abstract class, override below 4 methods in your filter.
* shouldFilter() – Must be set to **‘true’**. If value is set to false then filter will not

be executed.

* run() – contains Filter logic. Executed once when filter is called.
* filterType() – Provides Filter Constant. Must be one Type(pre, post, route, error)
* filterOrder() – Provides order for Filter. Any int type number like: 0, 56, 98.

**EurekaServer:**

**FolderStructure:**



**EurekaServerApplication.java:**

**package** com.app;

**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);

}

}

**application.yml:**

server:

port: 8761

eureka:

client:

register-with-eureka: false

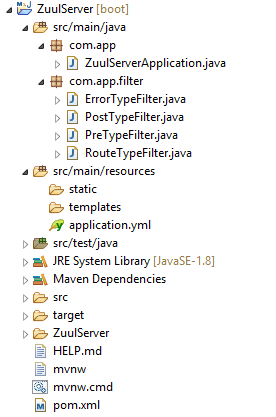
fetch-registry: false

**output:**



**ZUUL SERVER:**

**FolderStructure:**



**ZuulServerApplication.java:**

**package** com.app;

**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;

@SpringBootApplication

@EnableDiscoveryClient

@EnableZuulProxy

**public** **class** ZuulServerApplication {

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

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

}

}

**ErrorTypeFilter.java:**

**package** com.app.filter;

**import** org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;

**import** org.springframework.stereotype.Component;

**import** com.netflix.zuul.ZuulFilter;

**import** com.netflix.zuul.exception.ZuulException;

@Component

**public** **class** ErrorTypeFilter **extends** ZuulFilter{

/\*\*Enable(true) or Disable Filter(false)\*\*/

**public** **boolean** shouldFilter() {

**return** **true**;

}

/\*\*Define Filter Logic Here\*\*/

**public** Object run() **throws** ZuulException {

System.***out***.println("FROM ERROR FILTER");

**return** **null**;

}

/\*\*Specify Filter Type\*\*/

**public** String filterType() {

**return** FilterConstants.***ERROR\_TYPE***;

}

/\*\*Provider Filter Order for Execution\*\*/

**public** **int** filterOrder() {

**return** 0;

}

}

**PostTypeFilter.java:**

**package** com.app.filter;

**import** org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;

**import** org.springframework.stereotype.Component;

**import** com.netflix.zuul.ZuulFilter;

**import** com.netflix.zuul.exception.ZuulException;

@Component

**public** **class** PostTypeFilter **extends** ZuulFilter{

/\*\*Enable(true) or Disable Filter(false)\*\*/

**public** **boolean** shouldFilter() {

**return** **true**;

}

/\*\*Define Filter Logic Here\*\*/

**public** Object run() **throws** ZuulException {

System.***out***.println("FROM POST FILTER");

**return** **null**;

}

/\*\*Specify Filter Type\*\*/

**public** String filterType() {

**return** FilterConstants.***POST\_TYPE***;

}

/\*\*Provider Filter Order for Execution\*\*/

**public** **int** filterOrder() {

**return** 0;

}

}

**PreTypeFilter.java:**

**package** com.app.filter;

**import** org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;

**import** org.springframework.stereotype.Component;

**import** com.netflix.zuul.ZuulFilter;

**import** com.netflix.zuul.exception.ZuulException;

@Component

**public** **class** PreTypeFilter **extends** ZuulFilter{

/\*\*Enable(true) or Disable Filter(false)\*\*/

**public** **boolean** shouldFilter() {

**return** **true**;

}

/\*\*Define Filter Logic Here\*\*/

**public** Object run() **throws** ZuulException {

System.***out***.println("FROM PRE FILTER");

**return** **null**;

}

/\*\*Specify Filter Type\*\*/

**public** String filterType() {

**return** FilterConstants.***PRE\_TYPE***;

}

/\*\*Provider Filter Order for Execution\*\*/

**public** **int** filterOrder() {

**return** 0;

}

}

**RouteTypeFilter.java:**

**package** com.app.filter;

**import** org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;

**import** org.springframework.stereotype.Component;

**import** com.netflix.zuul.ZuulFilter;

**import** com.netflix.zuul.exception.ZuulException;

@Component

**public** **class** RouteTypeFilter **extends** ZuulFilter{

/\*\*Enable(true) or Disable Filter(false)\*\*/

**public** **boolean** shouldFilter() {

**return** **true**;

}

/\*\*Define Filter Logic Here\*\*/

**public** Object run() **throws** ZuulException {

System.***out***.println("FROM ROUTE FILTER");

**return** **null**;

}

/\*\*Specify Filter Type\*\*/

**public** String filterType() {

**return** FilterConstants.***ROUTE\_TYPE***;

}

/\*\*Provider Filter Order for Execution\*\*/

**public** **int** filterOrder() {

**return** 0;

}

}

**application.yml:**

server:

port: 9999

spring:

application:

name: ZUUL-PROXY

eureka:

client:

service-url:

default-zone: http://localhost:8761/eureka

zuul:

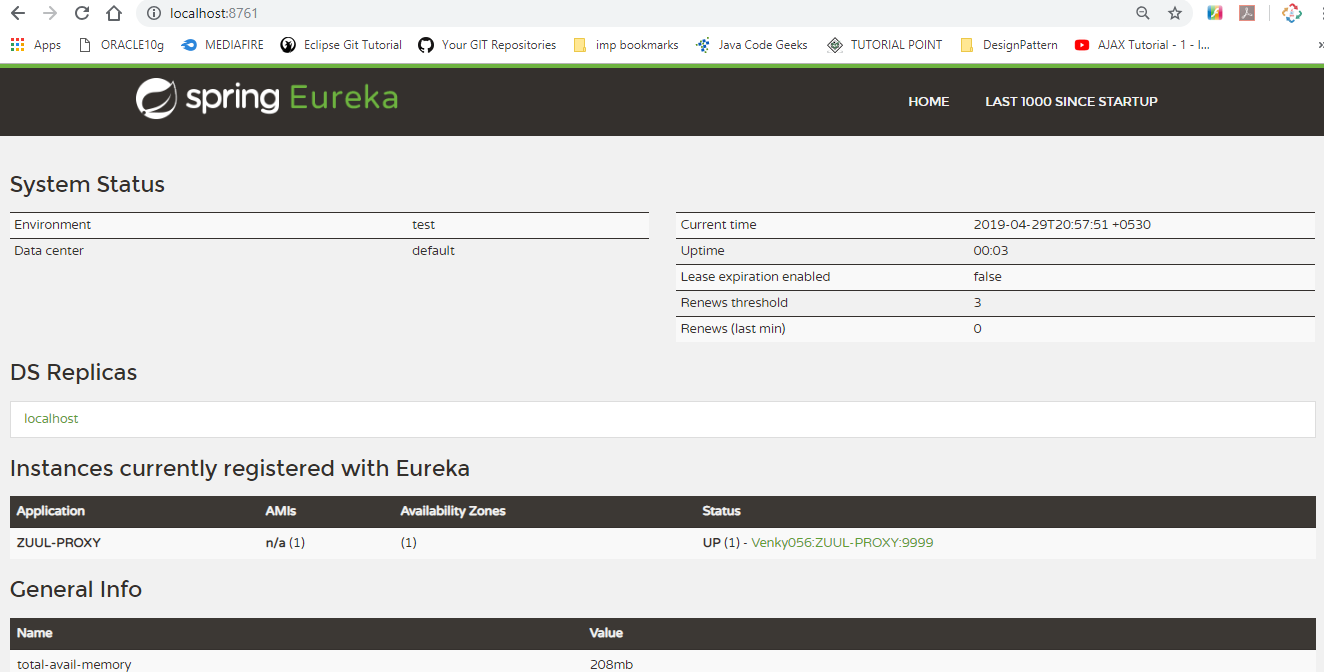
routes:

item:

path: /item-api/\*\*

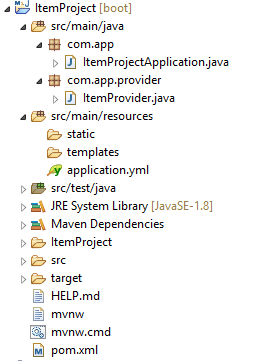
service-id: ITEM-SERVICE

**Output:**



**ItemProject:**

**FolderStructure:**



**ItemProjectApplication.java:**

**package** com.app;

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

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

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

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** ItemProjectApplication {

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

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

}

}

**ItemProvider.java:**

**package** com.app.provider;

**import** org.springframework.beans.factory.annotation.Value;

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

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

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

@RestController

@RequestMapping("/item")

**public** **class** ItemProvider {

@Value("${server.port}")

**private** String port;

@GetMapping("/find")

**public** String findItem() {

**return** "ITEM FOUND:"+port;

}

}

**application.yml:**

server:

port: 9900

spring:

application:

name: ITEM-SERVICE

eureka:

client:

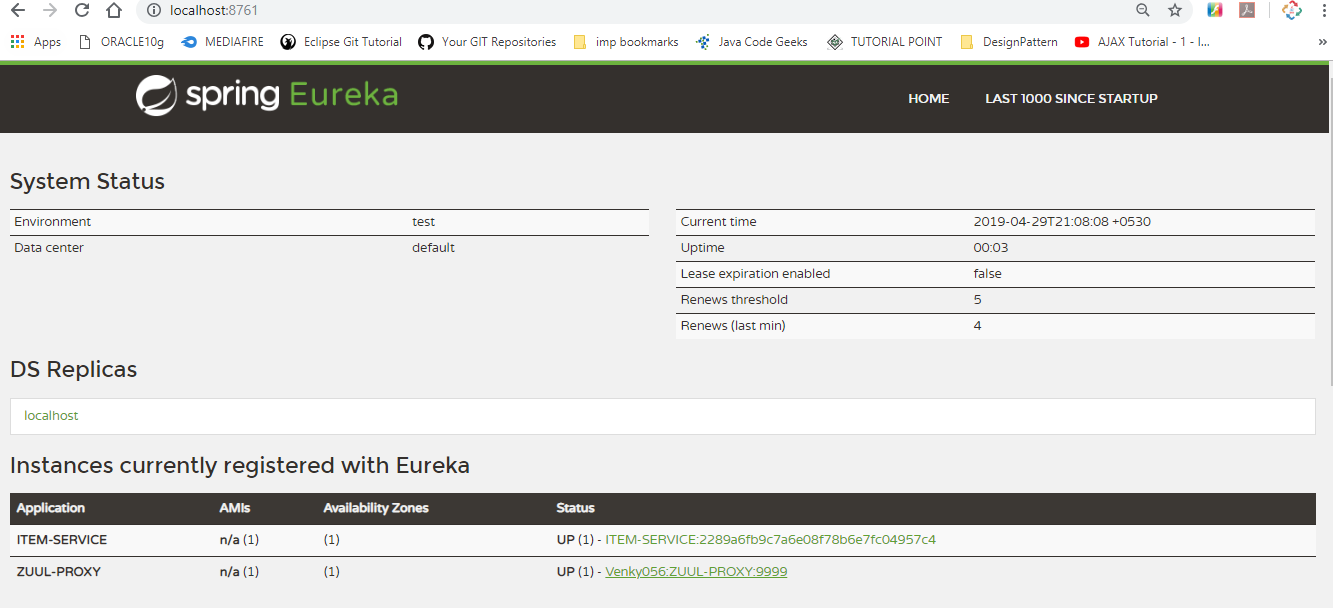
service-url:

default-zone: http://localhost:8761/eureka

instance:

instance-id: ${spring.application.name}:${random.value}

**output:**



**BrowserOutput:**



**ConsoleOutput:**



**PROJECT LOMBOK**

This is open source JAVA API is used to avoid writing (or generating) common code for Bean/Model/Entity classes.

That is like:

1. Setters and Getters
2. toString() method
3. Default and Parameterized constructor
4. hashCode() and equals() methods.

* Programmer can write these methods manually or generate using IDE. But if any modification(s) are done in those classes then again generate set/get methods also delete and write code for new : toString, hashCode, equals and Param const ( it is like repeated task)
* By using Lombok API which reduces writing code or generating task for Beans. Just apply annotations, it is done.
* To use lombok, while creating Spring Boot Project choose dependency: Lombok (or) Add below dependency in pom.xml

(For spring boot project: do not provide version. it is provided by spring boot parent only.)

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<scope>provided</scope>

</dependency>

For non-spring boot projects

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.6</version>

</dependency>

**Installation of Lombok in IDE:**

1) Open STS/Eclipse (any workspace)

2) Create spring boot project with Lombok Dependency (or) maven project with above Lombok Dependency.

3) Update Maven Project

4) Close STS.

5) Go to lombok jar location

For e.g.: **C:\Users\<username>\.m2\repository\org\projectlombok\lombok\1.18.6**

6) open command Prompt here

🡺 Shift + Mouse Right click

🡺Choose “Open Command Window Here”

🡺Type cmd given below

**Java -jar lombok-1.18.6.jar**

🡺Wait for few minutes (IDEs are detected)

🡺 Click on Install/Update

🡺 Finish

7) Open STS/Eclipse and start coding

**=-=-=-=-Example application=-=-=-=-=**

***#1*** create spring Boot starter project

* File >🡺new 🡺 spring starter project 🡺 enter details:

GroupId : com.app

ArtifactId : SpringBootLombok

Version: 1.0

* Choose dependency : lombok (only)

#2 Create Model class with below annotations

**@Getter //Generates get methods**

**@Setter //Generates set methods**

**@ToString //override toString method**

**@NoArgsConstructor //generate default constructor**

**@RequiredArgsConstructor //override hashcode , equals method**

\*\*) To use @RequiredArgsConstructor which generates constructor using variable annnoated with @NonNull. If no variable found having @NonNull, then it is equal to generating “Default constructor” only

**Note:**

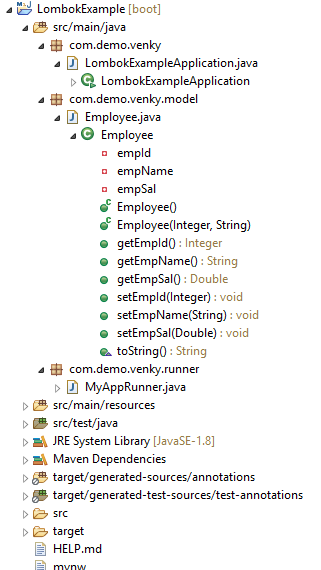
**\*#\*:** Apply @Data (package : lombok.Data) over Bean/Model which generates Set, get, toString, equals, hashCode and RequiredArgs Constructor.

E.g.:

@Data

public class Employee {………….}

**FolderStructure:**



**LombokExampleApplication.java:**

**package** com.demo.venky;

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

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

@SpringBootApplication

**public** **class** LombokExampleApplication {

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

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

}

}

**Employee.java:**

**package** com.demo.venky.model;

**import** lombok.Getter;

**import** lombok.NoArgsConstructor;

**import** lombok.NonNull;

**import** lombok.RequiredArgsConstructor;

**import** lombok.Setter;

**import** lombok.ToString;

//@Data //Generates all (get,set,constructors tostring )methods

@Getter //Generates get methods

@Setter //Generates set methods

@ToString //override toString method

@NoArgsConstructor //generate default constructor

@RequiredArgsConstructor //override hashcode , equals method

**public** **class** Employee {

@NonNull

**private** Integer empId;

@NonNull

**private** String empName;

**private** Double empSal;

}

**MyAppRunner.java:**

**package** com.demo.venky.runner;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.stereotype.Component;

**import** com.demo.venky.model.Employee;

@Component

**public** **class** MyAppRunner **implements** CommandLineRunner {

**public** **void** run(String... args) **throws** Exception {

Employee e1=**new** Employee();

e1.setEmpId(10);

e1.setEmpName("AA");

e1.setEmpSal(6.66);

Employee e2=**new** Employee();

e2.setEmpId(20);

e2.setEmpName("BB");

e2.setEmpSal(7.77);

Employee e3=**new** Employee();

e3.setEmpId(30);

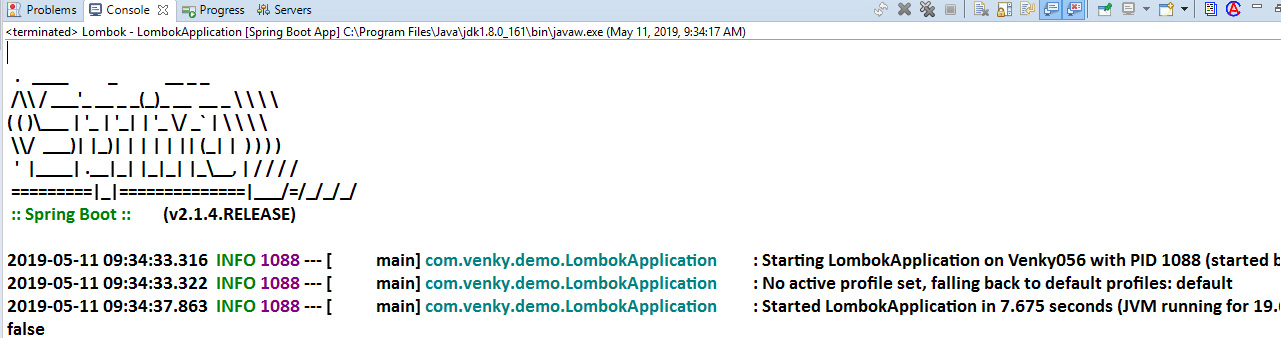
e3.setEmpName("CC");

e3.setEmpSal(9.99);

}

}

**Output:**



**Spring Boot Message Queues (MQ)**

🡺In case of real time application large data needs to be transferred and processed.

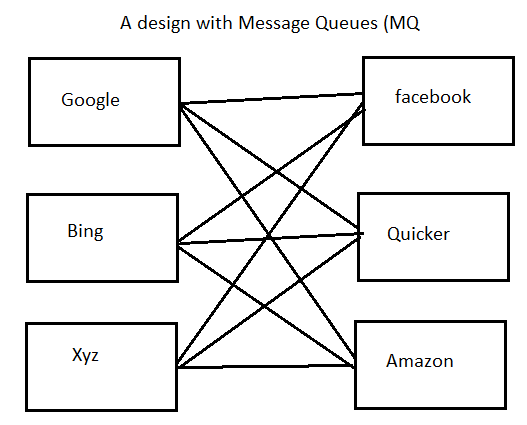
Like google server to amazon, facebook, … etc.

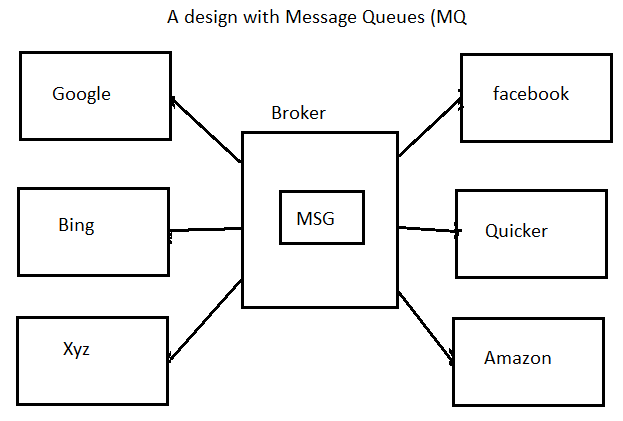
🡺Data (Message) transfer can be done using Message Queues.

🡺Message Queues are used for

1. Log aggregation (Read Large Log files from server and sent to another place)
2. Web Activities (Know what users are searching and provide related links and Advertisement in client websites)
3. Command Instruction (send message to Printer/Email one invoice on receive)
4. Data Streaming (Read large data from files, Networks, Databases continuously) etc..

🡺In case of multiple clients sharing data without MQ, may look like this:





🡺MQ can be implemented using Language based technologies (or API)

Ex: JMS (Java Message Service)

🡺Global MQ (between any type of client) Advanced Message Queuing protocol (AMQP)

🡺Apache ActiveMQ, Rabbit MQ, Apache Atrims are different Service providers (Broker software’s) for JMS

🡺Apache Kafka is a service Provider (Broker software) for AMQP.

**Spring Boot with Apache ActiveMQ**

JAVA JMS is simplified using Spring Boot which reduces writing basic configuration for ConnectionFactory, Connection, Session, Destination Creation, Send/Receive message etc....

JMS supports 2 types of clients. Those are

1. Producer (Client): Sends message
2. Consumer (Client): Read Message

Messages are exchanged using **MessageBroker** called as **MOM** (Message Oriented Middleware)

**Types of Communications in MQ’s:**

1. P2P (Peer-To-Peer): sending 1 message to one consumer.
2. Pub/Sub (Publish and Subscribe multiple consumers

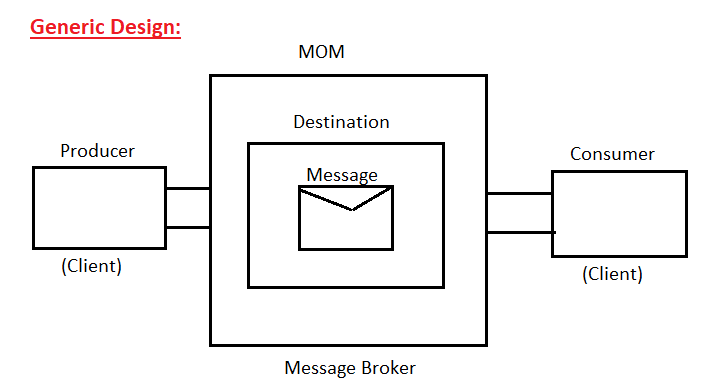
\*\*\* JMS supports two types of communications

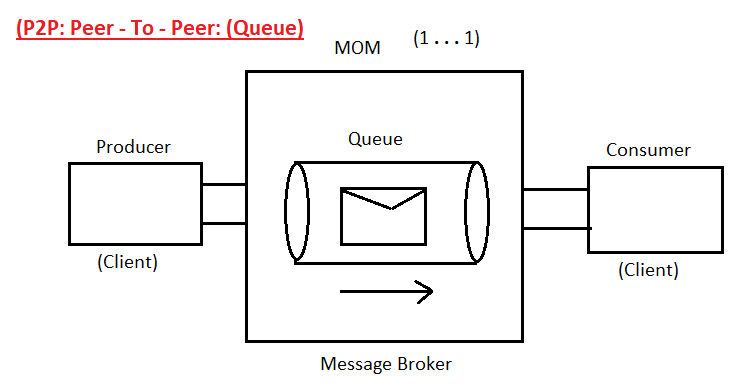
**Note:**

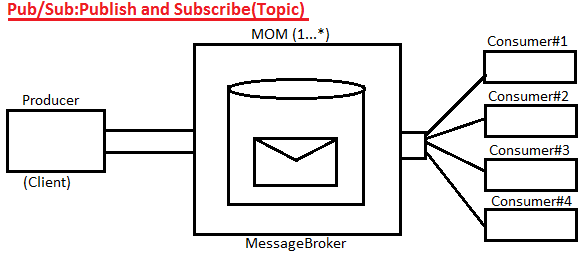
1. Destination is Special memory created in MOM which holds messages.
2. Here Queue Destination is used for P2P.

**Limitations of JMS:**

1. Used between Java Applications.
2. Message (Data) size should be smaller
3. Only one MOM (one instance) runs at a time.
4. Large data takes lot of time to process.
5. If Pub/Sub model is implemented with more consumers then process will be very slow
6. Day may not be delivered (data lose) in case of MOM stops Restart.







**Steps to Implements ActiveMQ:**

**Step#1** Create one Spring boot starter application using dependencies:

(or add below dependency in pom.xml)

<dependency>

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

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

</dependency>

**Step#2** In application.properties file provide common keys like MQ brokerUrl, user, password.

🡺If we do not specify any type then it behaves as **P2P (Queue).** To make it as Pub/Sub (Topic).

**application.properties:**

spring.activemq.broker-url=tcp://localhost:61616

spring.activemq.user=admin

spring.activemq.password=admin

spring.jms.pub-sub-domain=false

**Step#3** If application is Producer type then use JmsTemplate object and call send() which will send message to MOM.

**Step#4** If Application is Consumer type then define one Listener class using destination.

**Use Code:** @JmsListener(destination=” \_\_\_\_\_\_\_“)

🡺It must be Enabled using code: **@EnableJms**

🡺In case of JmsTemplate (C) @EnableJms is not required.

**Download and setup for ActiveMQ:**

**Download Link:**

🡺Go to below location

[**https://activemq.apache.org/components/classic /download/**](https://activemq.apache.org/components/classic%20/download/)

🡺Click on: apache-activemq-5.15.9-bin.zip

🡺Extract to one folder

🡺Go to ..\apache-activemq-5.15.9\bin\win64

🡺Double click on activemq.bat file

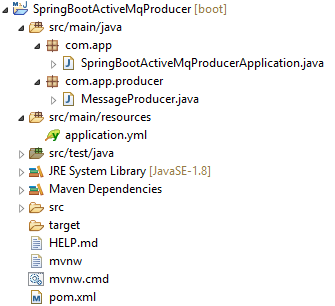
🡺Go to browser and type URL:[**http://localhost:8161/admin**](http://localhost:8161/admin)

🡺Create multiple consumer applications in same or different workspaces and set

**spring.jms.pub-sub-domain=true**

In application.properties file (or yml).

**Producer FolderStructure:**



**SpringBootActiveMqProducerApplication.java:**

**package** com.app;

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

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

@SpringBootApplication

**public** **class** SpringBootActiveMqProducerApplication {

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

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

}

}

**MessageProducer.java:**

**package** com.app.producer;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.jms.core.JmsTemplate;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** MessageProducer **implements** CommandLineRunner{

@Autowired

**private** JmsTemplate template;

@Override

**public** **void** run(String... args) **throws** Exception {

template.send("my-tpca", (ses)->ses.createTextMessage("AAAAAAAAA"));

System.***out***.println("sent from Producer");

}

}

**application.yml:**

spring:

activemq:

broker-url: tcp://localhost:61616

user: admin

password: admin

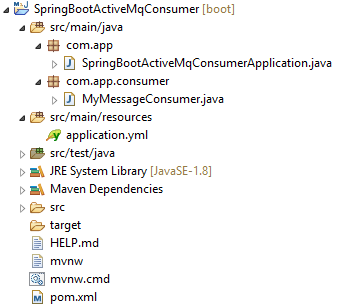
jms:

pub-sub-domain: true

**ConsoleOutput:**



**Consumer FolderStructure:**



**SpringBootActiveMqConsumerApplication.java:**

**package** com.app;

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

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

@SpringBootApplication

**public** **class** SpringBootActiveMqConsumerApplication {

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

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

}

}

**MyMessageConsumer.java:**

**package** com.app.consumer;

**import** org.springframework.jms.annotation.EnableJms;

**import** org.springframework.jms.annotation.JmsListener;

**import** org.springframework.stereotype.Component;

@EnableJms

@Component

**public** **class** MyMessageConsumer {

@JmsListener(destination = "my-tpca")

**public** **void** readMessage(String msg) {

System.***out***.println("from consumer");

System.***out***.println("msg is:"+msg);

}

}

**application.yml:**

spring:

activemq:

broker-url: tcp://localhost:61616

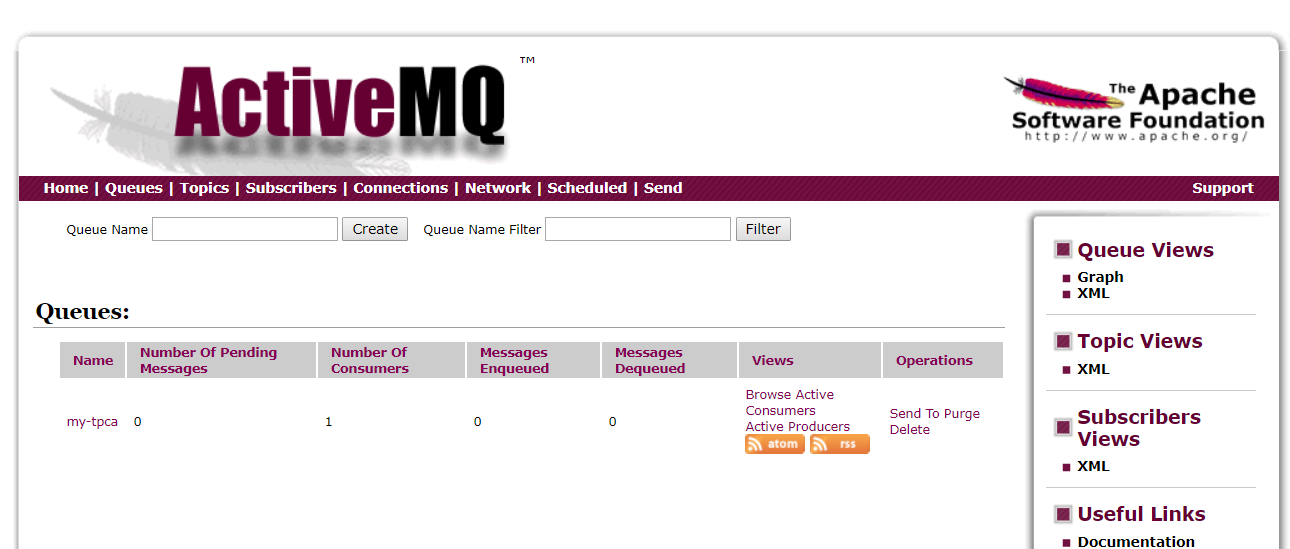
user: admin

password: admin

jms:

pub-sub-domain: true

**BrowserOutput:**



**Spring Boot Apache Kafka Integration**

Apache Kafka is used for

1. Distributed Messaging System, which works for multiple destinations of any type (any Language +Plugin required), to send or receive Messages.

B. Supports Realtime Data Streaming. It means read continuous and Large data from external source like Float Files, Database, networks, etc.…

C. Message Brokers will persist the message (save into their memory) to avoid data lose in case of consumer non-available or broker is down.

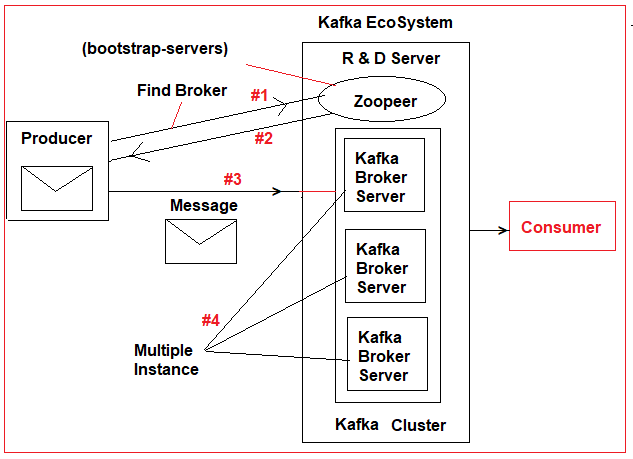
D. Kafka Default communication type is Pub/Sub (Topics).

E. Kafka Supports Load Balancing for Broker Softwire’s to make execution faster. It is known as Kafka Cluster.

F. All these Broker instances must be Registered with Registry and Discovery (R&D) Server. Kafka comes with default **“Zookeeper R&D Server”.**

G. This complete Kafka Software is called as Kafka **Ecosystem** (= Kafka Cluster + R&D Server)

H. Kafka works as Protocol independent i.e. works for TCP, FTP, SMTP, HTTP… etc.)



**Execution Flow:**

🡺Producer Application should get Message Broker details from R & D Server (zookeeper) known as bootstrap-server).

🡺Producer gets unique-id of Message Broker server and sends message to Broker.

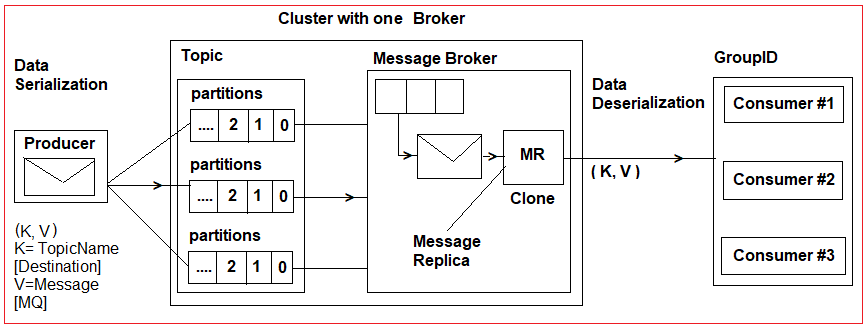
🡺MessageBroker will send this message to one or multiple consumers.

🡺Producer sends data <k, V> format in Serialized (Converting to binary/Characters formats). Here K=Destination (Topic name) and V= Message.

🡺Every Message will be partitioned into Multiple parts in Topic (Destination) to avoid large data sending, by making into small and equal parts (some time size may vary).

🡺Broker reads all partitions data and creates its replica (Clone/Mirror obj) to send message to multiple consumers based on **Topic** and **Group-Id**.

🡺At Consumer side Deserialization must be applied on K, V to read data. Consumer should also be linked with bootstrap-server to know its broker.



🡺Partitions are used to breakdown large message into multiple parts and send same to multiple brokers to make data destination in parallel.

🡺Message Replica: it creates multiple copies to one message to publish one message to multiple Consumers.

**Kafka Producer and Consumer Setup Details:**

🡺For Producer Application we should details in application.properties (or). yml

🡺Those are

**bootstrap-servers=localhost:9092**

**key-serializer=StringSerializer**

**value-serializer=StringSerializer**

🡺By using this Spring Boot creates instance of **“KafkaTemplate<K, V>”** then we can call send(k, v) method which will send data to Consumer.

=>Here: K=Topic Name, V= Data/Message

🡺For Consumer Application we should provide details in application.properties (or) .yml

🡺Those are

**bootstrap-servers=localhost:9092**

**key-deserializer=StringDeserializer**

**value-deserializer=StringDeserializer**

**group-id=MyGroupId**

🡺By using this Spring Boot configures the Consumer application, which must be implemented using: @KafkaListener(topics=”­­­\_\_\_\_\_\_\_“, groupId=”\_\_\_\_\_\_\_“)

-----------------------------------------------------------------------------------------------------------------

**\*\*\* bat files in kafka to be created\*\*\***

**1.Cluster.bat**

🡺Starts Zookeeper with Kafka Cluster design

.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

**2Server.bat**

🡺Starts Kafka Server (Message Broker)

.\bin\windows\kafka-server-start.bat .\config\server.properties

**Coding Steps:**

**Step#1** Create new Spring boot Starter project with dependencies: web, kafka

GroupId: com.app

ArtifactId: SpringBootKafkaApp

version: 1.0

**Kafka Dependency:**

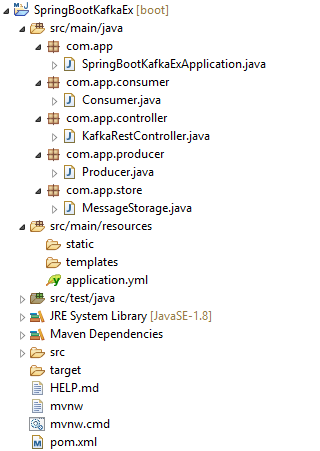
<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

**FolderStructure:**



**Step#2** add key= value pairs in application (. properties/. yml) file.

server:

port: 9988

spring:

kafka:

producer:

bootstrap-servers: localhost:9092

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: org.apache.kafka.common.serialization.StringSerializer

consumer:

bootstrap-servers: localhost:9092

key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

group-id: groupId

my:

app:

topicname: sampletopic

**Step#3 Define one starter class**

**package** com.app;

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

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

@SpringBootApplication

**public** **class** SpringBootKafkaExApplication {

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

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

}

}

**Step#4 Define Producer code**

**package** com.app.producer;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.kafka.core.KafkaTemplate;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** Producer {

@Value("${my.app.topicname}")

**private** String topic;

@Autowired

**private** KafkaTemplate<String, String> template;

**public** **void** sendMessage(String message) {

template.send(topic, message);

}

}

**Step#4 Define Message Storage class**

**package** com.app.store;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** MessageStorage {

**private** List<String> list=**new** ArrayList<String>();

**public** **void** put(String message) {

list.add(message);

}

**public** String getAll() {

**return** list.toString();

}

}

**Step#5: Define Consumer class**

**package** com.app.consumer;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.kafka.annotation.KafkaListener;

**import** org.springframework.stereotype.Component;

**import** com.app.store.MessageStorage;

@Component

**public** **class** Consumer {

@Autowired

**private** MessageStorage storage;

@KafkaListener(topics="${my.app.topicname}",groupId="groupId")

**public** **void** consume(String message) {

storage.put(message);

}

}

**Step#6: Define Kafka Controller class**

**package** com.app.controller;

**import** org.springframework.beans.factory.annotation.Autowired;

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

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

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

**import** com.app.producer.Producer;

**import** com.app.store.MessageStorage;

@RestController

**public** **class** KafkaRestController {

@Autowired

**private** Producer producer;

@Autowired

**private** MessageStorage storage;

@RequestMapping("/send")

**public** String readInMessage(@RequestParam String message) {

producer.sendMessage(message);

**return** "message sent!!";

}

@RequestMapping("/view")

**public** String viewOutMessage() {

**return** storage.getAll();

}

}

**\*\*\*Run Starter class and enter URLs:**

<http://localhost:9988/send?message=OK>

http://localhost:9988/view

**output:**

**Spring Boot with Apache Camel**

**Routing:** It is a process of sending large data from Application (Source) to another Application (Destination). Here data can be File System (.xml, .txt, .csv, .xlsx, .json, etc..), Database (Oracle DB, MySQL Db) or Message Queues using JMS (Active MQ) etc..

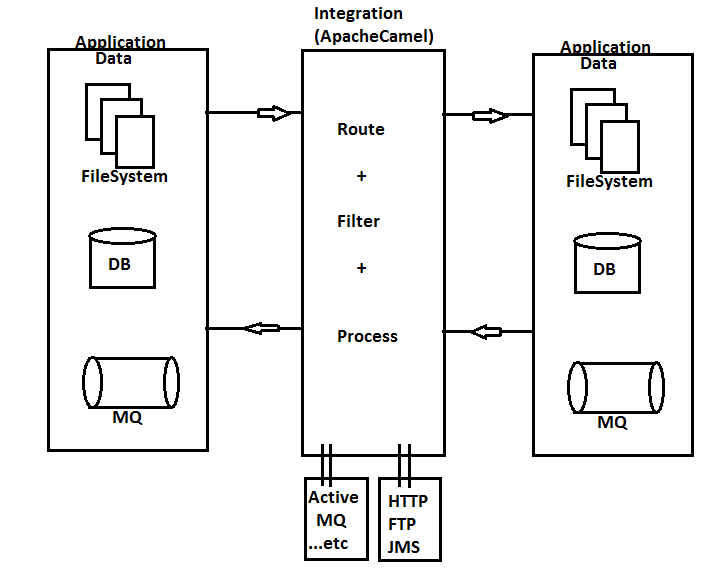
🡺Apache Camel is OpenSource and Light weight “Conditional based Routing Engine” which supports filtering and processing.

🡺Apache Camel also supports different language like PHP, Python, and JavaScript… etc.

🡺Compared to Spring Batch Integration tool Apache camel is a light weight tool.

🡺Camel supports reading data from different sources even like HTTP, FTP, JMS protocols based.

**Daigram:**



**Implementing Camel Routing in Spring boot:**

**Step#1** in pom.xml, we must add below dependency which supports Spring boot integration.

<dependency>

<groupId>org.apache.camel</groupId>

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

<version>2.23.2</version>

</dependency>

**Step#2:** Camel avoid main method (thread) execution control and run as independent while working with Spring boot, our starter class is main method. So, we should add key=value in properties file as

**application.properties:**

camel.springboot.main-run-controller=true

**Step#3:** Define one Route Builder class and configure details of routing, filtering and processing.

🡺To implement this, we need to write one class using (Abstract class) RoutingBuilder provided by Apache camel having one abstract method: configure() which contains coding format like:

from (SourceLocation)

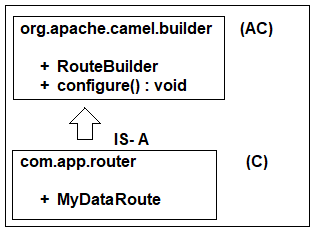
. [filter] . [process].

.to (DestinationLocation)

🡺Here Location can be URL/Local File System DB, JMS (MessageQueues)… etc.

**Coding Steps:**

**Step#1:** Create Spring Boot starter application with dependencies: Apache Camel



GroupId: org.sathyatech

ArtifactId: SpringBootApacheCamel

Version: 1.0

**Step#2:** open application.properties (.yml) and add main method-control key as true.

**application.properties:**

camel.springboot.main-run-controller=true

**(OR)**

**application.yml:**

camel:

springboot:

main-run-controller=true

**Step#3:** Define Router Builder class with file transfer logic.

package com.app.router;

@Component

public class MyFilesRouter extends RouteBuilder {

public void configure() throws Exception{

//with static location

from (“file:D:\\source”).to(“file:D:\\destination”);

}

}

**Step#4:** Create two folders: source and Destination in “D: drive”.

**Step#5:** Start Application and place files in “D:/source” which will be copied to “D:/destination”.

**EIP Patterns by Apache Camel:** EIP stand for “Enterprise Integration Patterns” used to define short form code for

* Data routing
* Data Filtering
* Data Processing

**#1 from (“file:source”) .to(“file:desti”);**

🡺It will copy files from source to destination by taking files in backup folder in source with name **.camel**.

🡺It supports even same file sending with new data (Operation to Override Program).

**#2 from (“file:source?noop=true”) .to(“file:destination”);**

🡺To avoid this, we should set as true.

**#3 from (“{{source}}”).to(“{{destination}}”)**

🡺Here {{location}} indicates dynamic location passing using Properties/Yml files, System args, command line inputs etc.

**Example EIPs:**

**#1Dynamic Location:**

Location (source/destination) can be passed at runtime using properties/yml files, config server, system arguments… etc.

🡺To indicate Location (URL file System, DB, MQ…) comes at runtime use format: {{location}}

**Code changes**

**a. applictaion.properties:**

camel.springboot.main-run-controller=true

my.source=file:D/source?noop=true

my.destination=file:/desti

**b. Router class code**

from(“{{my.source}}”).to(“{{my.destination}}”);

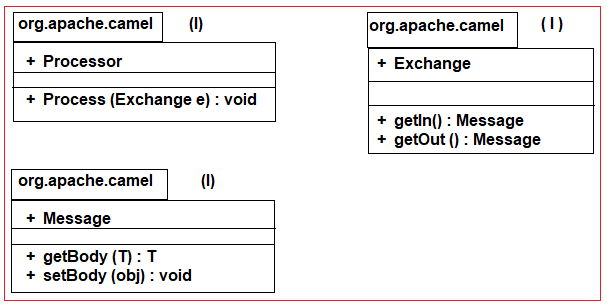
**#2 Processing Data:**

In Realtime data will be converted or modified based on requirements. Like XML->JSON, JSON->Simple Text, XML->CSV, CSV->SQL Format etc. i.e. data converted from one format to another format which can be done using 3 supporting interfaces. Those are:

1.Processor

2.Exchange

3.Message



**Code: Write inside Router (Impl) class:**

from (“file:D:/source”)

.process(new Processor() {

public void process (Exchange ex) throws Exception

{

//#1 Read Input Message

Message m = ex.getIn();

#2 Read Body from message

String body = m.getBody(String.class);

//#3 do processing

body =”modified ::”+body;

//#4 Writer data to out message

Message m2 = ex.getOut();

//#5 set body to out message

m2.setBody(body);

}

})

.to(“file:D:/destination?fileName=myFile.txt”);

**\*\*\*Note:** Here file name indicates new name for modified message. It can also be passed at run time.

**Using Lambda Expression## code:**

from(“file:D:/source”)

.process(ex 🡺{

String body=ex.getIn().getBody(String.class);

Body=”New AA modified ::”+body;

ex.getOut().setBody(body);

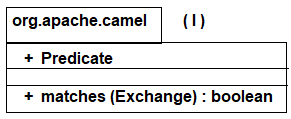
})

.to(“file:D:/destination?fileName=myFile.txt”);

**#3 Filters using Predicates:**

Predicate is a type expression which returns either true or false based on condition checking.

=>If true next level steps are executed else false execution stopped here only.



=>ValueBuilder (C) is used to execute required Predicates, using method: contains(), startWith(), isNull(), …etc.

=>We can get ValueBuilder (C) object using methods body(), header().

=>Header() indicates checking on file name, extension, location … etc.

=>Body() indicates file data/ content check

**Ex#1 File name having work sample**

from(“file:D:/source”)

.filter(

header(Exchange.FILE\_NAME).contains(“sample”)

.contains(“sample”)

)

.to(“file:D:/destination”);

**Ex#2 File body starts with word java**

from(“file:D:/source”)

.filter(body().startWith(“java”))

.to(“file:D:/destination”);

**Conditional based Routing [Choose –when-otherwise]**

=>Filter are used to check predicates and if true executes the next level process but it does execute for false case.

=>To Handle switch-case concept for data routing use choose-when-other.

**Format looks like:**

From(“source”)

. choice()

. when(condition#1).to(“destinatation#1”)

. when(“condition#2).to(“destinatation#2”)

….

otherwise().to(“destinatation#n”)

**Example:**

from(“file:D:/source”)

. choice()

.when(body().startWith(“java”))

.to(“file:D:/destinatation?fileName=a.txt”)

. when(body(). startWith(“xml”)).to(“file:D:/destinatation?fileName=b.txt”)

. when(body(). startWith(“xml”)).to(“file:D:/destinatation?fileName=b.txt”)

. otherwise().to(“file:D://destinatation?fileName=d.txt”);

**Apache Camel Intergration with Active MQ(JMS)**

🡺Patterns used to communicate with MQ using camel is: jms<type>:<destination>

🡺Here type can be queue or topic.

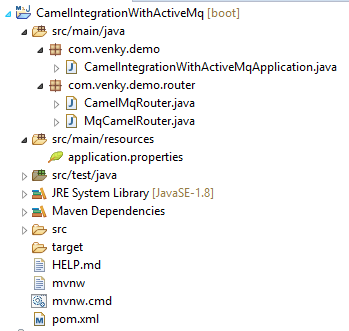
Example: jms:queue:info

jms:topic:news etc.

🡺For this coding along with ActiveMQ and Camel Dependencies we should add ActiveMQ-pool and camel-jms integration dependencies.

**Step#1** Create Spring boot starter project with dependencies: Apache Camel, ActiveMQ.

**FolderStructure:**



**Step#2** add below dependencies in pom.xml file

<dependency>

<groupId>org.apache.activemq</groupId>

<artifactId>activemq-pool</artifactId>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-jms</artifactId>

<version>2.23.2</version>

</dependency>

**Step#3** Provide camel and ActiveMQ properties

**---application.properties---**

camel.springboot.main-run-controller=true

spring.activemq.broker-url=tcp://localhost:61616

spring.activemq.user=admin

spring.activemq.password=admin

**Step#4** Define Routers

**---a)Camel to MQ Router---**

**package** com.venky.demo.router;

**import** org.apache.camel.builder.RouteBuilder;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** CamelMqRouter **extends** RouteBuilder{

@Override

**public** **void** configure() **throws** Exception {

from("file:D:/MICROSERVICES PRACTICE-RAGHU/CAMEL/source").to("jms:queue:outdata");

}

}

**---b) MQ to camel Router---**

**package** com.venky.demo.router;

**import** org.apache.camel.builder.RouteBuilder;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** MqCamelRouter **extends** RouteBuilder{

@Override

**public** **void** configure() **throws** Exception {

from("jms:queue:outdata").to("file:D:/MICROSERVICES PRACTICE-RAGHU/CAMEL/desti");

} }

**Step#5** Run starter class and start ActiveMQ using bat

**Step#6** Login to MQ (<http://localhost:8161/admin>)

🡺Click on menu Queue

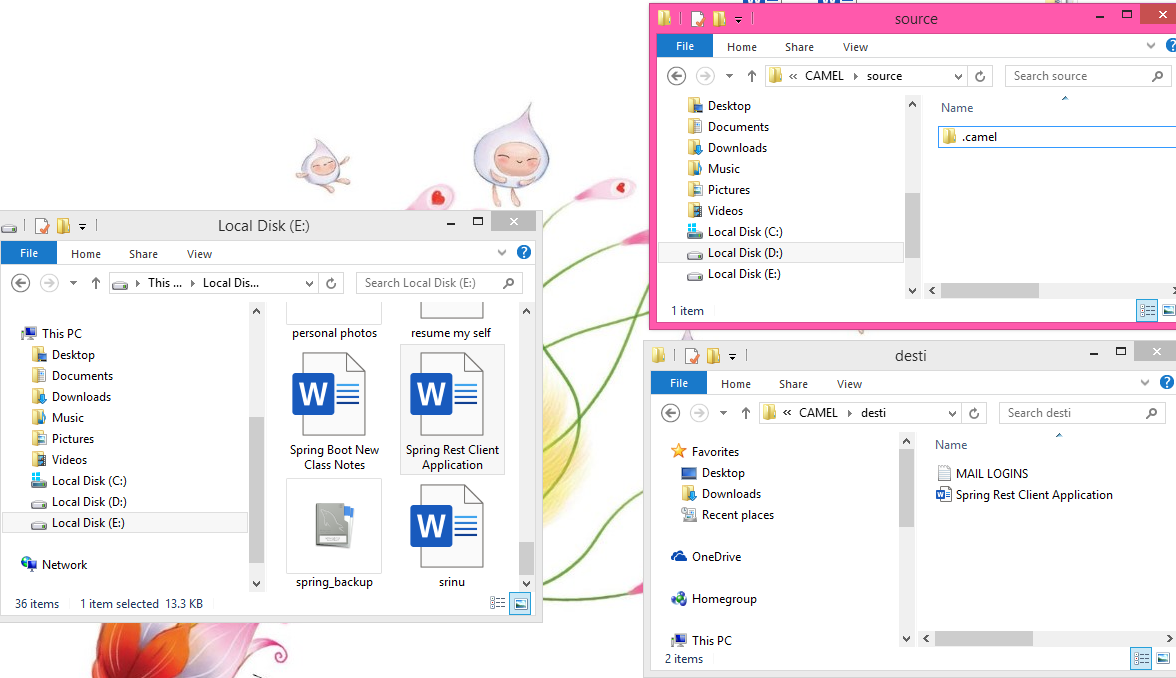
🡺Enter Queue name and click create [ 2 queues: outdata, indata]

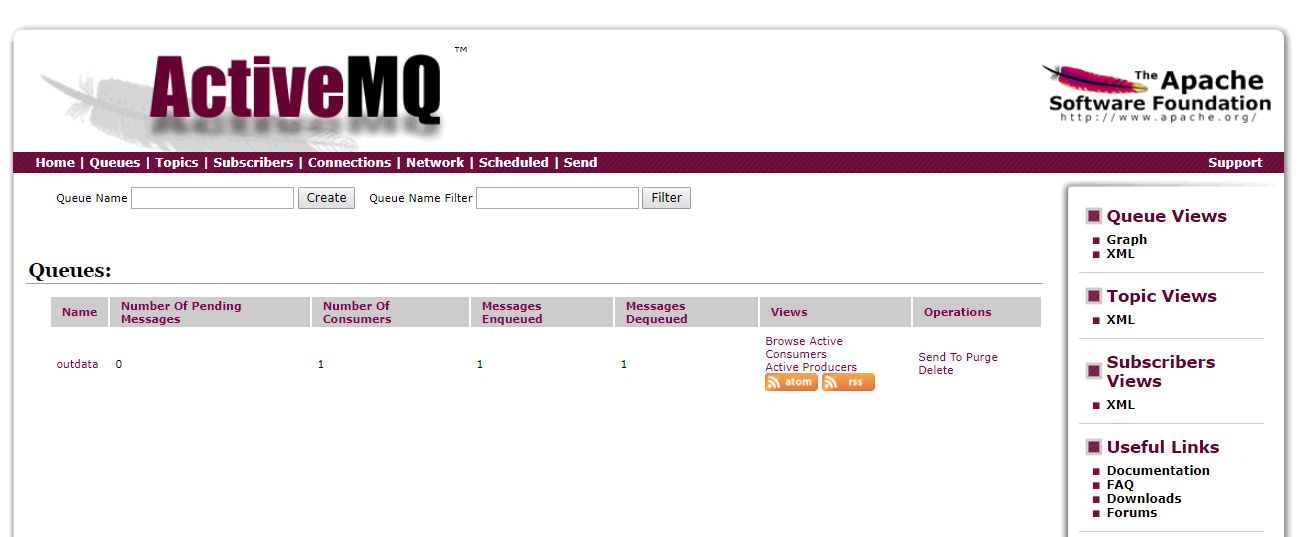
🡺click on “sendTo” option on “indata” queue Enter message and press send. File will be copied to destination folder.

🡺Go to D:/source folder and place text file having any message.MQ reads this from source.

=>Go to MQ =>click on queue name =>click messageId

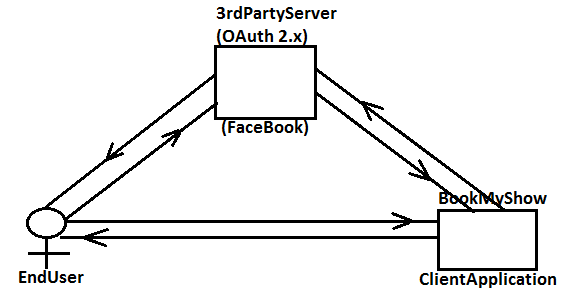
**Output:**





**Open Authorization (OAuth 2.x)**

OAuth 2.x is standard and framework which provides 3rd party security services to client application which are registered, on behalf of end user.



🡺These 3rd party Applications are also called as “Authorization and Resource Servers”.

🡺OAuth 2.x standard is widely used in small/medium scale/daily used, business application.

🡺OAuth2 Provide SSO (Single Sign on) for multiple applications acting as a one Service.

🡺Example Client Application are: bookmyshow, yetra.com, quora, redbus, makemytrip, Netflix, mediafire, avast.com, carwale.com, zomato.com.

🡺**\*\*\***OAuth2 may not be used for high level and Large scale applications (Ex:- Banking Creditcard, Stock Market… etc). These Spring Security ORM is used mostly.

🡺Few Authorization and Resource Servers are : Google, facebook, Redit, Github, Twitter, Linkedin… etc.

**Work flow of OAuth2:**

**Diagram:**

****

**OneTime setup for OAuth2:**

**Step #1** Choose any one (or more) 3rd party “Authorization and Resource Server”.

Ex:-- facebook, Google cloud platform (GCP) Github, Linkedin, Twitter… etc.

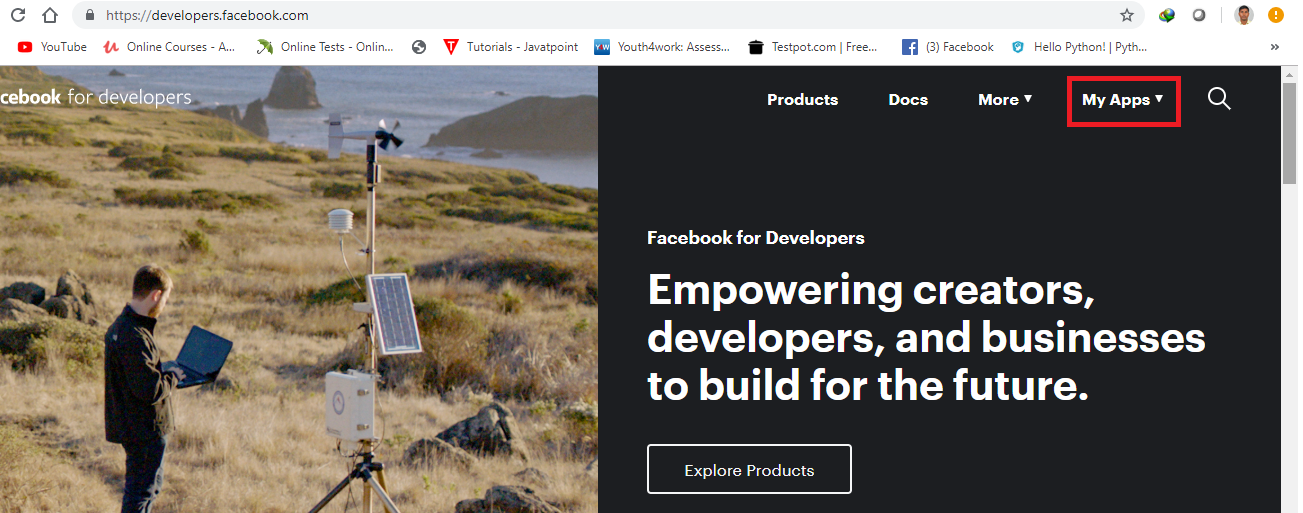
**Step #2** Here choosing Facebook as 3rd party server for open Authorization link is:

<https://developers.facebook.com/>

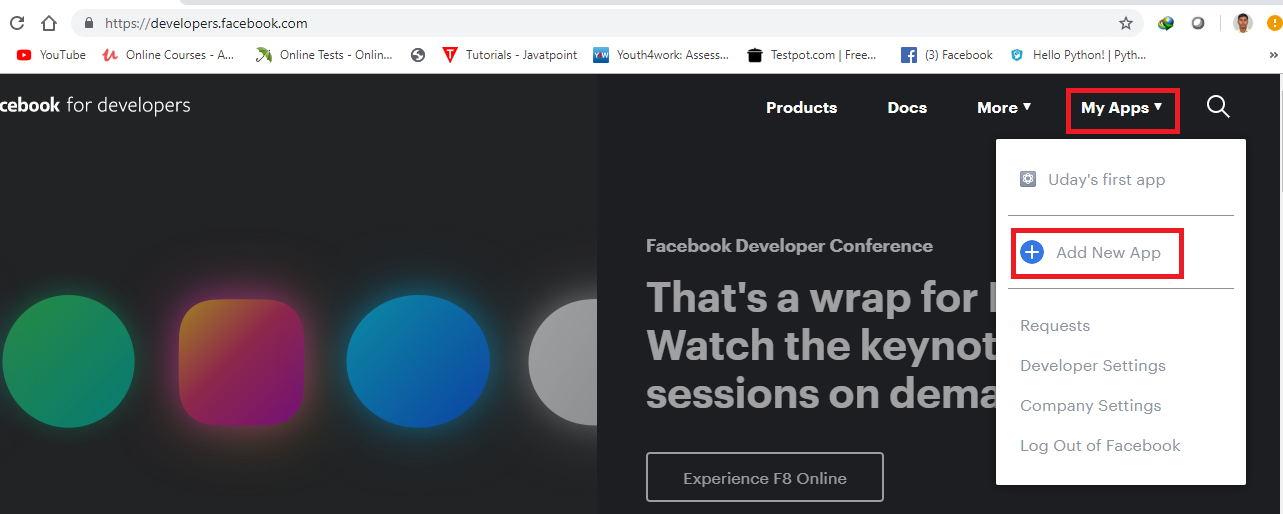
**Step #3** Define one new (client) Application in FB Authorization server which generates ClientId (AppId) and secrete (App Secret)

🡺Goto facebook developer page

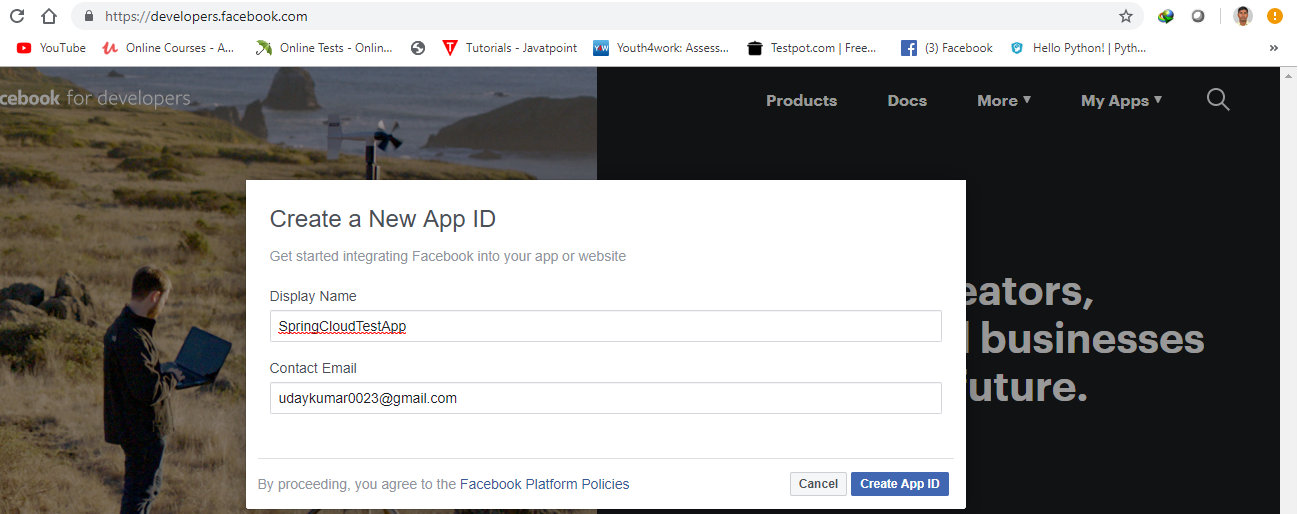
🡺Click on top right corner ”MyApps”



🡺Choose “Add New App”



🡺Provide Display name (ex: BootTest) and email id : [udaykumar0023@gmail.com](mailto:udaykumar0023@gmail.com).



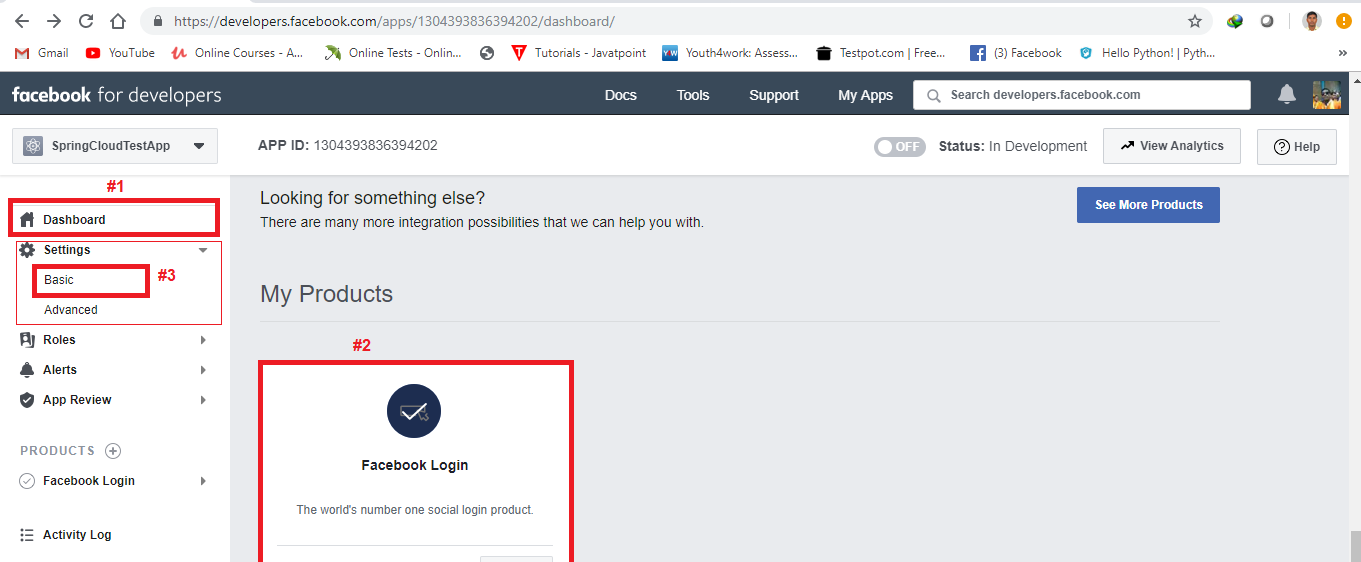
🡺Click on “Create AppId”.

🡺Complete Security verification

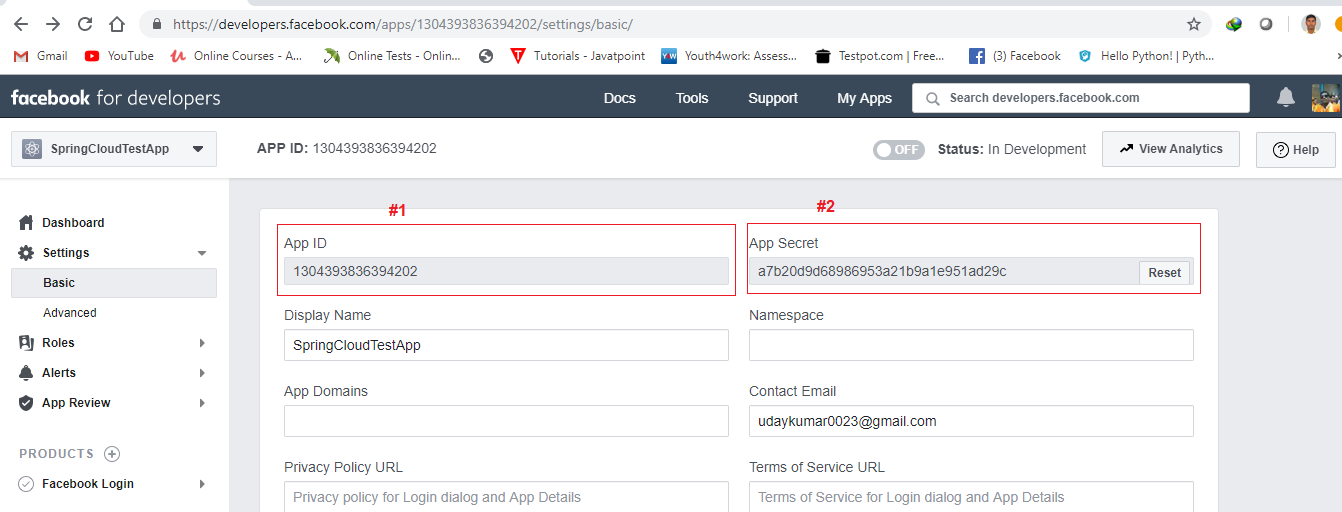
🡺Click on “Dashboard”

🡺Click on “facebook Login” setup button

🡺Click on “Settings” >>”Basic



🡺Basic AppId (ClientId) and App Secret (Client Secret)



**Step #4** Create one SpringBoot Starter app with dependencies “Security” and “Cloud OAuth2”.

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-oauth2</artifactId>

</dependency>

**Step #5** Create application.yml file using below key value pairs.

**application.yml:**

server:

9898

security:

oauth2:

//Auth Server details

client:

clientId: <your clientId here>

clientSecret: <Your secret here>

accessTokenUri:

userAuthorizationUri:

tokenName:

authenticationSchema:

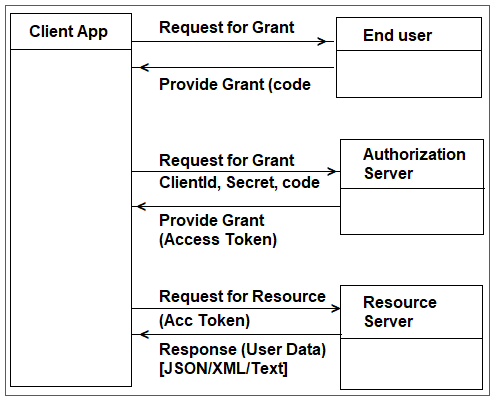
clientAuthenticationScheme:

#Resource Server details

resource:

userInfoUri:

**Request Execution flow :**



**Components Involved**: ClientApp, User(Browser), Auth Server (with Token Generator)and Resource Server (User data in XML/JSON).

**Step#6** Define SecurityConfig class (SecurityInit is not required, Handled by spring Boot only).

🡺Here Authentication Details not required to configure as we are using 3rd party security services.

🡺In Authorization config specify which URLs needs type **“EveryOneAccess“**

[PermitAll]

Package com.app.config;

@Configuration

@EnableOAuth2Sso

public class SecurityConfig extends WebSecurityConfigurerAdapter {

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRquests().antMatchers(“/”,”/login”).permitAll()

.anyRequest().authenticated();

}

}

**Step#7** Define RestController for User (or Any

package com.app.controller;

@RestController

pubic classs UserRestController {

@RequestMapping(“/user”)

public Principal showUser(Principal p) {

return p;

}

@RequestMapping(“/home”)

public String showData() {

return “Hello”;

}

}

**Step#8** Define one UI Page ex: index.html under src/main/resource, in static folder

**Index.html:**

<html><body>

<h1>Welcome to Login Page/</h1>

<a href=”user”>Facebook</a>

(or)

<hr/>

<form action=”#” method=”post”>

User : <input type=”text”>

Pwd : <input type=”password”>

<input type=”submit”>

</form>

</body></body></html>

**Step#9** Run application and Enter URL <http://localhost:9898>

**Step#10** Click on facebook Link and accept name (or) enter details.

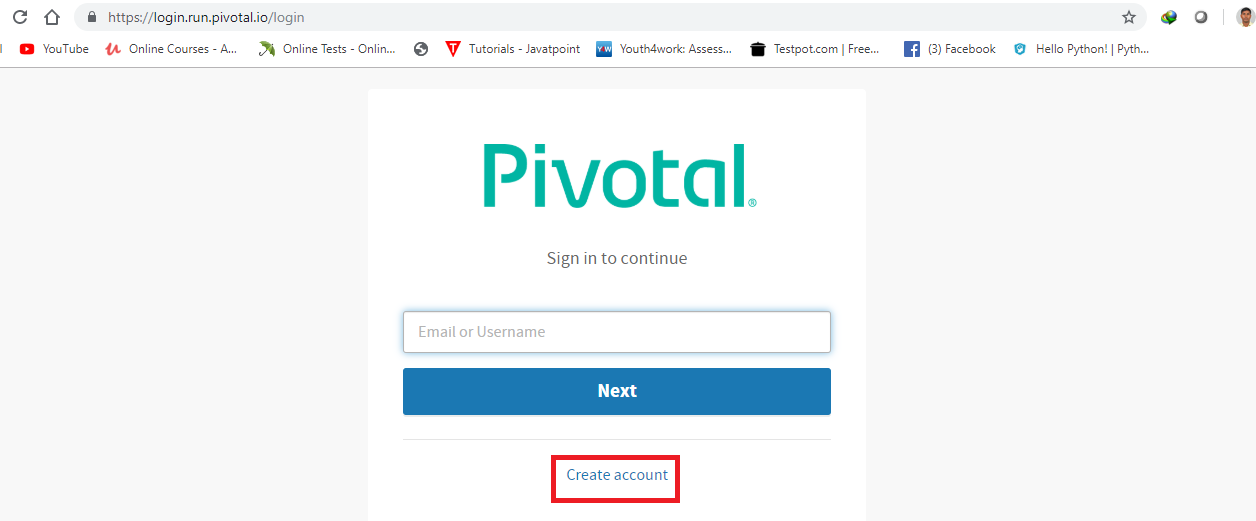
**Spring Boot PCF deployment**

**Pivotal Cloud Foundry (PCF)**

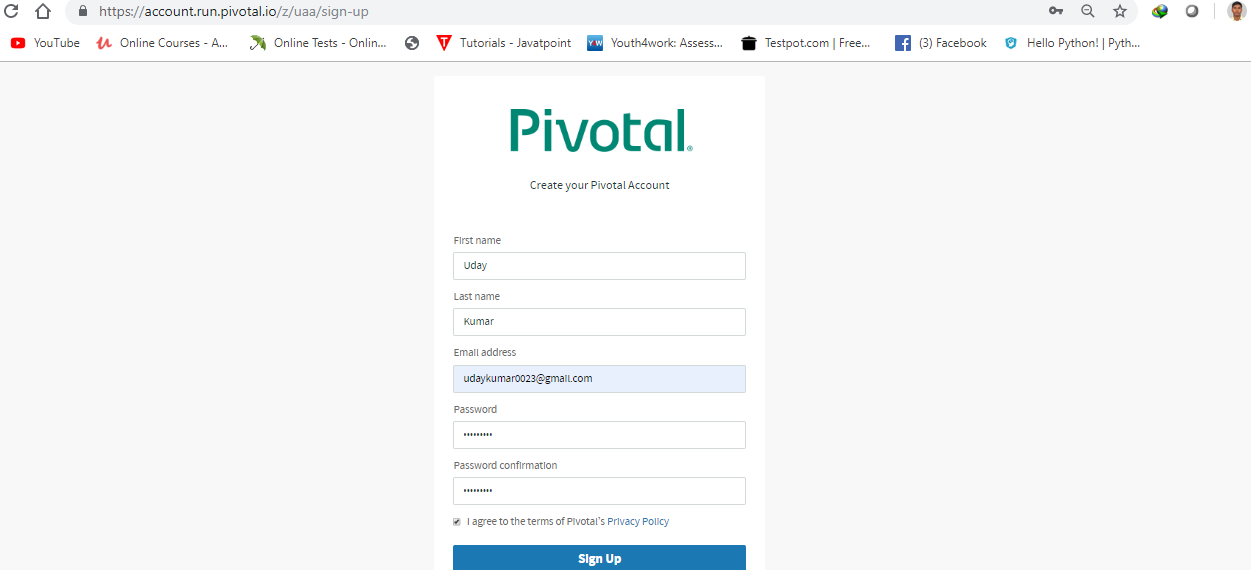
Pivotal Cloud Foundry is a Cloud Deployment Server provides service to Spring Boot and Microservices Applications mainly with all kinds of Environment setup like Databases, server, log tracers etc.

**#PCF Account Setup#**

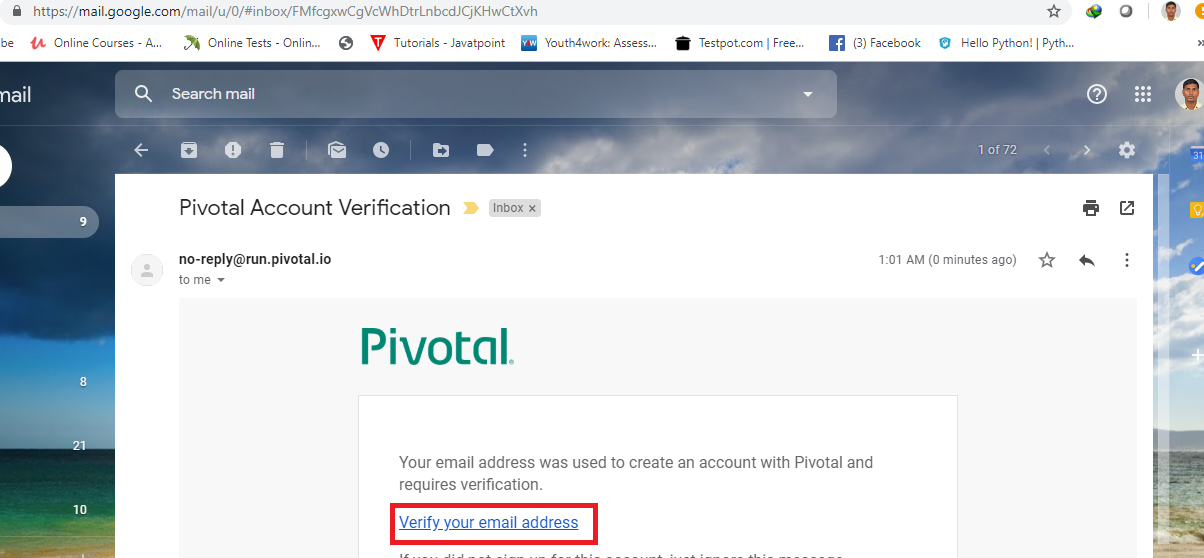
**Step#1:**  Goto <https://login.run.pivotal.io/login> And click on Create new account



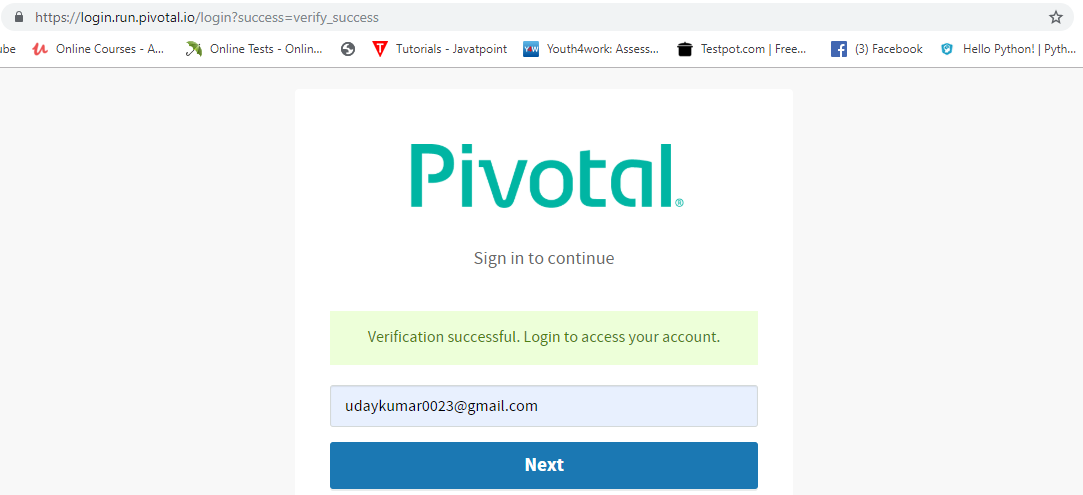
**Step#2:** Enter details name, email, password and conform password Register



**Step#3:** Goto Email Account and verify PCF link



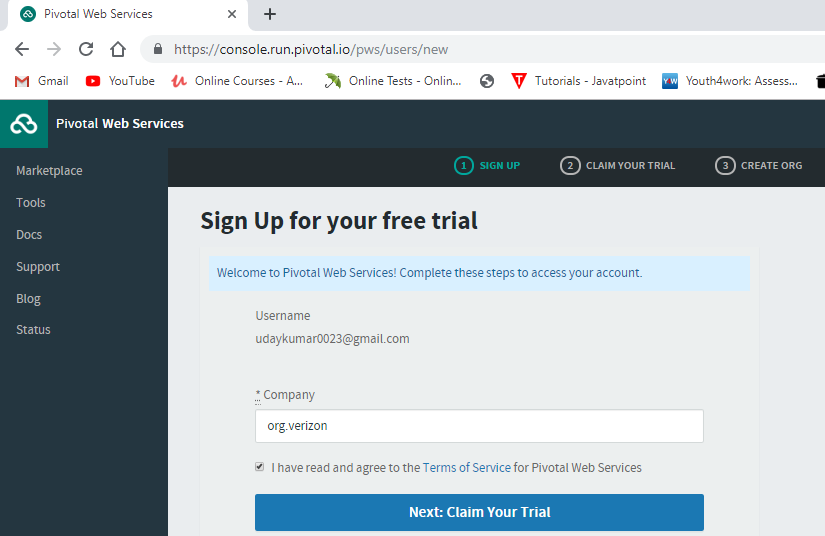
**Step#4:** Login to PCF account (Above URL)



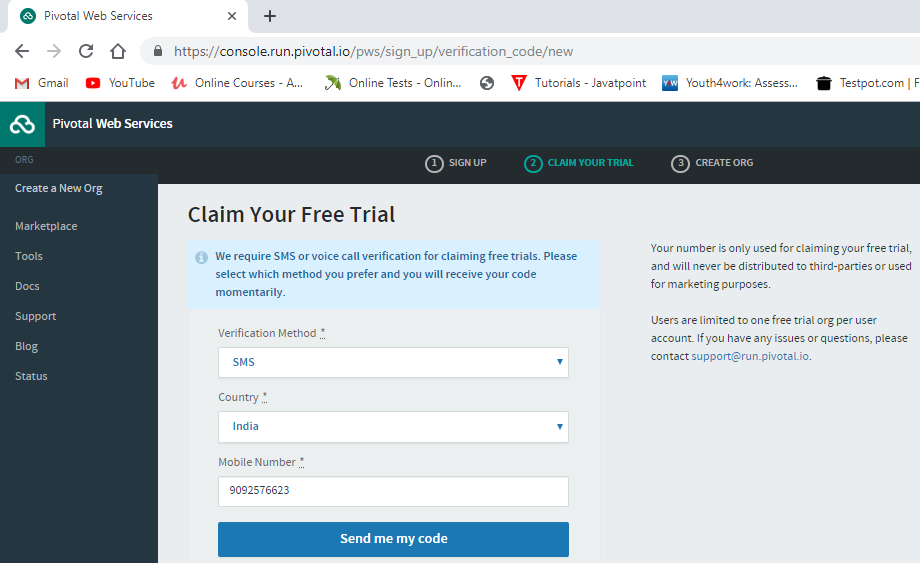
🡺Click on pivotal web service

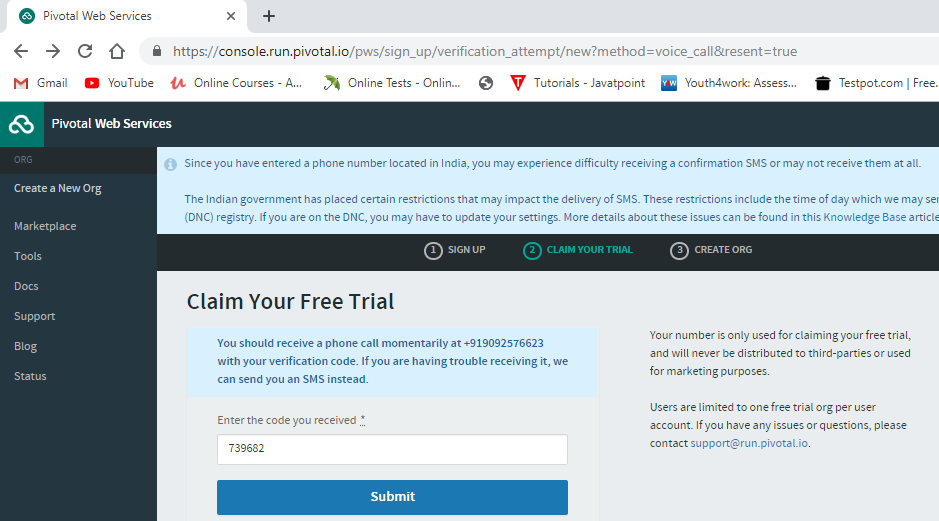
**Step#5:** Provide initial details like

🡺Company name

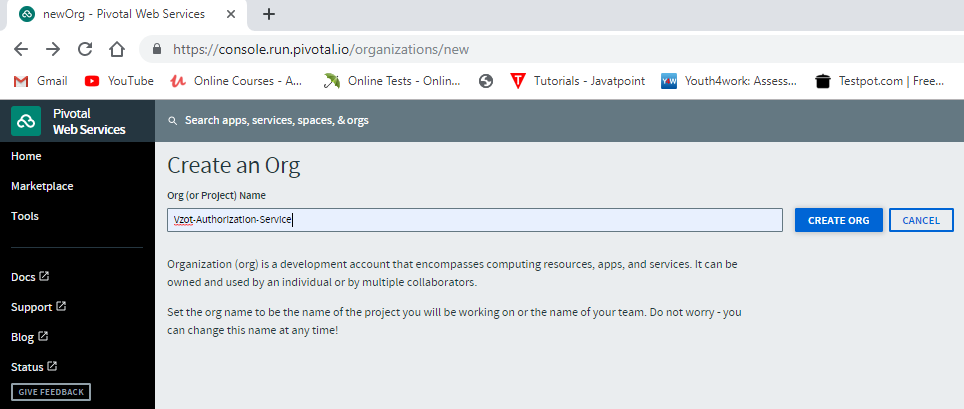


🡺Mobile number and verify





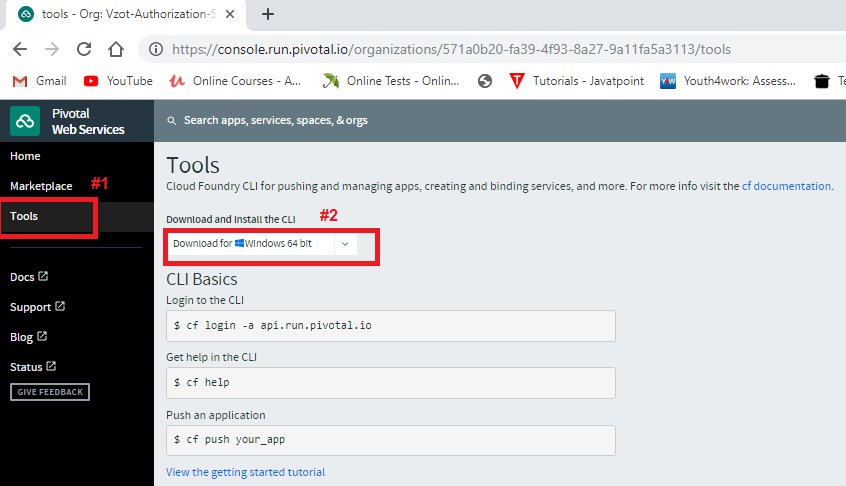
🡺Organization (org) name > finish



**Step#6:** Download and setup PCF CLI

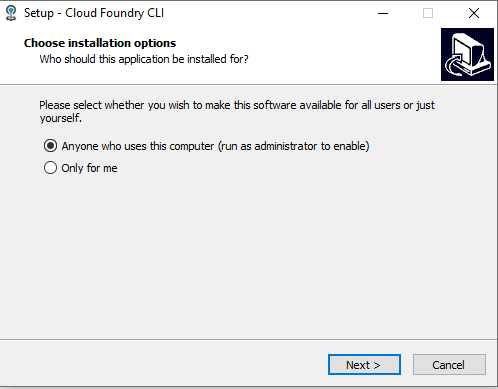
->Click on “Tools” option

->Choose OS and Bit and Download Software

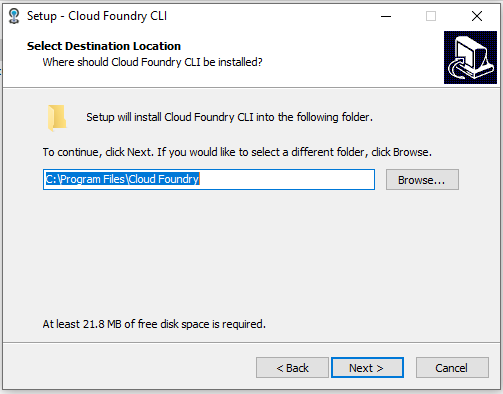


->Extract ZIP into Folder

->Click on “cf\_installer.exe”.



🡺next



🡺next 🡺next 🡺Finish

**Step#7:** Login and Logout Commands

🡺Goto cmd prompt

1.Login command is

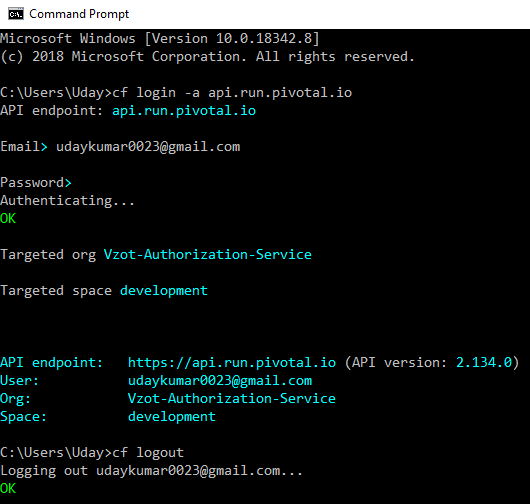
cf login –a api.run.pivotal.io

Email >

Password >

2.Logout command is

cf logout



**TASK:**

🡺Spring Boot WEB MVC + Thymeleaf + Data JPA + H2 (Curd Operation) + Bootstrap

**Step#8:** Create Org and space in PCF

🡺Login to PCF using browser

🡺Click on home 🡺 Click on create button

🡺Enter org name **ex:** sample-org

🡺finish

**\*\*\*** An org “sample-org” is created with default space (workspace) “development” is created.

=>A space (workspace) is a folder or location where project is stored and running.

**Step#9:** Create one Spring Boot Starter Project in STS.

🡺File 🡺 new 🡺 Spring Starter Project

🡺Enter Details 🡺 choose Dependencies (Ex: web)

🡺Finish

**Step#10:** Write Code for RestController, Services, Dao, and properties/yml files etc.

**Step#11:** Clean Application

🡺Right click 🡺 Run As 🡺 Maven Clean

**…Wait for status BUILD SUCCESS..**

🡺This step will clear all old folders and files which are in “project-name/target” folder.

**Step#12:** Do setup of JDK in workspace

>Windows > Preferences

>search with “Installed JRE”

>Click on “Add” > browse for location

Ex: “C:\Program Files\Java\jdk1.8.0\_201”

>Choose new and delete old JRE

>Apply and close

**Step#13:** Update JRE to Project

=>Right click on Project > build path

=>Configure build path > Choose JRE System Lib.

=>Edit > select workspace JRE/JDK

=>Apply and close.

**Step#14:** Install Application (build project)

=>Right click > Run As > Maven install

..wait for few minutes, still BUILD SUCCESS..

\*\*(If failed, then Update Maven Project, select force Update and finish.

Repeat step#11, 13 and then 14)

**Step#14:** Refresh Project to see updates in “target” folder.

=>Right click on Project > refresh (F5)

.. then ..

=>Right click on target folder

=>Properties (or Alt+Enter)

=>click on “Show In System Explorer”

=>Open target folder > find --.jar file

(Ex: SpringBootSampleApp-1.0.jar)

**Step#16:** Open command and login to PCF from here

=>shift + mouse right click

=>choose “open cmd window here”

=>Login to PCF

**cmd/>cf login –a api.run.pivotal.io**

Enter Email Id and password

=>Push Application to created org & space

**cmd/>cf push [PROJECT-NAME] –p [JARNAME].jar**

Ex:- cf push sathyatech –p SpringBootSampleApp-1.0.jar

.. wait for 5 minutes to upload project..

**Step#17:** Once success then go to PCF WebConsole (browser) and click on space “development”

To see total apps and running app.

=>Click on Route (URL) to execute app

=>Enter Paths to URL ex: /home/show …

**Step#18:** LogoutPCF from cmd prompt

**cmd/> cf logout**

\*\*\* BUILD FAIL at CLEAN

Then Force Update maven Project

🡺Project 🡺 alt + F5 🡺 choose Force update

🡺Apply and close [Finish]

\*\*\* BUILD FAIL at INSTALL

Then update JDK new version, in place of JRE

\*\*\* showing Cached Errors

=>Goto .m2 folder and delete that folder

=>come back to project > force update project

**TASK:**

=>Create Project with MySQL DB with CURD operations and upload to PCF

=>Go to Market place and chose mysql provider enter details

Instance name : myappsql

Choose or/space:

>choose plan > finish

=>Come to Home >Click on org > space > service

=>click on MySQl services > Bind App

=>select Our Project > bind > re-start app

=>Enter URL in browser and execute.