The Spring Cloud project is an umbrella project from the Spring team that implements a set of common patterns required by distributed systems.Spring Cloud by itself is not a cloud solution. Rather, it provides a number of capabilities that are essential when developing applications targeting cloud deployments that follows the Twelve-Factor application principles.

The cloud-ready solutions that are developed using Spring Cloud are portable across many cloud providers such as Cloud Foundry, AWS, Heroku, and so on. Built on Spring's "**convention over configuration**" approach, Spring Cloud defaults all configurations by hiding the complexities, and by providing simple declarative configurations to build echo-systems.

Spring Cloud offers many choices of cloud solutions based on their requirements. For example, the service registry can be implemented using popular options such as Eureka, ZooKeeper, or Consul. The components of Spring Cloud are fairly decoupled.

Note

What is the difference between Spring Cloud and Cloud Foundry?

Spring Cloud is a developer kit for developing Internet-scale Spring Boot applications, whereas Cloud Foundry is an open-source Platform as a Service for building, deploying, and scaling applications.

Spring Cloud releases

The Spring Cloud project is an overarching Spring project that includes a combination of different components. The versions of these components are defined in the BOM.
we will use the **Brixton.RELEASE** version of the Spring Cloud:

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Brixton.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
```

The spring-cloud-starter-parent defines different versions of its sub components as follows:

```
<spring-cloud-aws.version>1.1.0.RELEASE</spring-cloud-aws.version>
<spring-cloud-bus.version>1.1.0.RELEASE</spring-cloud-bus.version>
<spring-cloud-cloudfoundry.version>1.0.0.RELEASE</spring-cloud-
cloudfoundry.version>
<spring-cloud-commons.version>1.1.0.RELEASE</spring-cloud-commons.version>
```

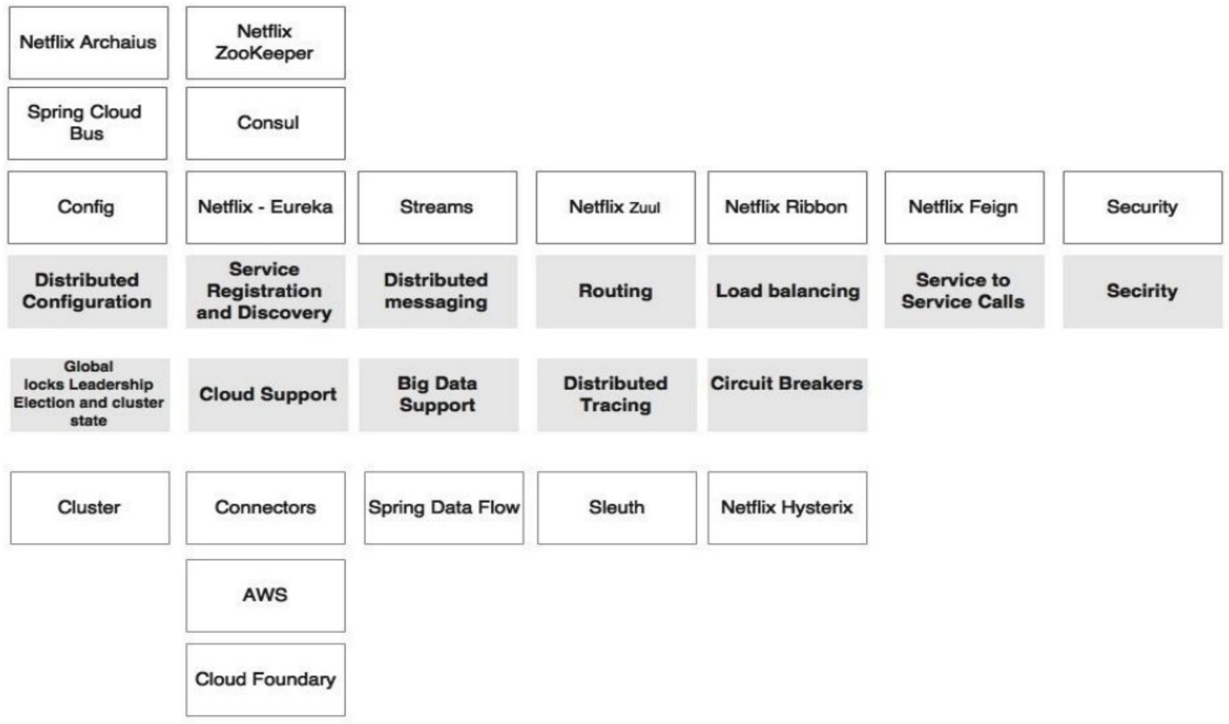
```
<spring-cloud-config.version>1.1.0.RELEASE</spring-cloud-config.version>
<spring-cloud-netflix.version>1.1.0.RELEASE</spring-cloud-netflix.version>
<spring-cloud-security.version>1.1.0.RELEASE</spring-cloud-security.version>
<spring-cloud-cluster.version>1.0.0.RELEASE</spring-cloud-cluster.version>
<spring-cloud-consul.version>1.0.0.RELEASE</spring-cloud-consul.version>
<spring-cloud-sleuth.version>1.0.0.RELEASE</spring-cloud-sleuth.version>
<spring-cloud-stream.version>1.0.0.RELEASE</spring-cloud-stream.version>
<spring-cloud-zookeeper.version>1.0.0.RELEASE </spring-cloud-zookeeper.version>
```

Note

The names of the Spring Cloud releases are in an alphabetic sequence, starting with A, following the names of the London Tube stations. **Angel** was the first release, and **Brixton** is the second release.

Components of Spring Cloud

Each Spring Cloud component specifically addresses certain distributed system capabilities. The grayed-out boxes at the bottom of the following diagram show the capabilities, and the boxes placed on top of these capabilities showcase the Spring Cloud sub projects addressing these capabilities:



The Spring Cloud capabilities are explained as follows:

Distributed configuration: Configuration properties are hard to manage when there are many microservice instances running under different profiles such as development, test, production, and so on. It is, therefore, important to manage them centrally, in a controlled way. The distributed configuration management module is to externalize and centralize microservice configuration parameters. **Spring Cloud Config** is an externalized configuration server with Git or SVN as the backing repository. Spring Cloud Bus provides support for propagating configuration changes to multiple subscribers, generally a microservice instance. Alternately, ZooKeeper or HashiCorp's Consul can also be used for distributed configuration management.

- **Routing**: Routing is an API gateway component, primarily used similar to a reverse proxy that forwards requests from consumers to service providers. The gateway component can also perform software-based routing and filtering. **Zuul** is a lightweight API gateway solution that offers fine-grained controls to developers for traffic shaping and request/response transformations
- **Load balancing**: The load balancer capability requires a software-defined load balancer module which can route requests to available servers using a variety of load balancing algorithms. **Ribbon** is a Spring Cloud sub project which supports this capability. Ribbon can work as a standalone component, or integrate and work seamlessly with Zuul for traffic routing.
- **Service registration and discovery**: The service registration and discovery module enables services to programmatically register with a repository when a service is available and ready to accept traffic. The microservices advertise their existence, and make them discoverable. The consumers can then look up the registry to get a view of the service availability and the endpoint locations. The registry, in many cases, is more or less a dump. But the components around the registry make the ecosystem intelligent. There are many subprojects existing under Spring Cloud which support registry and discovery capability. **Eureka, ZooKeeper, and Consul** are three sub projects implementing the registry capability.
- **Service-to-service calls**: The **Spring Cloud Feign** sub project under Spring Cloud offers a declarative approach for making RESTful service-to-service calls in a **synchronous** way. The declarative approach allows applications to work with **POJO** (**Plain Old Java Object**) interfaces instead of low-level HTTP client APIs. Feign internally uses reactive libraries for communication.

Circuit breaker: The circuit breaker sub project implements the circuit breaker pattern. The circuit breaker breaks the circuit when it encounters failures in the primary service by diverting traffic to another temporary fallback service. It also automatically reconnects back to the primary service when the service is back to normal. It finally provides a monitoring dashboard for monitoring the service state changes. The Spring Cloud Hystrix project and Hystrix Dashboard implement the circuit breaker and the dashboard respectively.

Global locks, leadership election and cluster state: This capability is required for cluster management and coordination when dealing with large deployments. It also offers global locks for various purposes such as sequence generation. The Spring Cloud Cluster project implements these capabilities using **Redis, ZooKeeper, and Consul**.

Security: Security capability is required for building security for cloud-native distributed systems using externalized authorization providers such as OAuth2. The Spring Cloud Security project implements this capability using customizable authorization and resource servers. It also offers SSO capabilities, which are essential when dealing with many microservices.

Big data support: The big data support capability is a capability that is required for data services and data flows in connection with big data solutions. The Spring Cloud Streams and the Spring Cloud Data Flow projects implement these capabilities. The Spring Cloud Data Flow is the re-engineered version of Spring XD.

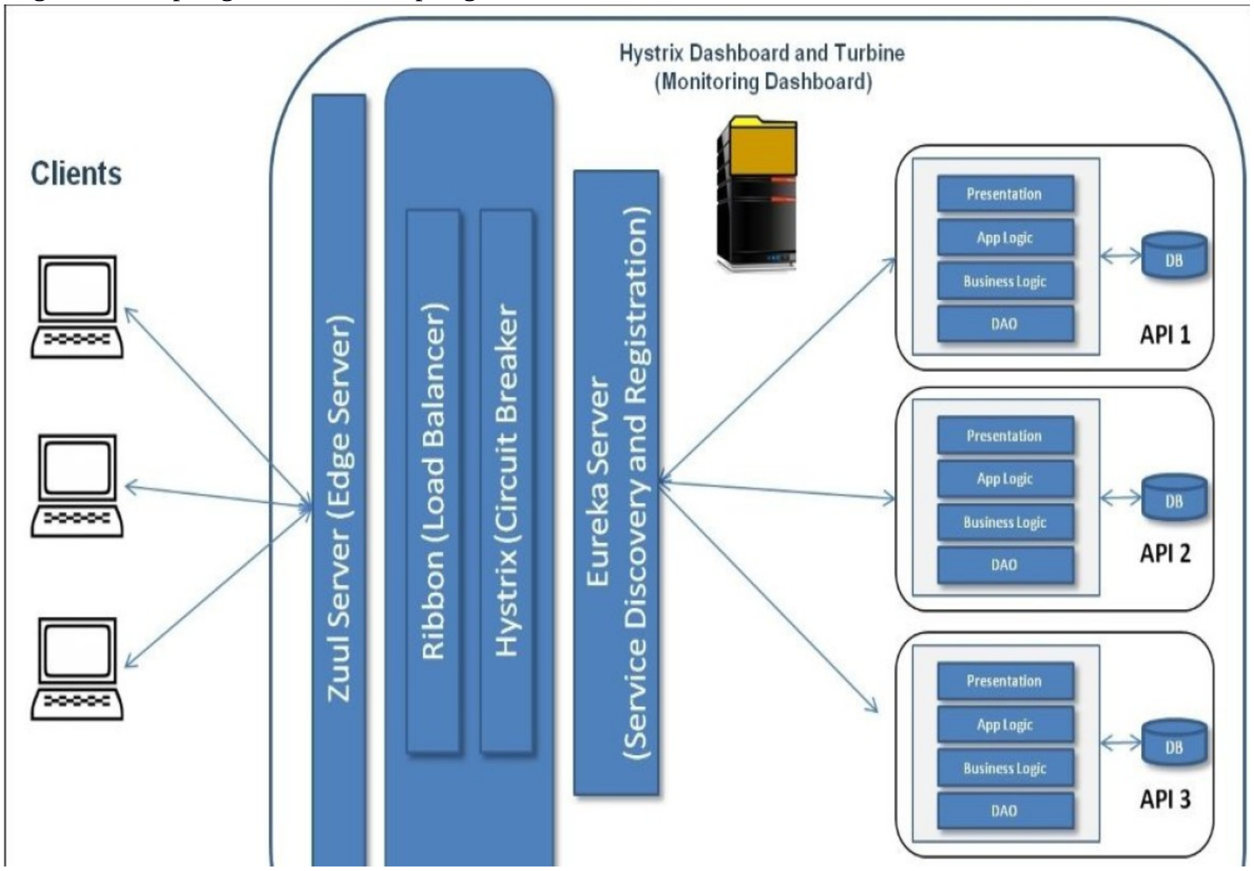
Distributed tracing: The distributed tracing capability helps to thread and correlate transitions that are spanned across multiple microservice instances. **Spring Cloud Sleuth** implements this by

providing an abstraction on top of various distributed tracing mechanisms such as **Zipkin and HTrace** with the support of a 64-bit ID.
<u>Distributed messaging:</u> Spring Cloud Stream provides declarative messaging integration on top of reliable messaging solutions such as **Kafka, Redis**, and **RabbitMQ**.
<u>Cloud support:</u> Spring Cloud also provides a set of capabilities that offers various connectors, integration mechanisms, and abstraction on top of different cloud providers such as the Cloud Foundry and AWS.

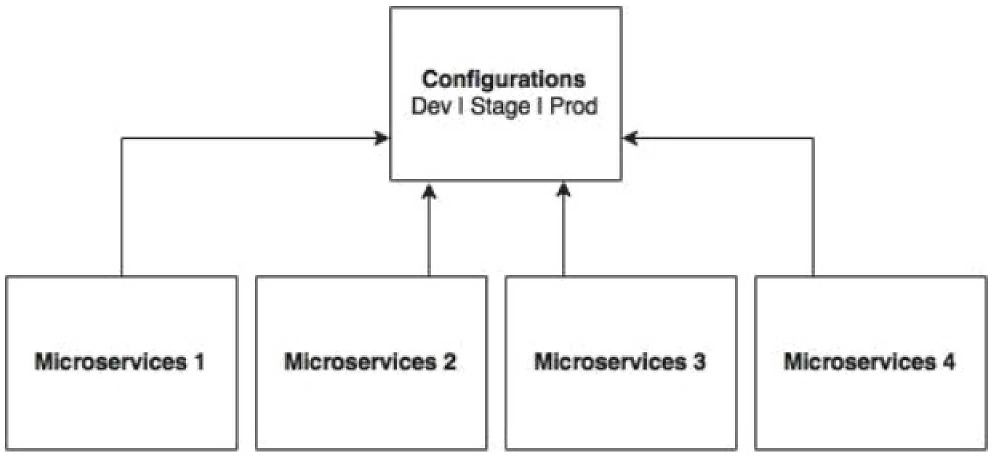
<u>Spring Cloud and Netflix OSS</u>

Many of the Spring Cloud components which are critical for microservices' deployment came from the Netflix Open Source Software (Netflix OSS) center. Netflix is one of the pioneers and early adaptors in the microservices space.
In order to manage large scale microservices, engineers at Netflix came up with a number of homegrown tools and techniques for managing their microservices. These are fundamentally crafted to fill some of the software gaps recognized in the AWS platform for managing Netflix services. Later, Netflix open-sourced these components, and made them available under the Netflix OSS platform for public use. These components are extensively used in production systems, and are battle-tested with large scale microservice deployments at Netflix.

Spring Cloud offers higher levels of abstraction for these Netflix OSS components, making it more Spring developer friendly. It also provides a declarative mechanism, well-integrated and aligned with Spring Boot and the Spring framework.

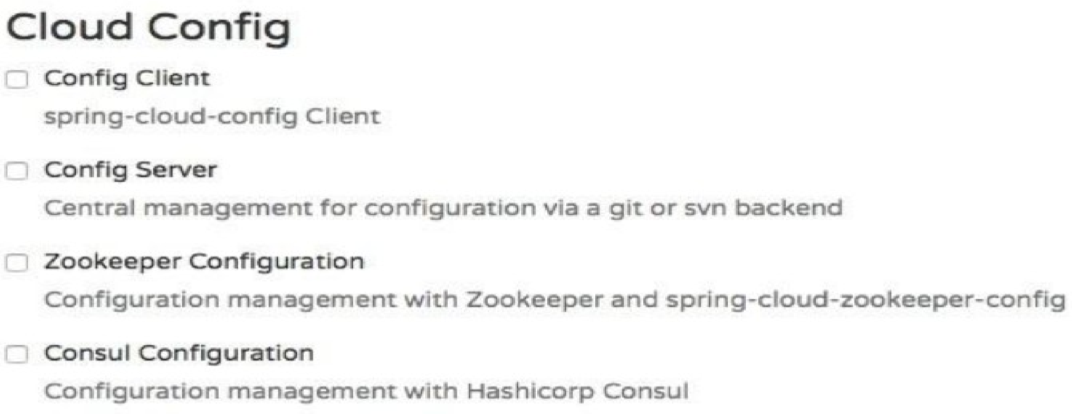


Spring Cloud Config
In Spring Boot applications, all configuration parameters were read from a property file packaged inside the project, either application.properties or application.yaml. This approach is good, since all properties are moved out of code to a property file. However, when microservices are moved from one environment to another, these properties need to undergo changes, which require an application re-build. This is violation of one of the Twelve-Factor application principles, which suggests one-time build and moving of the binaries across environments.
A better approach is to use the concept of profiles which is used for partitioning different properties for different environments. The profile-specific configuration will be named application-{profile}.properties. For example,application-development.properties represents a property file targeted for the development environment. However, the disadvantage of this approach is that the configurations are statically packaged along with the application. Any changes in the configuration properties require the application to be rebuilt.
Hence It is always recommended to externalize and centralize the configurations. **Spring Cloud Config** is an externalized configuration server with Git or SVN as the backing repository which externalizes the application configurations to SVN/GIT repository.



As shown in the preceding diagram, all microservices point to a central server to get the required configuration parameters. **The microservices then locally cache these parameters to improve performance. The Config server propagates the configuration state changes to all subscribed microservices so that the local cache's state can be updated with the latest changes**. The Config server also uses profiles to resolve values specific to an environment.

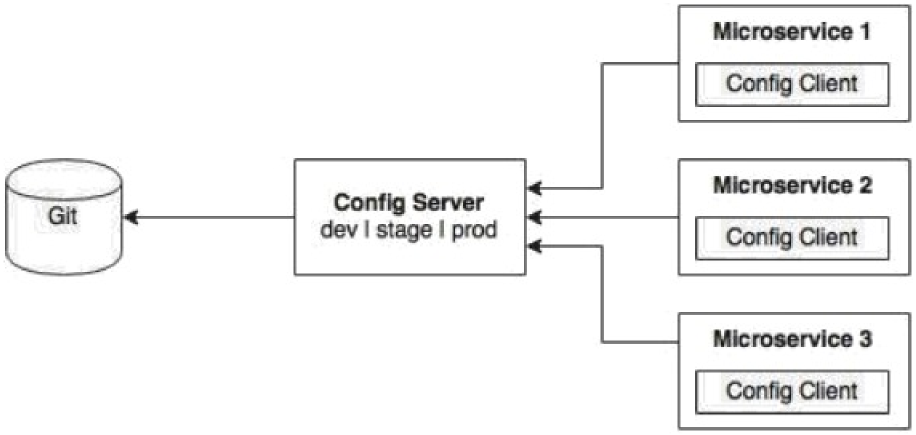
As shown in the following screenshot, there are multiple options available under the Spring Cloud project for building the configuration server. Config Server, Zookeeper Configuration, and Consul Configuration are available as options. However, we will use the Spring Config server implementation:



Cloud Config

- [ ] Config Client
  spring-cloud-config Client
- [ ] Config Server
  Central management for configuration via a git or svn backend
- [ ] Zookeeper Configuration
  Configuration management with Zookeeper and spring-cloud-zookeeper-config
- [ ] Consul Configuration
  Configuration management with Hashicorp Consul

The Spring Config server stores properties in a version-controlled repository such as Git or SVN. The Git repository can be local or remote. A highly available remote Git server is preferred for large scale distributed microservice deployments.
The Spring Cloud Config server architecture is shown in the following diagram:



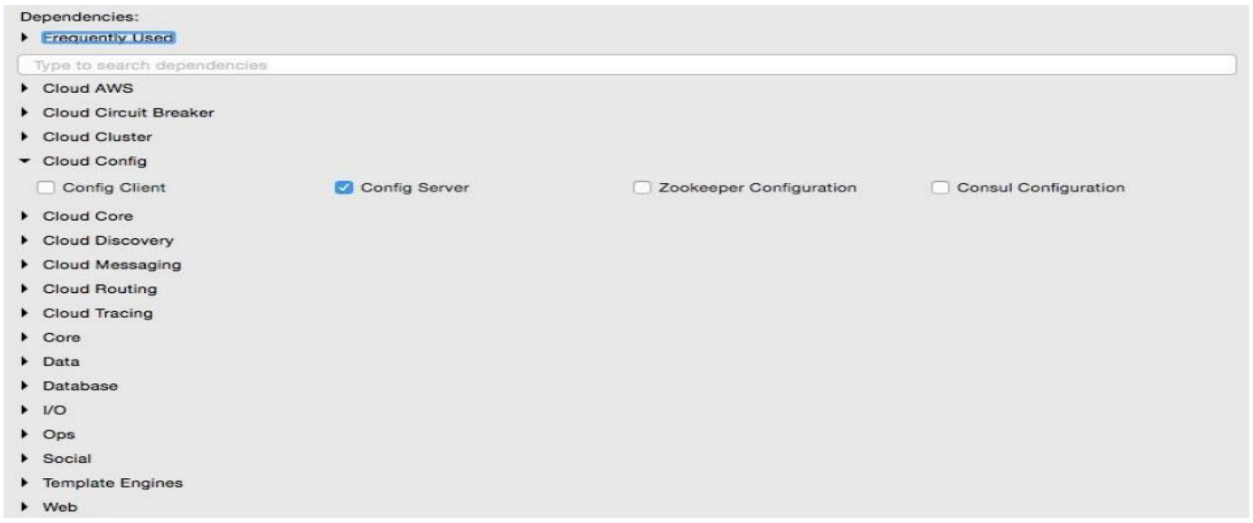
As shown in the preceding diagram, the Config client embedded in the Spring Boot MicroServices does a configuration lookup from a central configuration server using a simple declarative mechanism, and stores properties into the Spring environment. The configuration properties can be application-level configurations such as server URLs, credentials, and so on.

Unlike Spring Boot, Spring Cloud uses a bootstrap context, which is a parent context of the main application. Bootstrap context is responsible for loading configuration properties from the Config server. The bootstrap context looks for **bootstrap.yaml** or **bootstrap.properties** for loading initial configuration properties. To make this work in a Spring Boot application, rename the file **application.*** to **bootstrap.***.

Setting up the Config server

The following steps need to be followed to create a new Config server using STS:

Step-1: Create a new Spring Starter Project, and select config-server and Actuator as shown in the following diagram:


Dependencies:

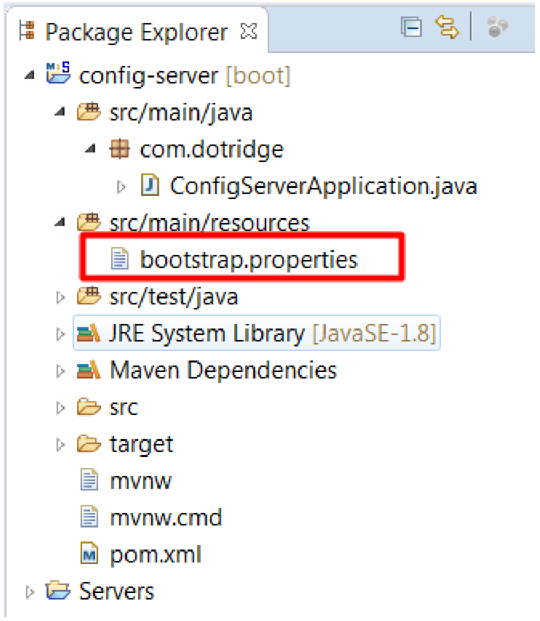
- Frequently Used
- Type to search dependencies
- Cloud AWS
- Cloud Circuit Breaker
- Cloud Cluster
- Cloud Config
  - [ ] Config Client
  - [✓] Config Server
  - [ ] Zookeeper Configuration
  - [ ] Consul Configuration
- Cloud Core
- Cloud Discovery
- Cloud Messaging
- Cloud Routing
- Cloud Tracing
- Core
- Data
- Database
- I/O
- Ops
- Social
- Template Engines
- Web

Step-2: Set up a Git repository. This can be done by pointing to a remote Git configuration repository like the one at **https://github.com/spring-cloud-samples/config-repo**. This URL is an indicative one, a Git repository used by the Spring Cloud examples. We will have to use our own Git repository instead.

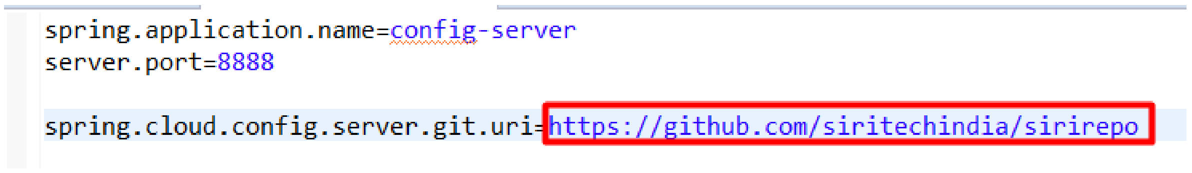
Note:

Alternately, a local filesystem-based Git repository can be used. In a real production scenario, an external Git is recommended.

Step-3: The next step is to change the configuration in the Config server to use the Git repository created in the previous step. In order to do this, rename the file application.properties to bootstrap.properties:


```
Package Explorer
config-server [boot]
  src/main/java
    com.dotridge
      ConfigServerApplication.java
  src/main/resources
    bootstrap.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
Servers
```

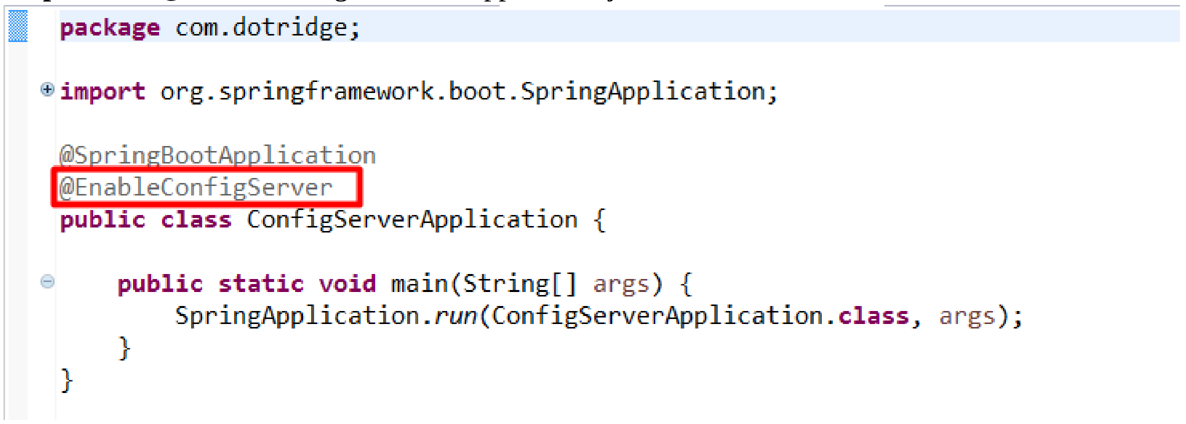
Step-4: Edit the contents of the new bootstrap.properties file to match the following:



```
spring.application.name=config-server
server.port=8888

spring.cloud.config.server.git.uri=https://github.com/siritechindia/sirirepo
```

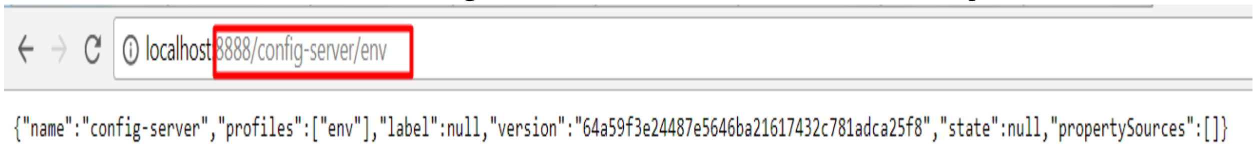
Port **8888** is the default port for the Config server. Even without configuring server.port, the Config server should bind to 8888.

Step-5: Add @EnableConfigServer in Application.java:



```
package com.dotridge;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {

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

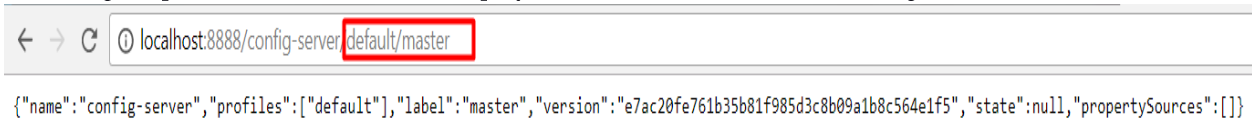
Step-6: Run the Config server by right-clicking on the project, and running it as a Spring Boot app. Visit http://localhost:8888/config-server/env to see whether the server is running. If everything is fine, this will list all environment configurations. Note that /**env** is an actuator endpoint.



```
localhost:8888/config-server/env
```

```
{"name":"config-server","profiles":["env"],"label":null,"version":"64a59f3e24487e5646ba21617432c781adca25f8","state":null,"propertySources":[]}
```

Step-7:.Check http://localhost:8888/config-server/default/master to see the properties specific to properties file. The browser will display the properties configured in properties files which were there in git repo. The browser should display contents similar to the following:



```
localhost:8888/config-server/default/master
```

```
{"name":"config-server","profiles":["default"],"label":"master","version":"e7ac20fe761b35b81f985d3c8b09a1b8c564e1f5","state":null,"propertySources":[]}
```

Understanding the Config server URL: http://localhost:8888/config-server/default/master

The first element(after **http://localhost:8888/**) in the URL is the application name. In the given example, the application name should be **config-server**. The application name is a logical name given to the application, using the **spring.application.name** property in bootstrap.properties of the Spring Boot application. Each application must have a unique name. The Config server will use the name to resolve and pick up appropriate properties from the Config server repository. The application name is also sometimes referred to as service ID.

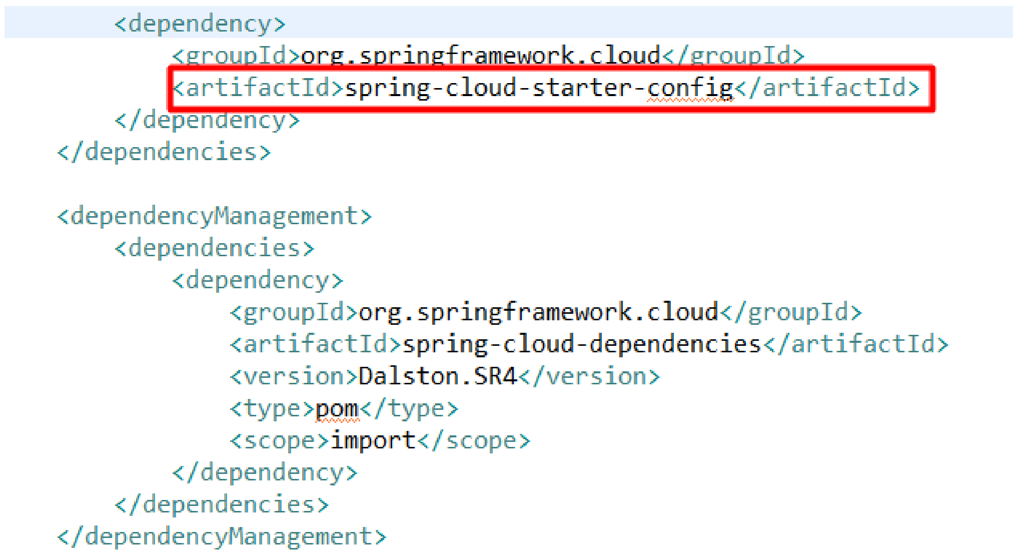
If there is an application with the name **patient-service**, then there should be a **patient-service.properties** in the configuration repository to store all the properties related to that application.

The second part (**default**)of the URL represents the profile. There can be more than one profile configured within the repository for an application. The profiles can be used in various scenarios. The two common scenarios are segregating different environments such as Dev, Test, Stage, Prod, and the like, or segregating server configurations such as Primary, Secondary, and so on. The first one represents different environments of an application, whereas the second one represents different servers where an application is deployed.

The profile names are logical names that will be used for matching the file name in the repository. The default profile is named default. To configure properties for different environments, we have to configure different files as given in the following example. In this example, the first file is for the development environment whereas the second is for the production environment:
application-development.properties
application-production.properties
These are accessible using the following URLs respectively:
http://localhost:8888/application/development,http://localhost:8888/application/production

The last part of the URL is the label, and is named master by default. The label is an optional Git label that can be used, if required. In short, the URL is based on the following pattern:
http://localhost:8888/{name}/{profile}/{label}.
The configuration can also be accessed by ignoring the profile. In the preceding example, all the following three URLs point to the same configuration:
http://localhost:8888/application/default
http://localhost:8888/application/master
http://localhost:8888/application/default/master

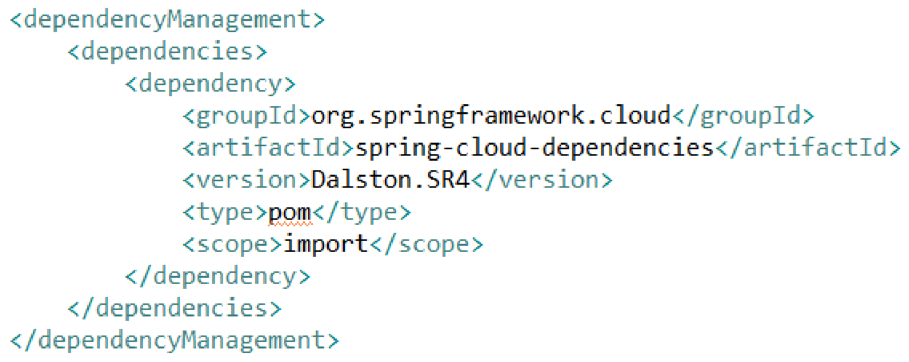
Accessing the Config Server from clients
Lets Consider we have the **patient-serice** micro service which will be now modified to use the Config Server. This service now should have spring config client configurations to communicate with the spring config server and hence forth we called it as spring config client. Follow these steps to use the Config server instead of reading properties from the application.properties file:
Step-1: Add the Spring Cloud Config dependency and the actuator (if the actuator is not already in place) to the pom.xml file. The actuator is mandatory for refreshing the configuration properties:



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

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Dalston.SR4</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

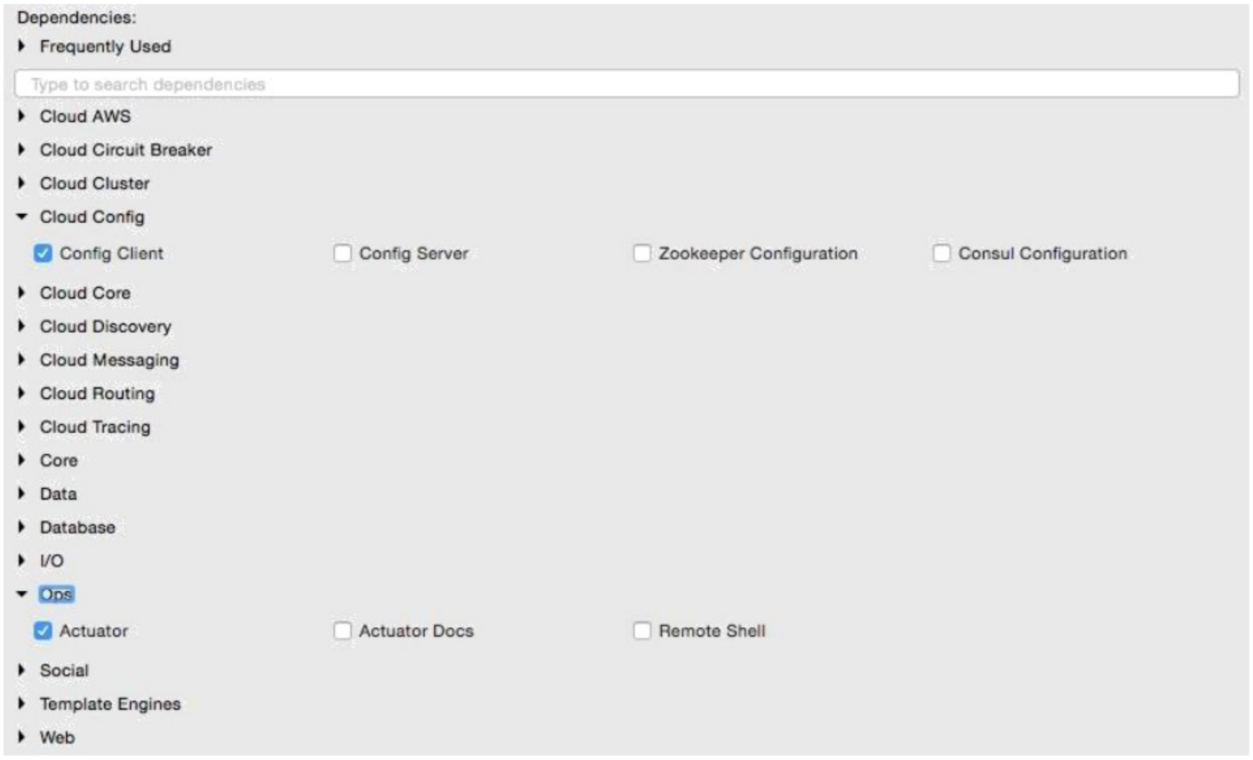
Note:

1) Since we are modifying the Spring Boot patient-service microservice that is already created, we have added the following to include the Spring Cloud dependencies. This is not required if the project is created from scratch

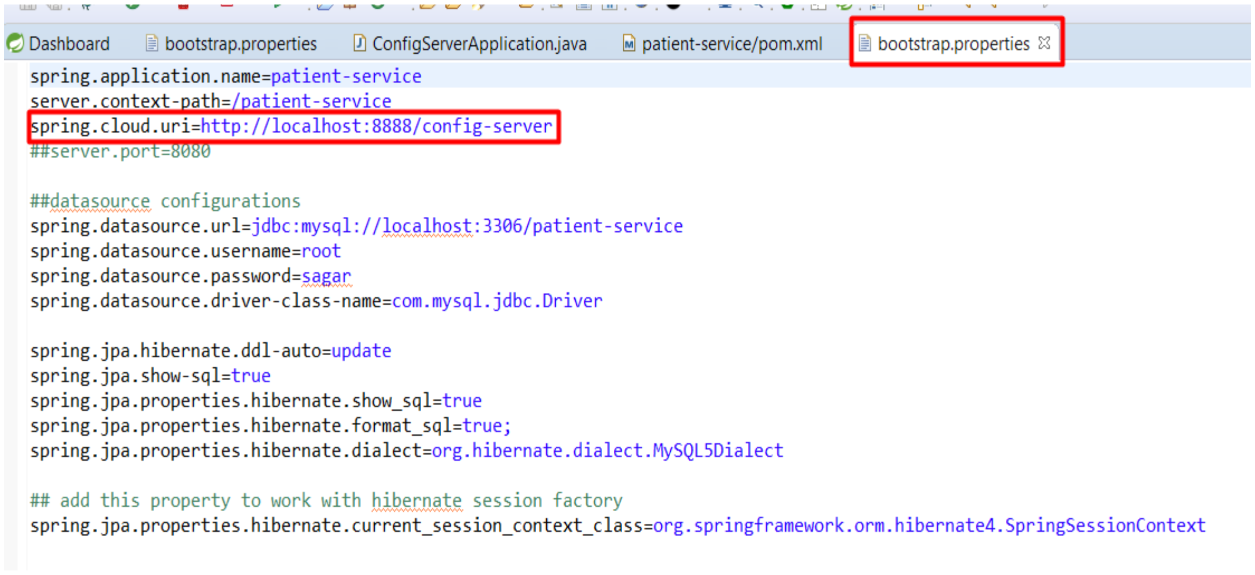


```
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Dalston.SR4</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

2) If the application is built from the ground up, select the libraries as shown in the following screenshot:


Dependencies:
- Frequently Used
- Cloud AWS
- Cloud Circuit Breaker
- Cloud Cluster
- Cloud Config
  - [✓] Config Client
  - [ ] Config Server
  - [ ] Zookeeper Configuration
  - [ ] Consul Configuration
- Cloud Core
- Cloud Discovery
- Cloud Messaging
- Cloud Routing
- Cloud Tracing
- Core
- Data
- Database
- I/O
- Ops
  - [✓] Actuator
  - [ ] Actuator Docs
  - [ ] Remote Shell
- Social
- Template Engines
- Web

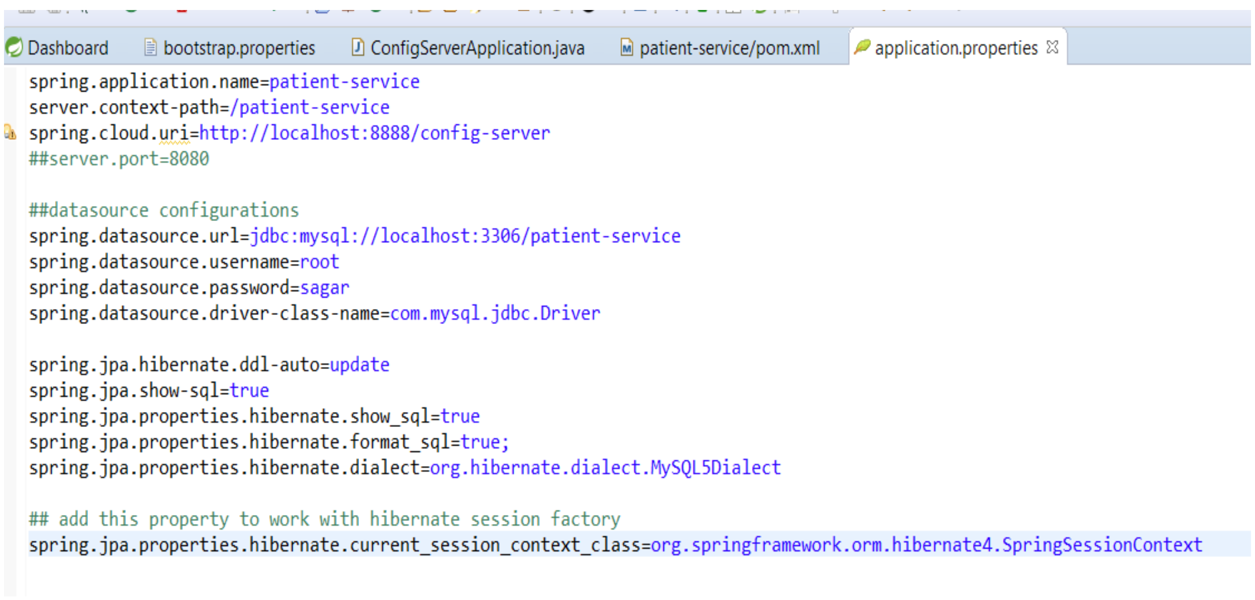
Step-2: Rename application.properties to bootstrap.properties, and add an application name and a configuration server URL. The configuration server URL is not mandatory if the Config server is running on the default port (8888) on the local host. The new bootstrap.properties file will look as follows:



```
spring.application.name=patient-service
server.context-path=/patient-service
spring.cloud.uri=http://localhost:8888/config-server
##server.port=8080

##datasource configurations
spring.datasource.url=jdbc:mysql://localhost:3306/patient-service
spring.datasource.username=root
spring.datasource.password=sagar
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true;
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

## add this property to work with hibernate session factory
spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext
```

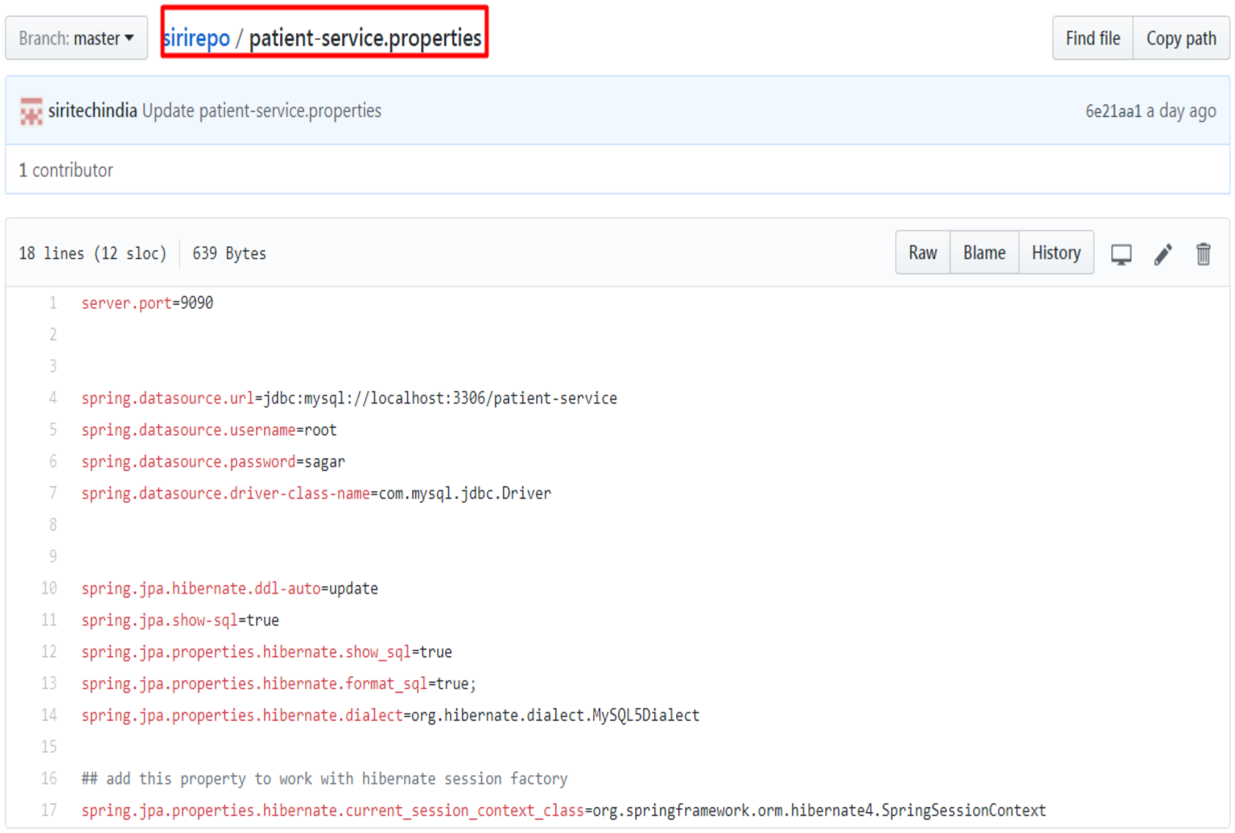
Note:

renaming application.properties file to bootstrap.properties file is not mandatory in config client micro services I.e. patient-service micro service. It will work even if we have application.properties file as shown below:



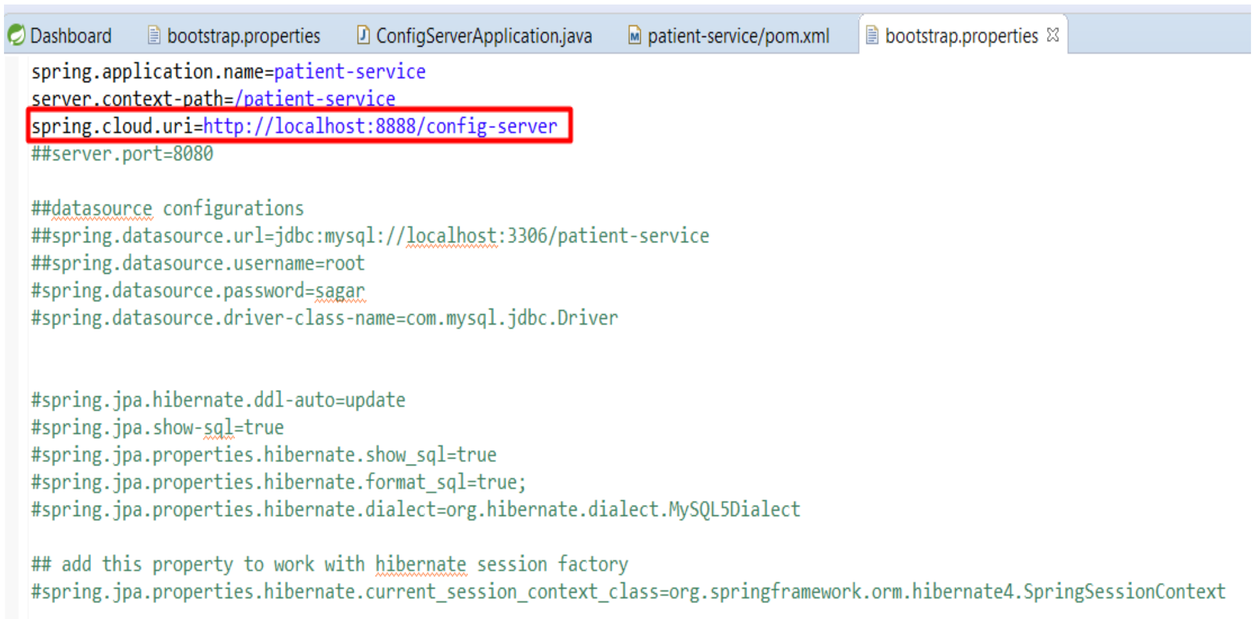
```
spring.application.name=patient-service
server.context-path=/patient-service
spring.cloud.uri=http://localhost:8888/config-server
##server.port=8080

##datasource configurations
spring.datasource.url=jdbc:mysql://localhost:3306/patient-service
spring.datasource.username=root
spring.datasource.password=sagar
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true;
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

## add this property to work with hibernate session factory
spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext
```

<u>Step-3:</u> Create a new configuration file for **patient-service.properties** in the Git repository as shown below:


Branch: master
sirirepo / patient-service.properties
Find file
Copy path
siritechindia Update patient-service.properties
6e21aa1 a day ago
1 contributor
18 lines (12 sloc) | 639 Bytes
Raw
Blame
History

```
server.port=9090


spring.datasource.url=jdbc:mysql://localhost:3306/patient-service
spring.datasource.username=root
spring.datasource.password=sagar
spring.datasource.driver-class-name=com.mysql.jdbc.Driver


spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true;
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

## add this property to work with hibernate session factory
spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext
```

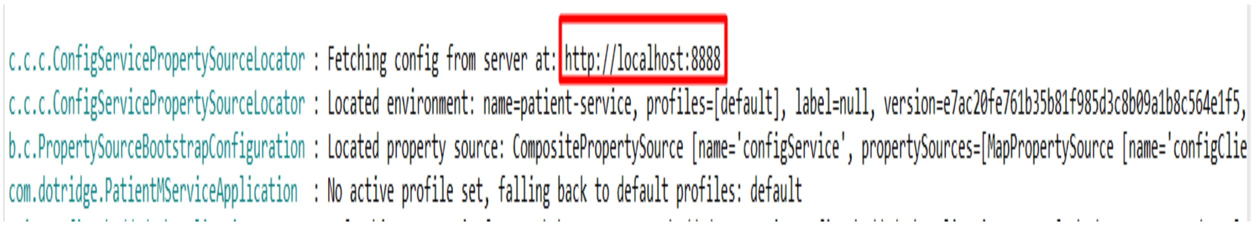
Note that **patient-service** is the service ID given to the patient microservice in the bootstrap.properties/application.properties file using the property **spring.application.name**. Move service-specific properties i.e. datasource etc properties from bootstrap.properties of patient-service microservice to the new patient-service.properties file created in git. Now after moving the datasource properties from bootstrap/application.properties file, it should look like as below:



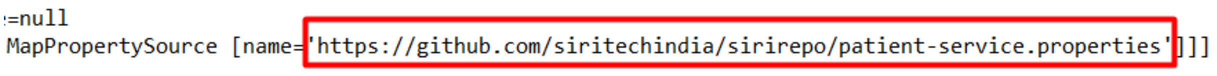
```
spring.application.name=patient-service
server.context-path=/patient-service
spring.cloud.uri=http://localhost:8888/config-server
##server.port=8080

##datasource configurations
##spring.datasource.url=jdbc:mysql://localhost:3306/patient-service
##spring.datasource.username=root
#spring.datasource.password=sagar
#spring.datasource.driver-class-name=com.mysql.jdbc.Driver


#spring.jpa.hibernate.ddl-auto=update
#spring.jpa.show-sql=true
#spring.jpa.properties.hibernate.show_sql=true
#spring.jpa.properties.hibernate.format_sql=true;
#spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

## add this property to work with hibernate session factory
#spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext
```

Step-4: Start the Config server. Then start the patient-service

Now when we run the patient-service, we should see the below log on console.



```
c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at: http://localhost:8888
c.c.c.ConfigServicePropertySourceLocator : Located environment: name=patient-service, profiles=[default], label=null, version=e7ac20fe761b35b81f985d3c8b09a1b8c564e1f5,
b.c.PropertySourceBootstrapConfiguration : Located property source: CompositePropertySource [name='configService', propertySources=[MapPropertySource [name='configClie
com.dotridge.PatientMServiceApplication  : No active profile set, falling back to default profiles: default
```



```
=null
MapPropertySource [name='https://github.com/siritechindia/sirirepo/patient-service.properties']]]
```

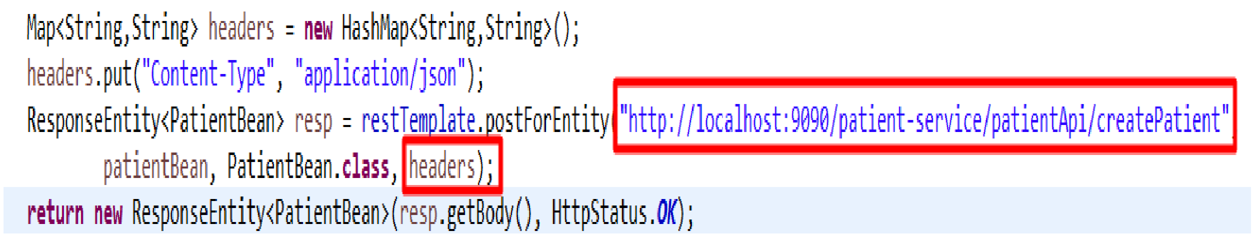
This is clear that,on bootstrapping of patient-service microservice, it is connecting to config server I.e. running on port 8888. Now Config Server connected to git, look-up for the properties file with the micro service name I.e patient-service.properties file, load the configurations from there and making available them to the Config Server client I.e. patient-service microservice. Hence patient-service micro service is connected to the database and will create the database tables according to the meta data descriptions in JPA Entities I.e. domains.

MicroService To MicroService communication

Feign as a declarative REST client

RestTemplate is used for making the synchronous call. When using RestTemplate, the URL parameter is constructed programmatically i.e we have to use the url which typically contains the host name,port context path,resource mapping url like "**http://localhost:9090/patient-**

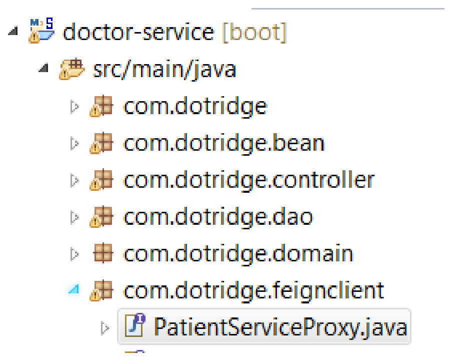
service/patientApi/createPatient", and data is sent across to the other service. In more complex scenarios, like posting data to another web service we have to get to the details of the HTTP APIs provided by RestTemplate to mentioned the content-type as header parameter and to pass the body with the data at a much lower level. See the below sample:



```
Map<String,String> headers = new HashMap<String,String>();
headers.put("Content-Type", "application/json");
ResponseEntity<PatientBean> resp = restTemplate.postForEntity("http://localhost:9090/patient-service/patientApi/createPatient",
        patientBean, PatientBean.class, headers);
return new ResponseEntity<PatientBean>(resp.getBody(), HttpStatus.OK);
```

Feign is a Spring Cloud Netflix library for providing a higher level of abstraction over REST-based service calls. Spring Cloud Feign works on a declarative principle. When using Feign, we write declarative REST service interfaces at the client, and use those interfaces to program the client.

For example: doctor has to register the patient. here **doctor-service** is a micro service should call another micro service **patient-service** and should post the patient data to create the patient.Here **doctor-service** is now acts as a client for **patient-service**, hence we will write a declarative REST service interface I.e.**PatientServiceProxy** at the doctor-service as shown below:



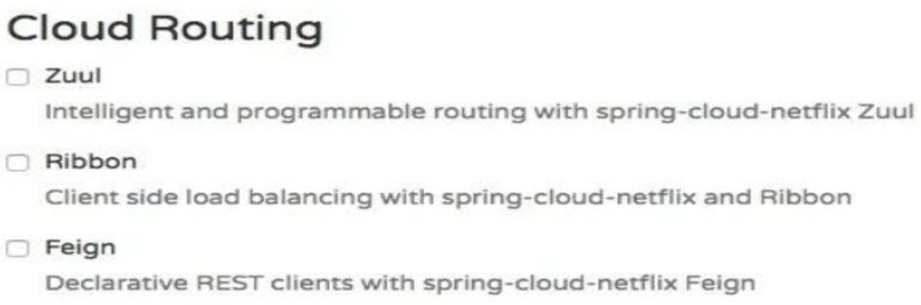
```
doctor-service [boot]
  src/main/java
    com.dotridge
    com.dotridge.bean
    com.dotridge.controller
    com.dotridge.dao
    com.dotridge.domain
    com.dotridge.feignclient
      PatientServiceProxy.java
```

we no need not worry about the implementation of this interface. This will be dynamically provisioned by Spring at runtime. With this declarative approach, we need not get into the details of the HTTP level APIs provided by RestTemplate.

Step-1: In order to use Feign, first we need to change the pom.xml file to include the Feign dependency as follows:

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

For a new Spring Starter project, **Feign** can be selected from the starter library selection screen, or from http://start.spring.io/. This is available under **Cloud Routing** as shown in the following screenshot:

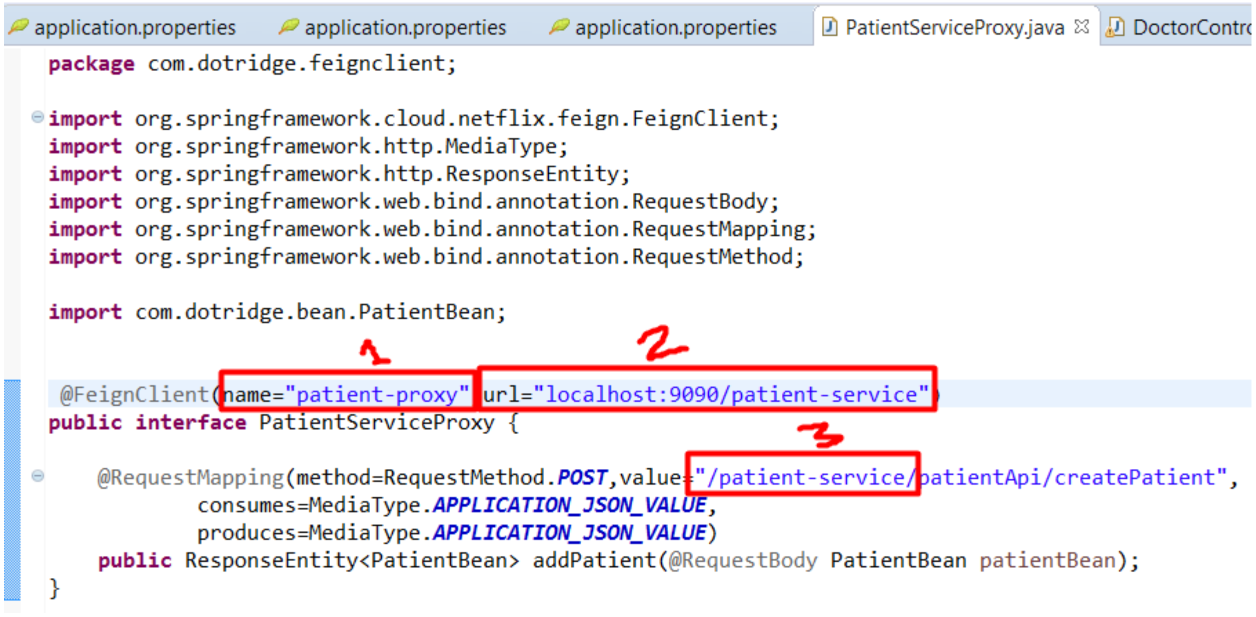

Cloud Routing

Zuul
Intelligent and programmable routing with spring-cloud-netflix Zuul

Ribbon
Client side load balancing with spring-cloud-netflix and Ribbon

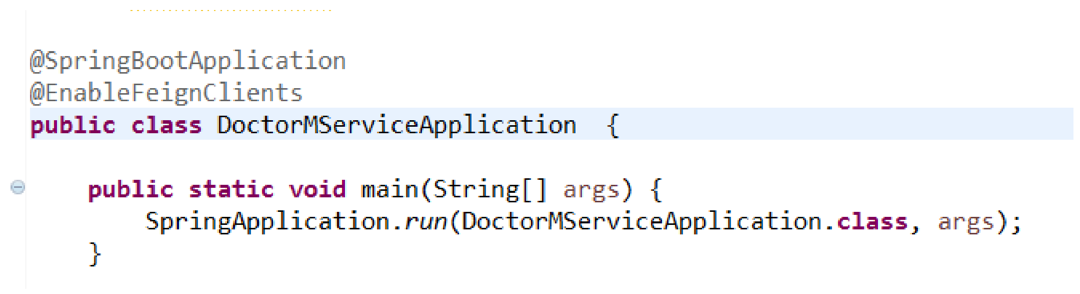
Feign
Declarative REST clients with spring-cloud-netflix Feign

<u>Step-2:</u> Create a PatientServiceProxy interface as below. This should acts as a Proxy interface of the actual **patient-service** micro service.



```
package com.dotridge.feignclient;

import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.dotridge.bean.PatientBean;

@FeignClient(name="patient-proxy",url="localhost:9090/patient-service")
public interface PatientServiceProxy {

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

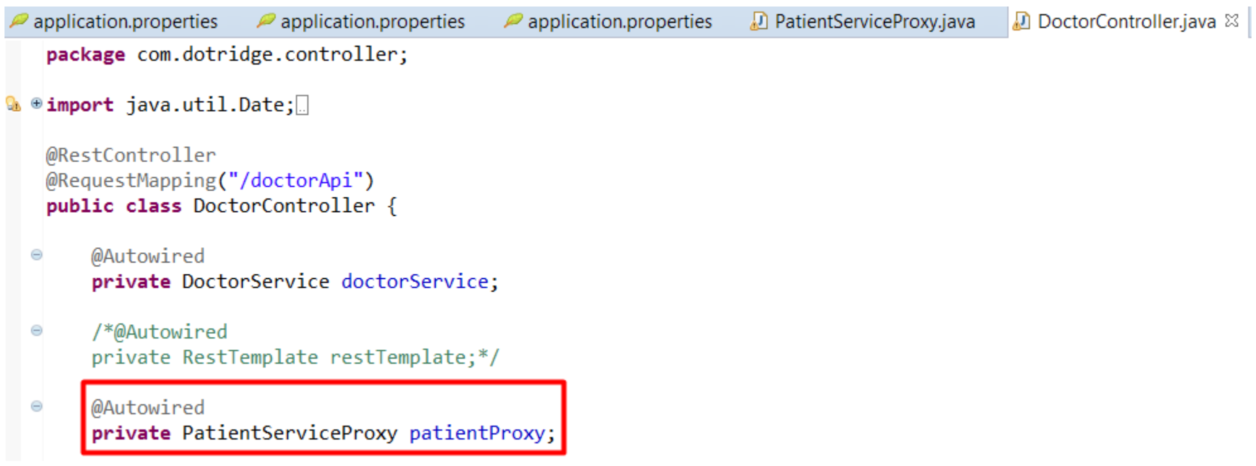
The **PatientServiceProxy** interface has a **@FeignClient** annotation. This annotation tells Spring to create a REST client based on the interface provided. The value could be a service ID or a logical name. The url indicates the actual URL where the target service is running. Either name or value is mandatory. In this case, since we have url, the name attribute is irrelevant. Use this service proxy to call the patient-service micro service. In the doctor-service microservice, we have to tell Spring that Feign clients exist in the Spring Boot application, which are to be scanned and discovered. This will be done by adding **@EnableFeignClients** at the class level of DoctorMServiceApplication. Optionally, we can also give the package names to scan.



```java
@SpringBootApplication
@EnableFeignClients
public class DoctorMServiceApplication  {

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

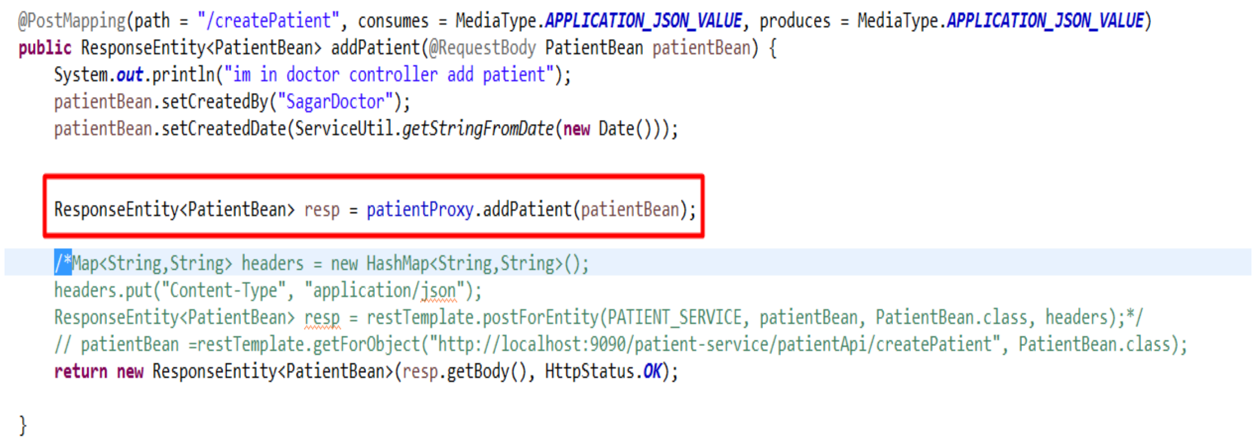
Step-3: Call patient-service micro service from doctor-micro service:

First autowired the **PatientServiceProxy** in Doctor Controller as shown below:



```java
package com.dotridge.controller;

import java.util.Date;

@RestController
@RequestMapping("/doctorApi")
public class DoctorController {

    @Autowired
    private DoctorService doctorService;

    /*@Autowired
    private RestTemplate restTemplate;*/

    @Autowired
    private PatientServiceProxy patientProxy;
```

Call patient-service micro service from doctor-micro service as shown below:



```java
@PostMapping(path = "/createPatient", consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<PatientBean> addPatient(@RequestBody PatientBean patientBean) {
    System.out.println("im in doctor controller add patient");
    patientBean.setCreatedBy("SagarDoctor");
    patientBean.setCreatedDate(ServiceUtil.getStringFromDate(new Date()));

    ResponseEntity<PatientBean> resp = patientProxy.addPatient(patientBean);

    /*Map<String,String> headers = new HashMap<String,String>();
    headers.put("Content-Type", "application/json");
    ResponseEntity<PatientBean> resp = restTemplate.postForEntity(PATIENT_SERVICE, patientBean, PatientBean.class, headers);*/
    // patientBean =restTemplate.getForObject("http://localhost:9090/patient-service/patientApi/createPatient", PatientBean.class);
    return new ResponseEntity<PatientBean>(resp.getBody(), HttpStatus.OK);

}
```

rerun the **doctor-service** microservice to see the effect.The URL of the patient service in the **PatientServiceProxy** interface is hardcoded as **url="localhost:9090/patient-service"**. This is because **@FeignClient** acts as a load balancer, if we forgot to hard code this url, it will try to connect to the load balancer to get the **patient-service** service details, as we didn't configure the load balancer so far, lets hard code this for a time being till we use client side load balancer i.e. Ribbon.

Note:

Value of the name attribute inside the **@FeignClient** annotation I.e.**patient-proxy** is no need to equals to our patient-service micro service name I.e. patient-service.

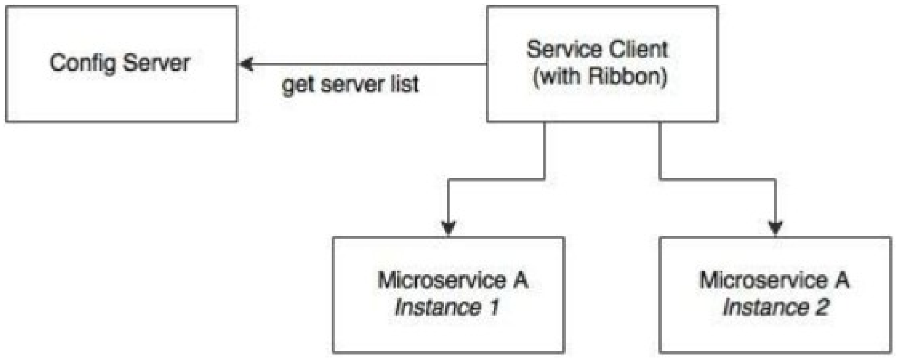
Ribbon For Load Balancing

So far, we were always running with a single instance of the microservice.The URL is hard coded both in client as well as in the service-to-service calls.Since there could be more than one service instance in the real world, this is not a recommended approach.

If there are multiple instances, then ideally, we should use a load balancer or a local DNS server to abstract the actual instance locations, and configure an alias name or the load balancer address in the clients.

The load balancer then receives the alias name, and resolves it with one of the available instances. With this approach, we can configure as many instances behind a load balancer. It also helps us to handle server failures transparent to the client.

This is achievable with **Spring Cloud Netflix Ribbon**. **Ribbon is a client-side load balancer which can do round-robin load balancing across a set of servers**. There could be other load balancing algorithms possible with the Ribbon library. Spring Cloud offers a declarative way to configure and use the Ribbon client.

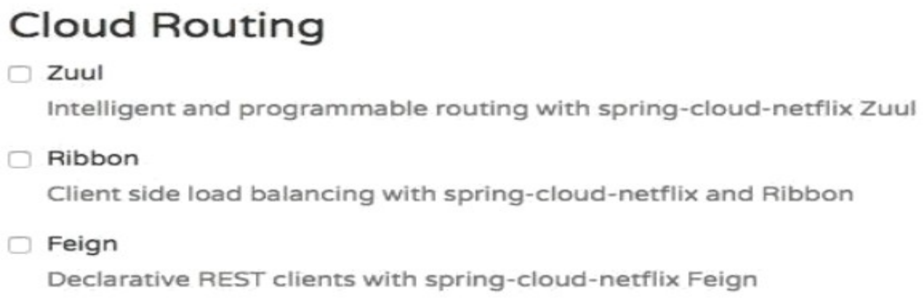

Config Server
get server list
Service Client (with Ribbon)
Microservice A Instance 1
Microservice A Instance 2

As shown in the preceding diagram, the Ribbon client looks for the Config server to get the list of available microservice instances, and, by default, applies a round-robin load balancing algorithm.

Step-1: In order to use the Ribbon client, we will have to add the following dependency to the pom.xml file:

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

In case of development from ground up, this can be selected from the Spring Starter libraries, or from **http://start.spring.io/**. Ribbon is available under Cloud Routing:

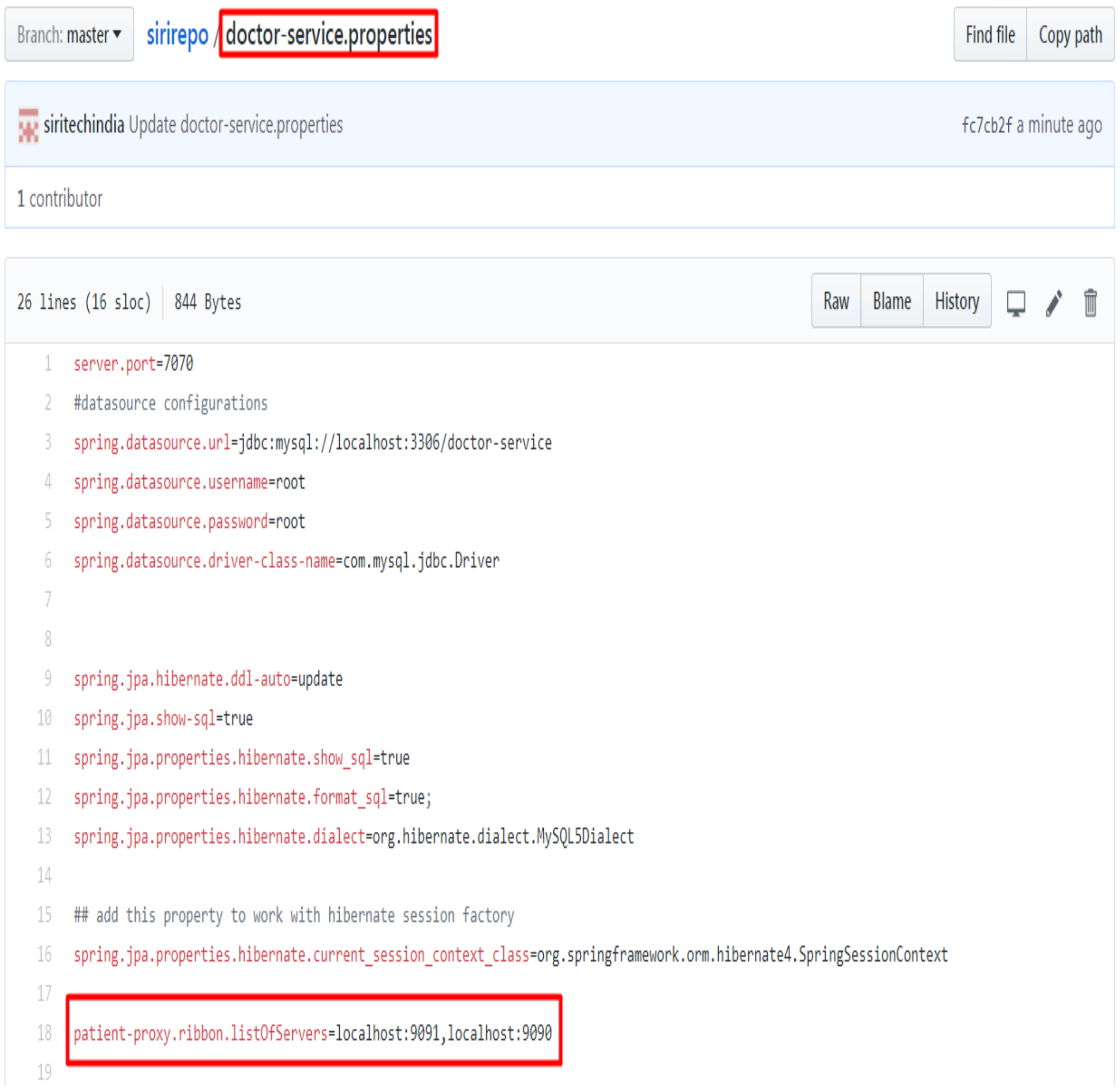

Cloud Routing

☐ Zuul
Intelligent and programmable routing with spring-cloud-netflix Zuul

☐ Ribbon
Client side load balancing with spring-cloud-netflix and Ribbon

☐ Feign
Declarative REST clients with spring-cloud-netflix Feign

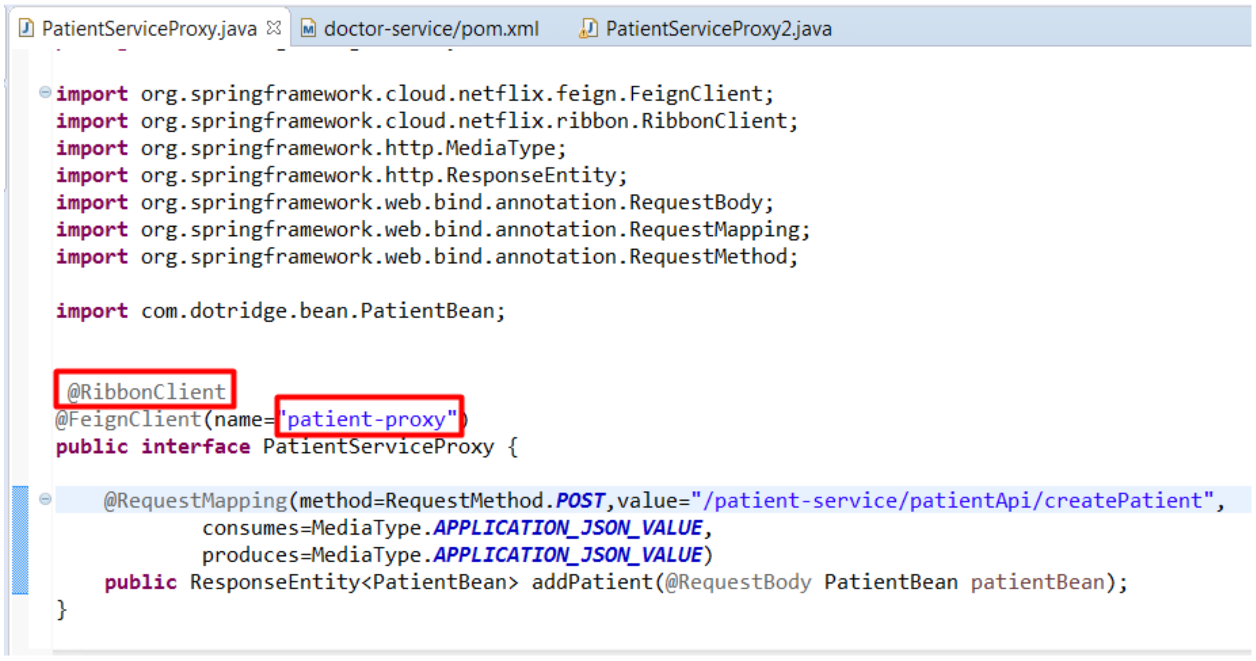
Step-2: Update the doctor-service microservice configuration file, doctor-service.properties file in git repository(as we are using config server),to include a new property to keep the list of the patient-service MicroServices as:



Branch: master
sirirepo / doctor-service.properties
Find file
Copy path
siritechindia Update doctor-service.properties
fc7cb2f a minute ago
1 contributor
26 lines (16 sloc) 844 Bytes
Raw
Blame
History

```
server.port=7070
#datasource configurations
spring.datasource.url=jdbc:mysql://localhost:3306/doctor-service
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver


spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true;
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

## add this property to work with hibernate session factory
spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext

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

```

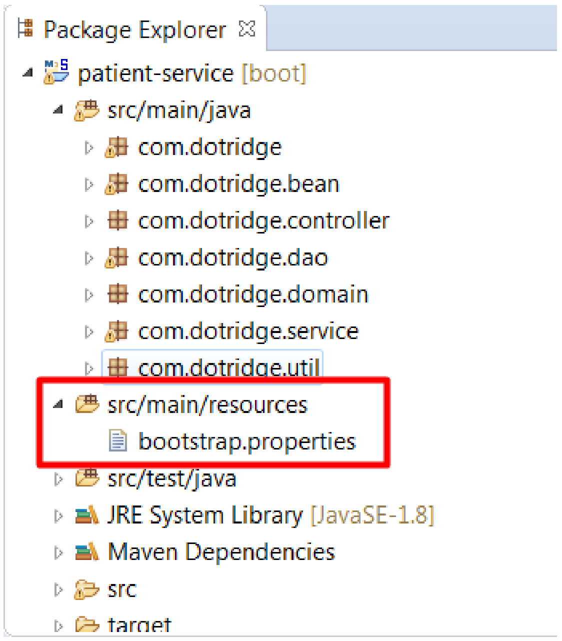
Going back and editing the PatientServiceProxy class to use the Ribbon client as below:



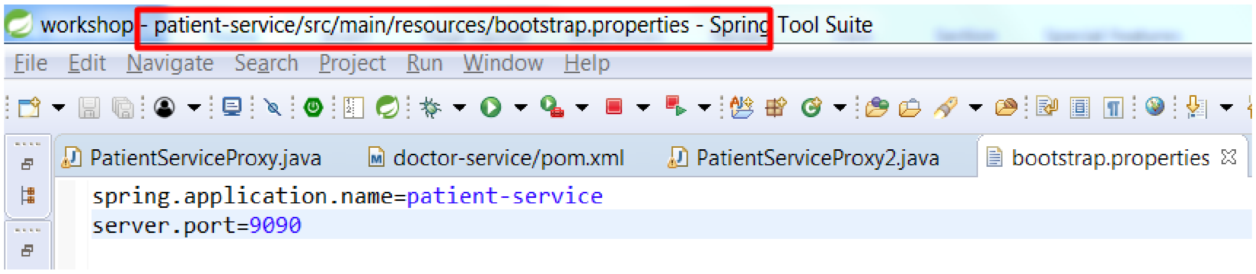
```java
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.cloud.netflix.ribbon.RibbonClient;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.dotridge.bean.PatientBean;


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

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

Here if we could observe, we have added **@RibbonClient** annotation to enable client side load balancing used by @FeignClient. In @FeignClient we have taken its name as **"patient-proxy"**. Now Make sure to add the **listOfServers** property with this name I.e.**<name of the feign client>.ribbon.lostOfServers**. Hence In our case it should be like **patient-proxy.ribbon.listOfServers=localhost:9091,localhost:9090**

<u>**Step-3:**</u> To See the effect of load balancing lets make the two instances of **patient-service** micro service. Patient-service micro service is as:

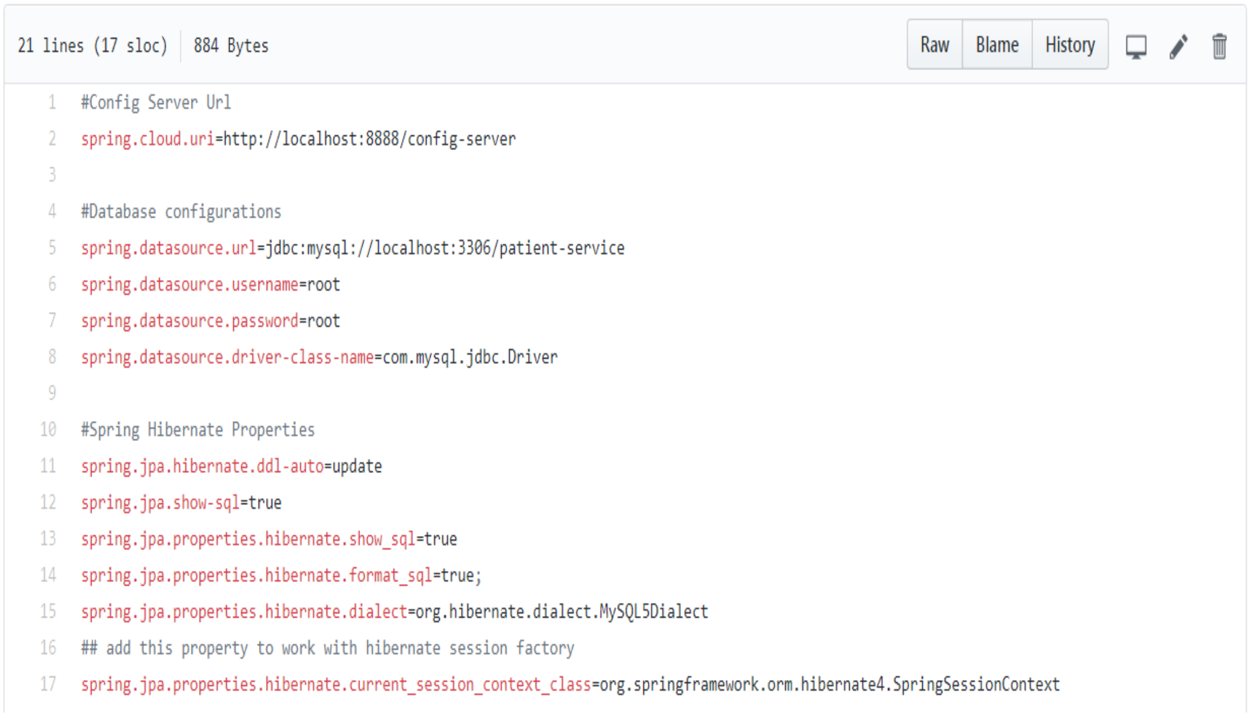


The bootstrap.properties file of patient-service micro service is as below:


workshop - patient-service/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

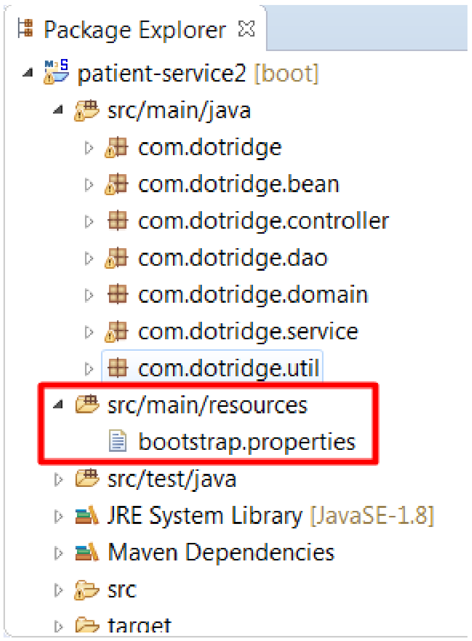
```
spring.application.name=patient-service
server.port=9090
```

As patient-service connecting to config-server to fetch its configuration properties from git, it should have a properties file with the name **patient-server.properties(which is <spring.application.name>.properties)** in git repository as shown below:

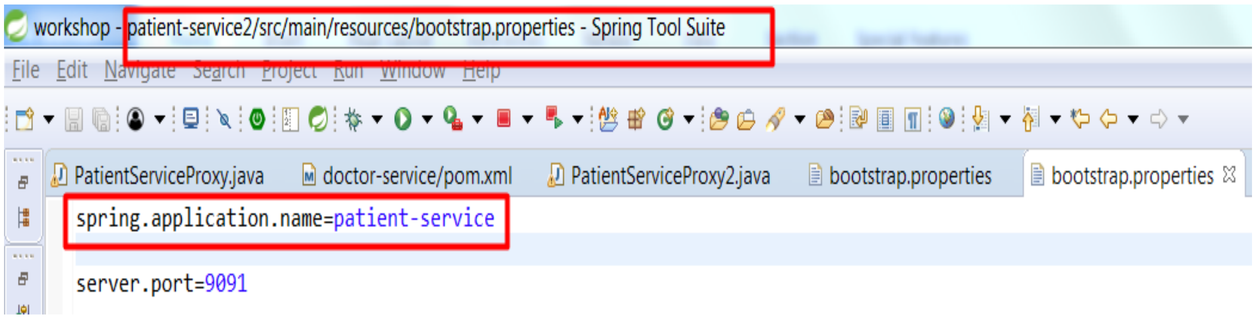


```
#Config Server Url
spring.cloud.uri=http://localhost:8888/config-server

#Database configurations
spring.datasource.url=jdbc:mysql://localhost:3306/patient-service
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

#Spring Hibernate Properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true;
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
## add this property to work with hibernate session factory
spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext
```

To make another instance of patient-service, copy and paste the same patient-service project in Spring Too Suit, it should be as below:

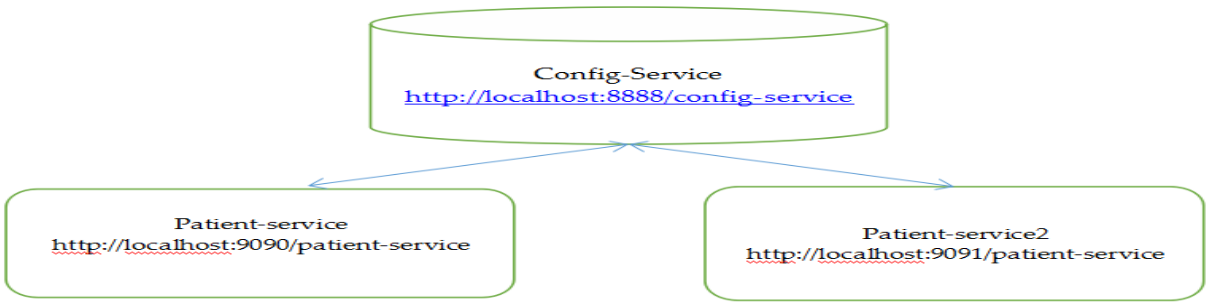


This Service is completely identical to the patient-service except its port number. It should run on different port than patient-service,except this all the properties of the patient-service should also applies to this including spring.application.name. See the below screen shot.


workshop - patient-service2/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=patient-service

server.port=9091
```

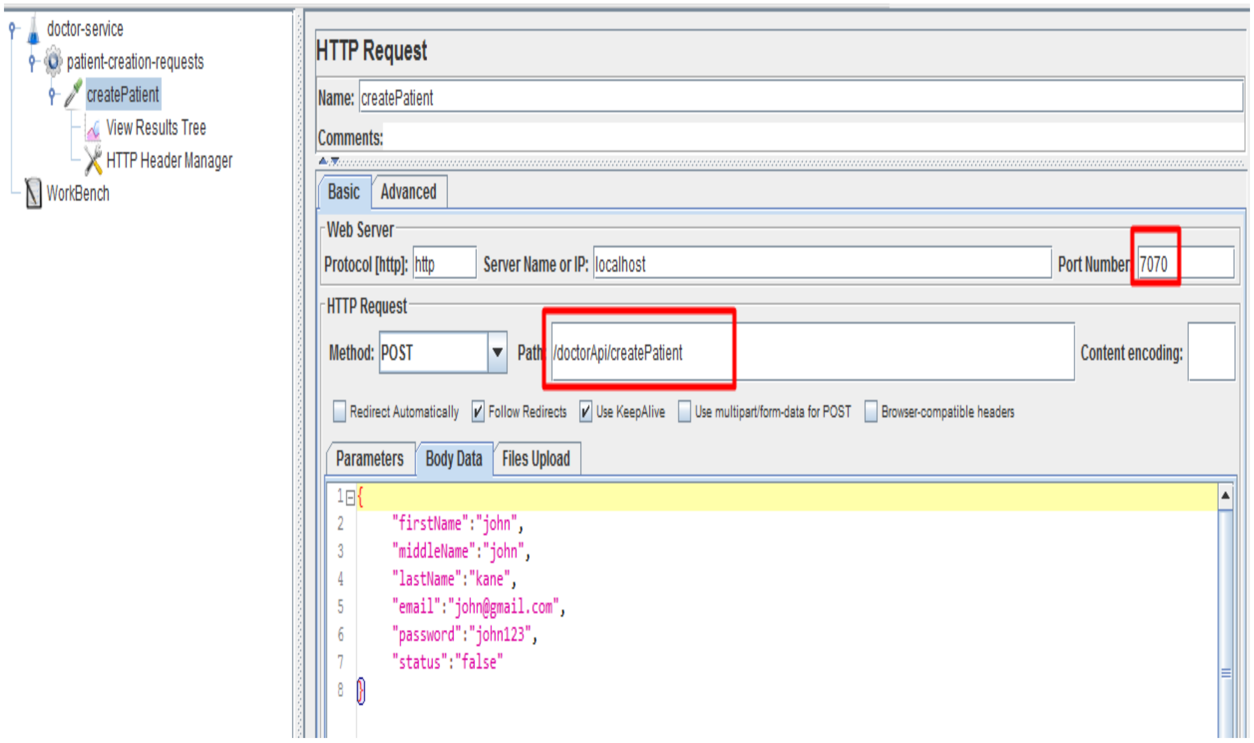
As it is also have the same name I.e. **sping.application.name=patient-service**, this will pull the same configuration parameters as patient-service have from the **config-service I.e.git**. Hence we have now two instance of patient-service one is running on port 9090 and other one is on 9091.


Config-Service
http://localhost:8888/config-service
Patient-service
http://localhost:9090/patient-service
Patient-service2
http://localhost:9091/patient-service

Step-3: Run the doctor-service microservice,When the it is bootstrapped,we see the following trace, which says there are two servers enlisted: **DynamicServerListLoadBalancer:{NFLoadBalancer:name=patient-proxy,currentlist of Servers=[localhost:9090, localhost:9091],Load balancer stats=Zone stats: {unknown=[Zone:unknown; Instance count:2; Active connections count: 0; Circuit breaker tripped count: 0; Active connections per server: 0.0;] }**

When we are creating patients from doctor-service micro service, we could see the log as below:

Request:

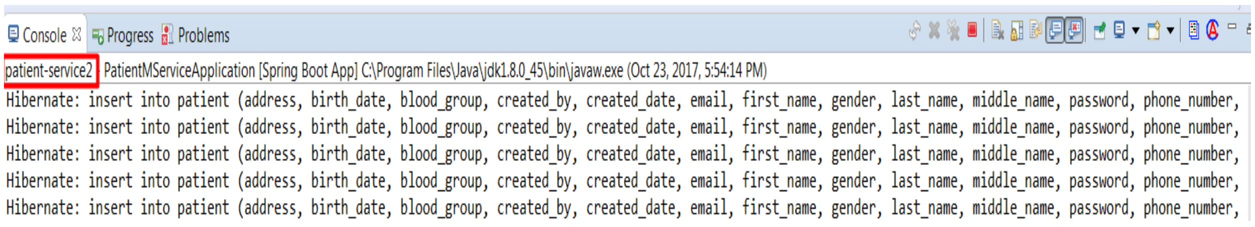


Name: createPatient
Comments:
Protocol [http]: http   Server Name or IP: localhost   Port Number: 7070
Method: POST   Path: /doctorApi/createPatient   Content encoding:
Redirect Automatically   [✓ Follow Redirects]   [✓ Use KeepAlive]   Use multipart/form-data for POST   Browser-compatible headers

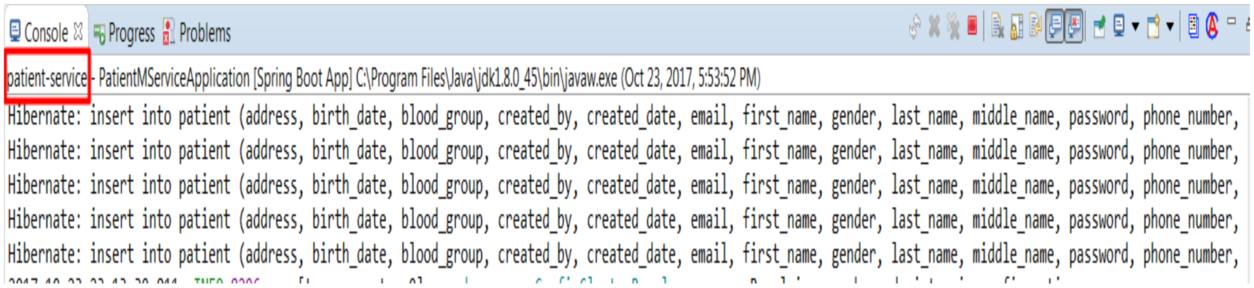
```
{
    "firstName":"john",
    "middleName":"john",
    "lastName":"kane",
    "email":"john@gmail.com",
    "password":"john123",
    "status":"false"
}
```

Here through Jmeter we are simulating this request for 10 users creation I.e. 10 patients records creation

Response:



```
patient-service2 PatientMServiceApplication [Spring Boot App] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 23, 2017, 5:54:14 PM)
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-service - PatientMServiceApplication [Spring Boot App] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct 23, 2017, 5:53:52 PM)
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,
```

So here 5 users created by patient-service instance and another 5 users were created by patient-service instance2. Here Load balancer uses the Round Robbin algorithm to find the patient-service instance and to hand over the request

Eureka for registration and discovery

So far, we have achieved externalizing configuration parameters as well as load balancing across many service instances. Ribbon-based load balancing is sufficient for most of the MicroServices requirements. However, this approach falls short in a couple of scenarios:

- ✓ If there is a large number of microservices, and if we want to optimize infrastructure utilization, we will have to dynamically change the number of service instances and the associated servers. It is not easy to predict and preconfigure the server URLs in a configuration file.
- ✓ When targeting cloud deployments for highly scalable microservices, static registration and discovery is not a good solution considering the elastic nature of the cloud environment.
- ✓ In the cloud deployment scenarios, IP addresses are not predictable, and will be difficult to statically configure in a file. We will have to update the configuration file every time there is a change in address.

The Ribbon approach partially addresses this issue. With Ribbon, we can dynamically change the service instances, but whenever we add new service instances or shut down instances, we will have to manually update the Config server. Though the configuration changes will be automatically propagated to all required instances, the manual configuration changes will not work with large scale deployments. When managing large deployments, automation, wherever possible, is paramount.

To fix this gap, the MicroServices should self-manage their life cycle by dynamically registering service availability, and provision automated discovery for consumers.

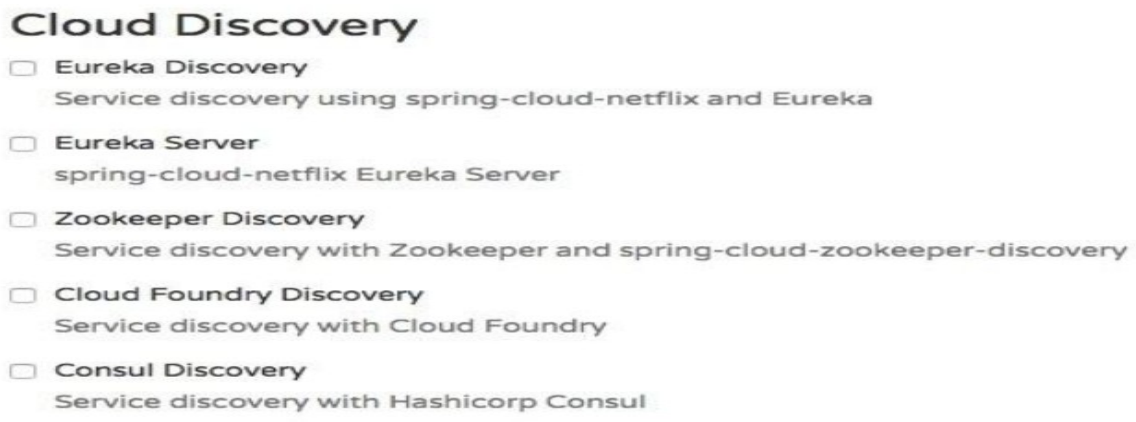
What is Dynamic Service Registration?

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:



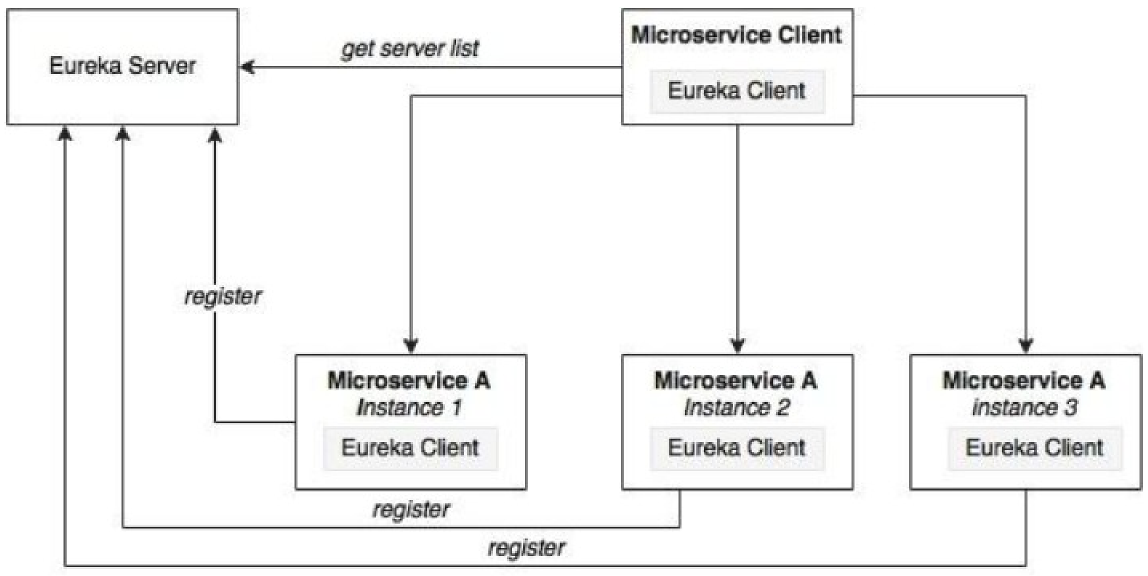
Cloud Discovery

- [ ] 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-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.

<u>How Registry will work?</u>

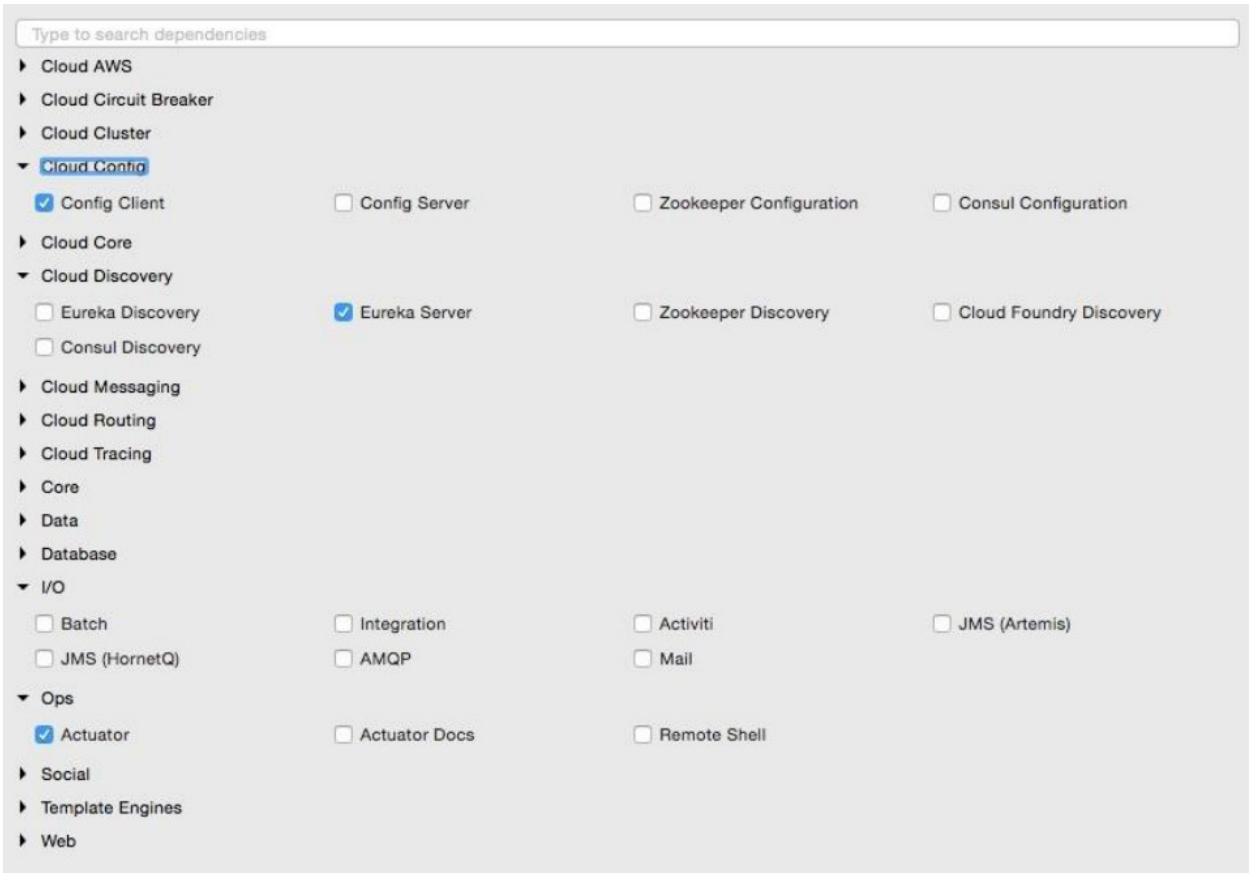
When a microservice is bootstrapped, it reaches out to the Eureka server, and advertises its existence with the binding information. Once registered, the service endpoint sends ping requests to the registry every 30 seconds to renew its lease. If a service endpoint cannot renew its lease in a few attempts,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

<u>Setting up the Eureka server</u>

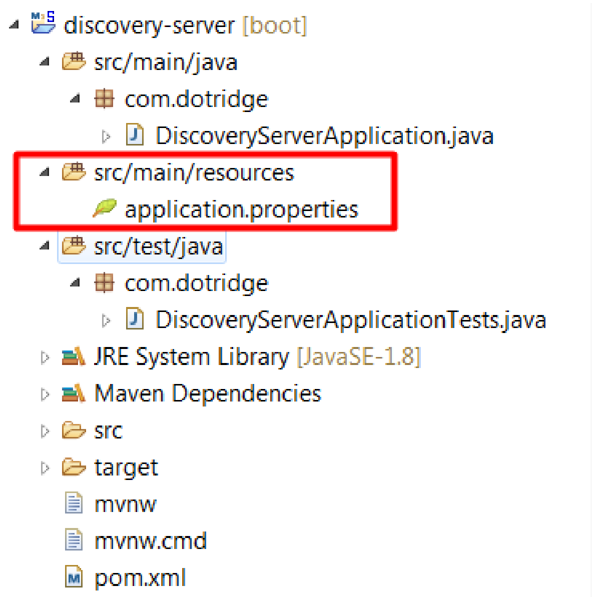
<u>Step-1:</u> Start a new Spring Starter project discovery-service, and select Config Client, Eureka Server, and Actuator:



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

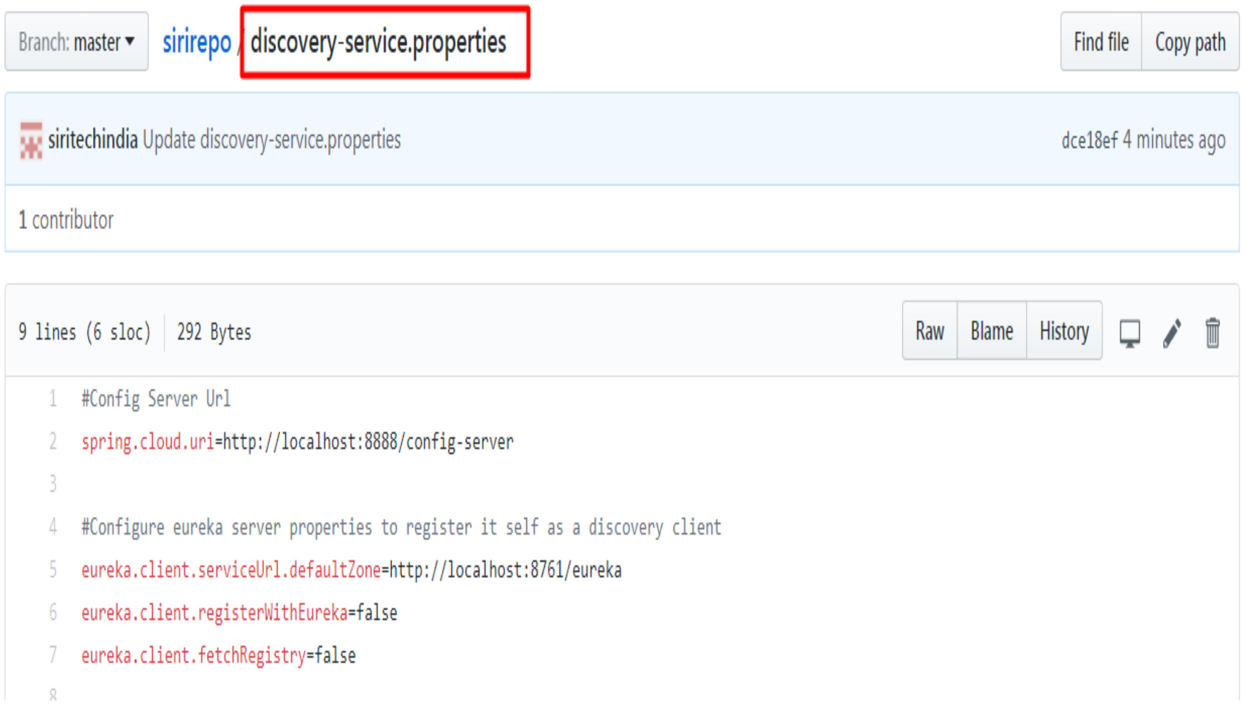
The project structure of the Eureka server i.e,discovery-server is shown in the following image:



```
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

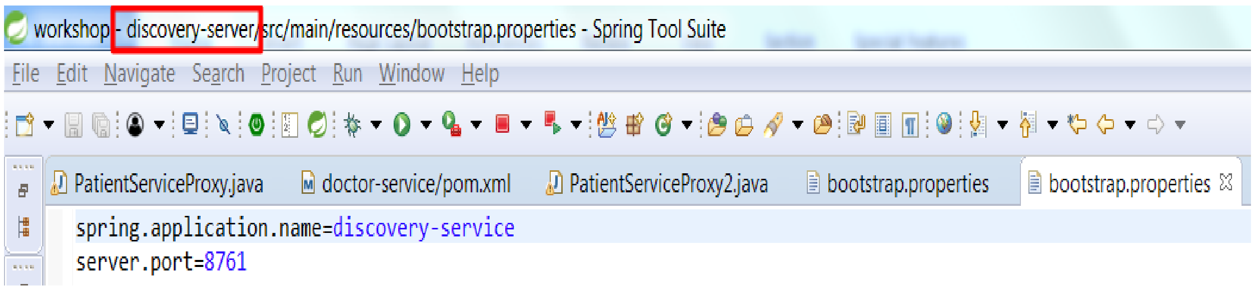
<u>Step-2:</u> Create a <spring.application.name>.properties I.e. discovery-server.properties file and commit to the git repository as discovery-service uses config-server to fetch the configuration properties. This file should have the config server url and eureka server properties as shown below:


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.

<u>Step-3:</u> Rename application.properties to bootstrap.properties since this is using the Config server. It should have application name and port as shown below:

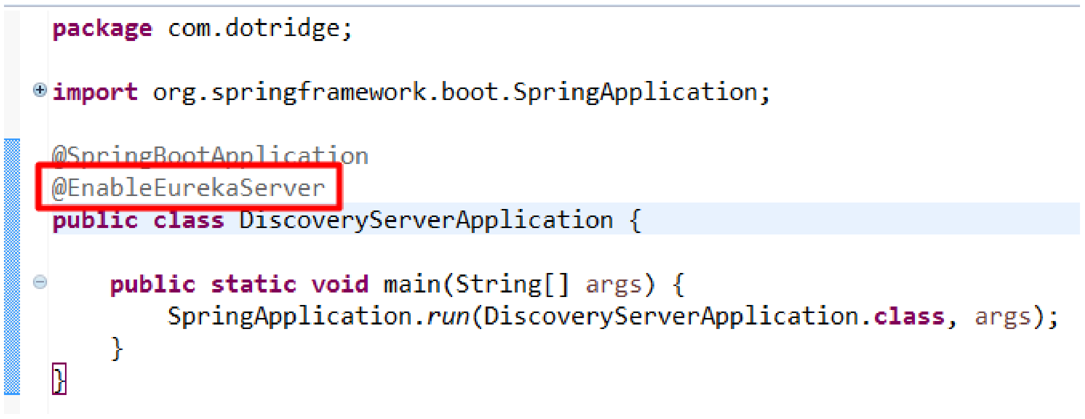

workshop - discovery-server/src/main/resources/bootstrap.properties - Spring Tool Suite
File Edit Navigate Search Project Run Window Help
PatientServiceProxy.java
doctor-service/pom.xml
PatientServiceProxy2.java
bootstrap.properties
bootstrap.properties

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

Note:

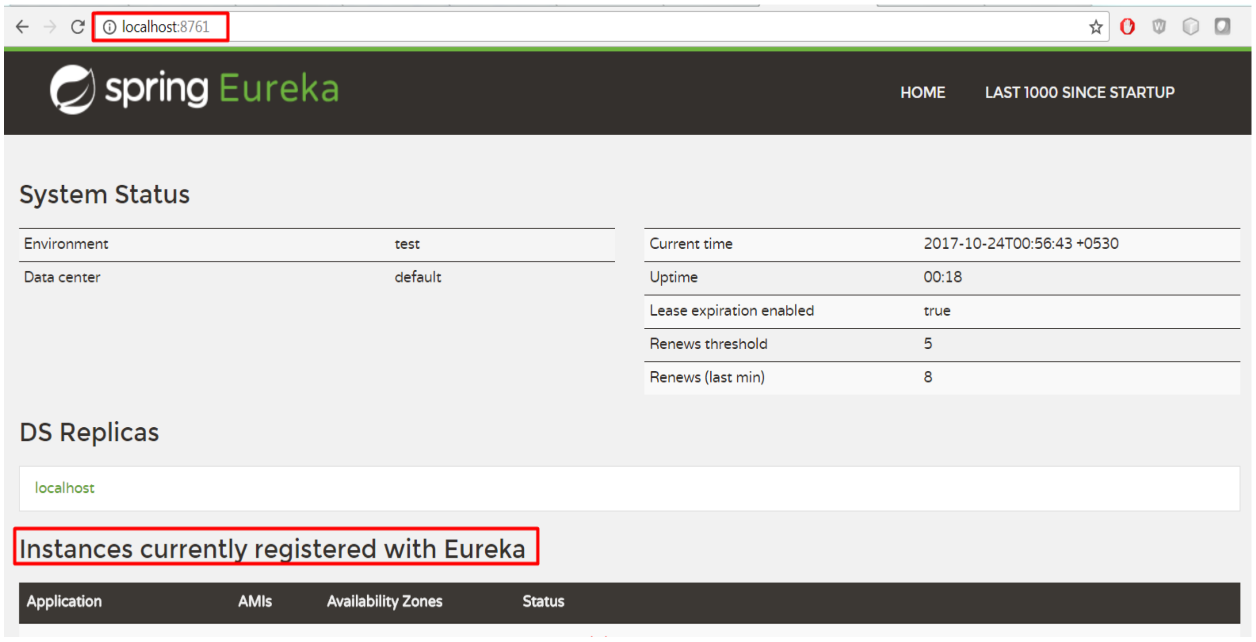
Renaming application.properties file to bootstrap.properties file is not necessary. But when we go for high availability of config-server and discovery-server we might need this.

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.



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

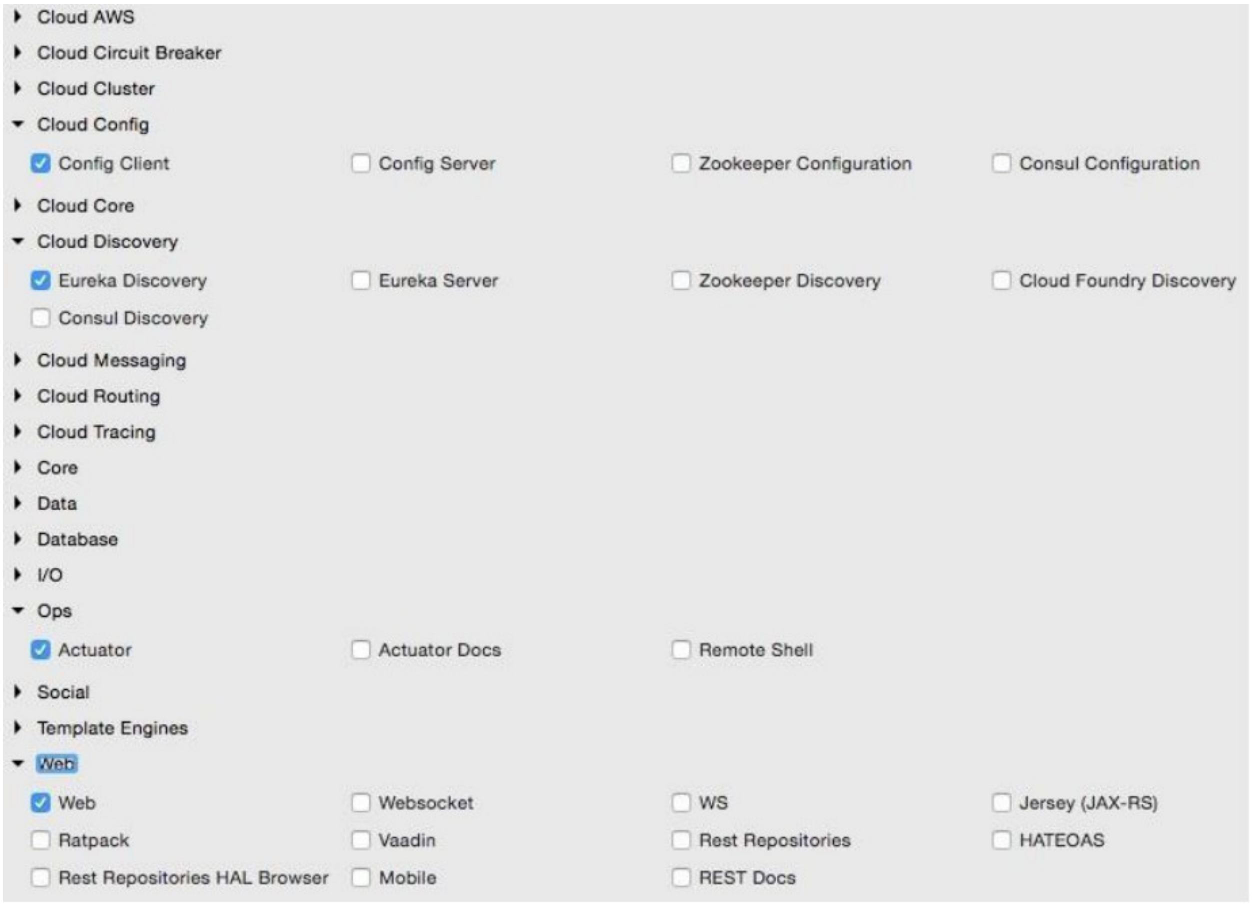
Application	AMIs	Availability Zones	Status

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.

Dynamic Discovery

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>
```

<u>Step-2:</u> 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:

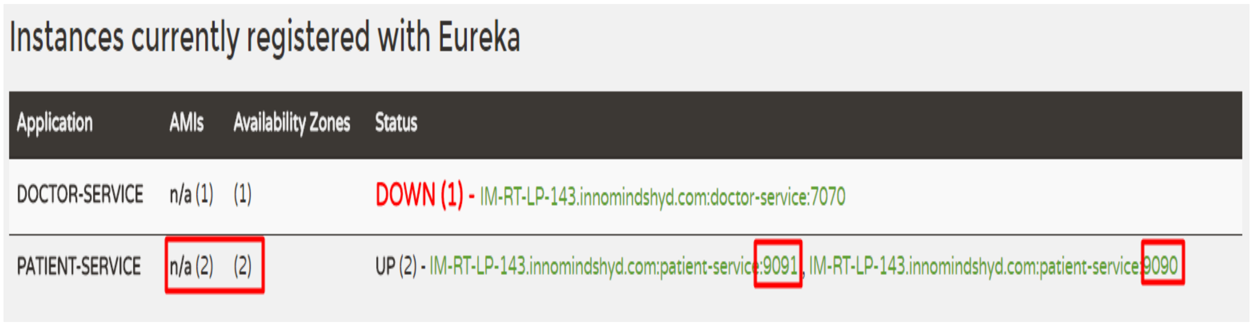
```
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
```

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

Step-3: 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.

```java
@SpringBootApplication
@EnableDiscoveryClient
public class PatientMServiceApplication {


    public static void main(String[] args) {

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

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



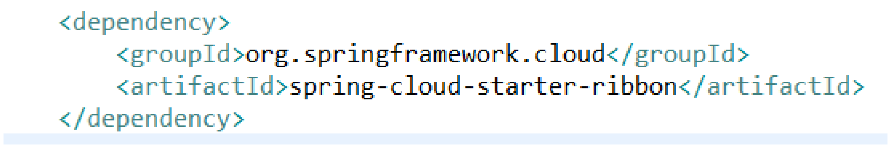
Instances currently registered with Eureka

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

Eureka Service with Load Balancing

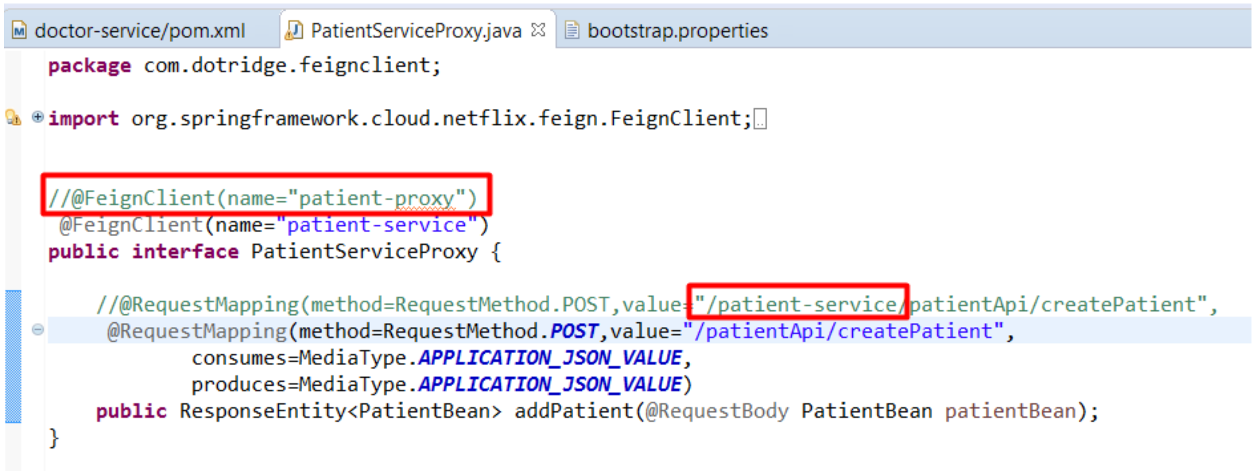
Time to fix the issue with doctor-service. We will remove our earlier Ribbon client, and use Eureka instead. Eureka internally uses Ribbon for load balancing. Hence, the load balancing behavior will not change.

Step-1: Remove the following dependency from doctor-service.properties file



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

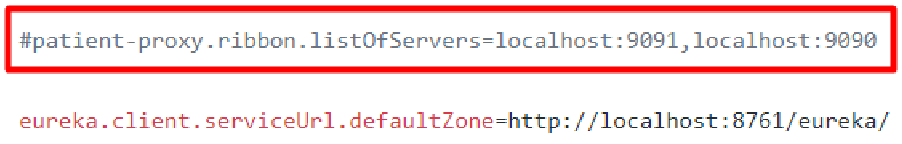
Step-2: Also remove the @RibbonClient annotation from the PatientServiceProxy class created in doctor-service micro service to communicate with the patient-service declaratively. Update @FeignClient(name="patient-proxy") to match the actual Patient microservices' service ID. In this case, patient-service 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.



```java
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);
}
```

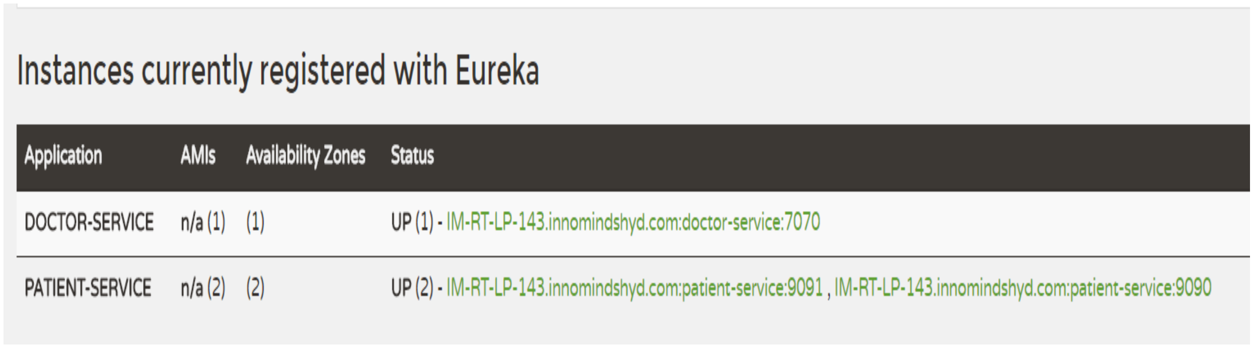
In the above screen shot we were commented the @FeignClient(name="patient-proxy"). when we are using this the patient-service microservice end point have been configured as “/patient-service/patientApi/createPatient” but now as we commented it and using load balancing supported by eureka, we no need 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.

Step-3: 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/
```

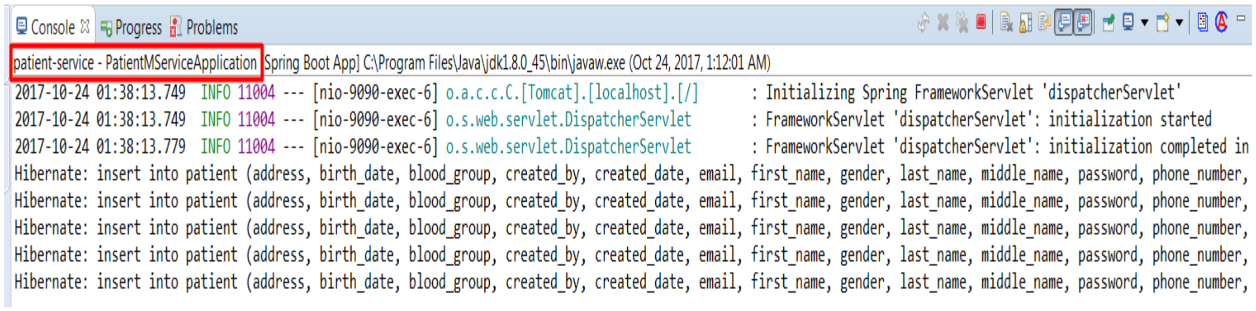
Step-4: Start the doctor-service. now will see that doctor-service is successfully registered with the eureka as shown below:



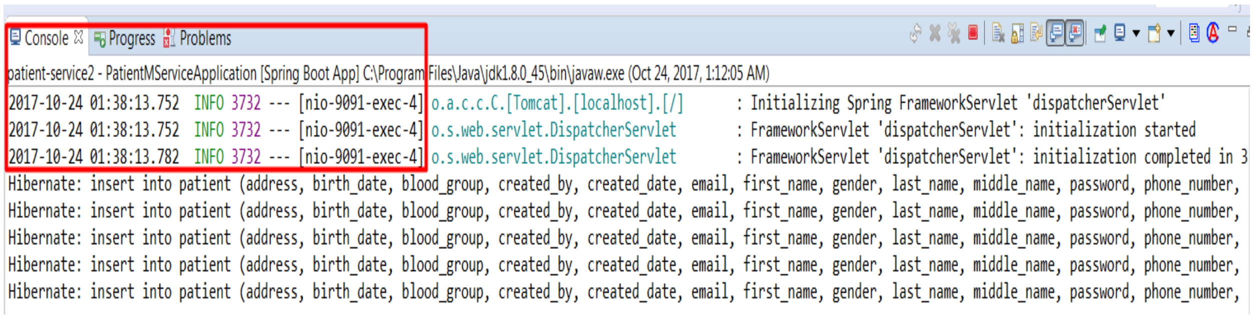
Instances currently registered with Eureka

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:

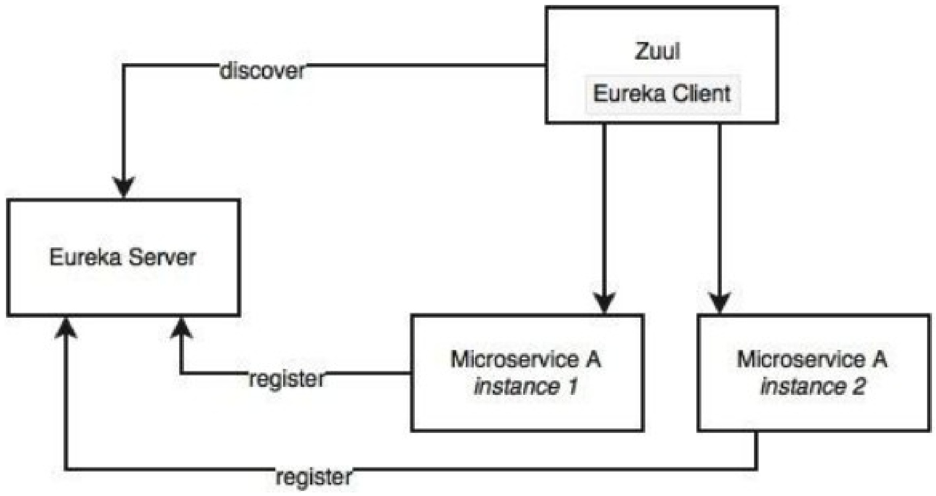
If we observe,**PatientServiceProxy** class where we are using @FiegnClient, earlier without load balaning we have used and attribute “url=localhost:9090/patient-service” of @FiegnClient annotation. But now as we are using Eureka Server discovery we no need to give the url rather the service name is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

Zuul proxy as the API gateway

In most microservice implementations, internal microservice endpoints should not exposed outside. They should kept as private services. A set of public services will be exposed to the clients using an API gateway.There are many reasons to do this:

- ✓ Only a selected set of microservices are required by the clients.
- ✓ If there are client-specific policies to be applied, it is easy to apply them in a single place rather than in multiple places. An example of such a scenario is the cross-origin access policy. It is hard to implement client-specific transformations at the service endpoint.
- ✓ If there is data aggregation required, especially to avoid multiple client calls in a bandwidth-restricted environment, then a gateway is required in the middle.

Zuul is a simple gateway service or edge service that comes from the Netflix family of microservice products.Unlike many enterprise API gateway products, Zuul provides complete control for us to configure or program based on specific requirements.



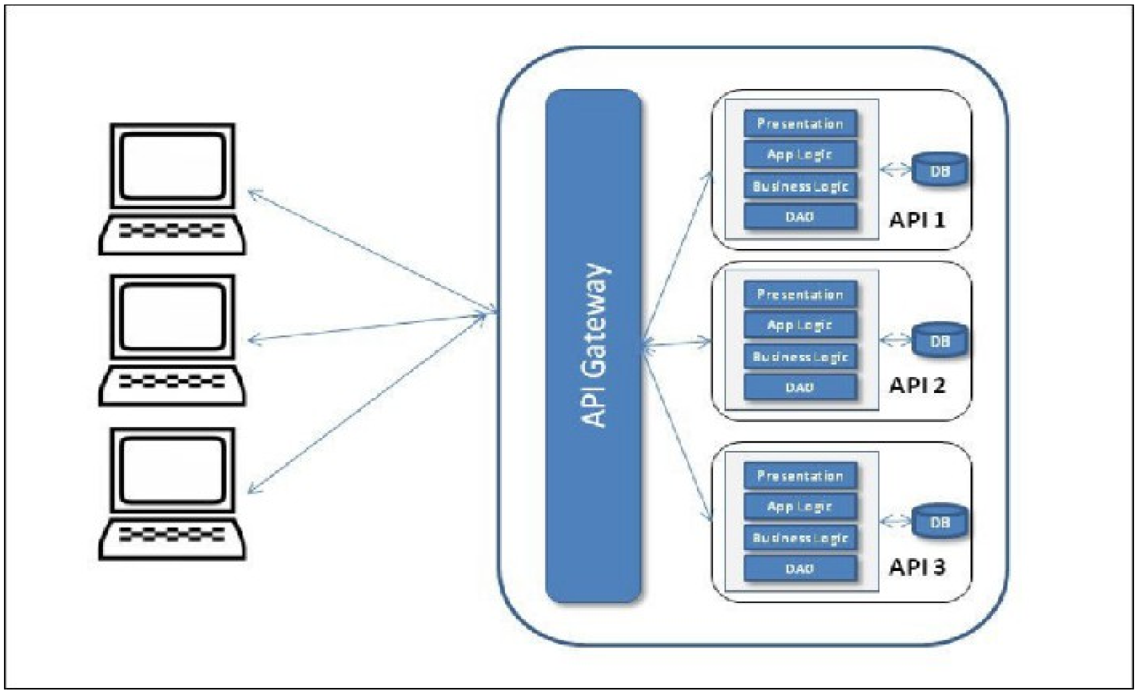
The Zuul proxy internally uses the Eureka server for service discovery, and Ribbon for load balancing between service instances.

The Zuul proxy is also capable of routing, monitoring, managing resiliency, security, and so on. In simple terms, we can consider Zuul as a reverse proxy service. With Zuul, we can even change the behaviors of the underlying services by overriding them at the API layer.

The API gateway provides the interface where different clients can access the individual services and solve the following problems:

if we want to send different responses to different clients for the same service. For example, a patient service could send different responses to a mobile client (minimal information) and a desktop client (detailed information) providing different details and something different again to a third-party client.

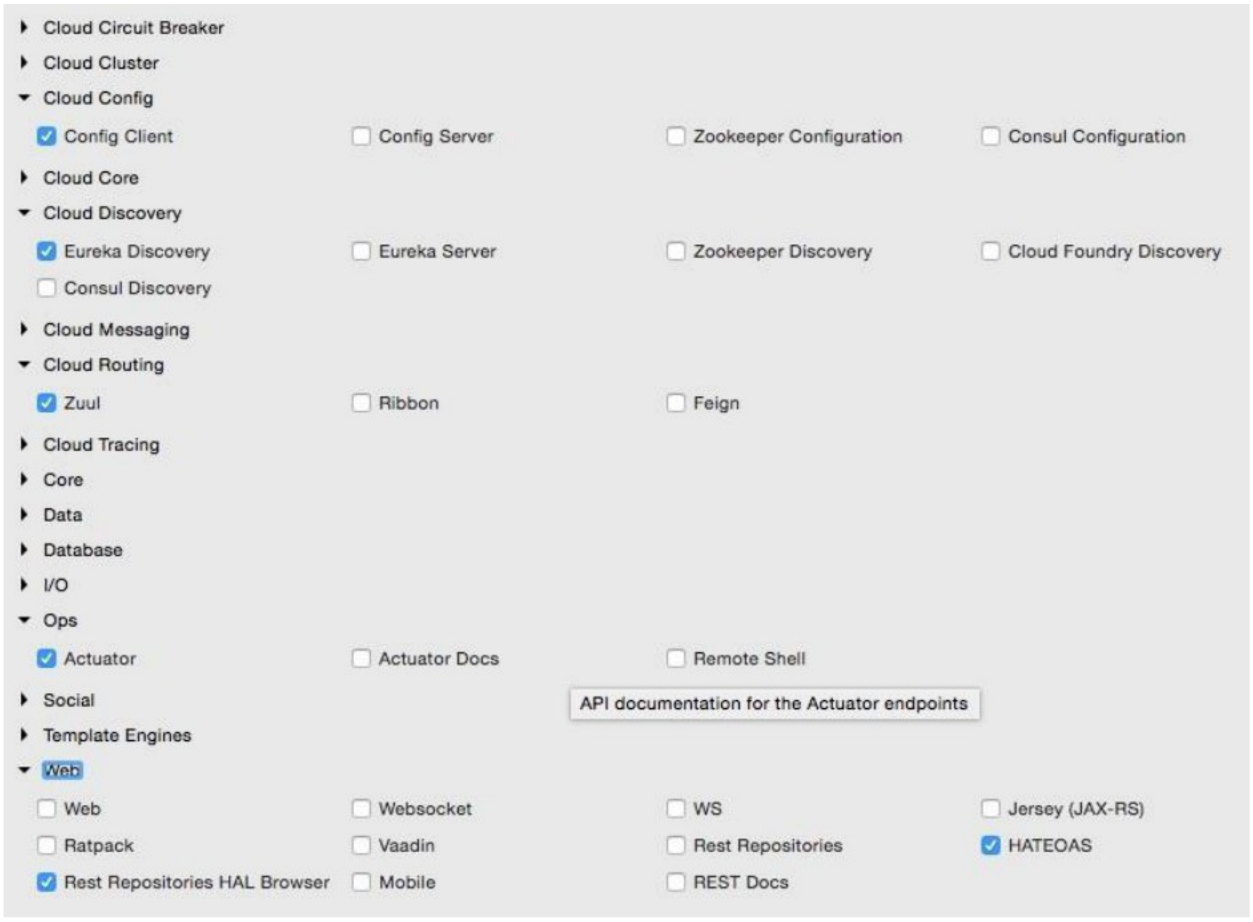
A response may require fetching information from two or more services:



<u>Setting up Zuul</u>

Unlike the Eureka server and the Config server, in typical deployments, Zuul is specific to a microservice. However, there are deployments in which one API gateway covers many MicroServices. In this case, we are going to add Zuul for each of our microservices: doctor-service,patient-service.

<u>Step-1:</u> Create a new Spring Starter project with name zuul-apigateway, and select Zuul, Config Client, Actuator, and Eureka Discovery

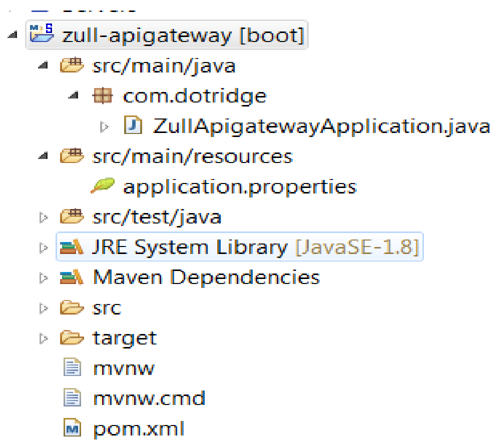


- Cloud Circuit Breaker
- Cloud Cluster
- Cloud Config
  - [✓] Config Client
  - [ ] Config Server
  - [ ] Zookeeper Configuration
  - [ ] Consul Configuration
- Cloud Core
- Cloud Discovery
  - [✓] Eureka Discovery
  - [ ] Eureka Server
  - [ ] Zookeeper Discovery
  - [ ] Cloud Foundry Discovery
  - [ ] Consul Discovery
- Cloud Messaging
- Cloud Routing
  - [✓] Zuul
  - [ ] Ribbon
  - [ ] Feign
- 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

Note:

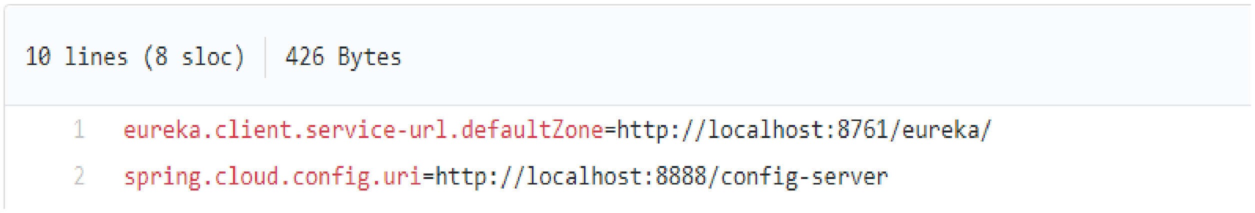
Also add Hysterics, Zuul proxy will depends on Hysterics Dashboard to monitor the each micro service.

The project structure for zuul-apigateway is shown in the following diagram:



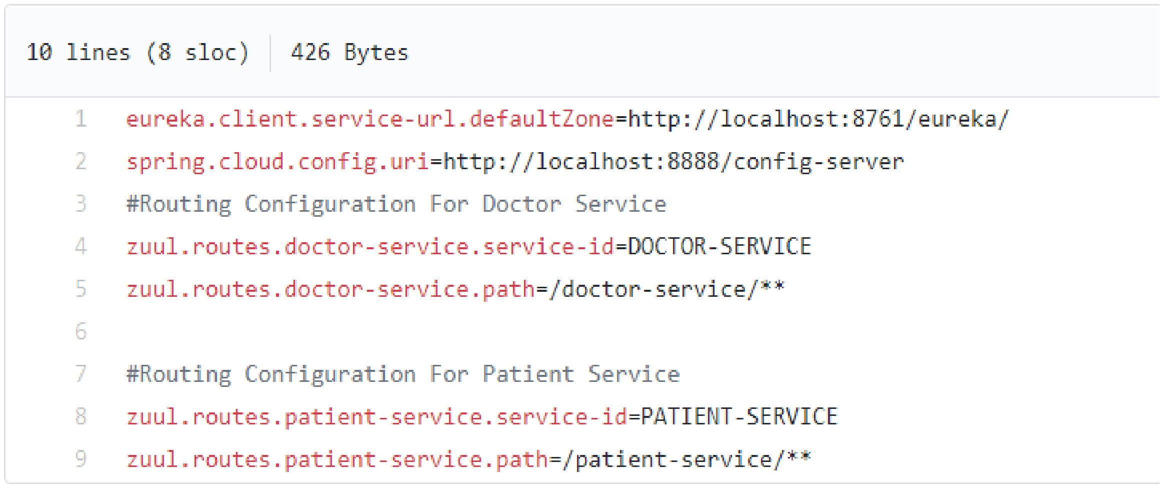
```
zull-apigateway [boot]
  src/main/java
    com.dotridge
      ZullApigatewayApplication.java
  src/main/resources
      application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Step-2: Integrate the API gateway with Eureka and the Config server. Create a **zuul-apigateway.property** file with the contents as shown below and commit to the Git repository.



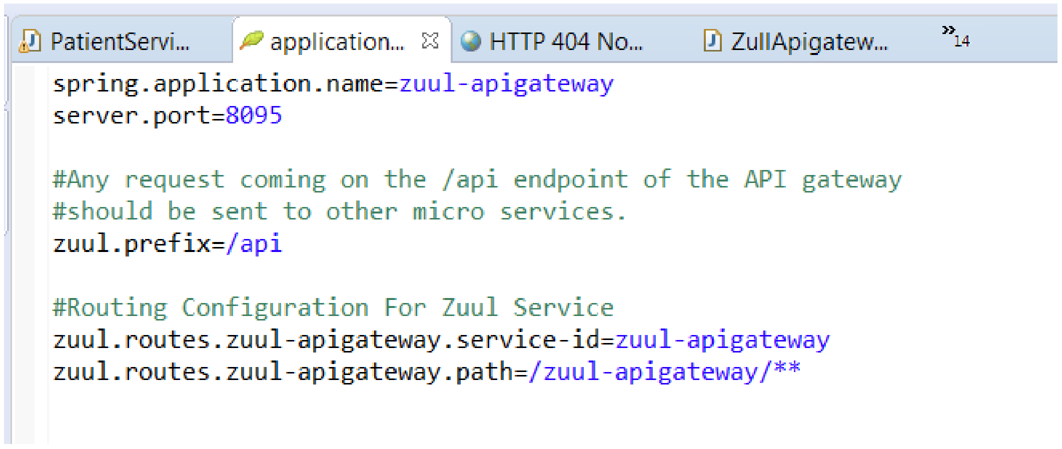
```
10 lines (8 sloc) | 426 Bytes

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
spring.cloud.config.uri=http://localhost:8888/config-server
```

<u>Step-3:</u> add the routing information for the doctor-service and patient-services as well. In the same zuul-apigateway.properties file as shown below:



```
10 lines (8 sloc) | 426 Bytes

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
spring.cloud.config.uri=http://localhost:8888/config-server
#Routing Configuration For Doctor Service
zuul.routes.doctor-service.service-id=DOCTOR-SERVICE
zuul.routes.doctor-service.path=/doctor-service/**

#Routing Configuration For Patient Service
zuul.routes.patient-service.service-id=PATIENT-SERVICE
zuul.routes.patient-service.path=/patient-service/**
```

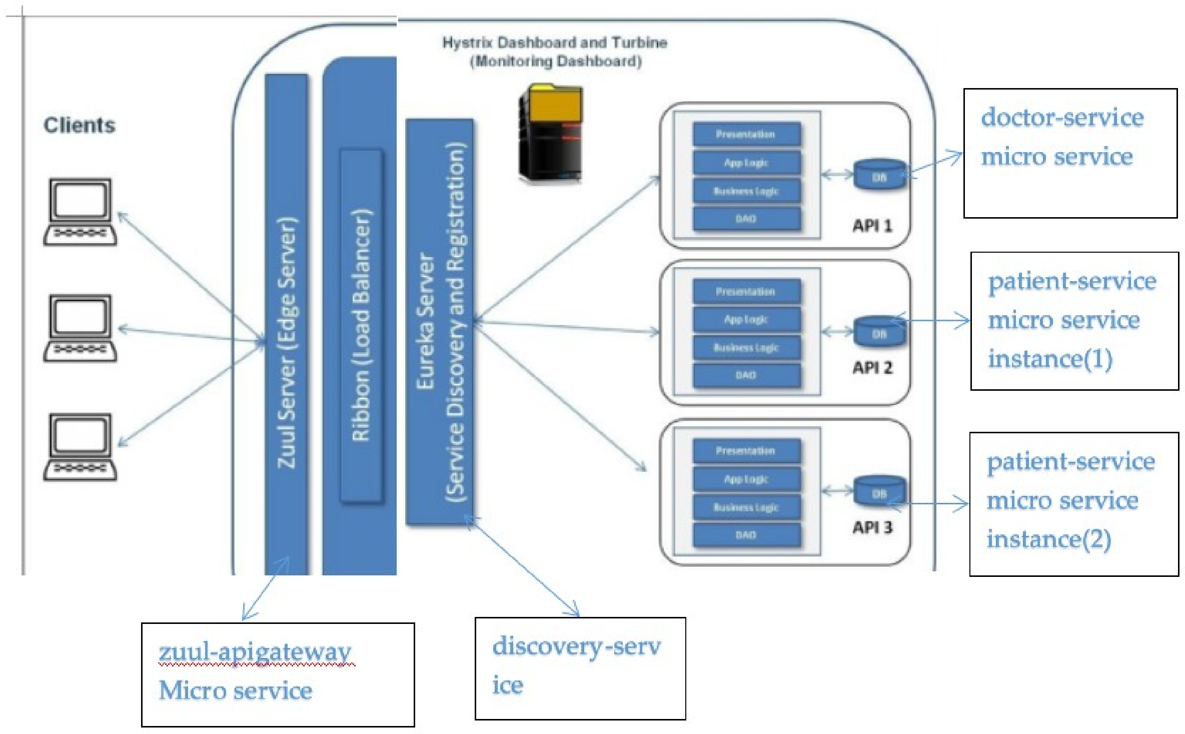
<u>Step-4:</u> update the application.properties file with the zuul proxy configurational parameters as and service id, port as shown below:



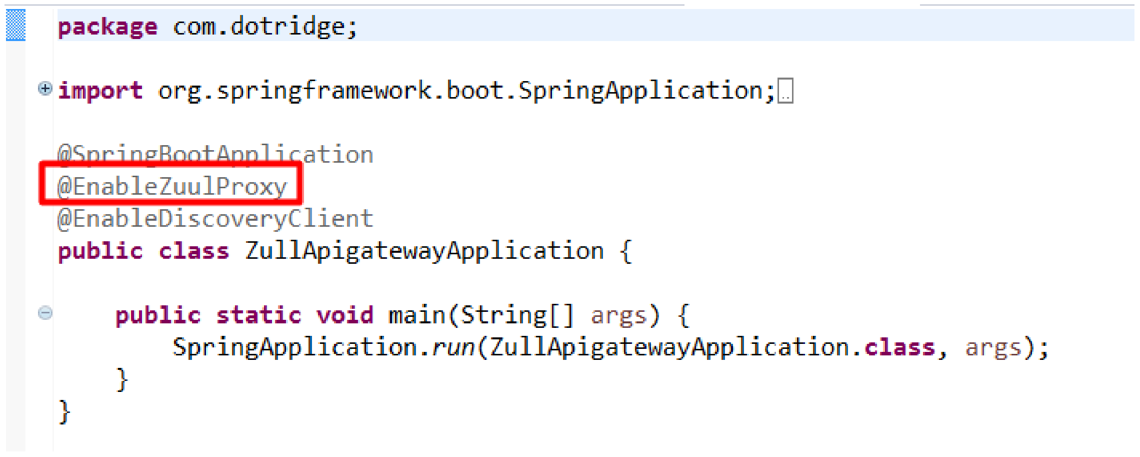
```
spring.application.name=zuul-apigateway
server.port=8095

#Any request coming on the /api endpoint of the API gateway
#should be sent to other micro services.
zuul.prefix=/api

#Routing Configuration For Zuul Service
zuul.routes.zuul-apigateway.service-id=zuul-apigateway
zuul.routes.zuul-apigateway.path=/zuul-apigateway/**
```

This configuration sets a rule on how to forward traffic.**Zuul.prefix:** represents the any request coming on the /api endpoint of the Zuul API gateway should be sent to doctor-service and patient-service micro services. zuul-apigateway is the service ID of the micro service, and it will be resolved using the Eureka server. In the above properties file, we have also configured the routing on to the zuul-apigateway its self.

The following diagram, represents the overall picture of the api gat way and micro services where and how its been configured and forwarding the requests for the appropriate micro service


Hystrix Dashboard and Turbine
(Monitoring Dashboard)
Clients
Zuul Server (Edge Server)
Ribbon (Load Balancer)
Eureka Server
(Service Discovery and Registration)
Presentation
App Logic
Business Logic
DAO
DB
API 1
API 2
API 3
doctor-service micro service
patient-service micro service instance(1)
patient-service micro service instance(2)
zuul-apigateway Micro service
discovery-serv ice

<u>Step-5:</u> add **@EnableZuulProxy** to tell Spring Boot that this is a Zuul proxy as shown below:

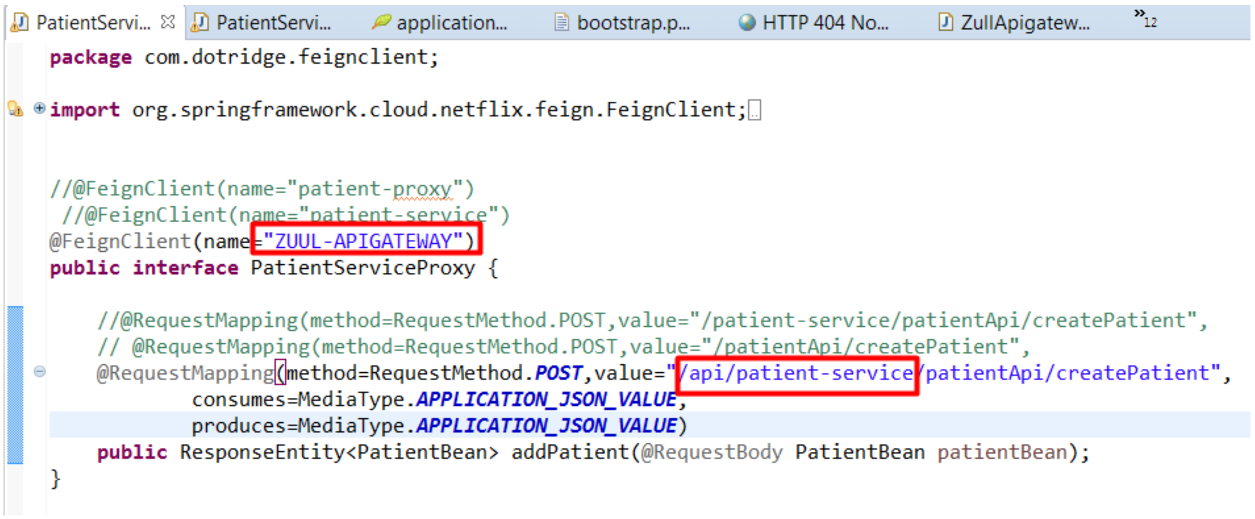


```java
package com.dotridge;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ZullApigatewayApplication {

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

Here we were also added an annotation **@EnableDiscoveryClient,** because zuul proxy will be the client for discovery server I.e.eureka, to get the list of registered micro services.

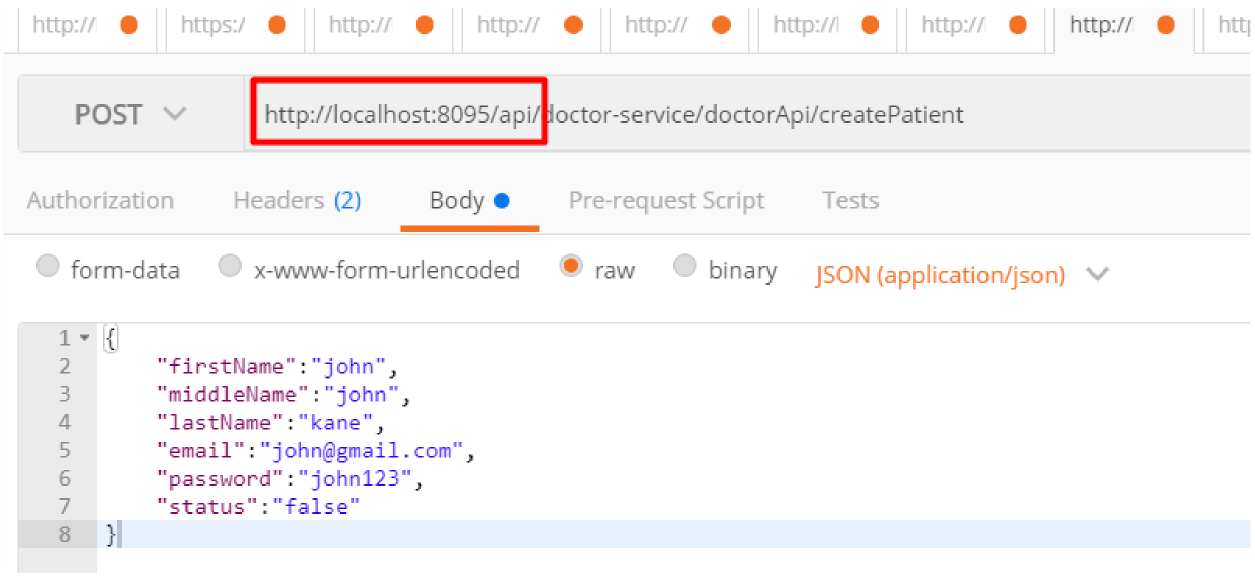
<u>Step-6:</u> Run this as a Spring Boot app. Before that, ensure that the Config server, the Eureka server, and the doctor-service microservice are running.

Now In order to check the working of the api gateway, lets go to doctor-service which contains FiegnClient I.e PatientServiceProxy where we are calling patient-service micro service. Now as we are using api gate way, we don't want to give the actual url I.e.private url **http://localhost:9090/patient-service/patientApi/createPatient** of the patient-service, rather we give the public api as shown below:



```
package com.dotridge.feignclient;

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

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

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

Now create a patient from doctor as shown below:



```
POST  http://localhost:8095/api/doctor-service/doctorApi/createPatient
```

```
{
    "firstName":"john",
    "middleName":"john",
    "lastName":"kane",
    "email":"john@gmail.com",
    "password":"john123",
    "status":"false"
}
```

If we could Observe, the doctor is not calling on its own private url I.e.**http://localhost:7070/doctor-service/doctorApi/createPatient**, rather it is calling now on public api gateway I.e.**http://localhost:8095/api** which is typically the url of the zuul-api gateway.

Now when we give the above request we could see the following logs:

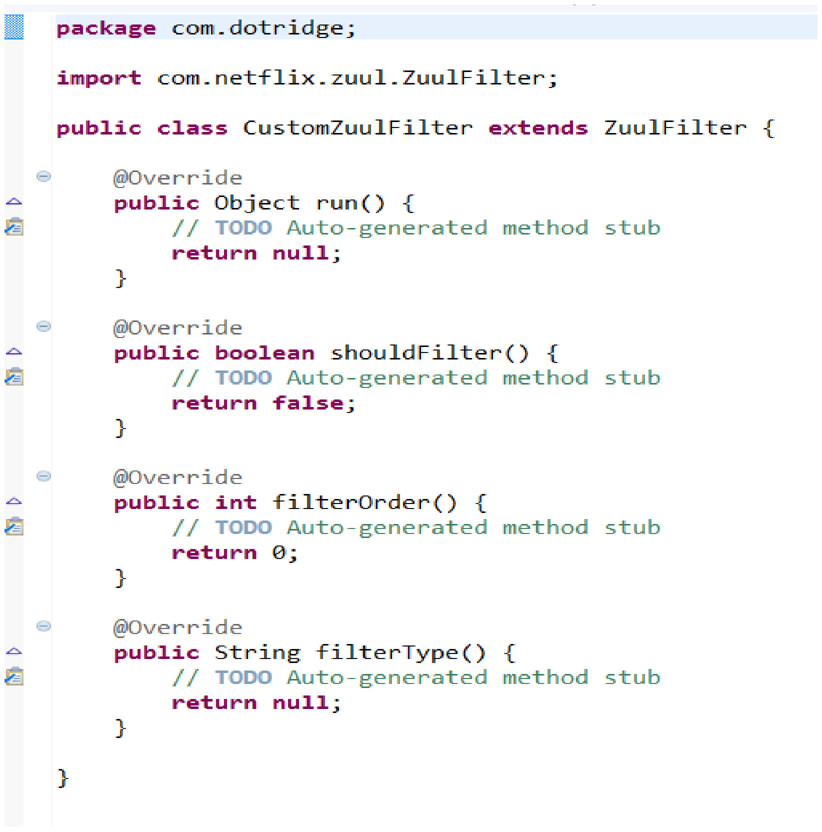
Zuul-apigateway:

```
2017-10-25 17:01:36.032  INFO 11804 --- [io-8095-exec-10] c.n.l.DynamicServerListLoadBalancer      : DynamicServerListLoadBalancer for client patient-service initialized: DynamicServerListLoadBalancer:{NFLoadBalancer:name=patient-service,current list of Servers=[IM-RT-LP-143.innomindshyd.com:9091, IM-RT-LP-143.innomindshyd.com:9090],Load balancer stats=Zone stats: {defaultzone=[Zone:defaultzone;	Instance count:2;	Active connections count: 0;	Circuit breaker tripped count: 0;	Active connections per server: 0.0;]

},Server stats: [[Server:IM-RT-LP-143.innomindshyd.com:9090;	Zone:defaultZone;	Total Requests:0;	Successive connection failure:0;	Total blackout seconds:0;	Last connection made:Thu Jan 01 05:30:00 IST 1970;	First connection made: Thu Jan 01 05:30:00 IST 1970;	Active Connections:0;	total failure count in last (1000) msecs:0;	average resp time:0.0;	90 percentile resp time:0.0;	95 percentile resp time:0.0;	min resp time:0.0;	max resp time:0.0;	stddev resp time:0.0]

, [Server:IM-RT-LP-143.innomindshyd.com:9091;	Zone:defaultZone;	Total Requests:0;	Successive connection failure:0;	Total blackout seconds:0;	Last connection made:Thu Jan 01 05:30:00 IST 1970;	First connection made: Thu Jan 01 05:30:00 IST 1970;	Active Connections:0;	total failure count in last (1000) msecs:0;	average resp time:0.0;	90 percentile resp time:0.0;	95 percentile resp time:0.0;	min resp time:0.0;	max resp time:0.0;	stddev resp time:0.0]

]}ServerList:org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList@41b4484e
```

This is proving that zuul-apigateway is contacting with the discovery-service to list get the service details and then apply the load balancing mechanism to communicate with the high available services.

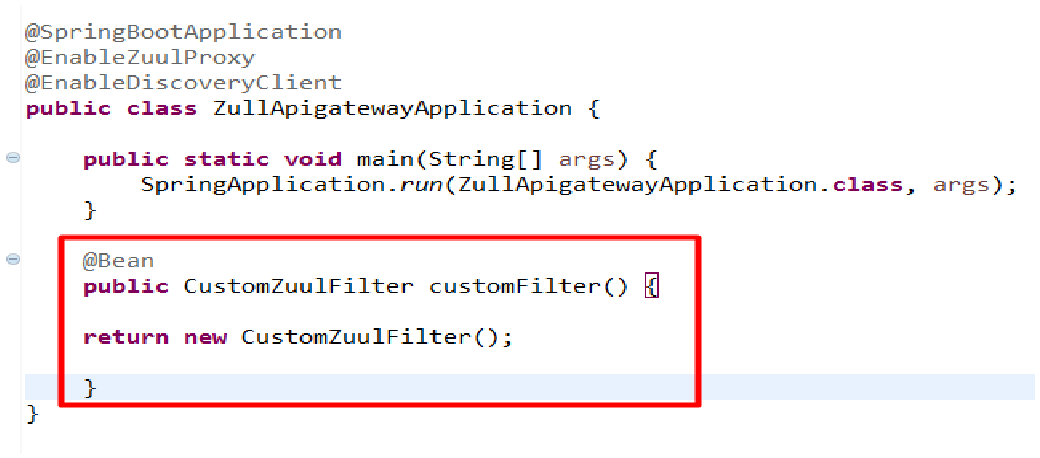
Filters in Zuul

Zuul also provides a number of filters. These filters are classified as pre filters, routing filters, post filters, and error filters. As the names indicate, these are applied at different stages of the life cycle of a service call. Zuul also provides an option for us to write custom filters. In order to write a custom filter, extend from the abstract ZuulFilter, and implement the following methods:



```java
package com.dotridge;

import com.netflix.zuul.ZuulFilter;

public class CustomZuulFilter extends ZuulFilter {

    @Override
    public Object run() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean shouldFilter() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public int filterOrder() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public String filterType() {
        // TODO Auto-generated method stub
        return null;
    }

}
```

Once a custom filter is implemented, add that class to the main context. In our case, add this to the ZullApigatewayApplication class as follows:



```java
@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ZullApigatewayApplication {

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

    @Bean
    public CustomZuulFilter customFilter() {

    return new CustomZuulFilter();

    }
}
```

In the preceding case, it just adds a new endpoint, and returns a value from the gateway. We can further use @Loadbalanced RestTemplate to call a backend service. Since we have full control, we can do transformations, data aggregation, and so on. We can also use the Eureka APIs to get the server list, and implement completely independent load-balancing or traffic-shaping mechanisms instead of the out-of-the-box load balancing features provided by Ribbon.

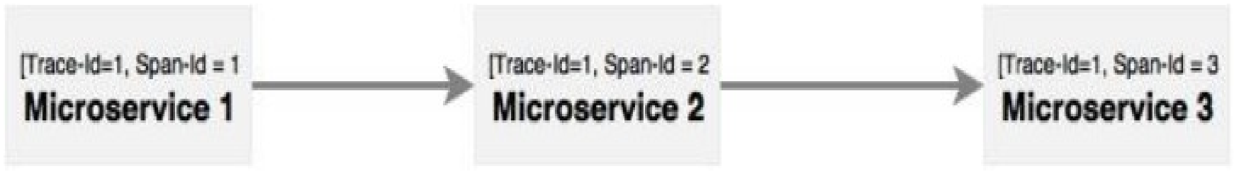
Note:

1. In the above approach we configured the API Gateway as the global to all the underlying micro services. If we need, even we can have the API Gateway on each and every individual micro service as well.
2. Always suggestible to take the zuul routing properties for other micro services in configuration server I.e.git and don't forgot to add the zuul proxy configurational parameters in application.properties file of zuul-apigateway. If we add these zuul configurational parameters in config server, zuul-apigateway wont work.

Distributed tracing with Spring Cloud Sleuth

In order to do end-to-end tracking of MicroServices need to have a correlation ID.Twitter's Zipkin, Cloudera's HTrace, and Google's Dapper systems are examples of distributed tracing systems. Spring Cloud provides a wrapper component on top of these using the Spring Cloud Sleuth library.

Distributed tracing works with the concepts of span and trace. The span is a unit of work; for example, calling a service is identified by a 64-bit span ID. A set of spans form a tree-like structure is called a trace. Using the trace ID, the call can be tracked end to end.



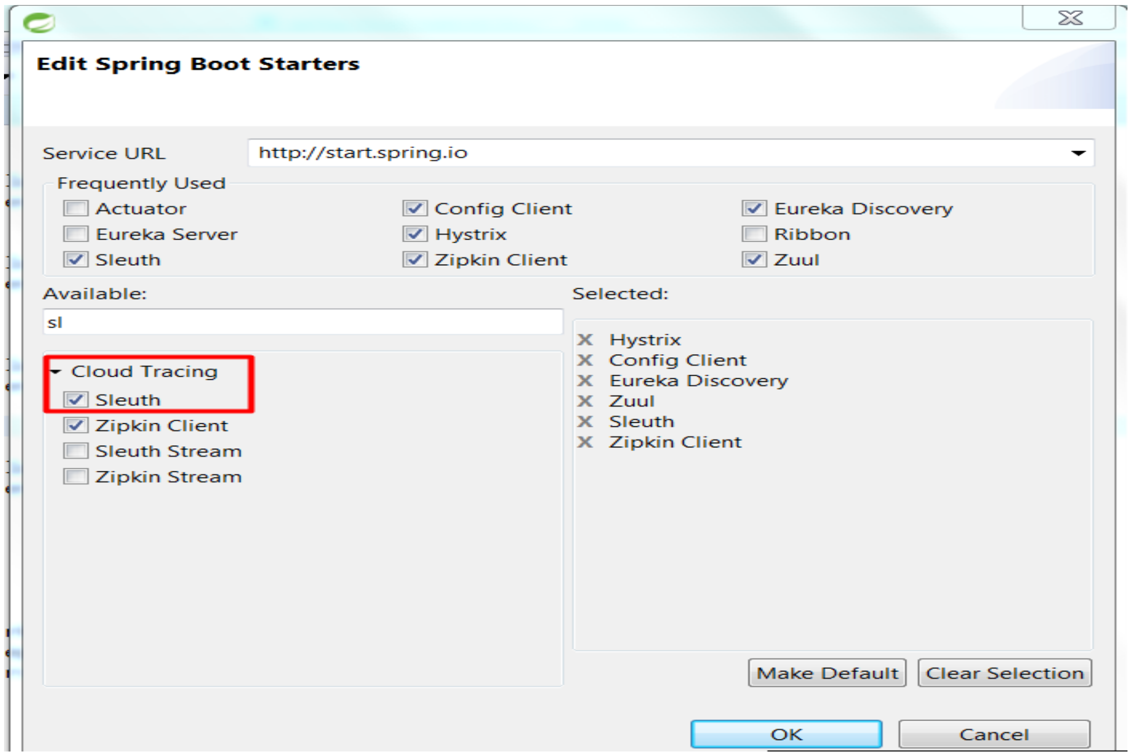
As shown in the diagram, Microservice 1 calls Microservice 2, and Microservice 2 calls Microservice 3. In this case, as shown in the diagram, the same trace ID is passed across all microservices, which can be used to track transactions end to end.

In order to demonstrate this, we will use the Zuul API Gateway and doctor-service,patient-service microservices.Without the trace ID, it is almost impossible to trace or link calls coming from the Website to Zuul API Gateway to doctor-service,patient-service microservice. In this case, it only involves two to three services, whereas in a complex environment, there could be many interdependent services.

Step-1: Update Zuul-apigateway API Gateway,doctor-service,patient-service micro services with the Sleuth dependency by adding the following dependency.

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

Add this dependency , by right click in zuul-apigateway pom.xml, select spring--->Edit Startes then the following window will open from there search for slueth and click on okay.



Edit Spring Boot Starters

Service URL: http://start.spring.io

Frequently Used

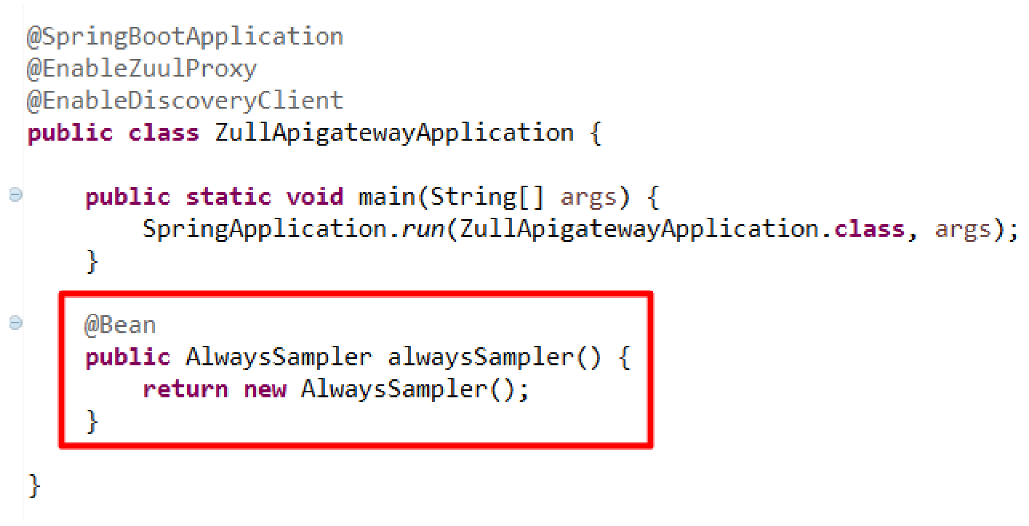
- [ ] Actuator
- [✓] Config Client
- [✓] Eureka Discovery
- [ ] Eureka Server
- [✓] Hystrix
- [ ] Ribbon
- [✓] Sleuth
- [✓] Zipkin Client
- [✓] Zuul

Available: sl

Cloud Tracing
- [✓] Sleuth
- [✓] Zipkin Client
- [ ] Sleuth Stream
- [ ] Zipkin Stream

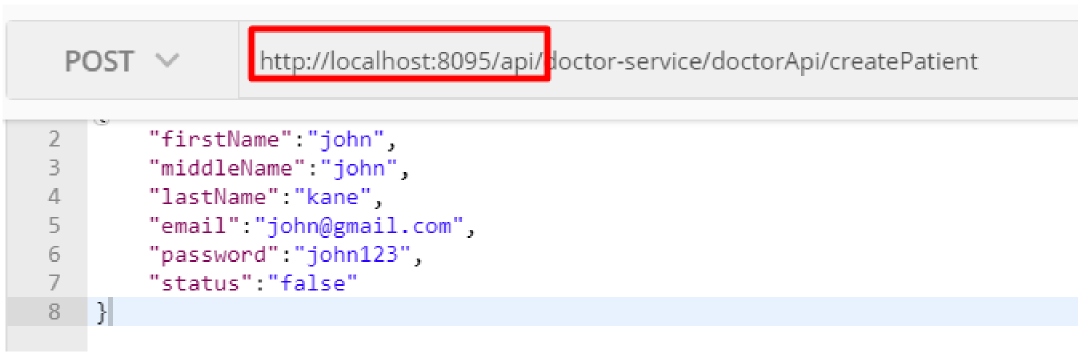
Selected:
- Hystrix
- Config Client
- Eureka Discovery
- Zuul
- Sleuth
- Zipkin Client

<u>Step-2:</u>Add the following piece of code in each and every micro service (in our case Zuul-apigateway,doctor-service,patient-service)to advise Sleuth when to start a new span ID in the Spring Boot Application class. In this case, AlwaysSampler is used to indicate that the span ID has to be created every time a call hits the service.



```java
@SpringBootApplication
@EnableZuulProxy
@EnableDiscoveryClient
public class ZullApigatewayApplication {

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

    @Bean
    public AlwaysSampler alwaysSampler() {
        return new AlwaysSampler();
    }

}
```

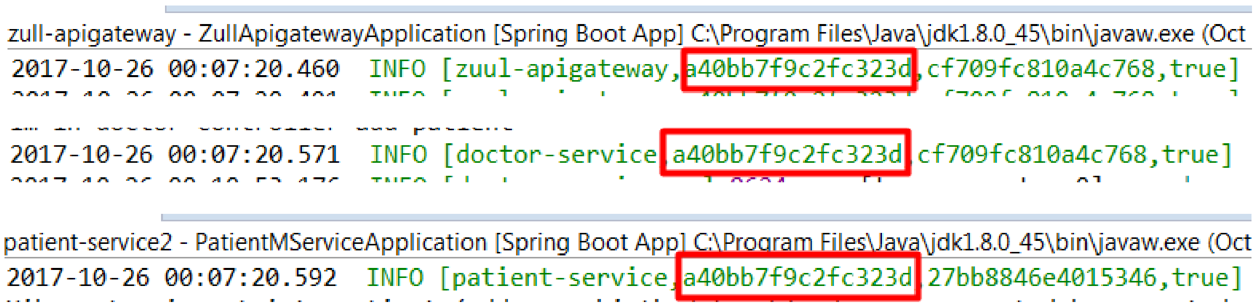
Step-3: Now Run all the restart all the services and run the doctor-service which is identified through api gate as below:



```
POST    http://localhost:8095/api/doctor-service/doctorApi/createPatient
```

```
    "firstName":"john",
    "middleName":"john",
    "lastName":"kane",
    "email":"john@gmail.com",
    "password":"john123",
    "status":"false"
}
```

Now if we could see the logs, we can get the following messages on the console:



```
zull-apigateway - ZullApigatewayApplication [Spring Boot App] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct
2017-10-26 00:07:20.460  INFO [zuul-apigateway,a40bb7f9c2fc323d,cf709fc810a4c768,true]

2017-10-26 00:07:20.571  INFO [doctor-service,a40bb7f9c2fc323d,cf709fc810a4c768,true]

patient-service2 - PatientMServiceApplication [Spring Boot App] C:\Program Files\Java\jdk1.8.0_45\bin\javaw.exe (Oct
2017-10-26 00:07:20.592  INFO [patient-service,a40bb7f9c2fc323d,27bb8846e4015346,true]
```

This looks like a normal log, except for the part in the beginning between the brackets. This is the core information that Spring Sleuth has added. This data follows the format of:

[application name, traceId, spanId, export]

Application name: This is the name we set in the properties file and can be used to aggregate logs from multiple instances of the same application.

TraceId: This is an id that is assigned to a single request, job, or action. Something like each unique user initiated web request will have its own traceId.

SpanId: Tracks a unit of work. Think of a request that consists of multiple steps. Each step could have its own spanId and be tracked individually.

Export – This property is a boolean that indicates whether or not this log was exported to an aggregator like Zipkin. Zipkin is used for analyzing logs created by Sleuth.

Zipkin

Zipkin was originally developed at Twitter, based on a concept of a Google paper that described Google's internally-built distributed app debugger – dapper. It manages both the collection and lookup of this data. To use Zipkin, applications are instrumented to report timing data to it.

In Complex Micro Services ecosystem,if we wants to troubleshooting latency problems or errors,Zipkin can be used which is having a graphical web application dashboard where we can filter or sort all traces based on the application, length of trace, annotation, or timestamp. By analyzing these traces, we can decide which components are not performing as per expectations, and we can fix them.

Internally it has 4 modules:

Collector: Once any component sends the trace data arrives to Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.

Storage: This module store and index the lookup data in backend. Cassandra, ElasticSearch and MySQL are supported.

Search: This module provides a simple JSON API for finding and retrieving traces stored in backend. The primary consumer of this API is the Web UI.

Web UI:A very nice UI interface for viewing traces.

Install Zipkin

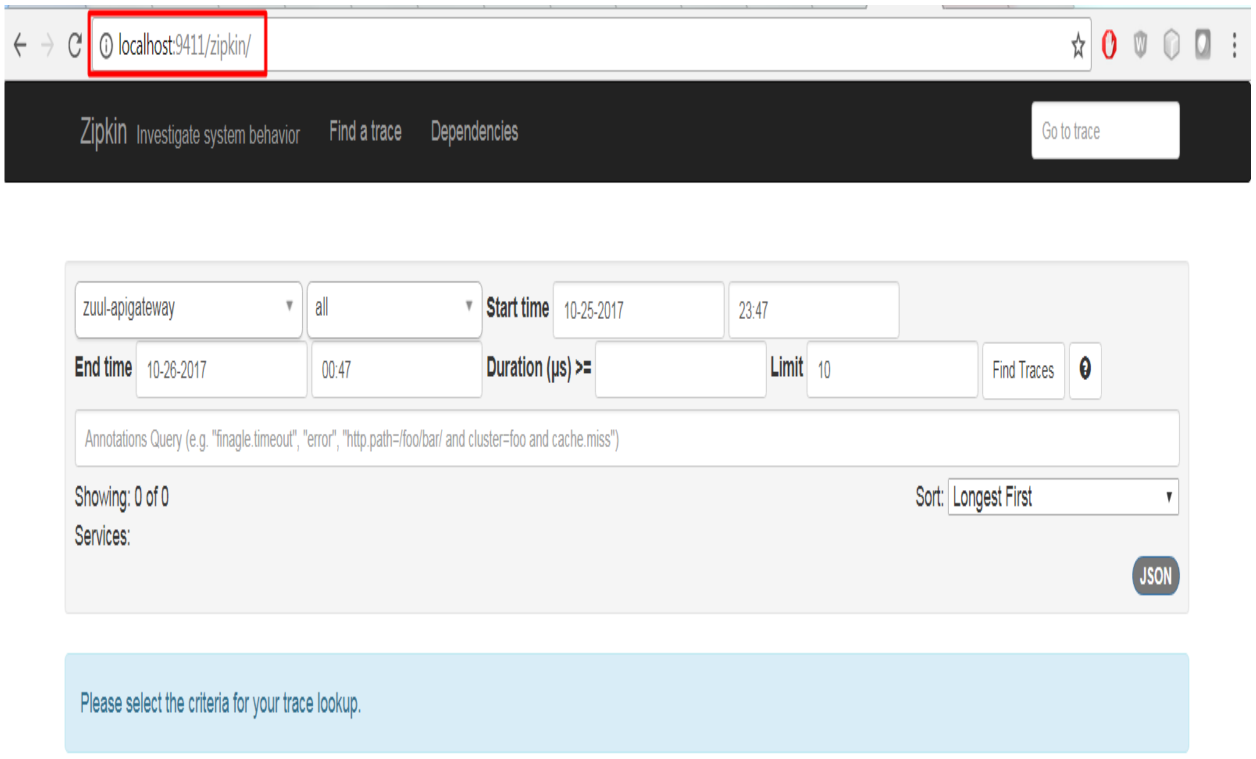
For windows installation, just download the latest Zipkin server from maven repository(https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exec) and run the executable jar file using below command.

java -jar zipkin-server-1.30.3-exec.jar

Above command will start the Zipkin server with default configuration. For advanced configuration, we can configure many other things like storage, collector listeners etc.

Once Zipkin is started, we can see the Web UI at http://localhost:9411/zipkin/

It is like as below:


localhost:9411/zipkin/

zuul-apigateway | all | Start time 10-25-2017 23:47
End time 10-26-2017 00:47 | Duration (μs) >= | Limit 10 | Find Traces

Annotations Query (e.g. "finagle.timeout", "error", "http.path=/foo/bar/ and cluster=foo and cache.miss")

Showing: 0 of 0
Services:
Sort: Longest First

Please select the criteria for your trace lookup.

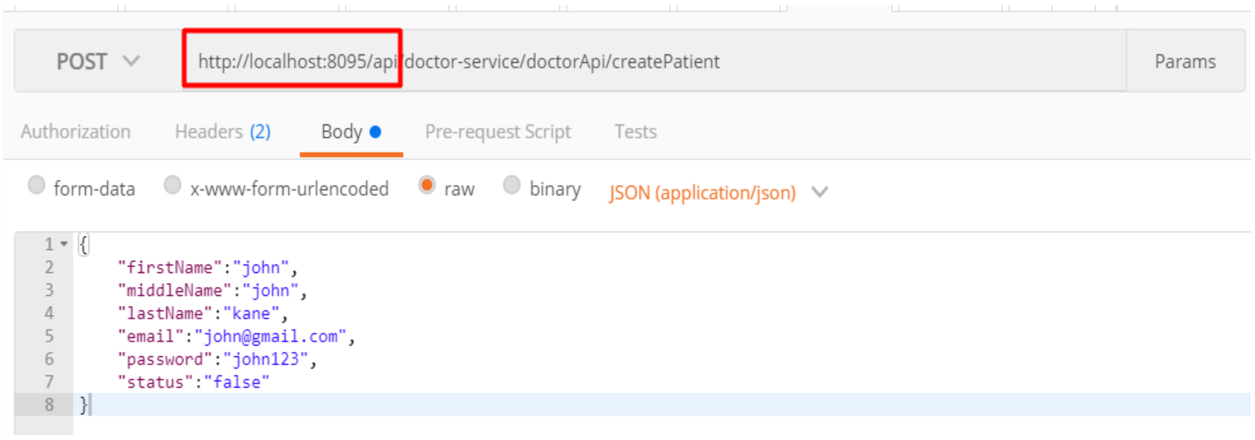
Step-2:

To install Zipkin in spring boot application ie. in our zuul-apigateway,doctor-service and patient-service micro services, we need to add Zipkin starter dependency as below:

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

Step-4: Now Restart all the service, and restart zipkin then open zipkin dashboard, we could see our micro services I.e.doctor-service,patient-service as zipkin clients as shown below:

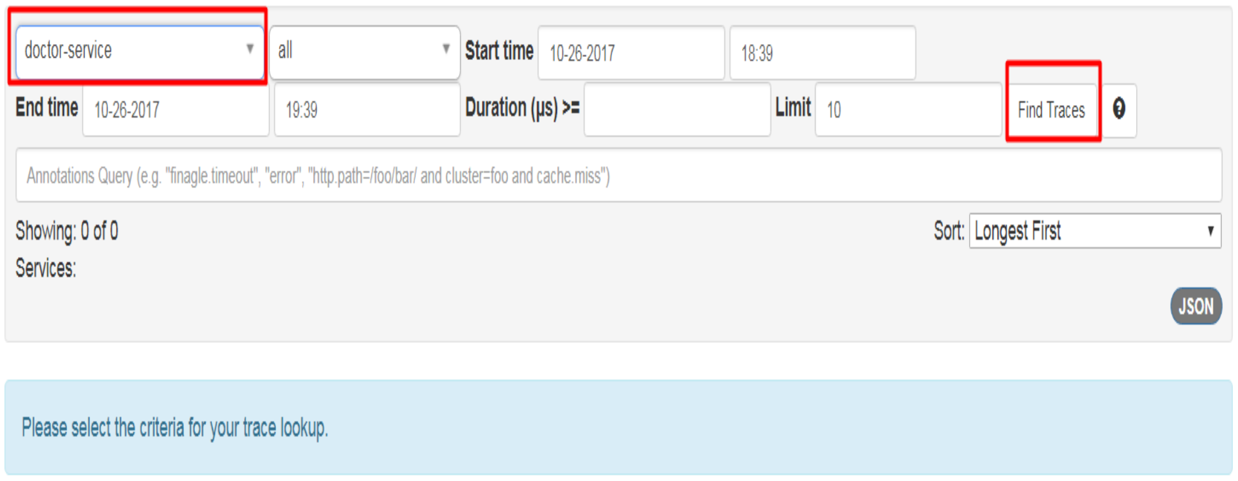
And give the request as below:



http://localhost:8095/api/doctor-service/doctorApi/createPatient

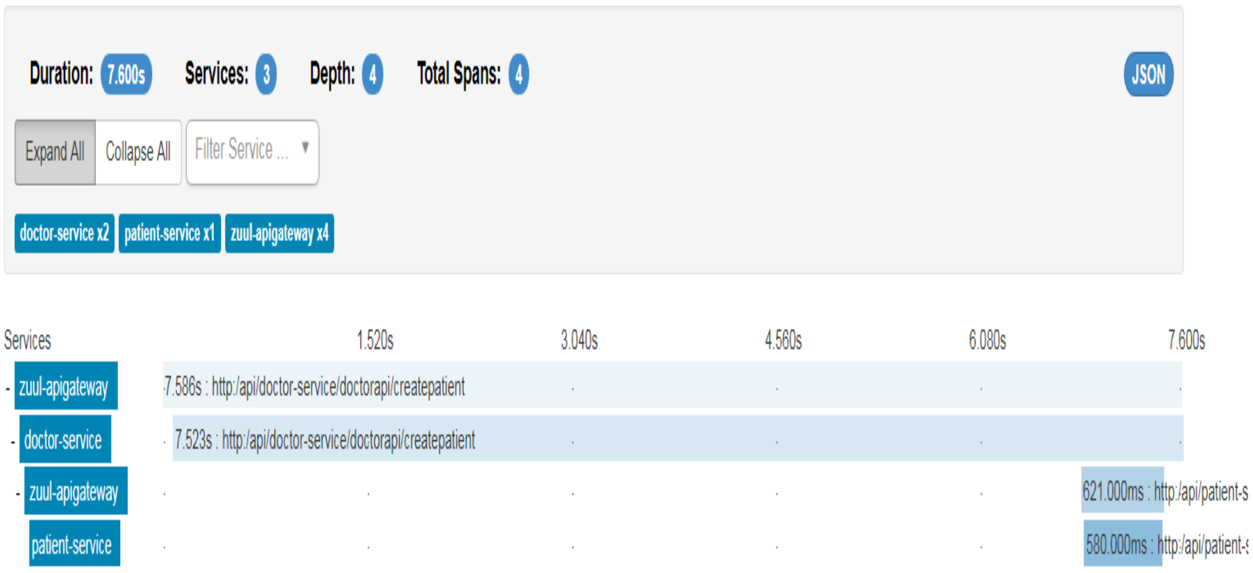
```
{
    "firstName":"john",
    "middleName":"john",
    "lastName":"kane",
    "email":"john@gmail.com",
    "password":"john123",
    "status":"false"
}
```

When we run the above request, we could see all our micro service which are having zipkin as maven dependency will be registered in zipkin server and can be monitored in its dashboard as shown below:



Please select the criteria for your trace lookup.

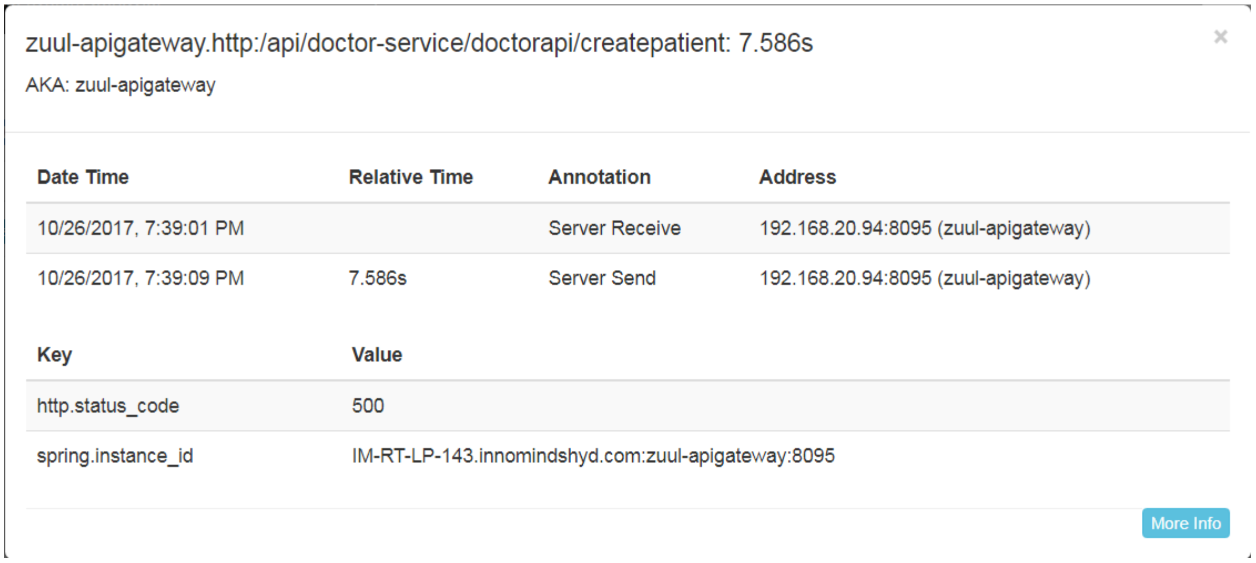
As shown above, When we click on find traces by selecting the micro service from the drop down, we could see the traces as below:



7.586s : http:/api/doctor-service/doctorapi/createpatient
7.523s : http:/api/doctor-service/doctorapi/createpatient
621.000ms : http:/api/patient-s
580.000ms : http:/api/patient-s

Now lets us analyze the flow:

1) Select zuul-apigateway, and we could see the following logs:



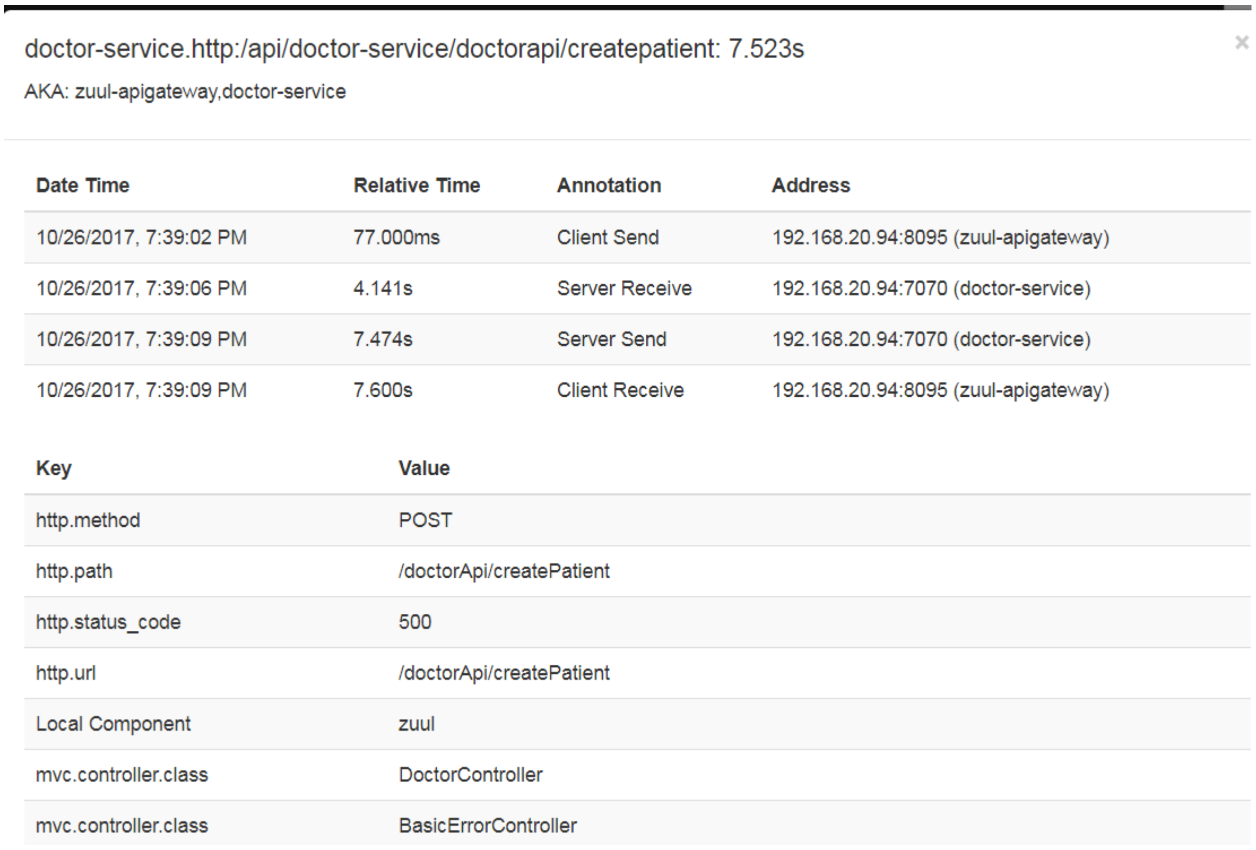
zuul-apigateway.http:/api/doctor-service/doctorapi/createpatient: 7.586s

AKA: zuul-apigateway

Date Time	Relative Time	Annotation	Address
10/26/2017, 7:39:01 PM		Server Receive	192.168.20.94:8095 (zuul-apigateway)
10/26/2017, 7:39:09 PM	7.586s	Server Send	192.168.20.94:8095 (zuul-apigateway)

Key	Value
http.status_code	500
spring.instance_id	IM-RT-LP-143.innomindshyd.com:zuul-apigateway:8095

2) click on doctor-service and we could see the logs as below:



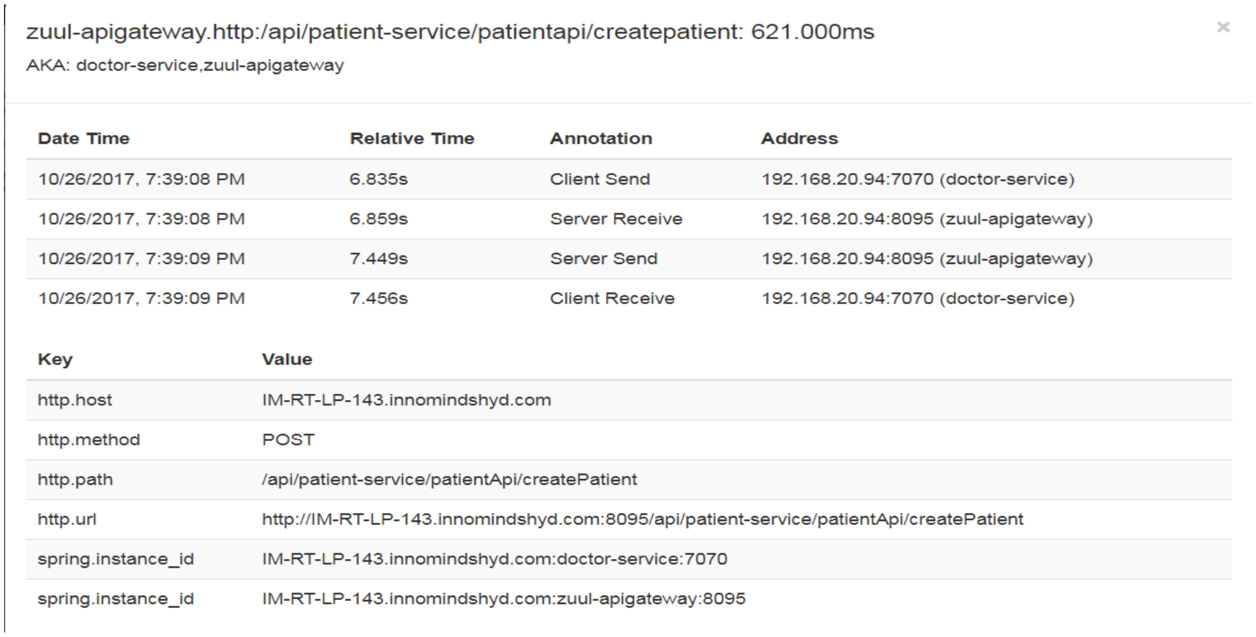
doctor-service.http:/api/doctor-service/doctorapi/createpatient: 7.523s

AKA: zuul-apigateway,doctor-service

Date Time	Relative Time	Annotation	Address
10/26/2017, 7:39:02 PM	77.000ms	Client Send	192.168.20.94:8095 (zuul-apigateway)
10/26/2017, 7:39:06 PM	4.141s	Server Receive	192.168.20.94:7070 (doctor-service)
10/26/2017, 7:39:09 PM	7.474s	Server Send	192.168.20.94:7070 (doctor-service)
10/26/2017, 7:39:09 PM	7.600s	Client Receive	192.168.20.94:8095 (zuul-apigateway)

Key	Value
http.method	POST
http.path	/doctorApi/createPatient
http.status_code	500
http.url	/doctorApi/createPatient
Local Component	zuul
mvc.controller.class	DoctorController
mvc.controller.class	BasicErrorController

Step-3: Now again click on zuul-apigateway, we could see the below logs:



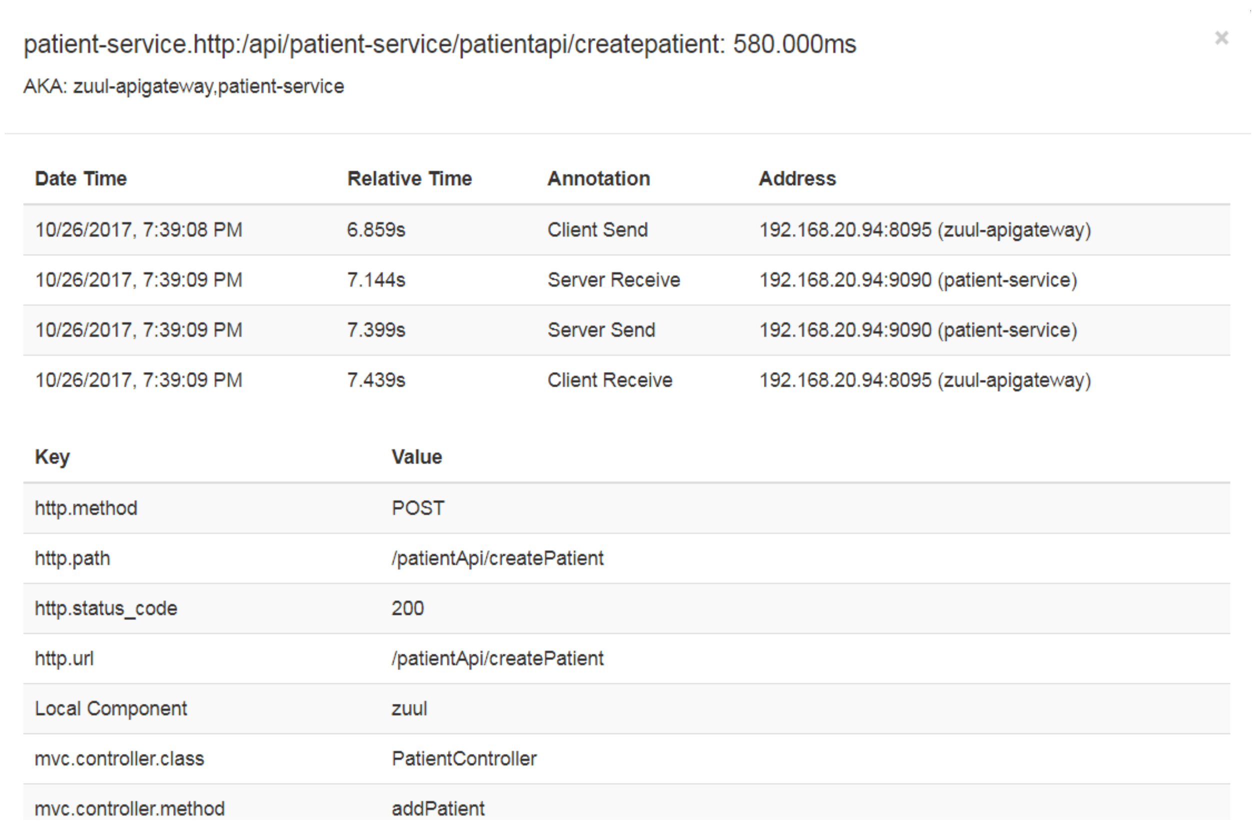
zuul-apigateway.http:/api/patient-service/patientapi/createpatient: 621.000ms

AKA: doctor-service,zuul-apigateway

Date Time	Relative Time	Annotation	Address
10/26/2017, 7:39:08 PM	6.835s	Client Send	192.168.20.94:7070 (doctor-service)
10/26/2017, 7:39:08 PM	6.859s	Server Receive	192.168.20.94:8095 (zuul-apigateway)
10/26/2017, 7:39:09 PM	7.449s	Server Send	192.168.20.94:8095 (zuul-apigateway)
10/26/2017, 7:39:09 PM	7.456s	Client Receive	192.168.20.94:7070 (doctor-service)

Key	Value
http.host	IM-RT-LP-143.innomindshyd.com
http.method	POST
http.path	/api/patient-service/patientApi/createPatient
http.url	http://IM-RT-LP-143.innomindshyd.com:8095/api/patient-service/patientApi/createPatient
spring.instance_id	IM-RT-LP-143.innomindshyd.com:doctor-service:7070
spring.instance_id	IM-RT-LP-143.innomindshyd.com:zuul-apigateway:8095

Step-4: now click on patient-service, we could see the below logs



patient-service.http:/api/patient-service/patientapi/createpatient: 580.000ms

AKA: zuul-apigateway,patient-service

Date Time	Relative Time	Annotation	Address
10/26/2017, 7:39:08 PM	6.859s	Client Send	192.168.20.94:8095 (zuul-apigateway)
10/26/2017, 7:39:09 PM	7.144s	Server Receive	192.168.20.94:9090 (patient-service)
10/26/2017, 7:39:09 PM	7.399s	Server Send	192.168.20.94:9090 (patient-service)
10/26/2017, 7:39:09 PM	7.439s	Client Receive	192.168.20.94:8095 (zuul-apigateway)

Key	Value
http.method	POST
http.path	/patientApi/createPatient
http.status_code	200
http.url	/patientApi/createPatient
Local Component	zuul
mvc.controller.class	PatientController
mvc.controller.method	addPatient

<u>The entire flow of above screen shots is explained below:</u>

With above request we are trying to registering the patient by the doctor. But doctor services were protected and hence accessed only behind the api gateway. Hence the first call went api gate way, then gateway scans the registry for the service and fetch them and then the doctor service will be invoked. In doctor service createPatinet method, we are calling patient-service which is also protected and can only be accused behind the api gate way, and hence the call again comes to api gateway and from there patient-service will be called. This complete flow will be monitored in Zipkin dashboard as shown above screen shots.