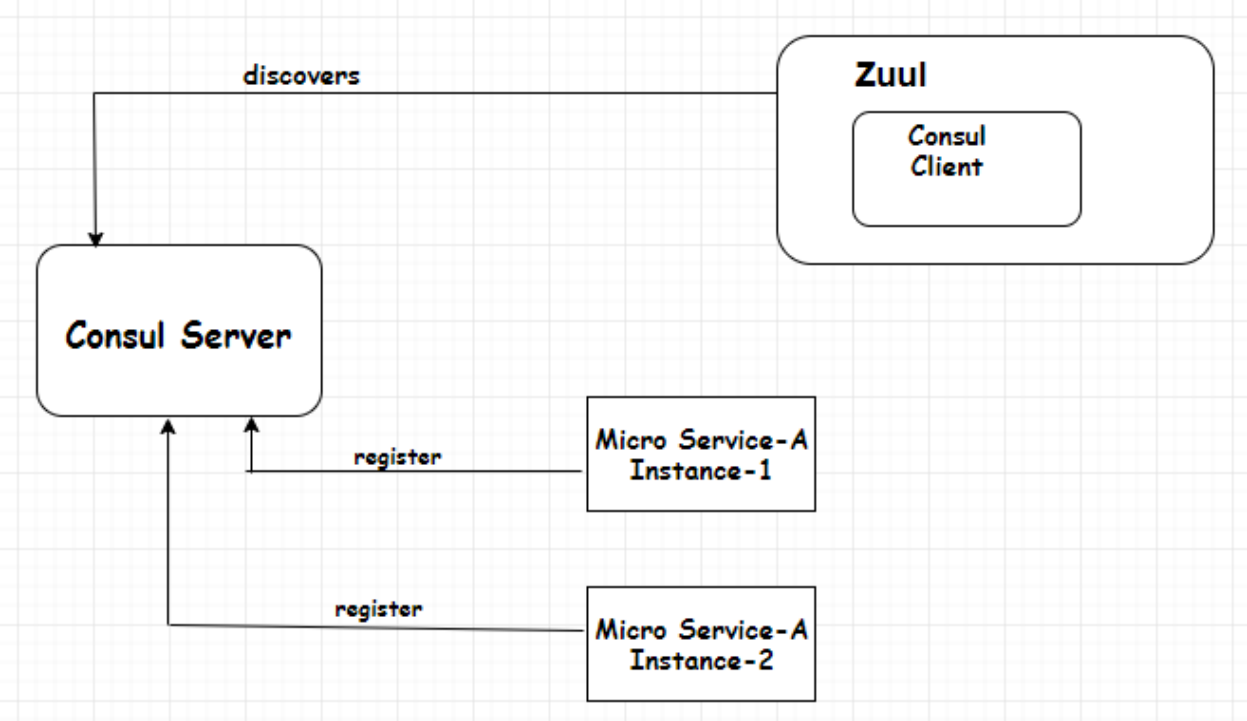
Zuul proxy as the API gateway

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.



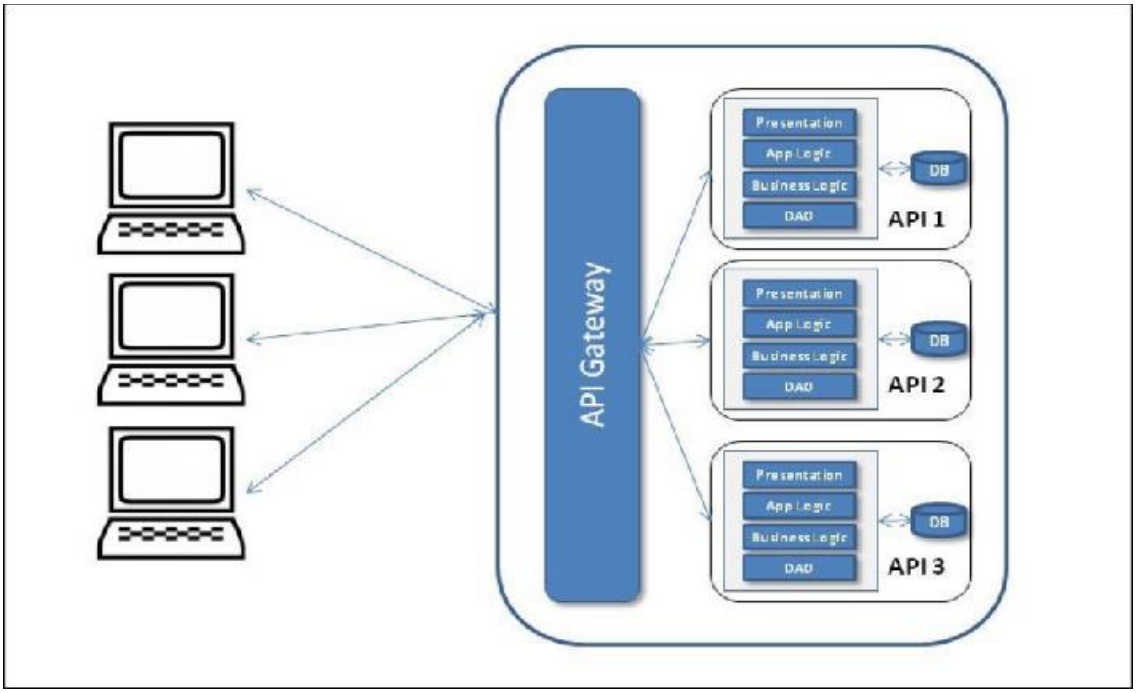
The Zuul proxy internally uses the discovery server for service discovery, and Ribbon for load balancing between service instances.

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.

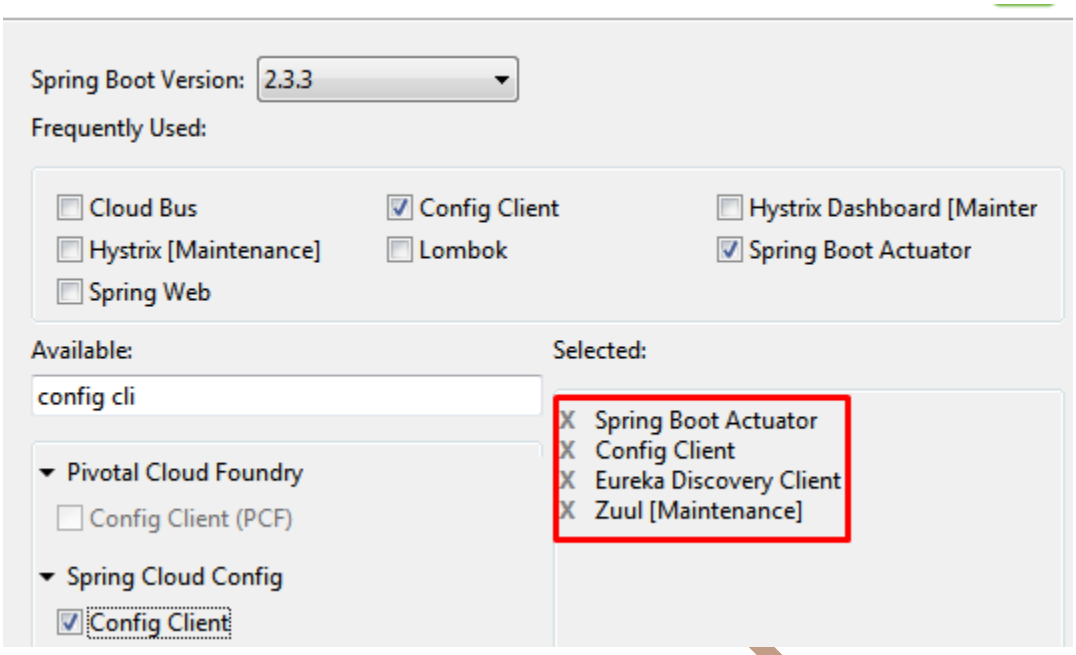
A response may require fetching information from two or more services:



Setting up Zuul

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.

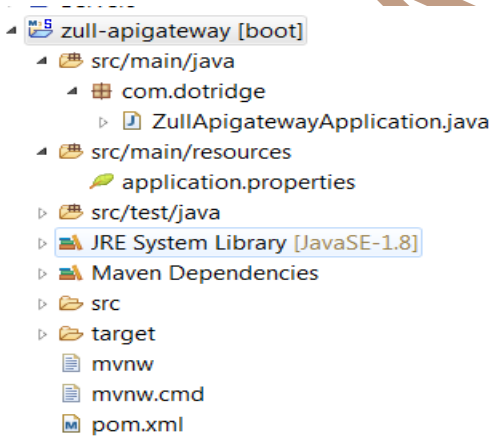
Step-1: Create a new Spring Starter project with name zuul-apigateway, and select Zuul, Config Client, Actuator, and Eureka Discovery



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:



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.

```
spring.cloud.config.uri=http://localhost:8888/config-server
```

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

```
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/**
```

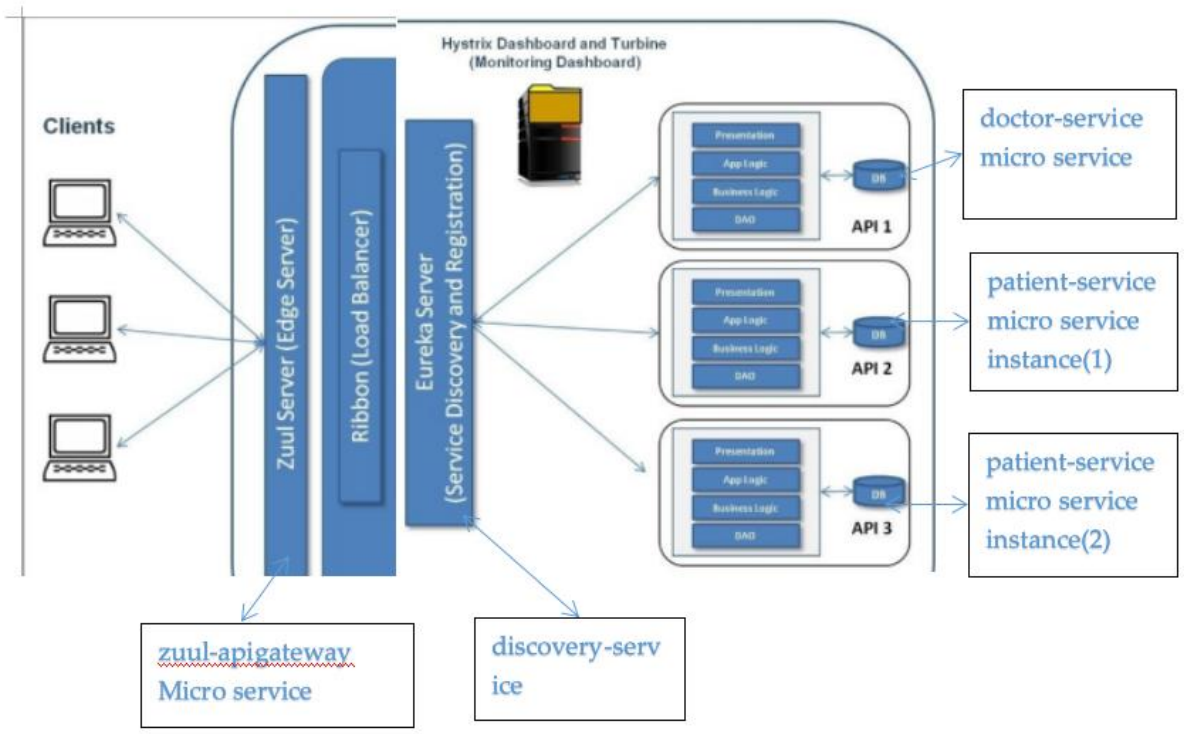
Step-4: update the application.properties file with the zuul proxy configurational parameters as and service id, port as shown below:

PatientServi... | application... | HTTP 404 No... | ZullApigatew... | 14

```
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



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

```
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);
    }
}
```

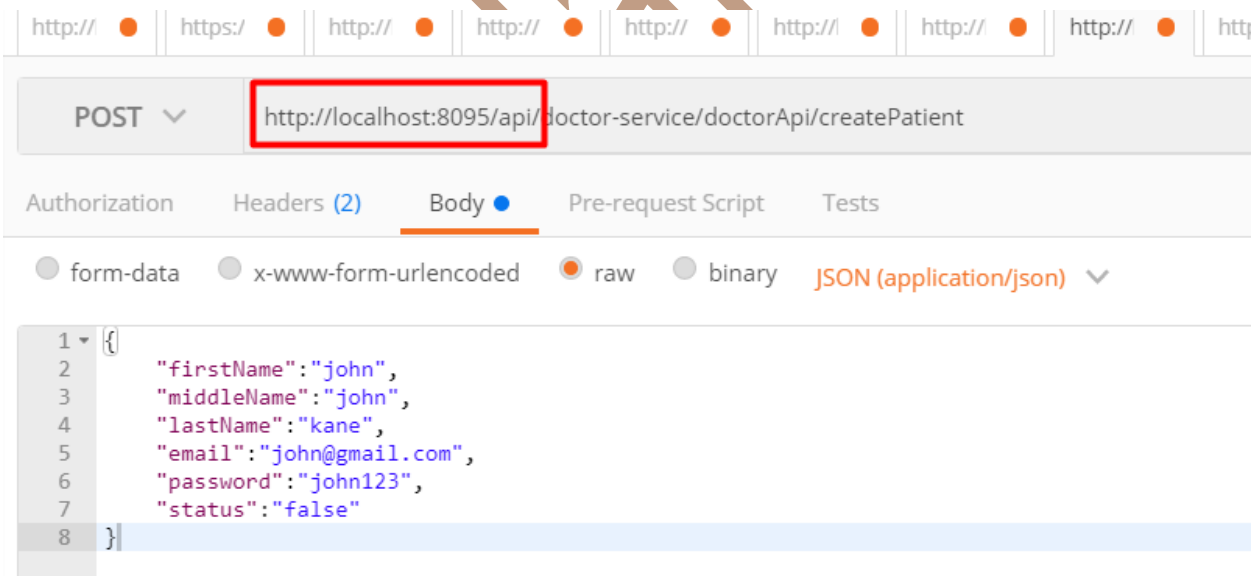
Here we were also added an annotation **@EnableDiscoveryClient,** because zuul proxy will be the client for discovery server I.e.eureka, to get the list of registered micro services.

Step-6: 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:

```
PatientServi...  PatientServi...  application...  bootstrap.p...  HTTP 404 No...  ZullApigatew...

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:



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.DomainExtractingServerLis**t@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.