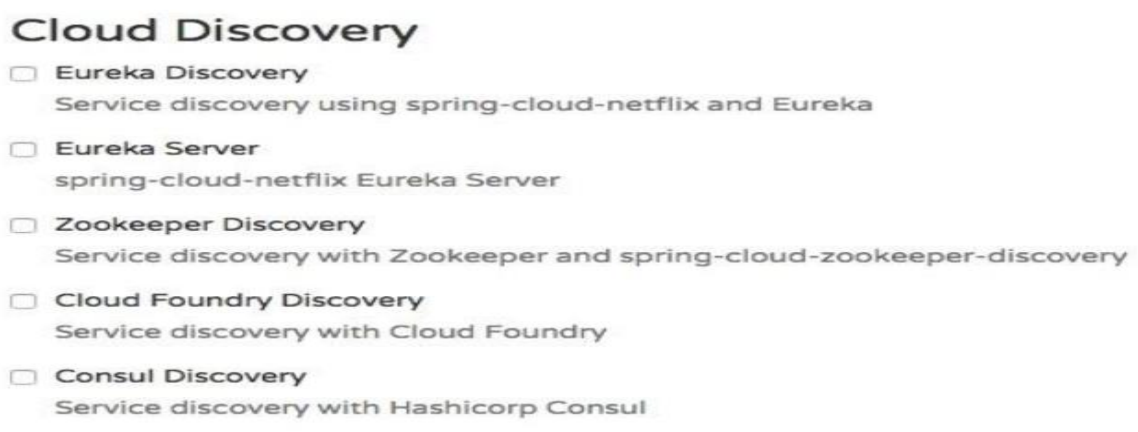
What is Dynamic Service Registration?

Dynamic Registration means, when a new service is started, it automatically enlists its availability in a central service registry. Similarly, when a service goes out of service, it is automatically delisted from the service registry. The registry always keeps up-to-date information of the services available, as well as their metadata.

What is Dynamic Discovery

Dynamic discovery is applicable from the service consumer's i.e.client point of view. Dynamic discovery is where clients look for the service registry to get the current state of the services, and then invoke the services accordingly. In this approach, **instead of statically configuring the service URLs, the URLs are picked up from the service registry**. Here the clients may keep a local cache of the registry data for faster access. In this approach, the state changes in the registry server will be propagated to the clients connected to it to avoid using stale data.

There are a number of options available for dynamic service registration and discovery. **Netflix Eureka, ZooKeeper, and Consul** are available as part of Spring Cloud, as shown in the http://start.spring.io/ screenshot below screenshot. **Etcd** is another service registry available outside of Spring Cloud to achieve dynamic service registration and discovery. we will focus on the Netflix Eureka implementation:

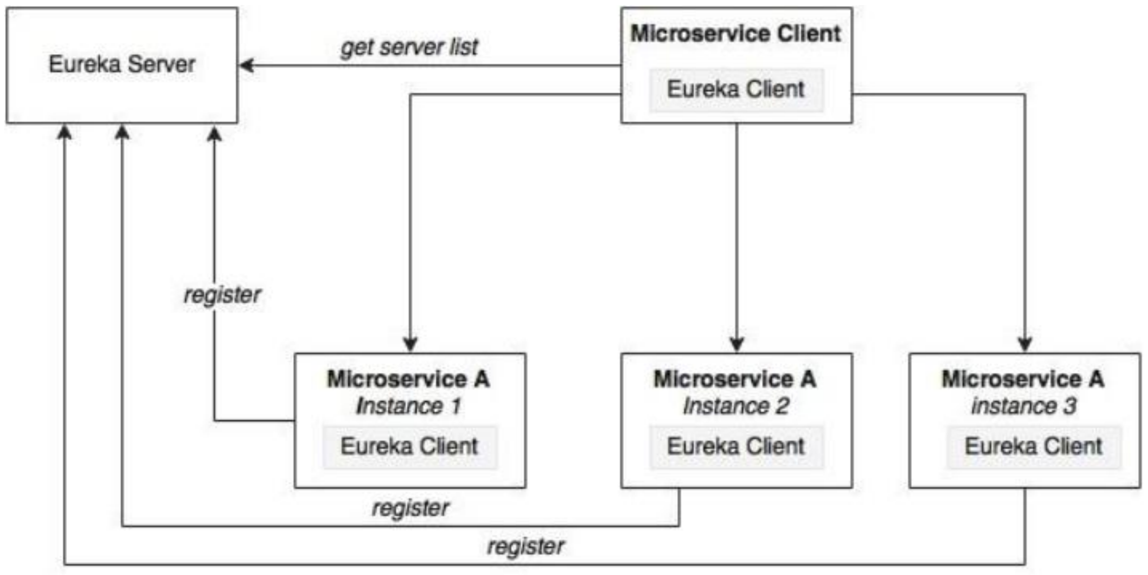

Cloud Discovery

- ☐ Eureka Discovery
  Service discovery using spring-cloud-netflix and Eureka
- ☐ Eureka Server
  spring-cloud-netflix Eureka Server
- ☐ Zookeeper Discovery
  Service discovery with Zookeeper and spring-cloud-zookeeper-discovery
- ☐ Cloud Foundry Discovery
  Service discovery with Cloud Foundry
- ☐ Consul Discovery
  Service discovery with Hashicorp Consul

What is Eureka?

Spring Cloud Eureka comes from Netflix OSS.Eureka is primarily used for *self/dynamic-registration, dynamic discovery, and load balancing*. **Eureka uses Ribbon for load balancing internally.**

Lets consider the following diagram:



As shown in the preceding diagram, Eureka consists of a server component and a client-side component. The server component is the registry in which all microservices register their availability.**The microservices use the Eureka client for registering their availability**. The consuming components will also use the Eureka client for discovering the service instances.

How Registry will work?

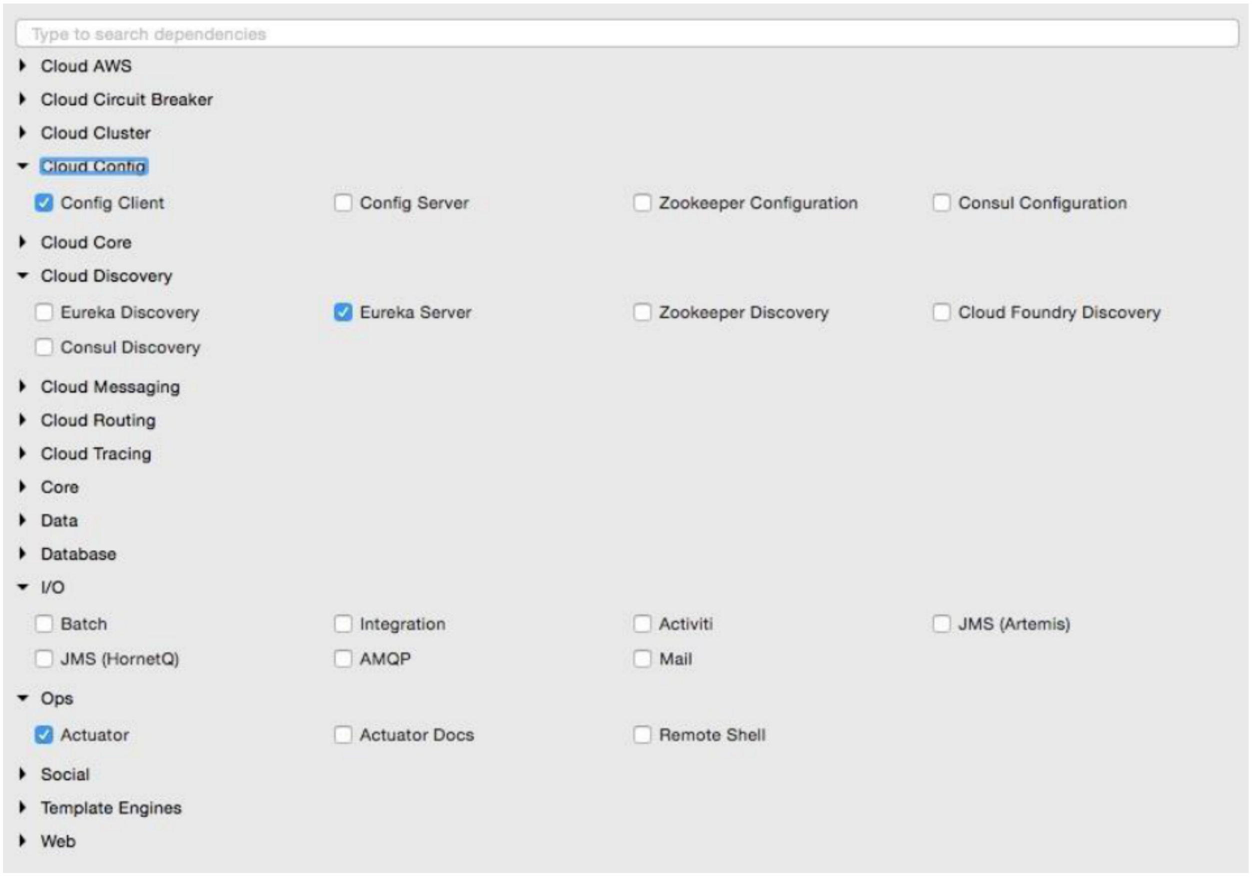
When a microservice is bootstrapped, it reaches out to the Eureka server, and advertises its existence with the binding information. **Once registered, the service endpoint sends ping requests to the registry every 30 seconds to renew its lease.** If a service endpoint cannot renew its lease in a few attempts, that service endpoint will be taken out of the service registry.

The registry information will be replicated to all Eureka clients so that the clients have to go to the remote Eureka server for each and every request. Eureka clients fetch the registry information from the server, and cache it locally. After that, the clients use that information to find other services. This information is updated periodically (every 30 seconds) by getting the delta updates between the last fetch cycle and the current one.

When a client wants to contact a microservice endpoint, the Eureka client provides a list of currently available services based on the requested service ID. The Eureka server is zone aware. Zone information can also be supplied when registering a service. When a client requests for a services instance, the Eureka service tries to find the service running in the same zone. The Ribbon client then load balances across these available service instances supplied by the Eureka client. The communication between the Eureka client and the server is done using REST and JSON.

Setting up the Eureka server

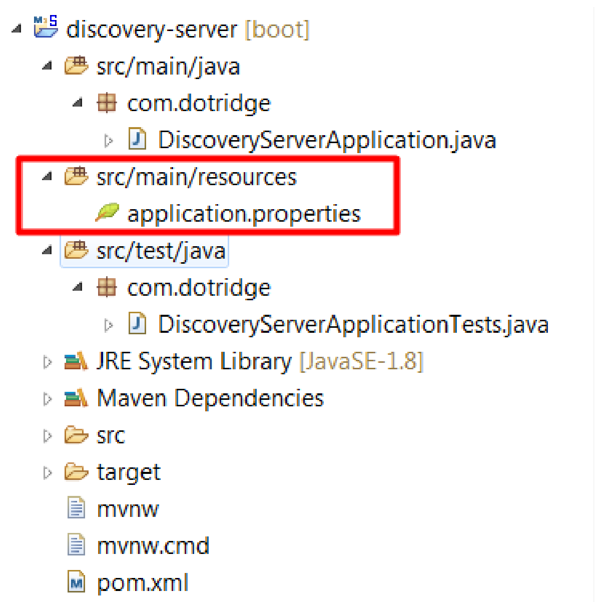
Step-1: Start a new Spring Starter project discovery-service, and select Config Client, Eureka Server, and Actuator:



Type to search dependencies

- Cloud AWS
- Cloud Circuit Breaker
- Cloud Cluster
- Cloud Config
  - [✓] Config Client
  - [ ] Config Server
  - [ ] Zookeeper Configuration
  - [ ] Consul Configuration
- Cloud Core
- Cloud Discovery
  - [ ] Eureka Discovery
  - [✓] Eureka Server
  - [ ] Zookeeper Discovery
  - [ ] Cloud Foundry Discovery
  - [ ] Consul Discovery
- Cloud Messaging
- Cloud Routing
- Cloud Tracing
- Core
- Data
- Database
- I/O
  - [ ] Batch
  - [ ] Integration
  - [ ] Activiti
  - [ ] JMS (Artemis)
  - [ ] JMS (HornetQ)
  - [ ] AMQP
  - [ ] Mail
- Ops
  - [✓] Actuator
  - [ ] Actuator Docs
  - [ ] Remote Shell
- Social
- Template Engines
- Web

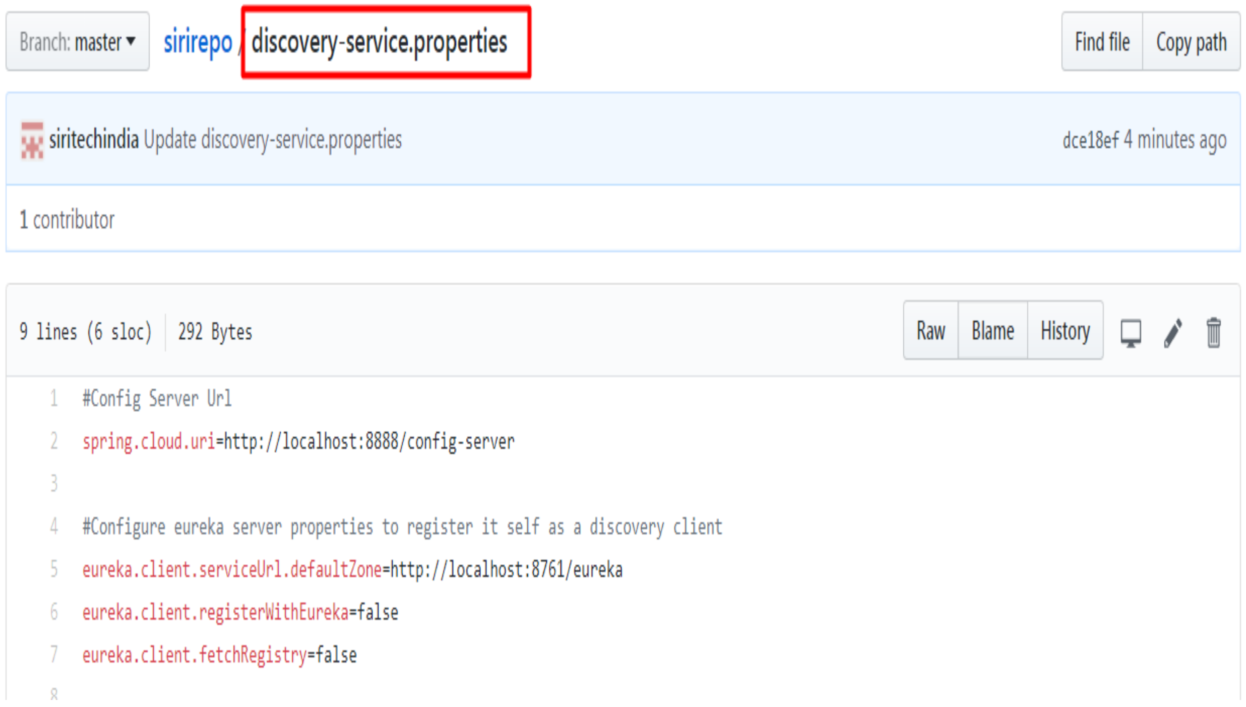
The project structure of the Eureka server i.e,discovery-server is shown in the following image:



```
discovery-server [boot]
  src/main/java
    com.dotridge
      DiscoveryServerApplication.java
  src/main/resources
    application.properties
  src/test/java
    com.dotridge
      DiscoveryServerApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Note that the main application is named DiscoveryServerApplication.java

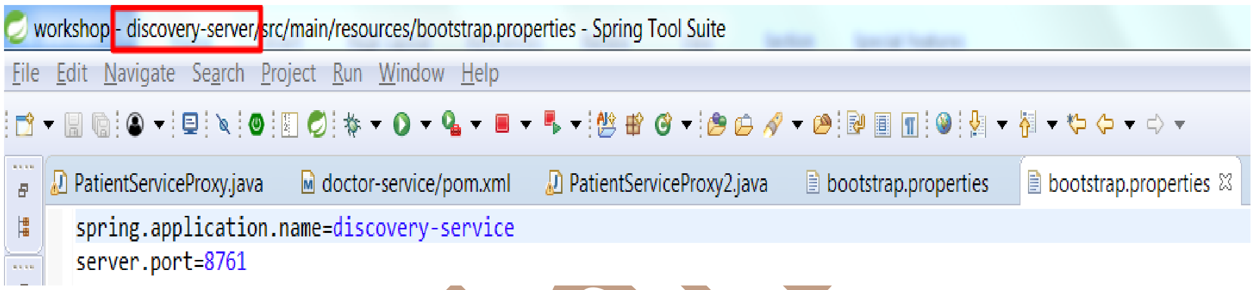
Step-2: Create a <spring.application.name>.properties I.e. discovery-server.properties file and commit to the git repository as discovery-service uses config-server to fetch the configuration properties. This file should have the config server url and eureka server properties as shown below:


Branch: master
sirirepo / discovery-service.properties
Find file
Copy path
siritechindia Update discovery-service.properties
dce18ef 4 minutes ago
1 contributor
9 lines (6 sloc) | 292 Bytes
Raw
Blame
History

```
#Config Server Url
spring.cloud.uri=http://localhost:8888/config-server

#Configure eureka server properties to register it self as a discovery client
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
```

The Eureka server can be set up in a standalone mode or in a clustered mode. We will start with the standalone mode. By default, the Eureka server itself is another Eureka client. This is particularly useful when there are multiple Eureka servers running for high availability. The client component is responsible for synchronizing state from the other Eureka servers. The Eureka client is taken to its peers by configuring the **eureka.client.serviceUrl.defaultZone** property.In the standalone mode, we point **eureka.client.serviceUrl.defaultZone** back to the same standalone instance. Even it is not necessary to configure the above eureka properties when it is running in standalone mode.

__Step-3:__ Rename application.properties to bootstrap.properties since this is using the Config server. It should have application name and port as shown below:

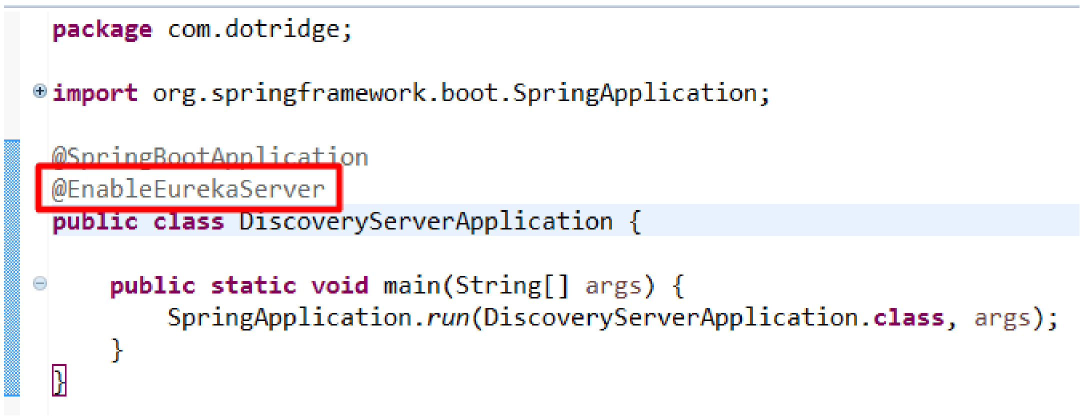

workshop - discovery-server/src/main/resources/bootstrap.properties - Spring Tool Suite
File Edit Navigate Search Project Run Window Help
PatientServiceProxy.java
doctor-service/pom.xml
PatientServiceProxy2.java
bootstrap.properties
bootstrap.properties

```
spring.application.name=discovery-service
server.port=8761
```

__Note:__

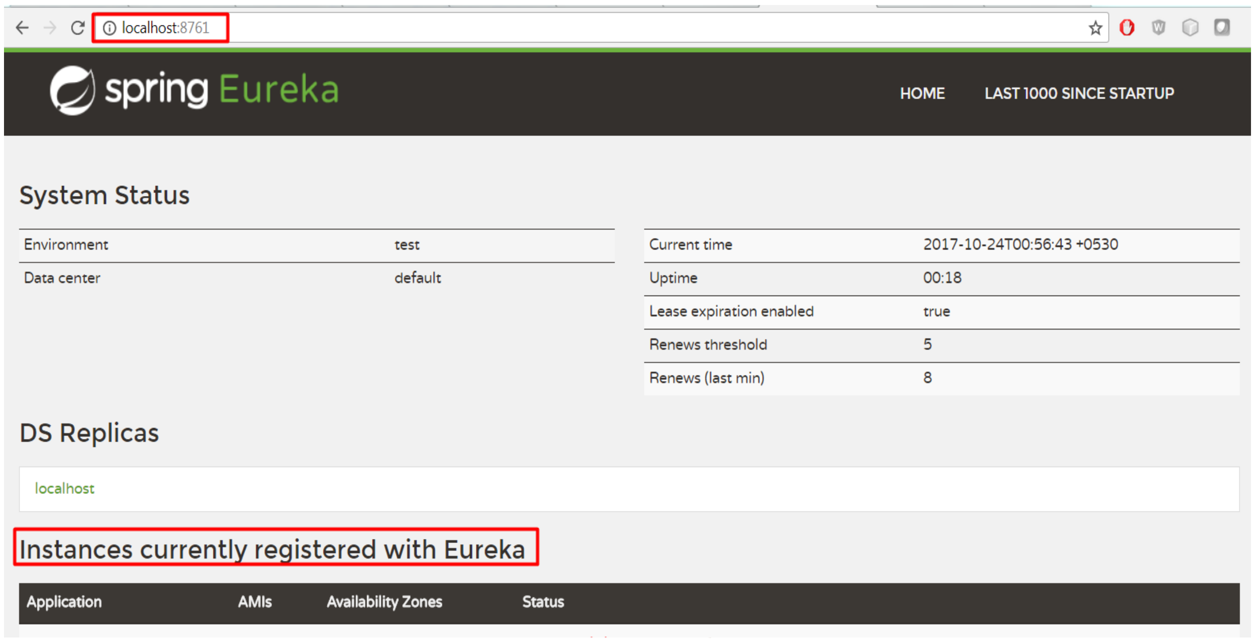
Renaming application.properties file to bootstrap.properties file is not necessary. But when we go for high availability of config-server and discovery-server we might need this.

__Step-4:__ In DiscoveryServerApplication, add **@EnableEurekaServer** annotation as shown below



```
package com.dotridge;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServerApplication {

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

__Step-5:__ We are now ready to start the Eureka server. Ensure that the Config server is also started. Right-click on the application and then choose Run As | Spring Boot App. Once the application is started, open http://localhost:8761 in a browser to see the Eureka console.


localhost:8761

spring Eureka

HOME LAST 1000 SINCE STARTUP

System Status

Environment	test
Data center	default

Current time	2017-10-24T00:56:43 +0530
Uptime	00:18
Lease expiration enabled	true
Renews threshold	5
Renews (last min)	8

DS Replicas

localhost

Instances currently registered with Eureka

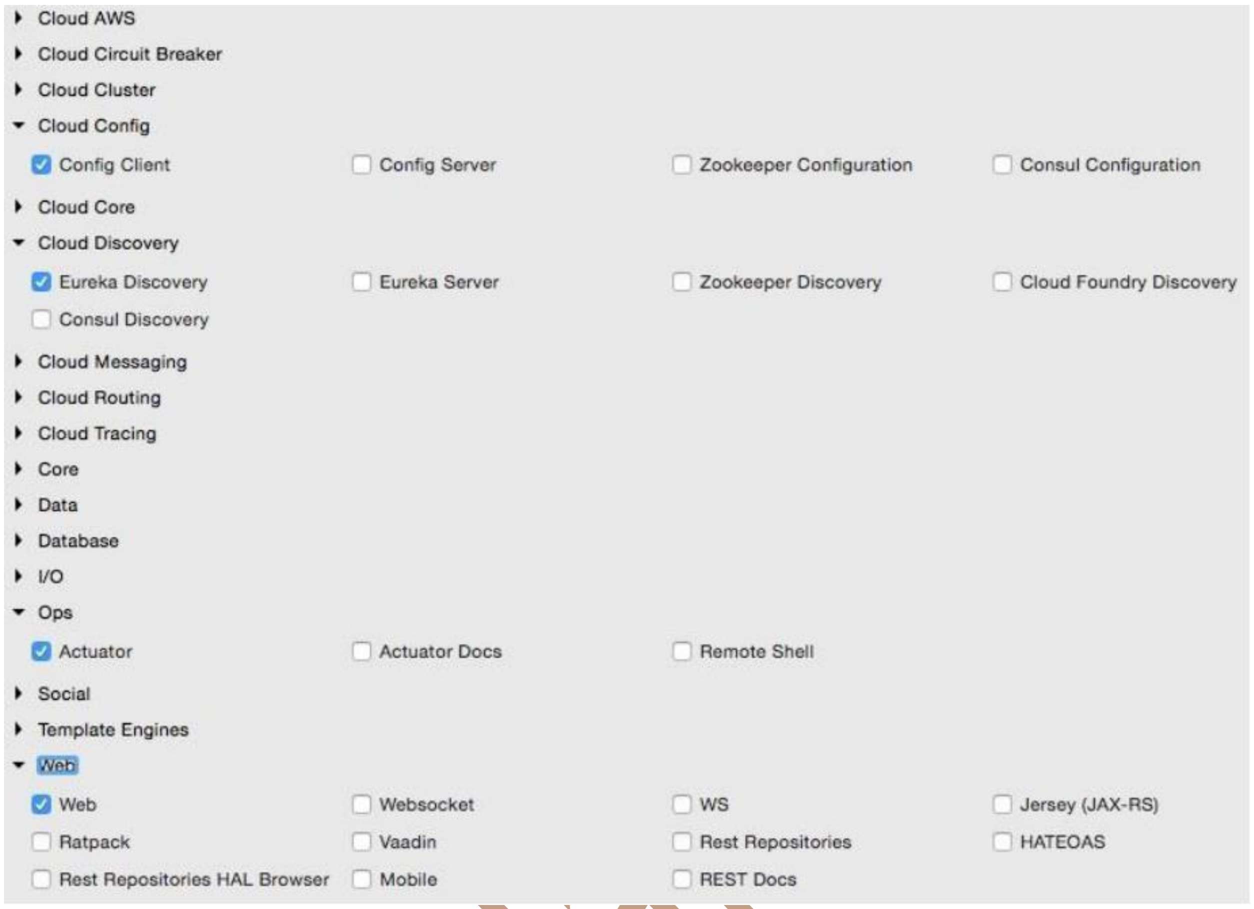
Application	AMIs	Availability Zones	Status

In the console, note that there is no instance registered under Instances currently registered with Eureka. Since no services have been started with the Eureka client enabled, the list is empty at this point.

<u>Dynamic Discovery</u>

Making a few changes to our microservice will enable dynamic registration and discovery using the Eureka service.

<u>Step-1:</u> add the Eureka dependencies to the pom.xml file. If the services are being built up fresh using the Spring Starter project, then select **Config Client**, **Actuator**, **Web** as well as **Eureka discovery** client as follows:



- Cloud AWS
- Cloud Circuit Breaker
- Cloud Cluster
- Cloud Config
  - [✓] Config Client
  - [ ] Config Server
  - [ ] Zookeeper Configuration
  - [ ] Consul Configuration
- Cloud Core
- Cloud Discovery
  - [✓] Eureka Discovery
  - [ ] Eureka Server
  - [ ] Zookeeper Discovery
  - [ ] Cloud Foundry Discovery
  - [ ] Consul Discovery
- Cloud Messaging
- Cloud Routing
- Cloud Tracing
- Core
- Data
- Database
- I/O
- Ops
  - [✓] Actuator
  - [ ] Actuator Docs
  - [ ] Remote Shell
- Social
- Template Engines
- Web
  - [✓] Web
  - [ ] Websocket
  - [ ] WS
  - [ ] Jersey (JAX-RS)
  - [ ] Ratpack
  - [ ] Vaadin
  - [ ] Rest Repositories
  - [ ] HATEOAS
  - [ ] Rest Repositories HAL Browser
  - [ ] Mobile
  - [ ] REST Docs

But as we are modifying our existing MicroServices, add the following additional dependency to all MicroServices in their pom.xml files:

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
    </dependencies>
```

Step-2: The following property has to be added to all microservices in their respective configuration files in config-server i.e.git repo. This will help the microservices to connect to the Eureka server. Commit to Git once updates are completed:

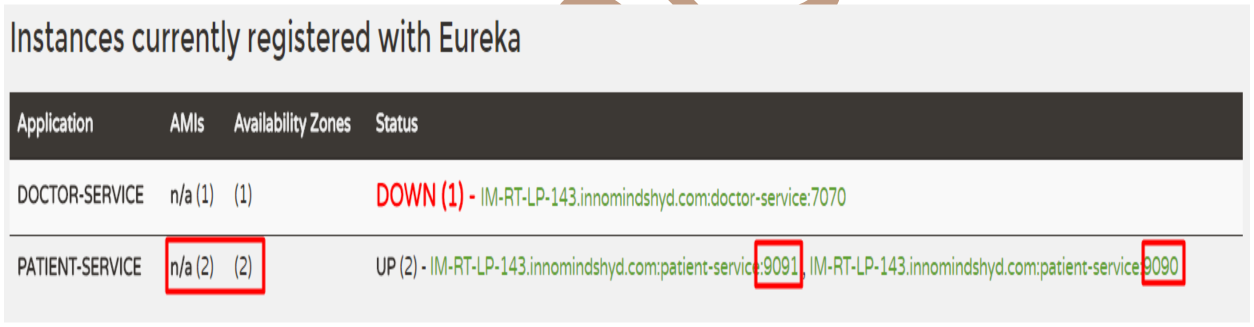
```

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

```

In our case add in doctor-service.properties, patient-service.properties files of config-server I.e.available in git repo.

<u>Step-3:</u> Add @EnableDiscoveryClient to all microservices in their respective Spring Boot main classes. This asks Spring Boot to register these services at start up to advertise their availability. Start all servers except doctor-service. Since we are using the Ribbon client on the doctor-service micro service, the behavior could be different when we add the Eureka client in the class path. We will fix this soon.

```
@SpringBootApplication
@EnableDiscoveryClient
public class PatientMServiceApplication {


    public static void main(String[] args) {

        SpringApplication.run(PatientMServiceApplication.class, args);
    }
```

<u>Step-4:</u> Going to the Eureka URL (http://localhost:8761), we can see that patient-service is up and running:



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
DOCTOR-SERVICE	n/a (1)	(1)	DOWN (1) - IM-RT-LP-143.innomindshyd.com:doctor-service:7070
PATIENT-SERVICE	n/a (2)	(2)	UP (2) - IM-RT-LP-143.innomindshyd.com:patient-service:9091, IM-RT-LP-143.innomindshyd.com:patient-service:9090

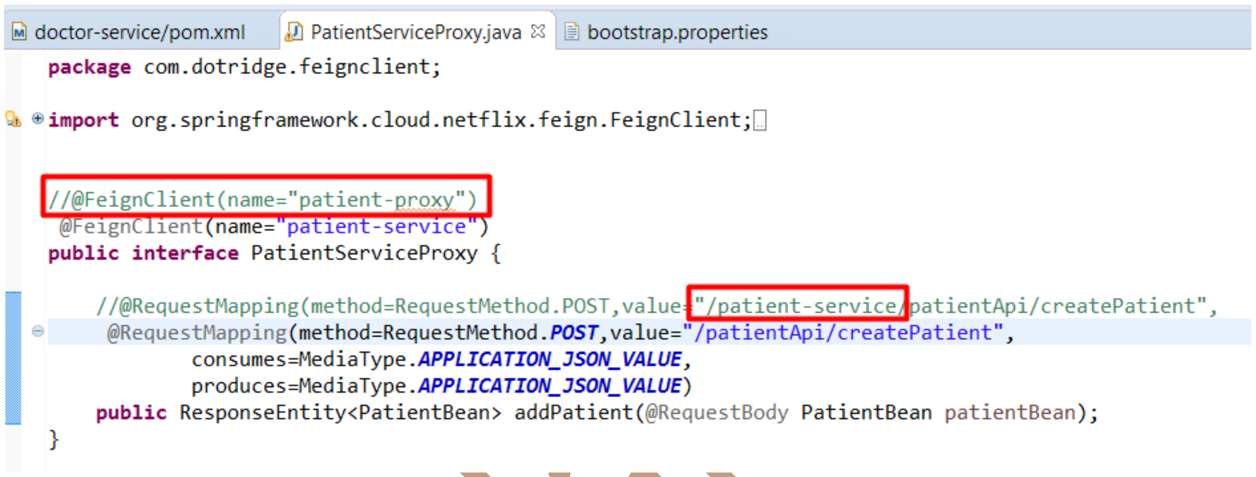
<u>Eureka Service with Load Balancing</u>

Time to fix the issue with doctor-service. We will remove our earlier Ribbon client, and use Eureka instead. Eureka internally uses Ribbon for load balancing. Hence, the load balancing behavior will not change.

<u>Step-1:</u> Remove the following dependency from doctor-service.properties file

```
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-ribbon</artifactId>
    </dependency>
```

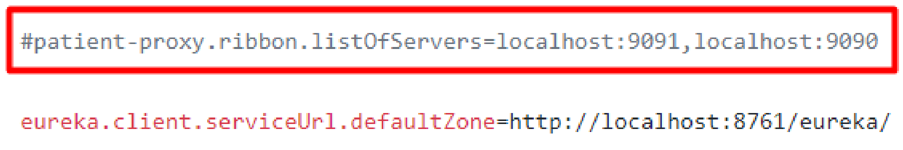
<u>Step-2:</u> Also remove the @RibbonClient annotation from the PatientServiceProxy class created in doctor-service micro service to communicate with the patient-service declaratively. Update @FeignClient(name="patient-proxy") to match the actual Patient microservices' service ID. In this case, patient-service is the service ID configured in the patient-service microservices' bootstrap.properties/application.properties file with the property spring.application.name. This is the name that the Eureka discovery client sends to the Eureka server. The service ID will be used as a key for the services registered in the Eureka server.



```
package com.dotridge.feignclient;

import org.springframework.cloud.netflix.feign.FeignClient;

//@FeignClient(name="patient-proxy")
 @FeignClient(name="patient-service")
public interface PatientServiceProxy {

    //@RequestMapping(method=RequestMethod.POST,value="/patient-service/patientApi/createPatient",
     @RequestMapping(method=RequestMethod.POST,value="/patientApi/createPatient",
            consumes=MediaType.APPLICATION_JSON_VALUE,
            produces=MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<PatientBean> addPatient(@RequestBody PatientBean patientBean);
}
```

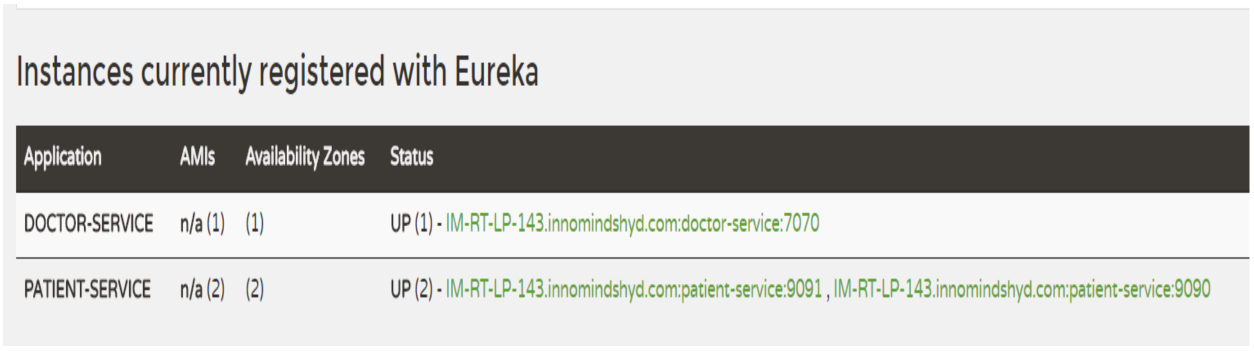
In the above screen shot we were commented the @FeignClient(name="patient-proxy"). when we are using this the patient-service microservice end point have been configured as "/patient-service/patientApi/createPatient" but now as we commented it and using load balancing supported by eureka, we no need to give "/patient-service" prefix with the "/patientApi/createPatient" as it is been now appending dynamically by eureka as it is registered with the eureka as a service id.

<u>Step-3:</u> Also remove the list of servers from the doctor-service.properties file of config-server I.e.available in git repo. With Eureka, we are going to dynamically discover this list from the Eureka server: patient-proxy.ribbon.listOfServers=localhost:9090, localhost:9091



```
#patient-proxy.ribbon.listOfServers=localhost:9091,localhost:9090

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

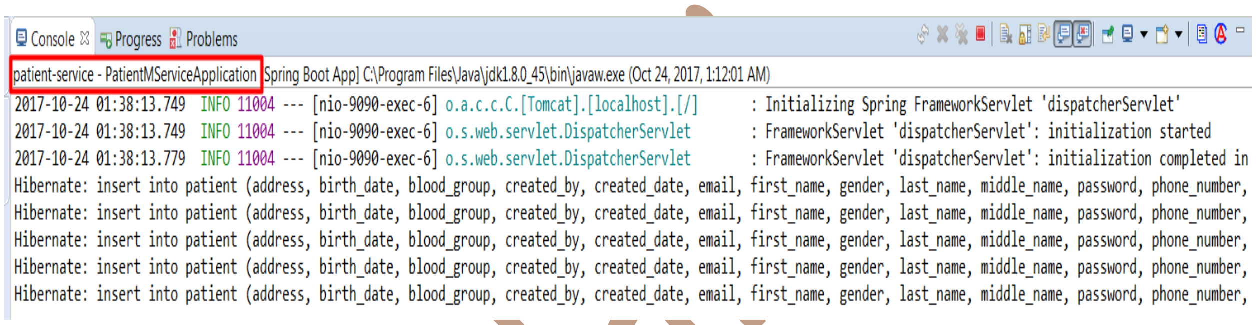
<u>Step-4:</u> Start the doctor-service. now will see that doctor-service is successfully registered with the eureka as shown below:



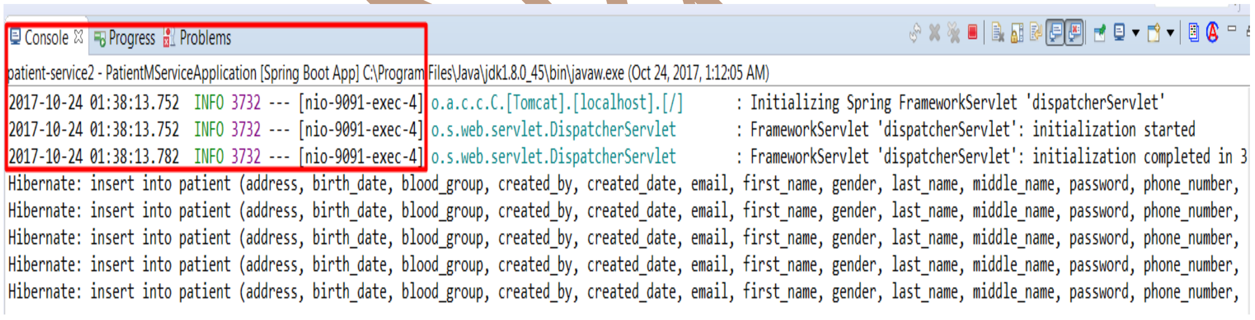
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
DOCTOR-SERVICE	n/a (1)	(1)	UP (1) - IM-RT-LP-143.innomindshyd.com:doctor-service:7070
PATIENT-SERVICE	n/a (2)	(2)	UP (2) - IM-RT-LP-143.innomindshyd.com:patient-service:9091 , IM-RT-LP-143.innomindshyd.com:patient-service:9090

Step-6: now again when we created the 10 users through the jmeter, we could see 5 will be handled by patient-service instance 1 running on 9090 and remaining 5 should be handled by another instance running on 9091.



```
patient-service - PatientMServiceApplication [Spring Boot App] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 24, 2017, 1:12:01 AM)
2017-10-24 01:38:13.749  INFO 11004 --- [nio-9090-exec-6] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
2017-10-24 01:38:13.749  INFO 11004 --- [nio-9090-exec-6] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
2017-10-24 01:38:13.779  INFO 11004 --- [nio-9090-exec-6] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
```



```
patient-service2 - PatientMServiceApplication [Spring Boot App] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 24, 2017, 1:12:05 AM)
2017-10-24 01:38:13.752  INFO 3732 --- [nio-9091-exec-4] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
2017-10-24 01:38:13.752  INFO 3732 --- [nio-9091-exec-4] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
2017-10-24 01:38:13.782  INFO 3732 --- [nio-9091-exec-4] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 3
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
Hibernate: insert into patient (address, birth_date, blood_group, created_by, created_date, email, first_name, gender, last_name, middle_name, password, phone_number,
```

This proves that Load balancing is now providing by Eureka Server

Note:

If we observe,**PatientServiceProxy** class where we are using @FiegnClient, earlier without load balaning we have used   and attribute "url=localhost:9090/patient-service" of @FiegnClient annotation. But now as we are using Eureka Server discovery we no need to give the url rather the service name is enough to given.