<u>Microservices</u>

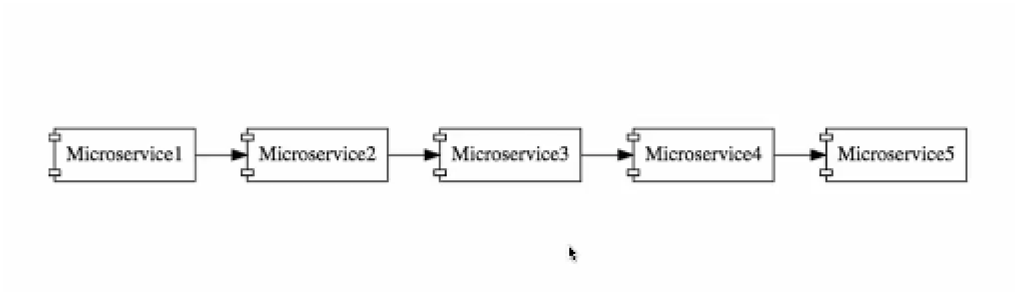
Small autonomous services that work together. -- **Sam Newman**

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. --- **Martin Fowler**

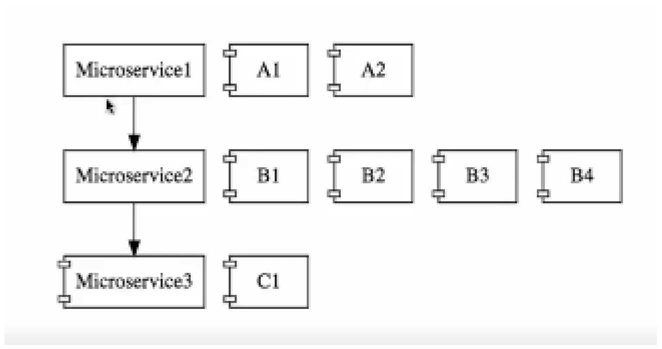
Important things for microservices :

1) REST

2) & Small Well Chosen Deployable Units (With very well thought out boundaries)



3) & Cloud Enabled (different no of instances of different microservices) – It means if the load is more on microservices3 (for ex), we should be able to bring a new instance of microservices3. Or we can also take down an instance without having huge problem.



challenges in building microservices:

1) Bounded Context

How do you identify each of the micro-services? Deciding the boundaries of microservices is an evolutionary process.

2) Configuration Management

Maintaining configuration for different environments (DEV, QA, STAGE, PROD) for all the microservices.

3) Dynamic scale up and scale down

The loads might be different on microservices at different instance of time. Load balancing among all the instances of the microservices.

4) Visibility and Monitoring

The functionality is distributed among the 10 microservices. If there is a bug, how do you identify where it is? You need to have a centralized log that says what happened with a particular request. Monitoring of microservices for ex: disk space, which microservice is down etc.

5) Pack of cards

If the microservices architecture is not well designed, it can be like pack of cards. If the fundamental microservice goes down, the entire application may go down. Therefore it is very important to have fault tolerance in microservice.

Spring Cloud: Solves the challenges associated with microservices implementation.

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns. They will work well in any distributed environment, including the developer's own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.

Spring Cloud is not one project. There are wide varieties of product under the umbrella of Spring Cloud.

One of the important project in Spring Cloud is 'Spring Cloud Netflix'. There are wide range of components that Netflix has open-sourced under the project 'Spring Cloud Netflix'.

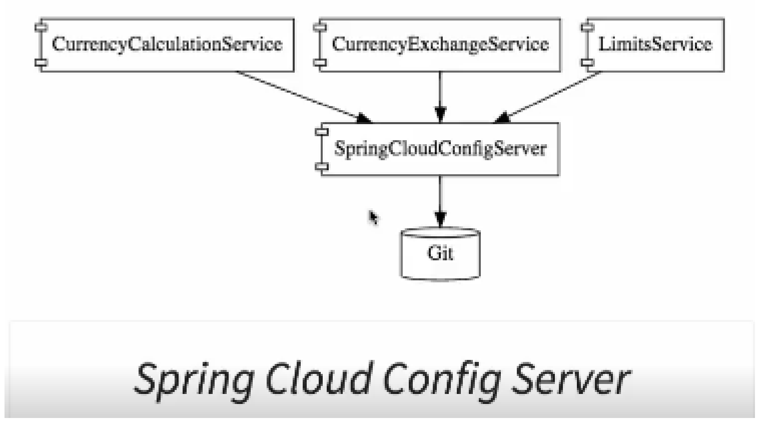
'Spring Cloud Config' is to provide the centralized configuration management.

'Spring Cloud bus' enables the microservices and infrastructure components like config server, API gateway to talk to each other.

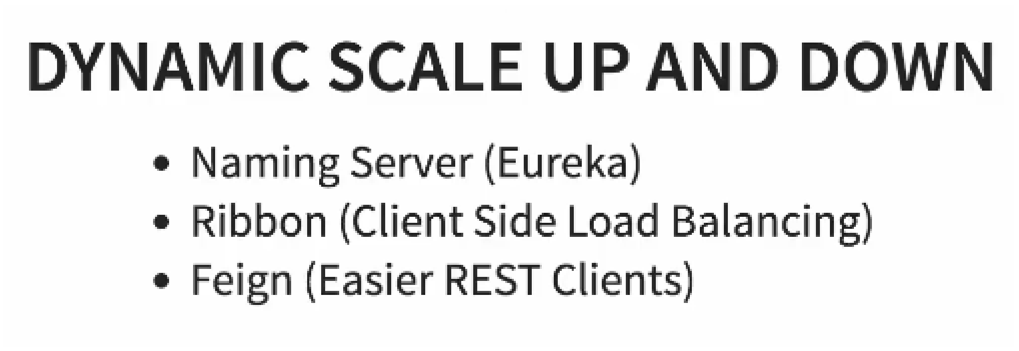
Finchley M2 release of Spring Cloud.

This provides solution to all the challenges mentioned above.

1) Configuration Management --> Spring Cloud Config Server (uses centralized config management) is used to expose all the configuration (stored in git repo) to all the microservices.



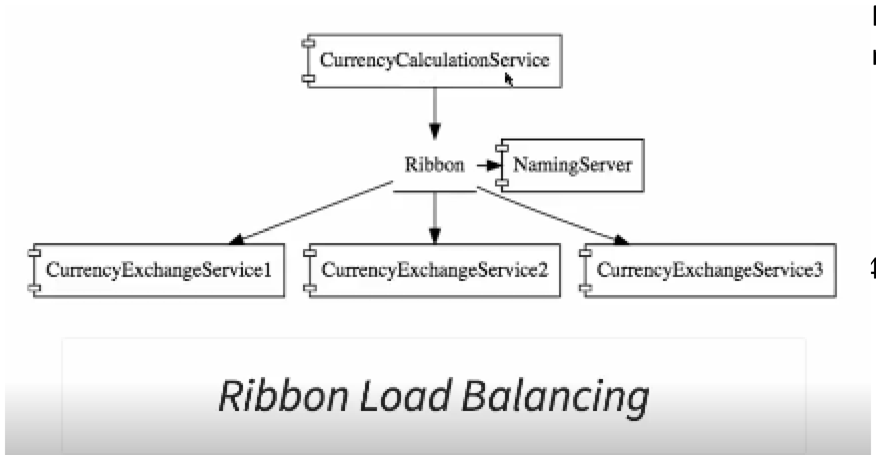
2) Dynamic scale up and scale down --> Naming server (Eureka).
Ribbon for client side load balancing.
Feign (easier REST clients) as a mechanism to write simple rest client


DYNAMIC SCALE UP AND DOWN

- Naming Server (Eureka)
- Ribbon (Client Side Load Balancing)
- Feign (Easier REST Clients)

Naming Server --> All the instances of all the microservices will register themselves with the naming server. Naming server has 2 imp features:
1) Service Registration
2) Service Discovery (in the below ex, CurrencyCalculationService can ask the Eureka NamingServer to get the instances of CurrencyExchangeService and NamingServer will provide the URL of CurrencyExchangeService to the CurrencyCalculationService). This helps in establishing the dynamic relationship between CurrencyCalculationService and the instances of CurrencyExchangeService.

Ribbon --> The CurrencyCalculationService will host Ribbon and it will make sure that the load is evenly distributed among the existing instances that it gets from the NamingServer.



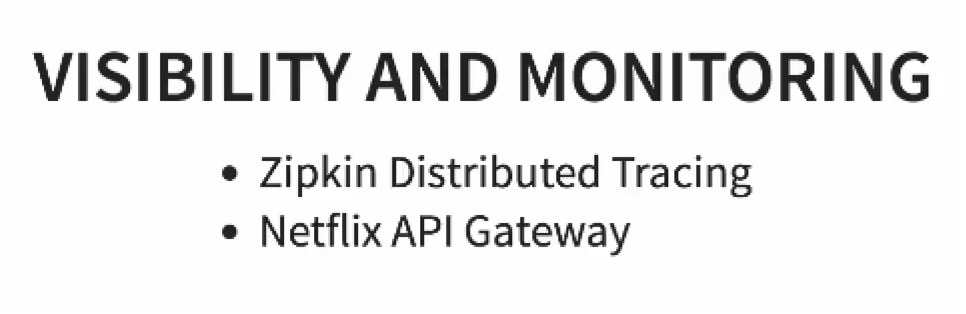
Ribbon Load Balancing

Feign --> in CurrencyCalculationService as a mechanism to write simple restful clients.

3) Visibility and Monitoring

Zipkin Distributed tracing server -> to trace request across multiple components

Netflix API gateway -> for common cross cutting concerns implementation



VISIBILITY AND MONITORING

- Zipkin Distributed Tracing
- Netflix API Gateway

Spring Cloud Sleuth --> to assign an ID to a request across multiple components.

Zipkin Distributed Tracing Server --> To trace a request across multiple components.

Microservices have some common features such as, logging, security, analytics etc. You do not want to implement these common features in every microservice. API Gateway provides solutions to these challenges. We will use Netflix Zool Gateway API.

Fault Toelrance will be implemented using Hystrix. If a service is down, Hystrix helps us in setting up a default response.

4) Fault tolerance

Hystrix --> If a service is down. Hystrix helps in configuring the default response.

Microservice Advantages:

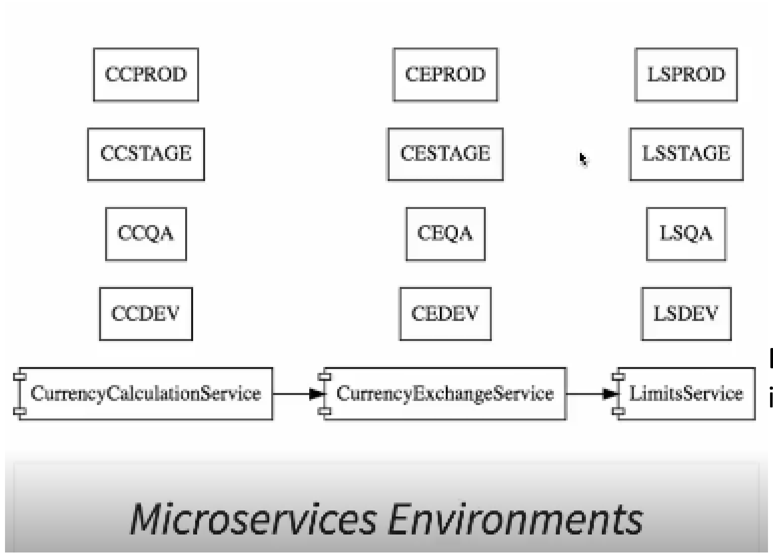
1) New technology and process adoption
   Each of the microservices can be built in different technology. For the new microservice we build we can bring in new processes

2) Dynamic scaling -> scaling based on load. Cloud enabled microservice can be scald up/down dynamically.

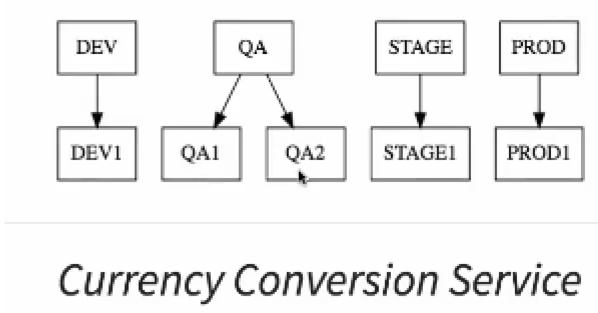
3) Faster release cycles --> bcz of smaller components it is much easier to release microservices compared to monolith applications. You can bring new feature faster to market.

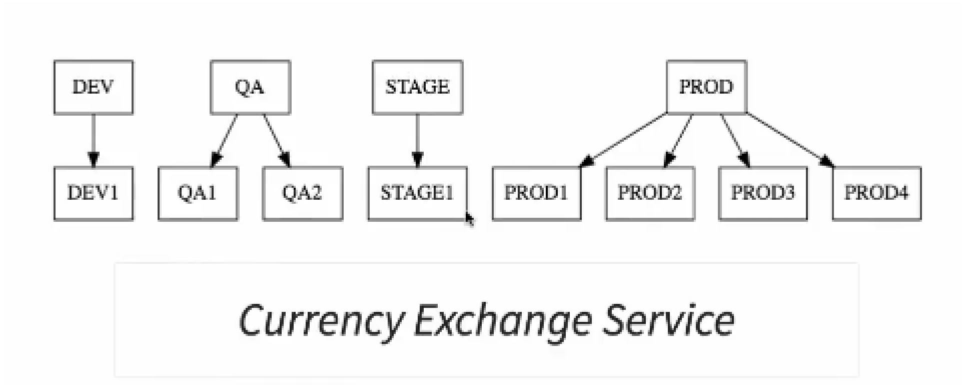
Components – Standardizing Ports and URL

Centralized application configuration:

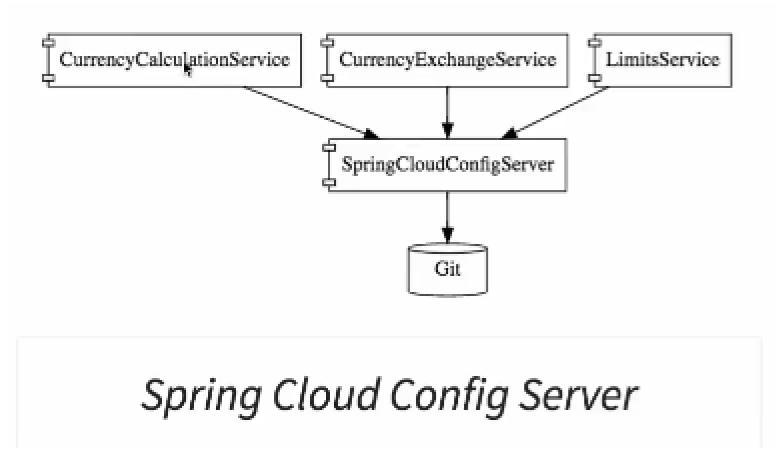


Different microservices may have different no of instances for a particular environment.





Put all the configurations into a git repository and connect SpringCloudConfigServer to the git repository. SpringCloudConfigServer will serve the configurations based on the different needs of the different instances in different environments for all the microservices. SpringCloudConfigServer will act as a centralised microservice configuration application.



Example configurations for ‘limits-service’ module :
limits-service-dev.properties
limits-service-qa.properties

Step1 – Part 2 – Setting up Limits Microservices

limits-service

dependencies: –

1) web -- For web aplications . Bcz we have web dependency on our classpath, the project will launch as a war inside tomcat.
2) devTools – to pick up the changes without restarting the tomcat
3) Actuator – Provides the monitoring capability around the application
4) config Client-- ‘Spring cloud config client’ to connect to springcloudconfigserver

----

Step 02 - Creating a hard coded limits service

Step 06 - Creating Local Git Repository

1) Create a new git repository (here git-localconfig-repo) .

2) Go to spring-cloud-config-server and add a link to the git folder. Right click Build Path Add source
3) Store all the config files (for ex; limits-service.properties). Commit the external git repository after you add a properties file.
4) Access those config files through spring-cloud-config-server.

Step 07 - Connect Spring Cloud Config Server to Local Git Repository

1) Copy the location of the github repository
2) Go to application.properties of the spring-cloud-config-server
3) spring.cloud.config.server.git.uri=file:///E:/workspace/git-localconfig-repo (This repository can be local as well as remote)
   file:// If it is a local git repository
4) http://localhost:8888/limits-service/default This url for spring-cloud-config-server throws error. Why?
   Limits-service Name of the configuration

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Jul 14 19:06:12 IST 2018
There was an unexpected error (type=Not Found, status=404).
No message available

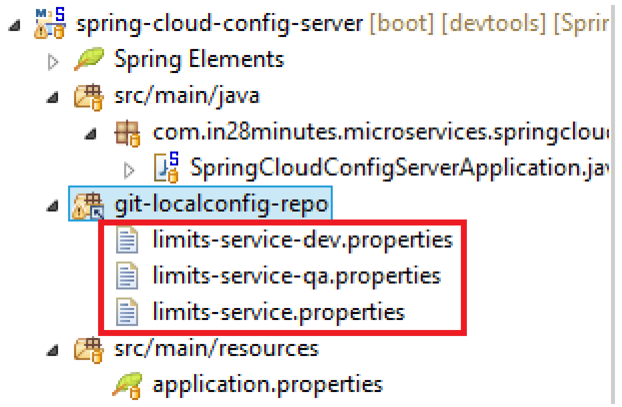
Answer: Whenever we create a spring-cloud-config-server, you need to enable it. Go to the SpringCloudConfigServerApplication class and annotate it with @EnableConfigServer.

Note: SpringCloudConfigServerApplication stores configuration for multiple projects and their different environments.

Step 08 - Configuration for Multiple Environments in Git Repository

Storing configurations for dev, qa, stage environment in got repository for limits-service.
Create 2 properties file as below:
limits-service-qa.properties
limits-service-dev.properties

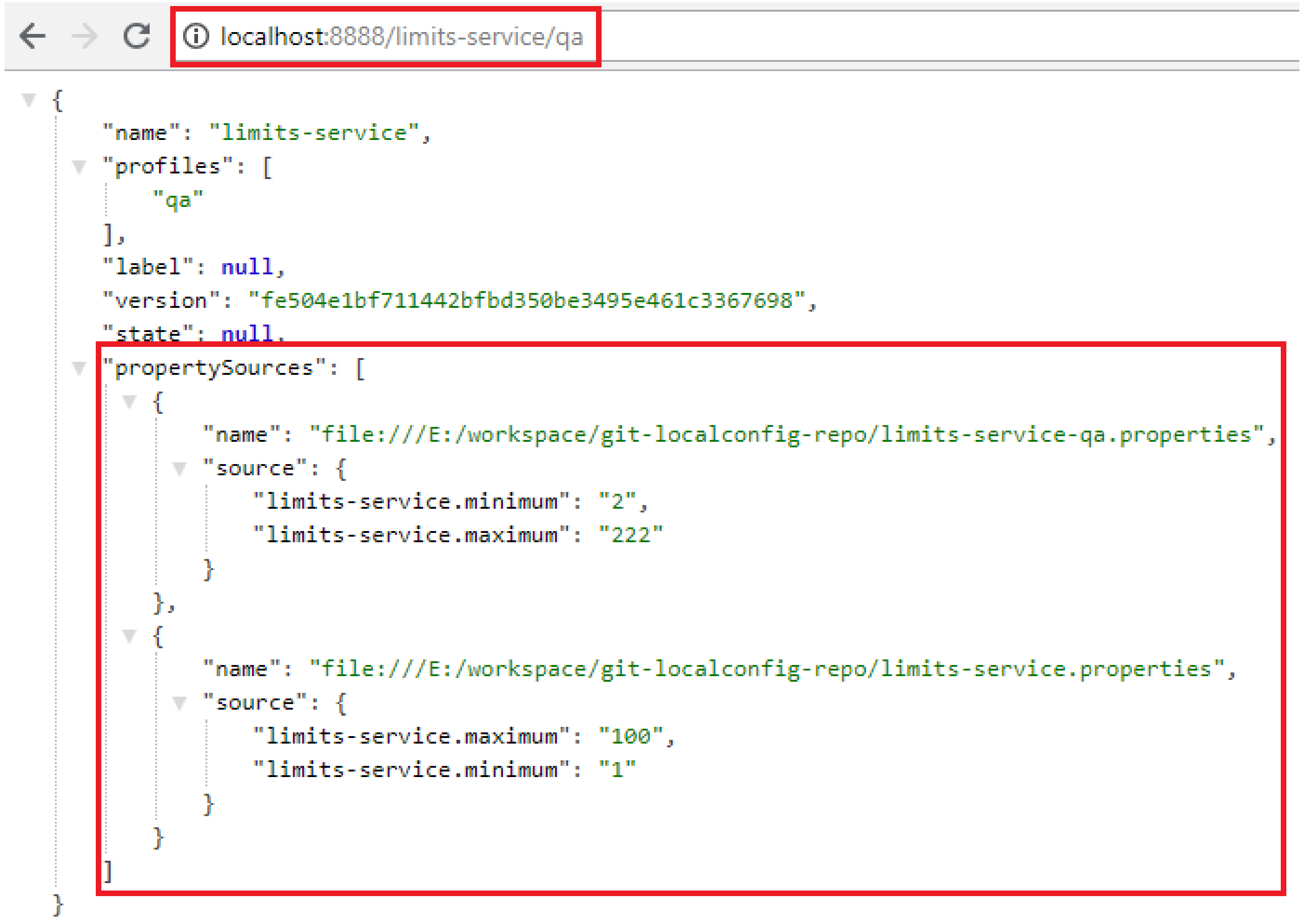
The above 2 configuration files will override the default 'limits-service.properties' for specific environment. You also have the flexibility to override the selected values.



```
spring-cloud-config-server [boot] [devtools] [Sprir
  Spring Elements
  src/main/java
    com.in28minutes.microservices.springclou
      SpringCloudConfigServerApplication.ja
  git-localconfig-repo
    limits-service-dev.properties
    limits-service-qa.properties
    limits-service.properties
  src/main/resources
    application.properties
```

Qn: How to access these 2 (dev & qa) properties in browser?

Ans: http://localhost:8888/limits-service/dev , http://localhost:8888/limits-service/qa


localhost:8888/limits-service/qa

```
{
    "name": "limits-service",
    "profiles": [
        "qa"
    ],
    "label": null,
    "version": "fe504e1bf711442bfbd350be3495e461c3367698",
    "state": null,
    "propertySources": [
        {
            "name": "file:///E:/workspace/git-localconfig-repo/limits-service-qa.properties",
            "source": {
                "limits-service.minimum": "2",
                "limits-service.maximum": "222"
            }
        },
        {
            "name": "file:///E:/workspace/git-localconfig-repo/limits-service.properties",
            "source": {
                "limits-service.maximum": "100",
                "limits-service.minimum": "1"
            }
        }
    ]
}
```

The thing to note in the above screen shot is that the lists of properties are in order of priority.

---

16. Step 09 - Connect Limits Service to Spring Cloud Config Server

We would like limits-service to pick up the configuration values from spring-cloud-config-server. In order to do we need to do the following:

1) Rename the application.properties to bootstrap.properties files for limits-service.
2) Configure url for spring-cloud-config-server in bootstrap.properties of limits-service.

   spring.cloud.config.uri=http://localhost:8888

   Note: By default, the configuration picked is limits-service.properties (default profile).

----

17. Step 10 - Configuring Profiles for Limits Service

Now, all the configurations for limits-service is coming from git repository.

Go to limits-service and add the following property to bootstrap.properties. *spring.profiles.active=dev*

You can also set the profiles using vm arguments or java aaplication arguments.

Note: If you make any change in configuration files of spring-cloud-config-server, you need to do 2 things for those changes to be refleaced into limits-service url.

1) Commit the changes in the git repository
2) Restart limits-service. Bcz , limits-service picks value from the current state of spring-cloud-config-server when limits-service is started, so if you make any change to properties files in git repository, you have to restart the limits-service.

Note: We can make use of 'refresh url' concept in order for limits-service to automatically pick up the changes values.

----

18. Step 11 - A review of Spring Cloud Config Server

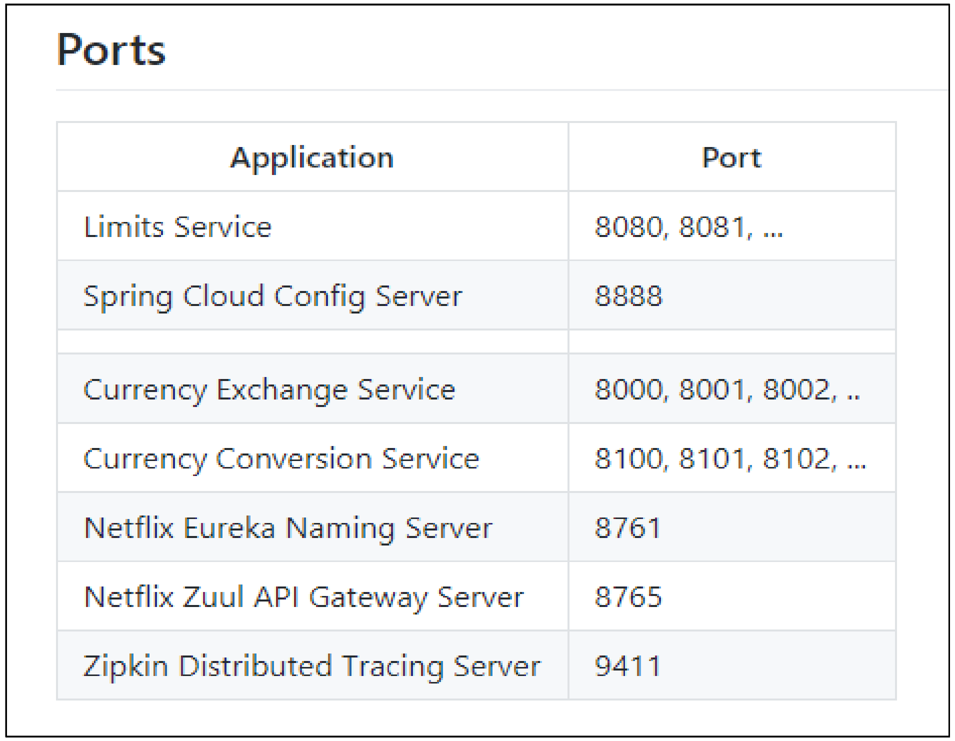
Spring Cloud Config Server is used to manage all the configurations related to all the microservices.

----

19. Step 12 - Introduction to Currency Conversion and Currency Exchange Microservice

20. Step 13 - Setting up Currency Exchange Microservice

currency-exchange-service



Ports

Application	Port
Limits Service	8080, 8081, ...
Spring Cloud Config Server	8888
Currency Exchange Service	8000, 8001, 8002, ..
Currency Conversion Service	8100, 8101, 8102, ...
Netflix Eureka Naming Server	8761
Netflix Zuul API Gateway Server	8765
Zipkin Distributed Tracing Server	9411

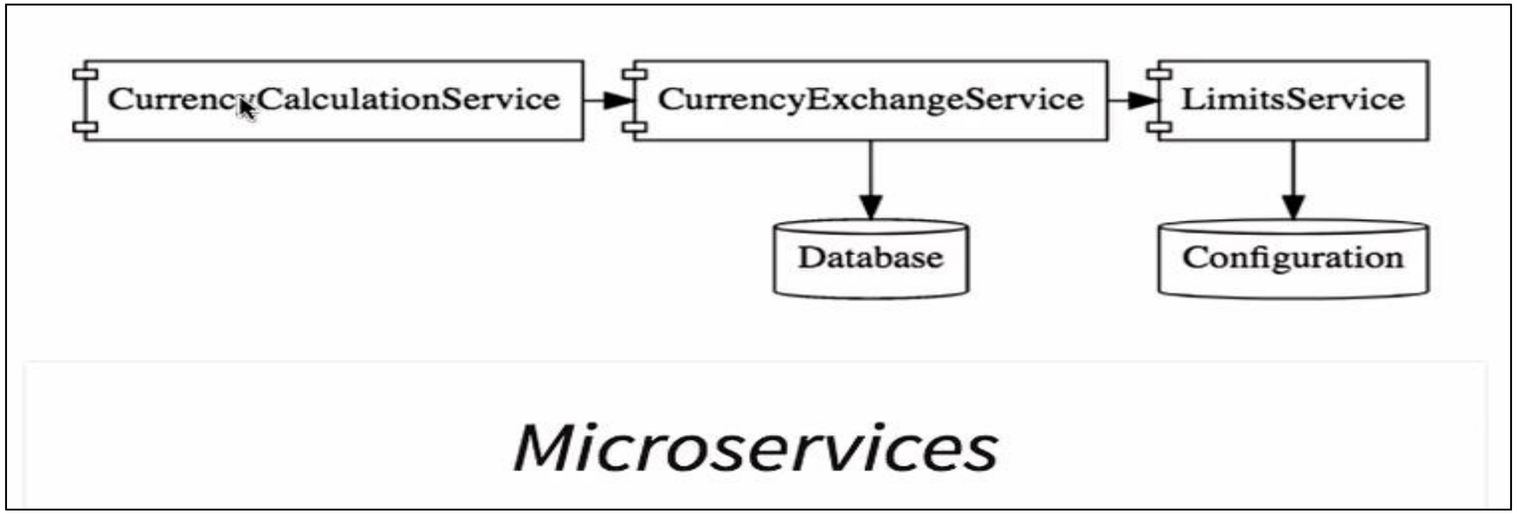
-- **21. Step 14 - Create a simple hard coded currency exchange service**

http://localhost:8000/currency-exchange/from/USD/to/INR

```
{
    "id": 1000,
    "from": "USD",
    "to": "INR",
    "conversionMultiple": 65
}
```

---

22. Step 15 - Setting up Dynamic Port in the Response



Currency-calculation-service will use currency-exchange-service to get the exchange value.

Adding a port to the currency-exchange-service response. -Dserver.port=8000 (vm argument) will override the configuration in application.properties.

---

23. Step 16 - Configure JPA and Initialized Data

To add an in-memory database, below dependencies are to be added :

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
	<groupId>com.h2database</groupId>
	<artifactId>h2</artifactId>
</dependency>
```

Now, define entities.

```
@Entity  // corresponds to database table
public class ExchangeValue {

	@Id
	private Long id;
}
```

Now, initialise some data into database. Create sql file. Data.sql.

To see the sql command use and enable the h2 console use:

```
spring.jpa.show-sql=true
spring.h2.console.enabled=true
```

Note: The fields of the entity class in java should not be a keyword in the used database.

http://localhost:8000/h2-console --> URL to access the H2 database console

Tip: Make sure that you use jdbc:h2:mem:testdb as JDBC URL.

Step 17 - Create a JPA Repository

Create ExchangeValuerepositorty.java.

```
public interface ExchangeValueRepository extends JpaRepository<ExchangeValue, Long>{

        ExchangeValue findByFromAndTo(String from, String to);

}
```

JpaRepository<ExchangeValue, Long> --> JpaRepository<What is the type of entity that it manages, What is the type of the ID filter>

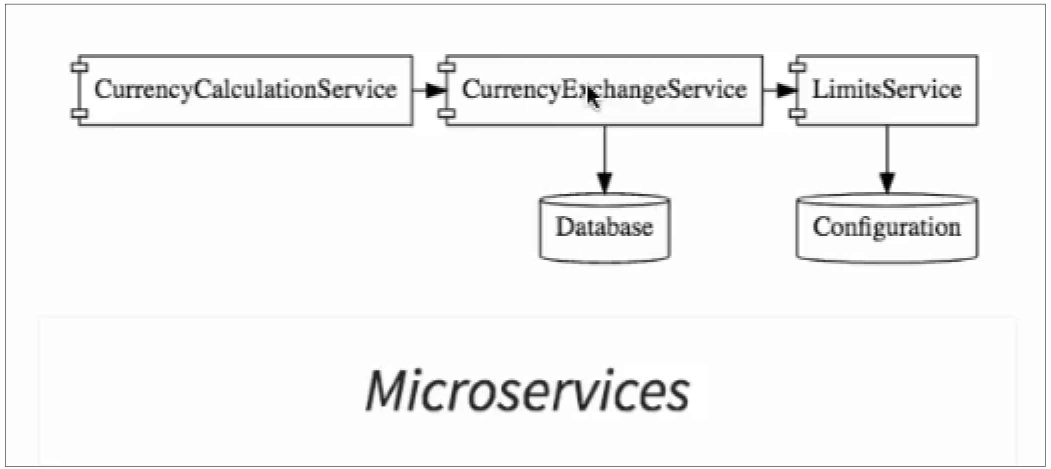
Apart from the default methods provided by spring data JPA, we can also create our own query methods.
Q.) How to write query method?

Ans: ReturnType findBy{Column1}and{Column2} (Datatype Column1, Datatype Column2); --> Just by declaring this method, Spring data JPA will provide the implementation.

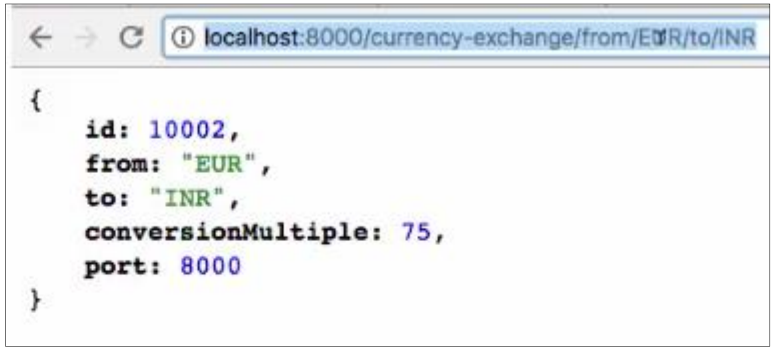
Step 18 - Setting up Currency Conversion Microservice

```
spring.application.name=currency-conversion-service
server.port=8100
```

Step 19 - Creating a service for currency conversion

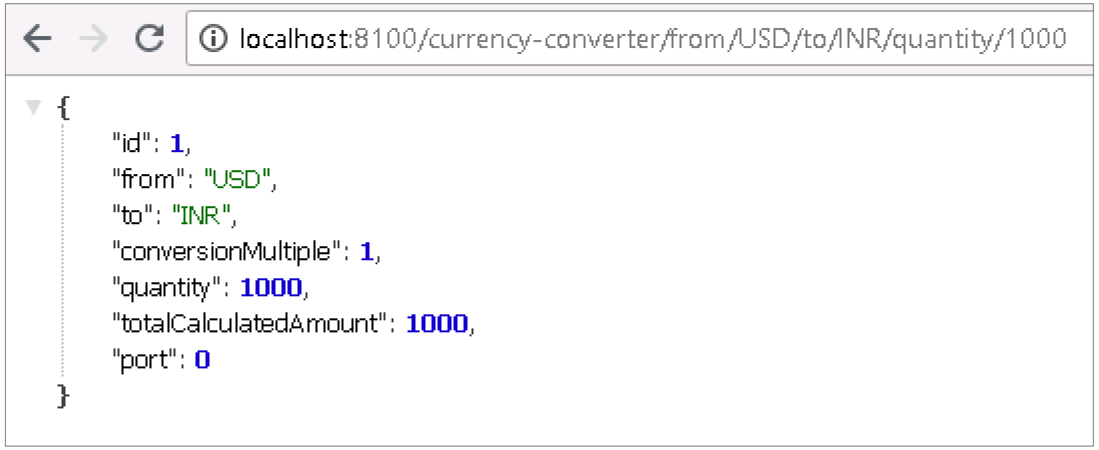


Right now, Currency-exchange-service returns the conversionMultiple.



```
localhost:8000/currency-exchange/from/EUR/to/INR
{
    id: 10002,
    from: "EUR",
    to: "INR",
    conversionMultiple: 75,
    port: 8000
}
```

The new microservice 'currency-conversion-service' will convert a given amount from one currency to other currency. Currency-conversion-service will in-turn use currency-exchange-service to get the conversionMultiple.



```
localhost:8100/currency-converter/from/USD/to/INR/quantity/1000
{
    "id": 1,
    "from": "USD",
    "to": "INR",
    "conversionMultiple": 1,
    "quantity": 1000,
    "totalCalculatedAmount": 1000,
    "port": 0
}
```

Step 20 - Invoking Currency Exchange Microservice from Currency Conversion Microservice

We will invoke currency-exchange-service from currency-conversion-service.

```
package com.in28minutes.microservices.currencyconversionservice;

import java.math.BigDecimal;
import java.util.HashMap;
import java.util.Map;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
public class CurrencyConversionController {

	@GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}")
	public CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to,
			@PathVariable BigDecimal quantity) {

		Map<String, String> uriVariables = new HashMap<>();
		uriVariables.put("from", from);
		uriVariables.put("to", to);
		ResponseEntity<CurrencyConversionBean> responseEntity = new RestTemplate().getForEntity(
				"http://localhost:8000/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.class,
                    uriVariables);
            CurrencyConversionBean response = responseEntity.getBody();

            return new CurrencyConversionBean(response.getId(), from, to, response.getConversionMultiple(), quantity,
                    quantity.multiply(response.getConversionMultiple()), response.getPort());
        }

}
```

Step 21 - Using Feign REST Client for Service Invocation

Feign --> Spring Cloud component.

Using RestTemplate we had to write lot of code even to access a simple URL. In microservice world, we have to depend on lot of rest URL calls. Hence, we need a simpler method to call a rest URL. Feign solves this problem. This makes it easy to call other restful microservices.

Feign also provides integration with *Ribbon,* which is a client side load balancing framework.

Steps to use Feign:
1) Add dependency for Feign.

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

2) Enable feign to scan for clients

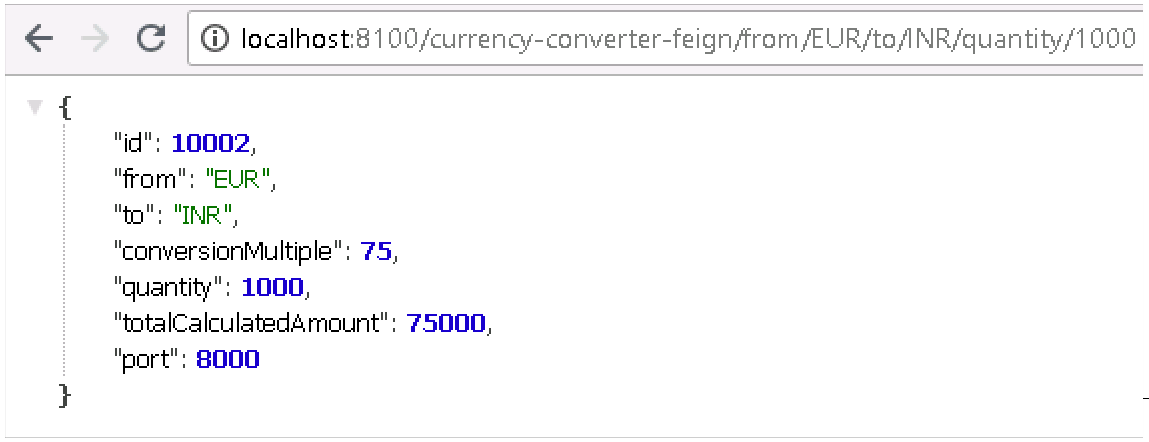
```
    @EnableFeignClients("Pass here the packages that Feign needs to scan")
```

3) Just like we need a repository to talk to JPA, we need to create a Feign proxy in order to talk to an external microservice. We are going to invoke currency exchange service.

@FeignClient --> This is a Feign client and this is going to use feign to talk to external microservice.
@FeignClient(name="Name of the service which we are going to call", url="The port where service is deployed")

```
@FeignClient(name="currency-exchange-service", url="localhost:8000")
public interface CurrencyExchangeServiceProxy {

	// Define a method to talk to the other microservice
	//@GetMapping("/currency-exchange/from/{from}/to/{to}")
	public CurrencyConversionBean retrieveExchangeValue
		(@PathVariable("from") String from, @PathVariable("to") String to);
}
```

Note: @PathVariable("**from**") String from, @PathVariable("**to**") String to --> We need to use **from** and **to** as the parameter, otherwise we will get the error- *PathVariable annotation was empty on param 0*.



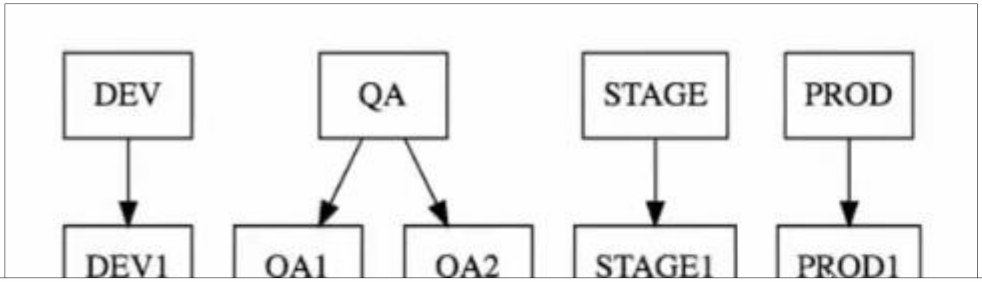
localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000

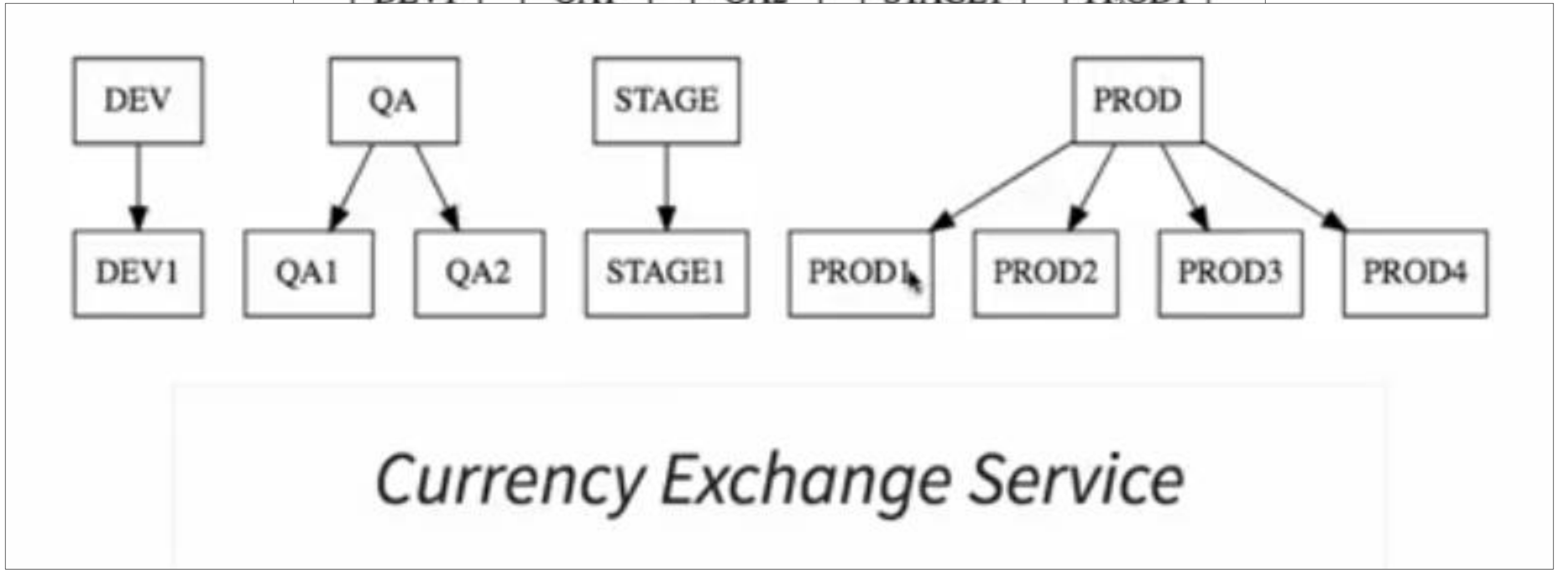
```
{
    "id": 10002,
    "from": "EUR",
    "to": "INR",
    "conversionMultiple": 75,
    "quantity": 1000,
    "totalCalculatedAmount": 75000,
    "port": 8000
}
```

Step 22 - Setting up client side load balancing with Ribbon

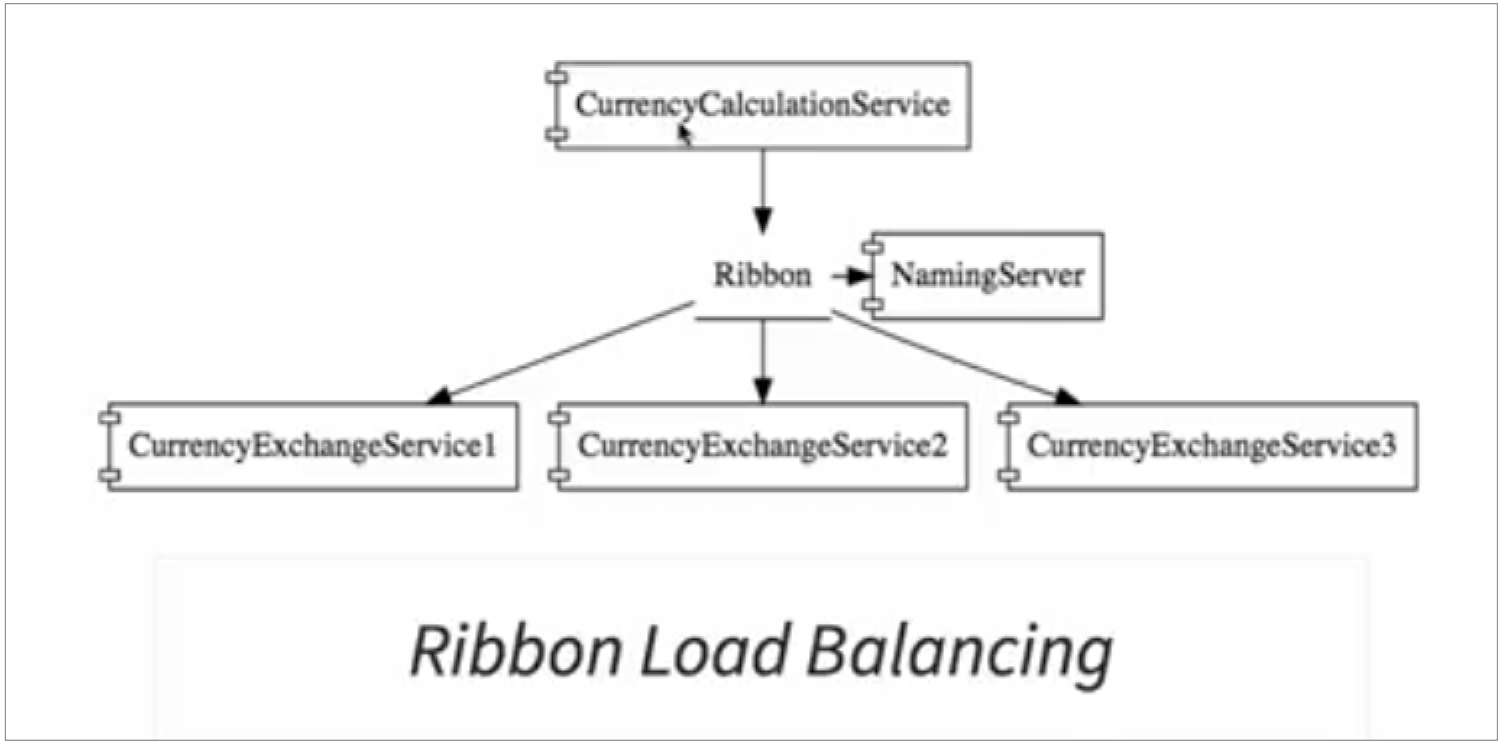
Right now, we are hard coding the url of currency-exchange-service inside currency-conversion-service.
@FeignClient(name="currency-exchange-service", url="**localhost:8000**")

```
public interface CurrencyExchangeServiceProxy {
}
```





We can move this url configuration to properties file but the problem is that currency-conversion-service can only talk to the one instance of currency-exchange-service (it may be PROD1, PROD2....). We want to make currency-conversion-service capable of talking to all the instances of currency-exchange-service. We will want to distribute the load between all the 4 instances of currency-exchange-service. Ribbon is the solution for this.



We are using Feign to call currency-exchange-service. Ribbon can make use of Feign configuration that we have already done and help us distribute the call among the different instances of the currency-exchange-service.

We will enable Ribbon on currency-conversion-service. Steps to do so:

1) pom.xml of currency-conversion-service.

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

2) Enable Ribbon on the proxy (Use @RibbonClient(“Name of the application”))

//@FeignClient(name="currency-exchange-service", url="localhost:8000") -->url attribute is no more required as we do not want to talk to only one instance, rather we would like to distribute the load among all the instances. So the new configuration is as below. The url's will be configured in application.properties.

```
@FeignClient(name="currency-exchange-service")
@RibbonClient(name="currency-exchange-service")
public interface CurrencyExchangeServiceProxy {

	// Define a method to talk to the other microservice
	//@GetMapping("/currency-exchange/from/{from}/to/{to}")
	public CurrencyConversionBean retrieveExchangeValue
		(@PathVariable("from") String from, @PathVariable("to") String to);
}
```

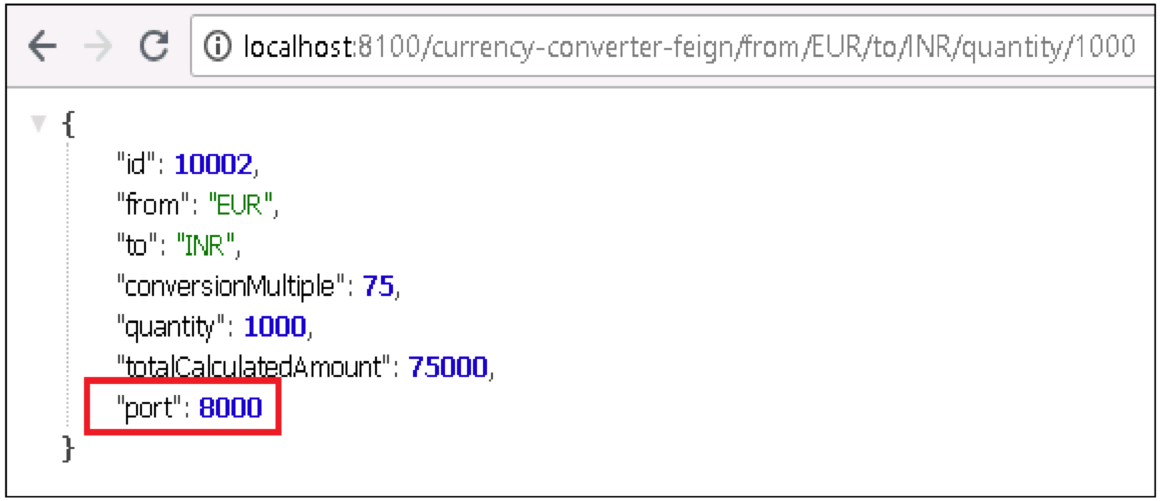
3) Configure what URL's the load for currency-exchange-service has to be distributed among. Go to application.properties of currency-conversion-service.

```
	currency-exchange-service.ribbon.listOfServers=http://localhost:8000, http://localhost:8001
```

----

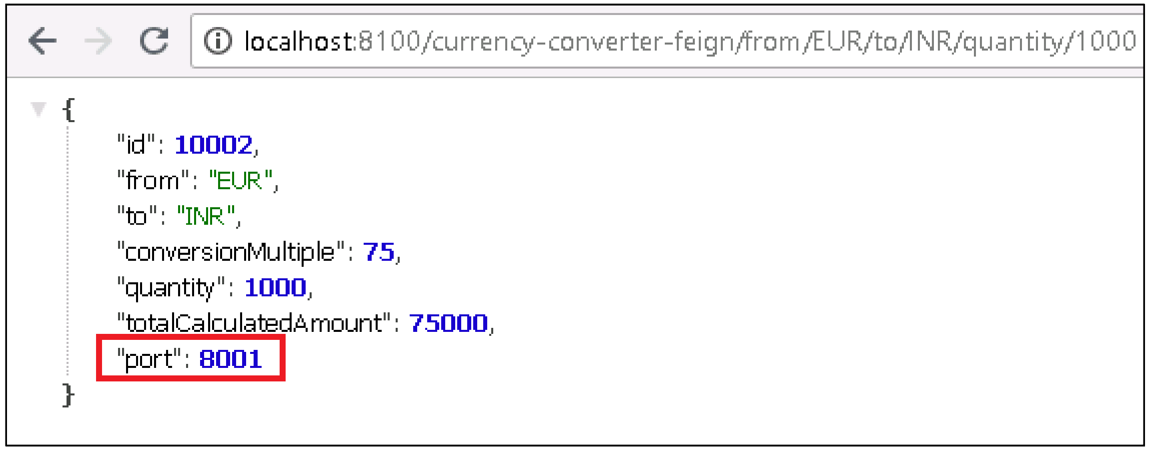
30. Step 23 - Running client side load balancing with Ribbon

Now the response of the url " http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000" will come from both of the instances of the currency-exchange-service.



localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000

```
{
    "id": 10002,
    "from": "EUR",
    "to": "INR",
    "conversionMultiple": 75,
    "quantity": 1000,
    "totalCalculatedAmount": 75000,
    "port": 8000
}
```



localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000

```
{
    "id": 10002,
    "from": "EUR",
    "to": "INR",
    "conversionMultiple": 75,
    "quantity": 1000,
    "totalCalculatedAmount": 75000,
    "port": 8001
}
```

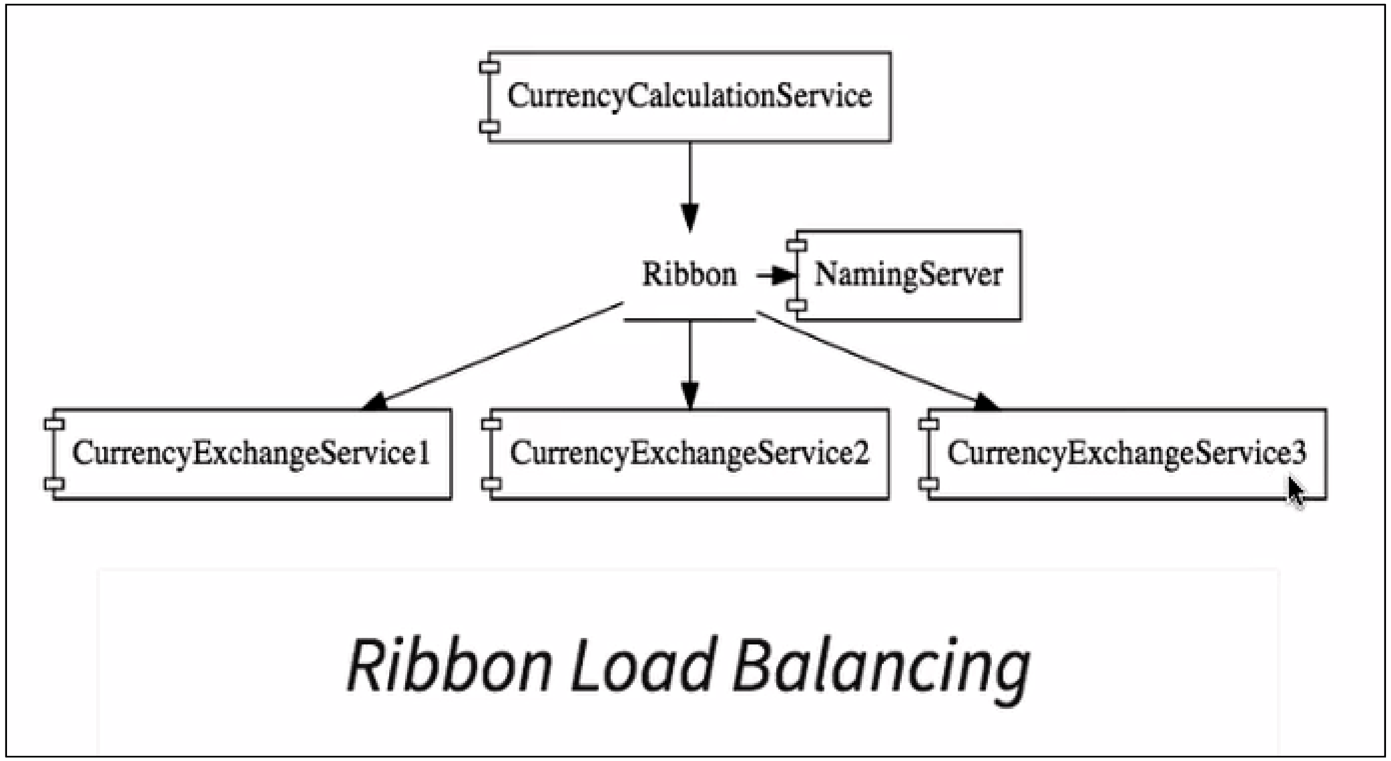
Ribbon is distributing the load between 2 active instances of the currency-exchange-service.

---

31. Step 24 - Understand the need for a Naming Server

In the previous step, Ribbon was able to dynamically redirect request to both the currency-exchange-services.

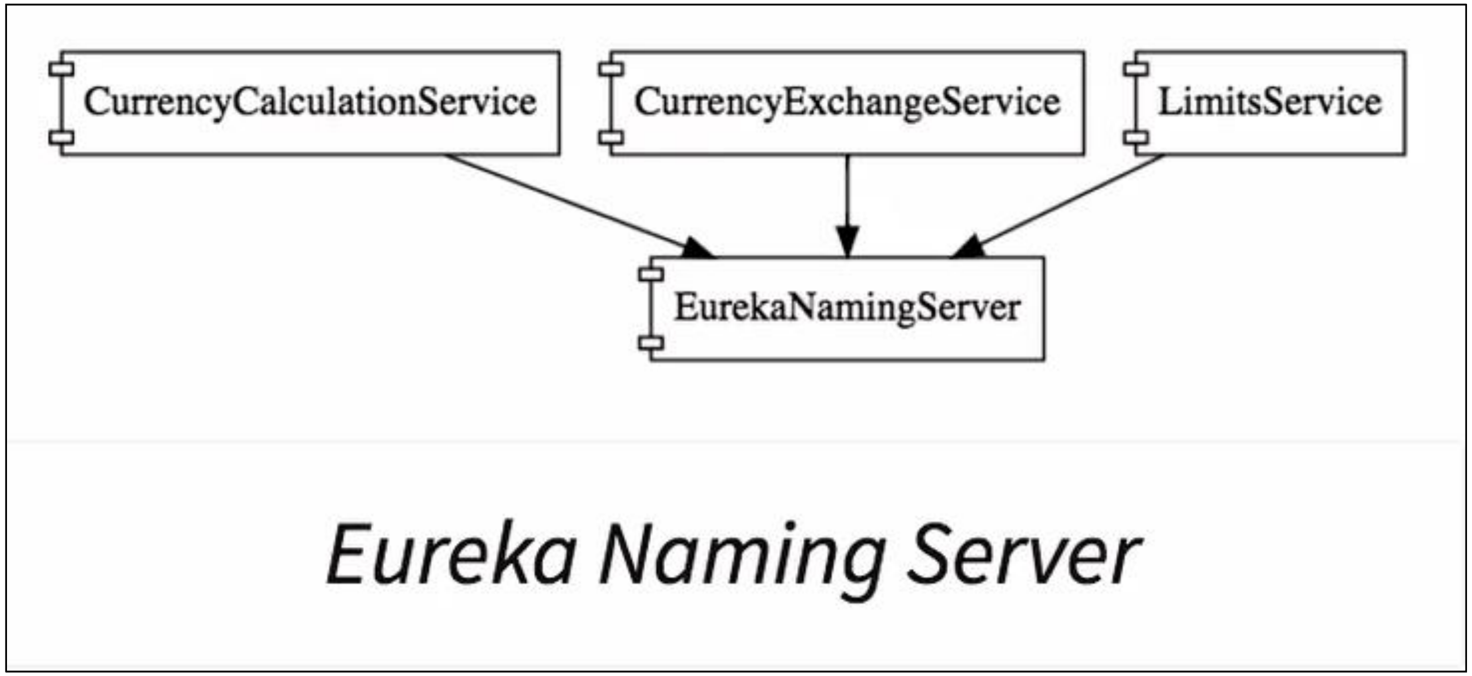
Suppose we start up a third instance of currency-exchange-service. Will Ribbon be able to include the third instance while distributing the load among all the instances? NO.



If we want Ribbon to include the third instance, we need to include the URL of the third instance to the application.properties of currency-conversion-service.

currency-exchange-service.ribbon.listOfServers=http://localhost:8000, http://localhost:8001, http://localhost:8002

So, we need to change the configuration file whenever a new server is created. What we would like to do rather is , based on the load we would want to increase/decrease the instances of currency-exchange-service. It becomes a maintenance headache. To resolve this issue, Naming server is used.



All the instances of all microservices would register with the naming server. Whenever a new instance of a microservice comes up, it would register itself with the Eureka Naming Server. This is called service registration. Whenever a service wants to talk to other service, it would first talk to the Naming Server and would ask about the running instances of the other service. This is called service discovery.

The 2 important features of the Naming Server are:

1) Service registration
2) Service discovery

---

32. Step 25 - Setting up Eureka Naming Server

Steps involved using the Naming server.

1) Create a component for the Eureka Naming Server.
   Dependencies: Eureka Server, Config Client, Actuator, DevTools

```
@SpringBootApplication
@EnableEurekaServer  // To enable eureka naming server
public class NetflixEurekaNamingServerApplication {

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

   Application.properties:

```
spring.application.name=netflix-eureka-naming-server
# Default port for the naming server is 8761
server.port=8761
# Do not want server itself to register with the Naming server
eureka.client.register-with-eureka=false
# Do not fetcha registry
eureka.client.fetch-registry=false
```

2) Update Currency-conversion-service to talk to the Eureka Naming Server
3) Update Currency-conversion-service to talk to the Eureka Naming Server
4) Configure Ribbon (currently installed in the currency-conversion-service) to find the details of the registered instances of the microservices

----

33. Step 26 - Connecting Currency Conversion Microservice to Eureka

2) Update Currency-conversion-service to talk to the Eureka Naming Server

Add eureka dependency to pom.xml of Currency-conversion-service.

```
   <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-eureka -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
  <version>1.0.1.RELEASE</version>
</dependency>
```

Open currencyConversionServiceApplication.java and annotate it with @EnableDiscoveryClient to register itself with the naming server.

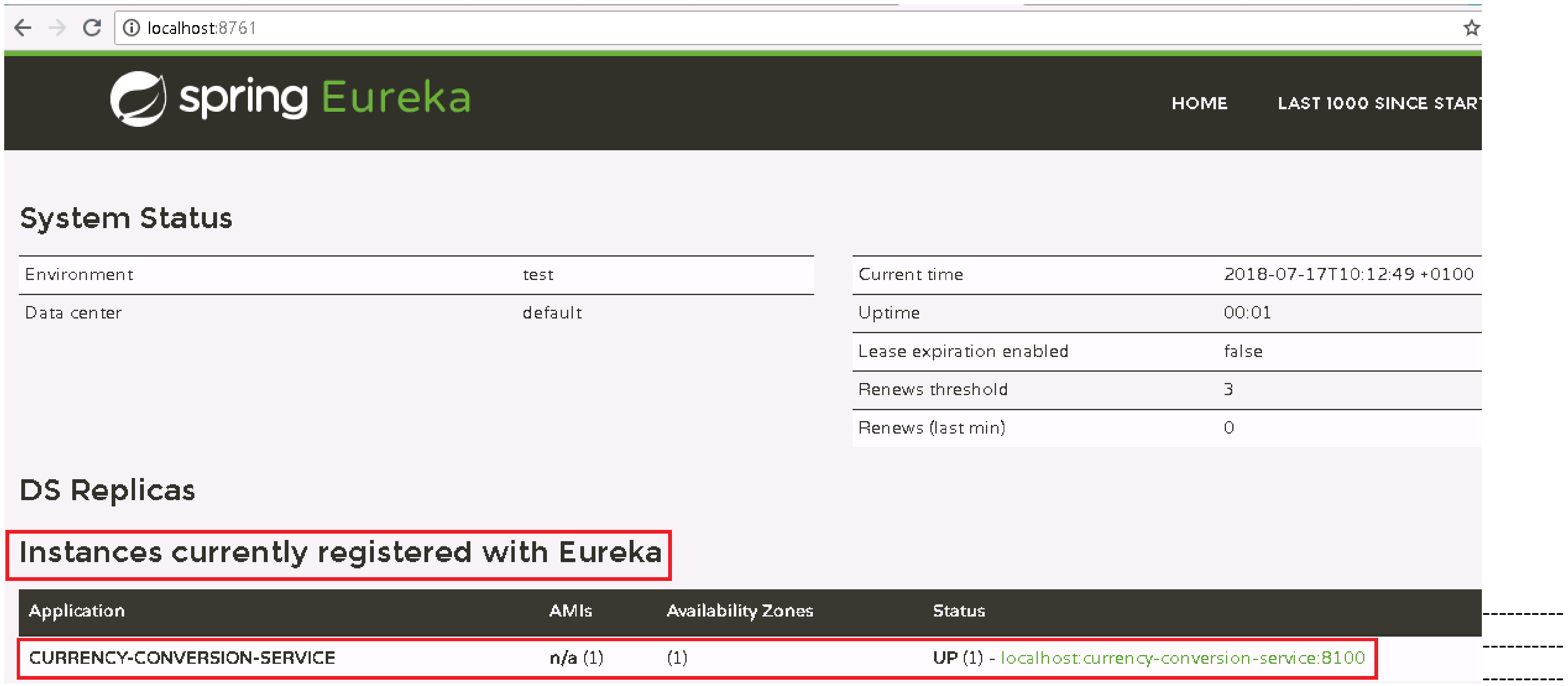
Configure the URL for Eureka Naming server in application.properties of currency-conversion-service.

```
Eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

Error in Eureka server page: EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

Solution: Add the Below dependency to pom.xml of currency-conversion-service (and to all the instances of microservices that you want to register with Eureka Naming server).

```
   <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-netflix-eureka-client -->
          <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-netflix-eureka-client</artifactId>
          </dependency>
```


localhost:8761

spring Eureka

HOME LAST 1000 SINCE STAR

System Status

Environment	test
Data center	default

Current time	2018-07-17T10:12:49 +0100
Uptime	00:01
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	0

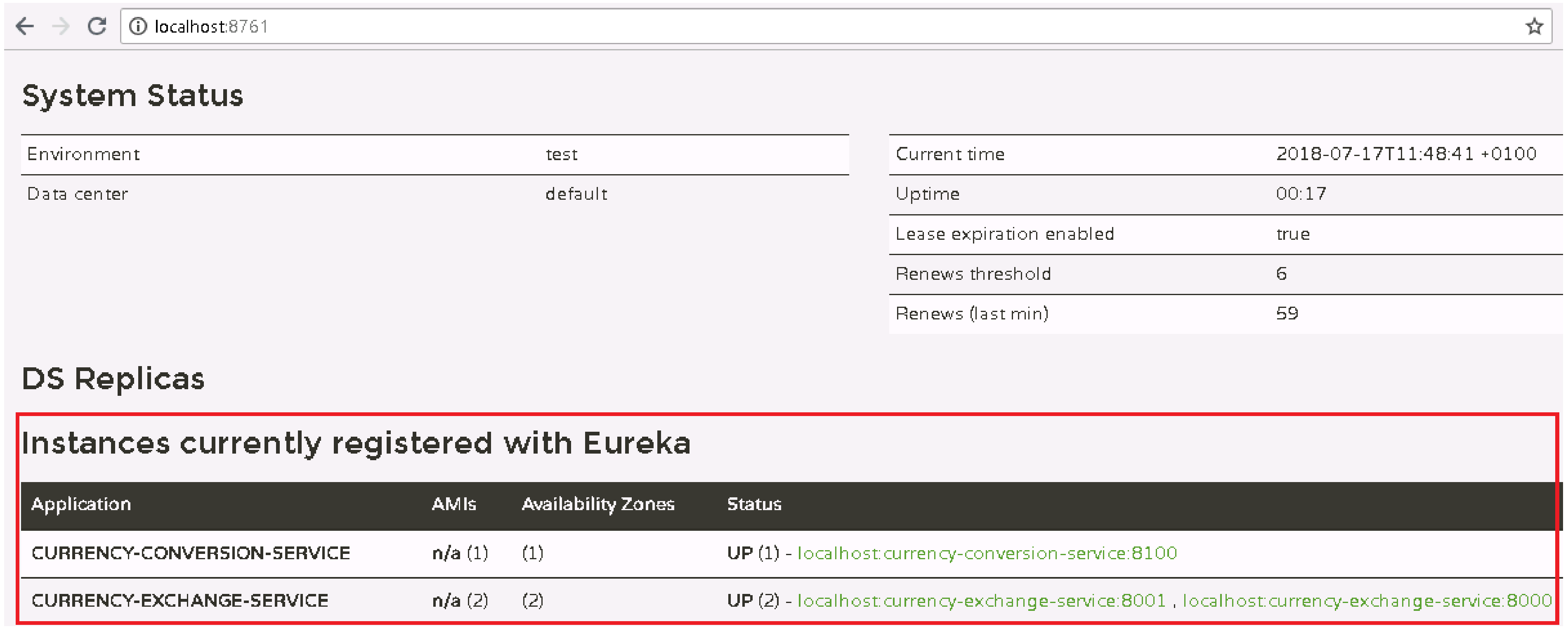
DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (1)	(1)	UP (1) - localhost:currency-conversion-service:8100

Step 27 - Connecting Currency Exchange Microservice to Eureka

Follow the same steps as in step 26 to register currency-exchange-service with Eureka Naming server


localhost:8761

System Status

Environment	test
Data center	default

Current time	2018-07-17T11:48:41 +0100
Uptime	00:17
Lease expiration enabled	true
Renews threshold	6
Renews (last min)	59

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (1)	(1)	UP (1) - localhost:currency-conversion-service:8100
CURRENCY-EXCHANGE-SERVICE	n/a (2)	(2)	UP (2) - localhost:currency-exchange-service:8001 , localhost:currency-exchange-service:8000

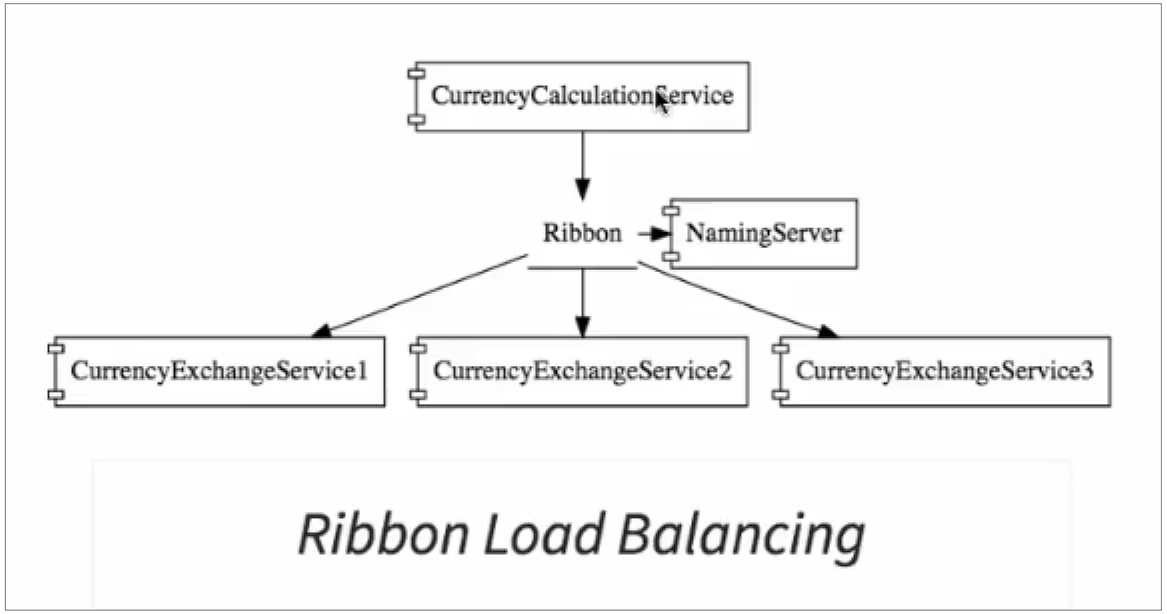
Step 28 - Distributing calls using Eureka and Ribbon

Till the last step we are configuring the url's of currency-exchange-service into application.properties of currency-conversion-service like below.
currency-exchange-service.ribbon.listOfServers=http://localhost:8000,http://localhost:8001

This step onwards, we will remove the above configuration and currency-conversion-service will ask Eureka Naming server directly for the instances of currency-exchange-service.

Ribbon asks the Naming server about the instances of currency-exchange-service. It will get the list back and invoke the appropriate instance of currency-exchange-service.

Step 29 - A review of implementing Eureka, Ribbon and Feign

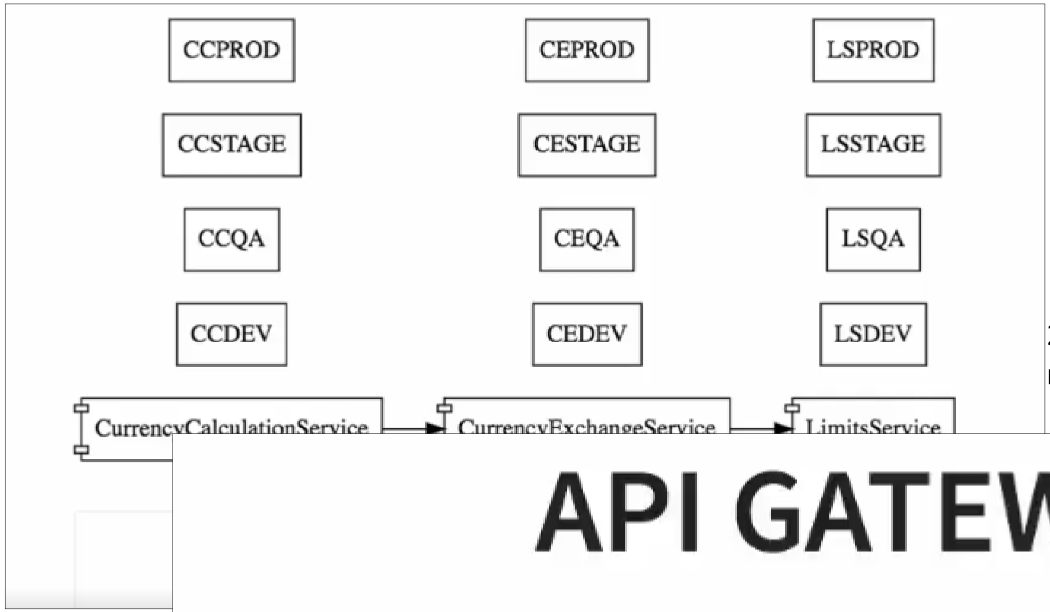


1) Started with creating couple of microservices.
2) currency-conversion-service is a consumer of currency-exchange-service.
3) When we started, we had the direct url connection between currency-conversion-service and currency-exchange-service.
4) Added Feign to currency-conversion-service to make it easy to call currency-exchange-service.
4) To introduce load balancing we introduced Ribbon.
5) Introduced Naming server to register all the microservices with it.
6) Connected all the microservices to Eureka Naming Server.
7) Instead of hard-coding the url's of currency-exchange-service into currency-conversion-service we configured currency-conversion-service to get the instances of currency-exchange-service from Eureka Naming server.
8) Dynamically able to bring up and send down currency-exchange-service instances.

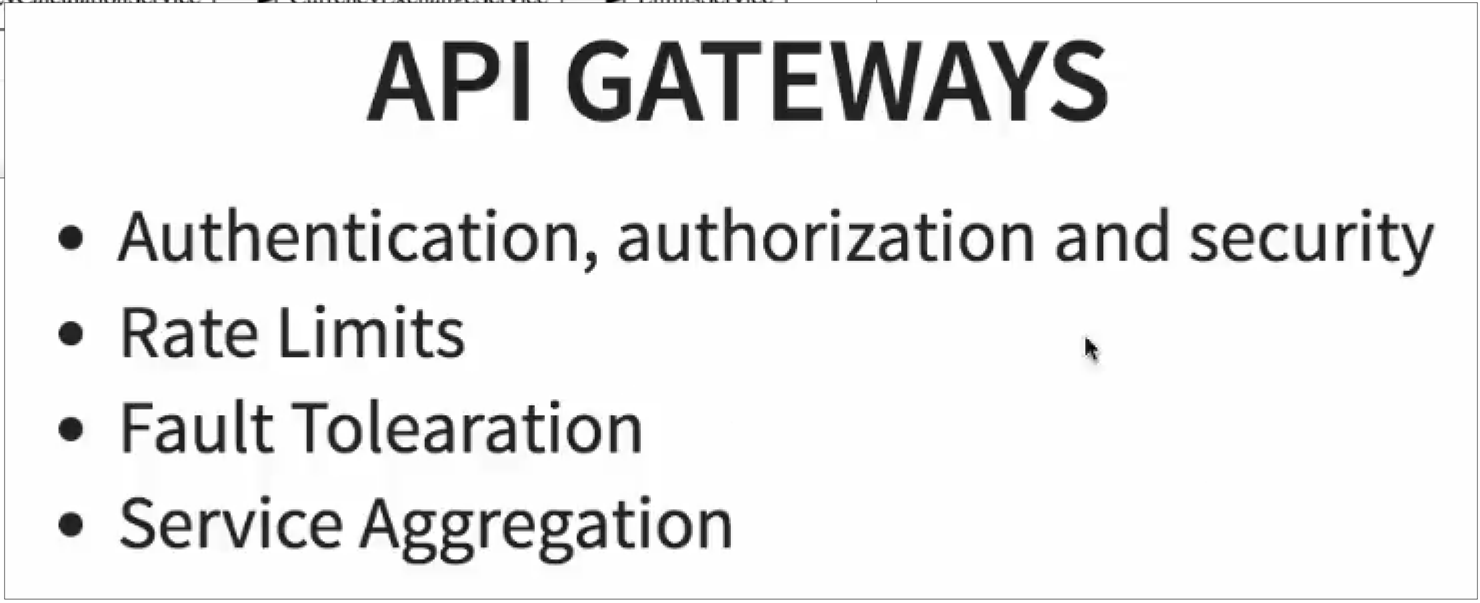
Step 30 - Introduction to API Gateways

In a microservice architecture, there are 2 common things:

1) Many microservices talking to each other



2) Common features for all the microservices



API GATEWAYS

- Authentication, authorization and security
- Rate Limits
- Fault Tolearation
- Service Aggregation

Rate Limits : For a specific client you would only want to allow certain no of calls per hour/day.
Fault Tolerance: Default response in case a service is not up.
Service Aggregation: If there is an external consumer who wants to call 15 different services as part of process. It is better to aggregate those 15 services and provide one service call for the external consumer.

These common features are implemented at the level of API gateways.

Instead of allowing microservices to call each other directly, we would make all the calls go through API gateway. API gateway will take care of providing the common features like authentication, making sure that all service calls are logged, making sure that the rate limits are adhered to, making sure that the services are fault tolerant. The API gateways can also provide aggregation services around multiple microservices. Bcz all the calls get routed through the API gateway, they also serve as a great place for debugging and doing analytics.

In the next step, we will implement a simple API gateway ZUUL.

Step 31 - Setting up Zuul API Gateway

API gateway implementation to intercept all the call between microservices.

Steps to set up a netfilx ZUUL server:
1) Create component for zuul api gateway server.
2) Decide what it should do when it intercepts a request.
3) To make sure that all the right requests are going through ZUUL API gateway.

1) Create component for zuul api gateway server.
--> Create module netflix-zuul-api-gateway-server.
-->

```
@EnableZuulProxy  // To enable the ZUUL proxy
@EnableDiscoveryClient  // To register with the Naming server
@SpringBootApplication
public class NetflixZuulApiGatewayServerApplication {

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

Step 32 - Implementing ZuulLogging Filter

Logging all the request that comes to Zuul API gateway.

Create a class ZuulLoggingFilter.java

```
package com.in28minutes.microservices.netflixzuulapigatewayserver;

import javax.servlet.http.HttpServletRequest;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;

@Component  // Wants spring to manage this class
```

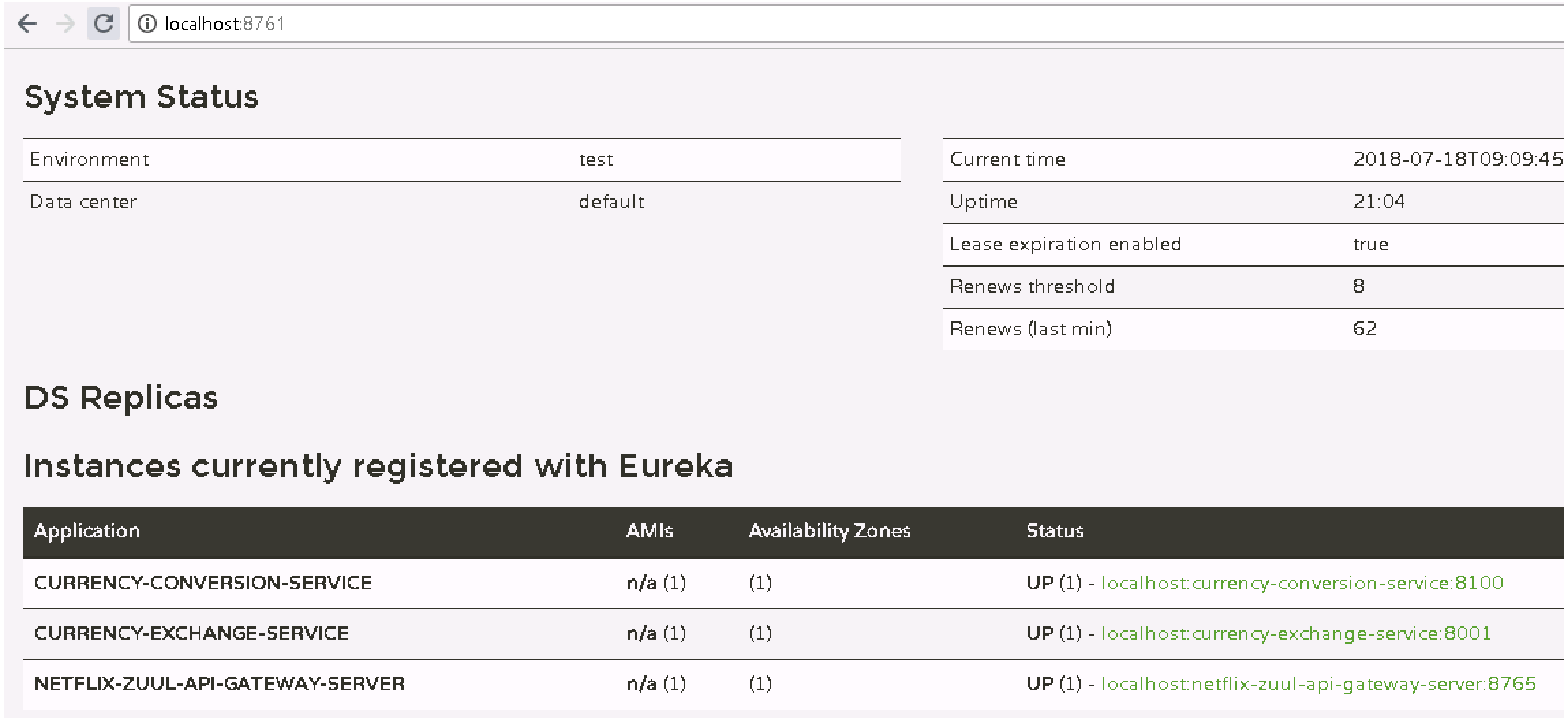
```
public class ZuulLoggingFilter extends ZuulFilter{

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Override
    /*Complete logic of interception goes here*/
    public Object run() throws ZuulException {
            HttpServletRequest request =
                            RequestContext.getCurrentContext().getRequest();
            logger.info("request -> {} requesr uri -> {}",
                            request, request.getRequestURI());
            return null;
    }

    @Override
    /* Should this filter be executed or not.
     * You can checkout the request using business logic and
     * decide whether you want to execute the filter or not*/
    public boolean shouldFilter() {
            return true; // execute for every request
    }

    @Override
    /* If you have more than 1 filter, for ex, ZuulLoggingFilter,
    ZuulSecurityFilter etc you can set priority among them.
     priority filter will be 1, 2, 3 and so on*/
    public int filterOrder() {
            return 1;
    }

    @Override
    /*This tells about when the filter should be executed.
     * 1) Before the request (return "pre")
     * 2) After the request (return "post")
     * 3) Only for error (return "error")*/
    public String filterType() {
            return “pre”;
    }

}
```

---Step 33 - Executing a request through Zuul API Gateway

Before launching Netflix-zuul-api-gateway-server ensure the below:

1) Launch Naming server
2) Launch one of the instances of currency-exchange-service
3) Launch currency-conversion-service

Now refresh Eureka server.



System Status

Environment	test
Data center	default

Current time	2018-07-18T09:09:45
Uptime	21:04
Lease expiration enabled	true
Renews threshold	8
Renews (last min)	62

DS Replicas

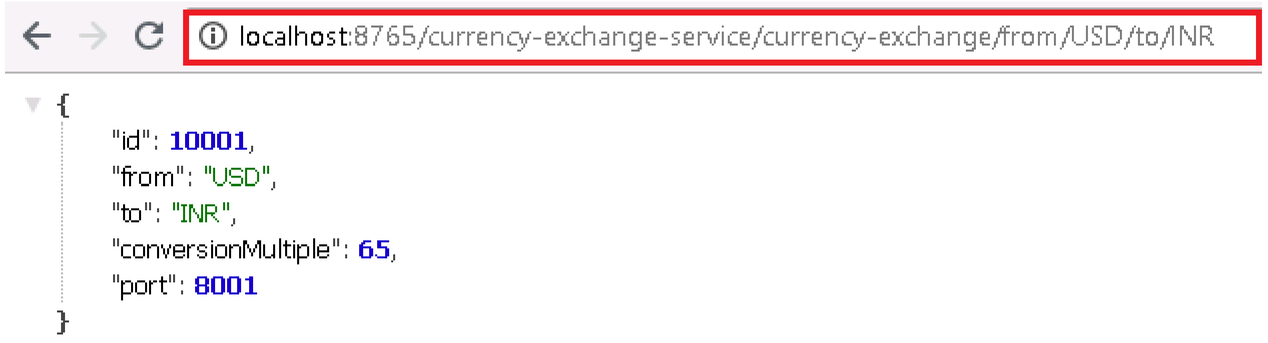
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (1)	(1)	UP (1) - localhost:currency-conversion-service:8100
CURRENCY-EXCHANGE-SERVICE	n/a (1)	(1)	UP (1) - localhost:currency-exchange-service:8001
NETFLIX-ZUUL-API-GATEWAY-SERVER	n/a (1)	(1)	UP (1) - localhost:netflix-zuul-api-gateway-server:8765

Currently we are accessing the currency-exchange-service directly through URL using http://localhost:8000/currency-exchange/from/USD/to/INR. Now access this url using the zuul api gateway. If a consumer directly calls this url, the request will not directly go through the zuul api gateway. How to make this request go through the zuul api gateway? We have configured the zuul api gateway server at port 8765. So the url for invoking a request through api gateway has to be of the below form:
http://localhost:8765/{application-name}/{uri of the service}

So the URL becomes: http://localhost:8765/currency-exchange-service/currency-exchange/from/USD/to/INR

The above URL is complete url to access a service through api gateway. We will get the below o/p if we execute the above url:



localhost:8765/currency-exchange-service/currency-exchange/from/USD/to/INR

```
{
    "id": 10001,
    "from": "USD",
    "to": "INR",
    "conversionMultiple": 65,
    "port": 8001
}
```

The request url is also printed (as implemented in the ZuulLoggingFilter.java) in the console log of netflix-zuul-api-gateway-server.

We would like to send the above request from currency-conversion-service (not manually) to currency-exchange-service.

Step 34 - Setting up Zuul API Gateway between microservice invocations

We will send request from currency-conversion-service (not manually) to currency-exchange-service via zuul-api-gateway-server. Class that actually calls the currency-exchange-service from currency-conversion-service is CurrencyExchangeServiceProxy.java.

Steps:

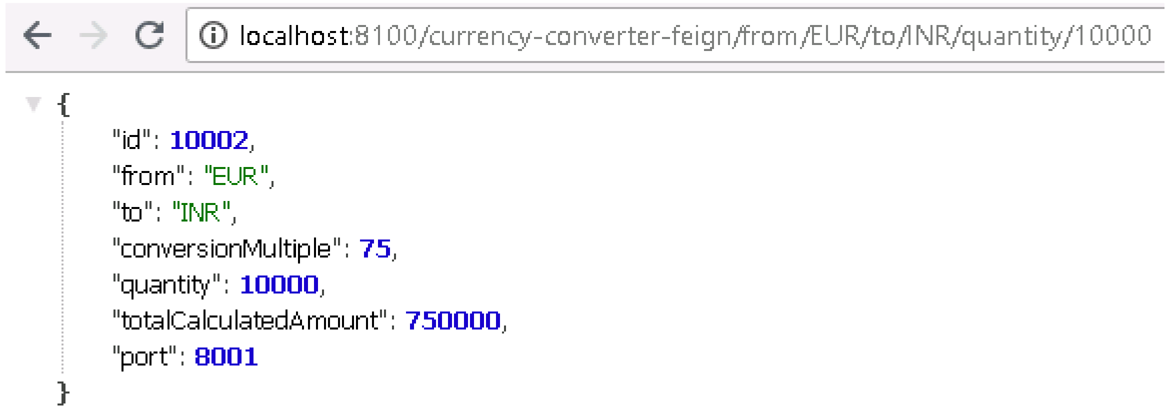
1) Restrict connection to currency-exchange-service in CurrencyExchangeServiceProxy.java by doing the below:
   // @FeignClient(name="currency-exchange-service")

2) Enable connection to netflix-zuul-api-gateway-server in CurrencyExchangeServiceProxy.java
   @FeignClient(name=" netflix-zuul-api-gateway-server ") → Feign client will talk to the naming server and get a uri for netflix-zuul-api-gateway-server

3) Append the name of the application to the service call

```
@FeignClient(name="netflix-zuul-api-gateway-server")
@RibbonClient(name="currency-exchange-service")
public interface CurrencyExchangeServiceProxy {

        @GetMapping("/currency-exchange-service/currency-exchange/from/{from}/to/{to}")
        public CurrencyConversionBean retrieveExchangeValue
                (@PathVariable("from") String from, @PathVariable("to") String to);
}
```

Now access the url of currency-conversion-service. This request will go through the zuul-api-gateway-server.



```
localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/10000
```

```
{
    "id": 10002,
    "from": "EUR",
    "to": "INR",
    "conversionMultiple": 75,
    "quantity": 10000,
    "totalCalculatedAmount": 750000,
    "port": 8001
}
```

The console log for netflix-zuul-api-gateway-server shows below:

```
2018-07-18 09:46:47.617  INFO 9152 --- [nio-8765-exec-6] c.i.m.n.ZuulLoggingFilter: request ->
org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@18489c9e requesr uri -> /currency-exchange-service/currency-
exchange/from/EUR/to/INR
```

Now, we want 2 things:

1) To intercept the request coming to currency-conversion-service.
   URL would be of form: http://localhost:8765/{application-name}/{uri of the service}
   http://localhost:8765/currency-conversion-service/currency-converter-feign/from/EUR/to/INR/quantity/10000.

   The console log for netflix-zuul-api-gateway-server corresponding to the above url is:

```
2018-07-18 09:58:40.787  INFO 9152 --- [trap-executor-0] c.n.d.s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints via configuration
2018-07-18 09:58:41.907  INFO 9152 --- [nio-8765-exec-7] c.i.m.n.ZuulLoggingFilter               : request ->
org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@21463f4a requesr uri -> /currency-conversion-service/currency-
converter-feign/from/EUR/to/INR/quantity/10000
2018-07-18 09:58:41.909  INFO 9152 --- [nio-8765-exec-7] s.c.a.AnnotationConfigApplicationContext : Refreshing SpringClientFactory-currency-
conversion-service: startup date [Wed Jul 18 09:58:41 BST 2018]; parent:
org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@1c2f3bc6
2018-07-18 09:58:41.947  INFO 9152 --- [nio-8765-exec-7] f.a.AutowiredAnnotationBeanPostProcessor : JSR-330 'javax.inject.Inject' annotation found
and supported for autowiring
2018-07-18 09:58:41.983  INFO 9152 --- [nio-8765-exec-7] c.netflix.config.ChainedDynamicProperty  : Flipping property: currency-conversion-
service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647
2018-07-18 09:58:41.986  INFO 9152 --- [nio-8765-exec-7] c.n.u.concurrent.ShutdownEnabledTimer    : Shutdown hook installed for: NFLoadBalancer-
PingTimer-currency-conversion-service
2018-07-18 09:58:41.986  INFO 9152 --- [nio-8765-exec-7] c.netflix.loadbalancer.BaseLoadBalancer  : Client: currency-conversion-service instantiated
a LoadBalancer: DynamicServerListLoadBalancer:{NFLoadBalancer:name=currency-conversion-service,current list of Servers=[],Load balancer
stats=Zone stats: {},Server stats: []}ServerList:null
2018-07-18 09:58:41.989  INFO 9152 --- [nio-8765-exec-7] c.n.l.DynamicServerListLoadBalancer      : Using serverListUpdater PollingServerListUpdater
2018-07-18 09:58:41.991  INFO 9152 --- [nio-8765-exec-7] c.netflix.config.ChainedDynamicProperty  : Flipping property: currency-conversion-
service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647
2018-07-18 09:58:41.991  INFO 9152 --- [nio-8765-exec-7] c.n.l.DynamicServerListLoadBalancer      : DynamicServerListLoadBalancer for client
currency-conversion-service initialized: DynamicServerListLoadBalancer:{NFLoadBalancer:name=currency-conversion-service,current list of
Servers=[localhost:8100],Load balancer stats=Zone stats: {defaultzone=[Zone:defaultzone;	Instance count:1;	Active connections count: 0;
	Circuit breaker tripped count: 0;	Active connections per server: 0.0;]
},Server stats: [[Server:localhost:8100;	Zone:defaultZone;	Total Requests:0;	Successive connection failure:0;	Total blackout
seconds:0;	Last connection made:Thu Jan 01 01:00:00 GMT 1970;	First connection made: Thu Jan 01 01:00:00 GMT 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@454b3747
2018-07-18 09:58:42.016  INFO 9152 --- [nio-8765-exec-3] c.i.m.n.ZuulLoggingFilter	: request ->
org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@24b37950 requesr uri -> /currency-exchange-service/currency-
exchange/from/EUR/to/INR
2018-07-18 09:58:43.001  INFO 9152 --- [erListUpdater-0] c.netflix.config.ChainedDynamicProperty  : Flipping property: currency-conversion-
service.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647
```

NOTE: It is evident from the abobe log that both the requests (/currency-conversion-service/currency-converter-feign/from/EUR/to/INR/quantity/10000 and /currency-exchange-service/currency-exchange/from/EUR/to/INR) are going through zuul-api-gateway-server).

2) To intercept the request coming to currency-exchange-service. (implemented before)

Tip: Zuul uses 'application-name' in the url to talk to Eureka and find out the Url of the service.

Step 35 - Introduction to Distributed Tracing

With the introduction of microservices, calling of services become complex. Let us assume that a service is not working fine and you want to debug it. Where do you look at? Currency-conversion-service or currency-exchange-service or netflix-zuul-api-gateway-server? How do we know what is happening in total with a specific request? One of the most important things while implementing microservice architecture is that we need to have distributed tracing. This is one place to know all about what happened with a particular request. We would like to have one centralized location where we can see the complete chain of what happened with a specific request.

We will use spring-cloud-sleuth with zipkin. One of the important thing is to assign a specific id to a request. Let us say a request is going through a set of application components, going through the api gateway to currency-conversion-service and again through the api gateway to currency-exchange service. How do we identify that both the request are the same one. Only way to identify this is to assign a request ID to the request.

Spring cloud sleuth assigns a unique ID to a request so that the request can be traced across components. Zipkin is a distributed tracing system. All the logs from all the microservices we would put in a MQ (we would use a Rabbit MQ) and we would send it out to the Zipkin server where it is consolidated and we would be able to look through the different requests and find what happened with a particular request.

Step 36-Implementing Spring Cloud Sleuth

In implementing Spring Cloud Sleuth, we first need to check where we want to use it. Spring Cloud Sleuth will add unique ID to a request so that you can trace it across multiple components. Here we will implement Spring Cloud Sleuth in the below components. However it can be implemented in other components as well.

currency-conversion-service
currency-exchange-service
netflix-zuul-api-gateway-server

There are 2 simple steps in adding Spring Cloud Sleuth to any component:

1) Add dependency

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

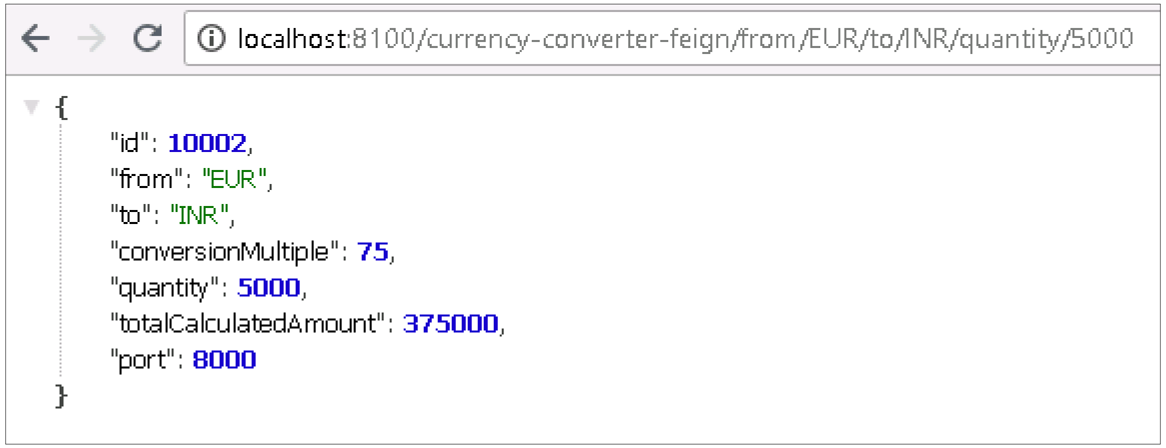
2) What all requests we would want to intercept?
Here we are tracing all the requests, so we have to create a Sampler in CurrencyExchangeServiceApplication.java (and in all of the spring modules in the java file where main method is present).

```
@Bean
public Sampler defaultSampler() {
        return Sampler.ALWAYS_SAMPLE;
}
```

NOTE: Exception while starting currency-conversion-service.
Factory method 'sleuthRxJavaSchedulersHook' threw exception; nested exception is java.lang.IllegalAccessError: tried to access method rx.plugins.RxJavaPlugins.reset()V from class org.springframework.cloud.sleuth.instrument.rxjava.SleuthRxJavaSchedulersHook

Solution: `spring.sleuth.rxjava.schedulers.hook.enabled=false (in application.properties of currency-conversion-service)`
https://gitter.im/spring-cloud/spring-cloud-sleuth/archives/2016/05/10

Executing url : http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/5000



localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/5000

```
{
    "id": 10002,
    "from": "EUR",
    "to": "INR",
    "conversionMultiple": 75,
    "quantity": 5000,
    "totalCalculatedAmount": 375000,
    "port": 8000
}
```

See the 2 console logs for the above url request:

(1) In currency-conversion-service

```
42b8de3cd83ec970  --> Request ID
```

```
2018-07-19 08:27:53.152  INFO [currency-conversion-service,42b8de3cd83ec970,42b8de3cd83ec970,true]
9552 --- [io-8100-exec-10] c.i.m.c.CurrencyConversionController    :
com.in28minutes.microservices.currencyconversionservice.CurrencyConversionBean@777dbaa0
```

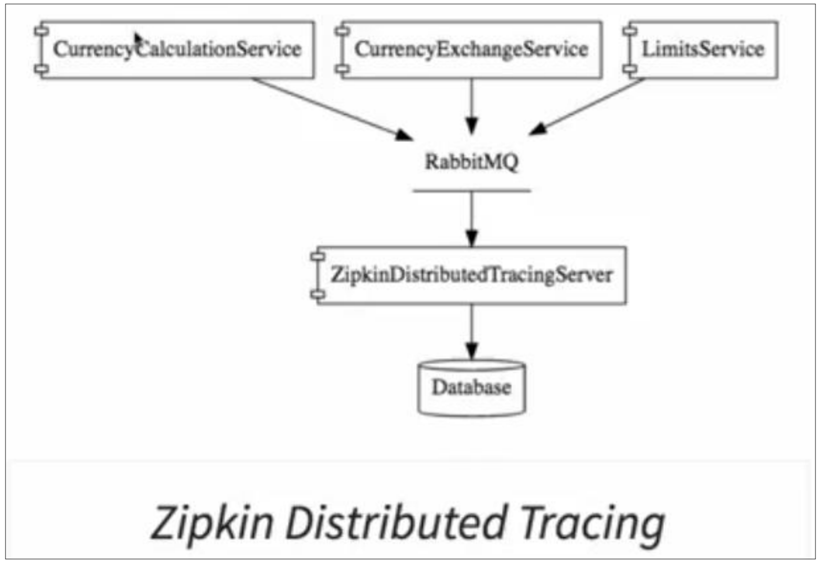
 (2)In currency-exchange-service
42b8de3cd83ec970 --> Request ID

```
2018-07-19 08:27:53.152  INFO [currency-conversion-service,42b8de3cd83ec970,42b8de3cd83ec970,true]
9552 --- [io-8100-exec-10] c.i.m.c.CurrencyConversionController     :
com.in28minutes.microservices.currencyconversionservice.CurrencyConversionBean@777dbaa0
```

Note: Both the above request ID's are same. As soon as we use spring-cloud-sleuth, it is assigning an ID to all the requests. Using a request ID we can trace a particular request across different components.

The only problem with the above approach is that the log for a request is distributed across multiple places (consoles).Here the need for a distributed tracing server (Zipkin) comes.

Step 37-Introduction to Distributed Tracing with Zipkin


CurrencyCalculationService
CurrencyExchangeService
LimitsService
RabbitMQ
ZipkinDistributedTracingServer
Database
Zipkin Distributed Tracing

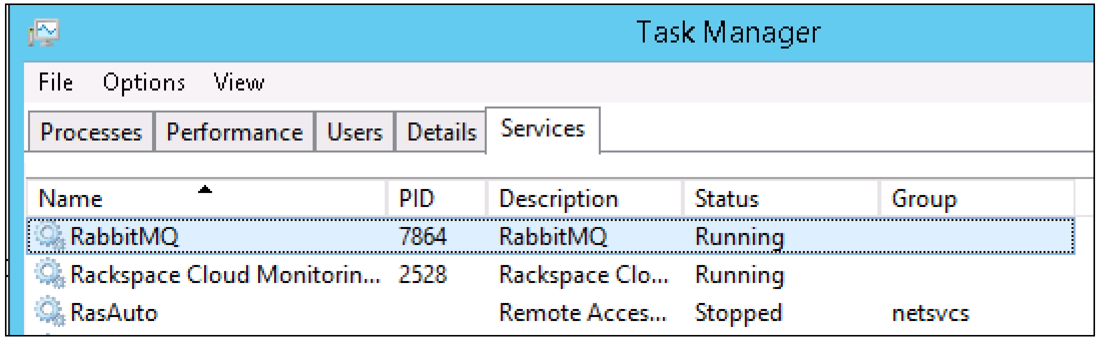
To centralize all the logs from all the microservices. We will use ZipkinDistributedTracingServer for centralized log. We will use Zipkin UI to look at requests.

Qn) How do I get all the logs from all the microservices to ZipkinDistributedTracingServer?
Ans: RabbitMQ. Whenever there is a log message, microservice will put it on the queue. ZipkinDistributedTracingServer will pick the log messages from the queue.

Typically, ZipkinDistributedTracingServer is connected to a database. We will use in-memory database.

Step 38 - Installing Rabbit MQ

In order to install RabbitMQ, we need erlang first.
http://www.erlang.org/downloads --> Erlang
https://www.rabbitmq.com/install-windows.html --> RabbitMQ
After the installations:
RabbitMQ will be running as a background service. You can control this service from the start menu.



Name	PID	Description	Status	Group
RabbitMQ	7864	RabbitMQ	Running	
Rackspace Cloud Monitorin...	2528	Rackspace Clo...	Running	
RasAuto		Remote Acces...	Stopped	netsvcs

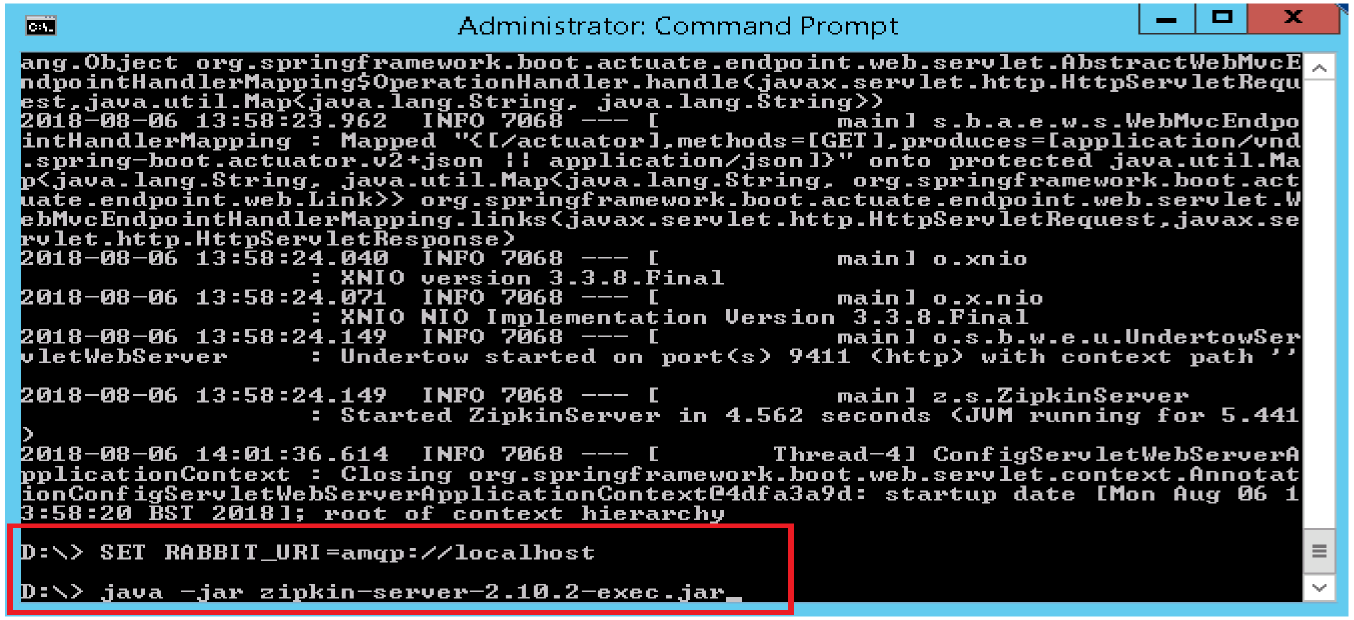
Step 39 - Setting up Distributed Tracing with Zipkin

https://zipkin.io/pages/quickstart --> For zipkin installation

Install ZipkinDistributedTracingServer and configure it to listen over RabbitMQ.

Before installing ZipkinDistributedTracingServer, ensure that the RabbitMQ is running as a background service and make these 2 configurations.

```
set RABBIT_URI=amqp://localhost
java -jar zipkin-server-2.7.0-exec.jar
```



```
ang.Object org.springframework.boot.actuate.endpoint.web.servlet.AbstractWebMvcE
ndpointHandlerMapping$OperationHandler.handle(javax.servlet.http.HttpServletRequ
est,java.util.Map<java.lang.String, java.lang.String>)
2018-08-06 13:58:23.962  INFO 7068 --- [           main] s.b.a.e.w.s.WebMvcEndpo
intHandlerMapping : Mapped "{[/actuator],methods=[GET],produces=[application/vnd
.spring-boot.actuator.v2+json || application/json]}" onto protected java.util.Ma
p<java.lang.String, java.util.Map<java.lang.String, org.springframework.boot.act
uate.endpoint.web.Link>> org.springframework.boot.actuate.endpoint.web.servlet.W
ebMvcEndpointHandlerMapping.links(javax.servlet.http.HttpServletRequest,javax.se
rvlet.http.HttpServletResponse)
2018-08-06 13:58:24.040  INFO 7068 --- [           main] o.xnio
                 : XNIO version 3.3.8.Final
2018-08-06 13:58:24.071  INFO 7068 --- [           main] o.x.nio
                 : XNIO NIO Implementation Version 3.3.8.Final
2018-08-06 13:58:24.149  INFO 7068 --- [           main] o.s.b.w.e.u.UndertowSer
vletWebServer     : Undertow started on port(s) 9411 (http) with context path ''

2018-08-06 13:58:24.149  INFO 7068 --- [           main] z.s.ZipkinServer
                 : Started ZipkinServer in 4.562 seconds (JVM running for 5.441
)
2018-08-06 14:01:36.614  INFO 7068 --- [       Thread-4] ConfigServletWebServerA
pplicationContext : Closing org.springframework.boot.web.servlet.context.Annotat
ionConfigServletWebServerApplicationContext@4dfa3a9d: startup date [Mon Aug 06 1
3:58:20 BST 2018]; root of context hierarchy

D:\> SET RABBIT_URI=amqp://localhost

D:\> java -jar zipkin-server-2.10.2-exec.jar_
```

http://localhost:9411/zipkin --> To access zipkin UI

Step 40 - Connecting microservices to Zipkin

Now we need our microservices to put the messages for zipkindistributedtracingserver on RabbitMQ. Once we place messages on RabbitMQ, these messages would be picked up and we will be able to see them on Zipkin dashboard.

We will connect currencyexchangeservice, currencycalculationservice along with apigateway to RabbitMQ. To be able to do this we need to add couple of dependencies to all the modules:

We are already logging the messages in to the log with the ID using the Sleuth.

Now, we would like to log message in the format that it (zipkin) expects/understands.

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

Now, how do I send a message to zipkin? Over MQ. We will use RabbitMQ which uses the amqp protocol. We are establishing a connection to the amqp bus. The default amqp installation which is used is RabbitMQ. This dependency is to get a connection to RabbitMQ. We would be able to put a zipkin message to RabbitMQ.

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

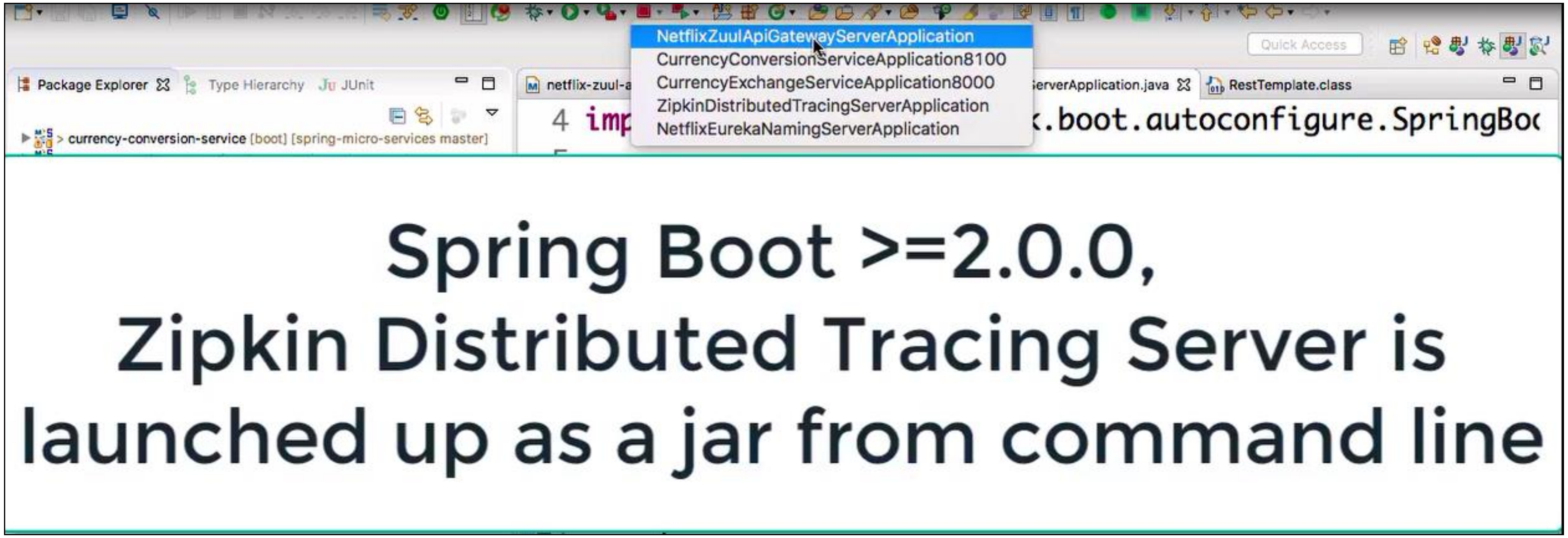
----

50. Step 41 - Using Zipkin UI Dashboard to trace requests

In this step we will fire a request and see if we can trace it in zipkindistributedtracingserver.

We need to run the below:

Currencycalculationservice, currencyexchangeservice, eurekanamingserver, netflix-zuul-api-gateway-server, zipkindistributedtracingserver

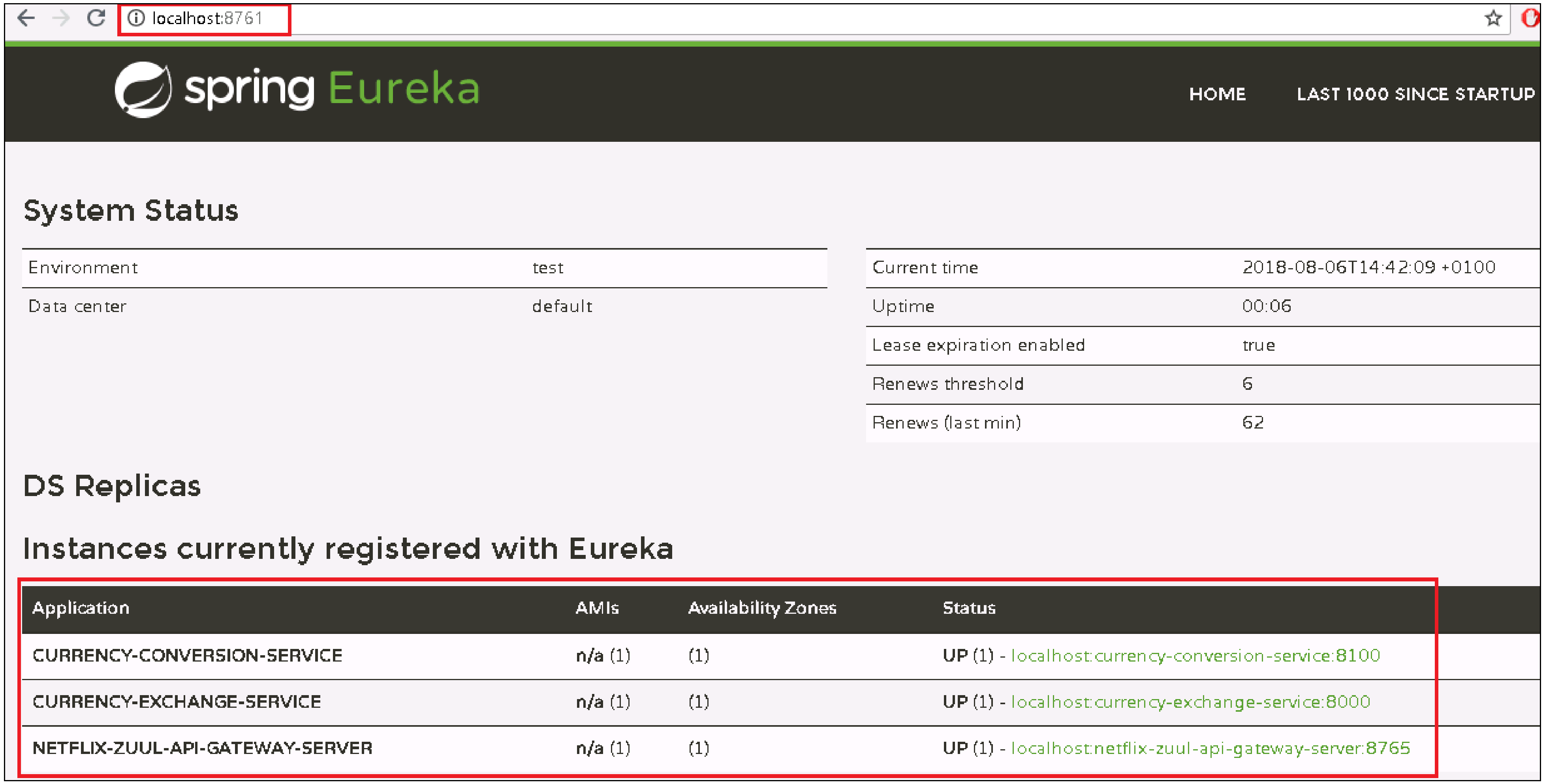

NetflixZuulApiGatewayServerApplication
CurrencyConversionServiceApplication8100
CurrencyExchangeServiceApplication8000
ZipkinDistributedTracingServerApplication
NetflixEurekaNamingServerApplication
Package Explorer
Type Hierarchy
JUnit
currency-conversion-service [boot] [spring-micro-services master]
RestTemplate.class
Quick Access

Spring Boot >=2.0.0,
Zipkin Distributed Tracing Server is launched up as a jar from command line

Launch in the below order:
Eurekanamingserver, zipkindistributedtracingserver, any order for microservices, netflix-zuul-api-gateway-server

Connect zipkindistributedtracingserver to Eureka (an exercise).

Screenshot for Eureka after running the above services:


localhost:8761

spring Eureka

HOME LAST 1000 SINCE STARTUP

System Status

Environment	test
Data center	default

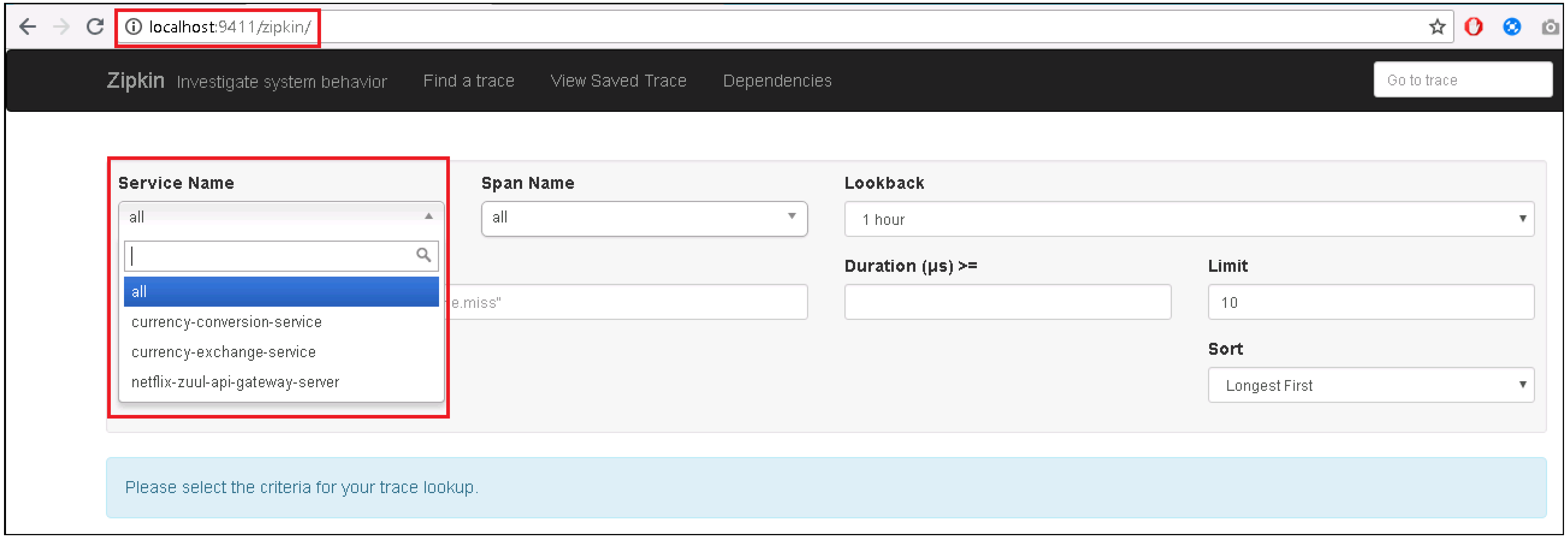
Current time	2018-08-06T14:42:09 +0100
Uptime	00:06
Lease expiration enabled	true
Renews threshold	6
Renews (last min)	62

DS Replicas

Instances currently registered with Eureka

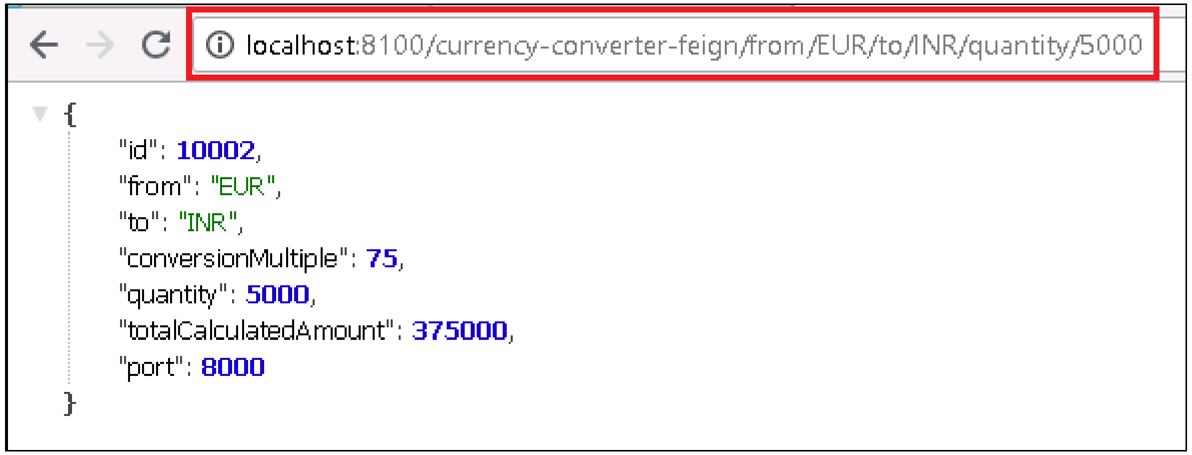
Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (1)	(1)	UP (1) - localhost:currency-conversion-service:8100
CURRENCY-EXCHANGE-SERVICE	n/a (1)	(1)	UP (1) - localhost:currency-exchange-service:8000
NETFLIX-ZUUL-API-GATEWAY-SERVER	n/a (1)	(1)	UP (1) - localhost:netflix-zuul-api-gateway-server:8765

Screenshot of Zipkin UI (showing all the modules):



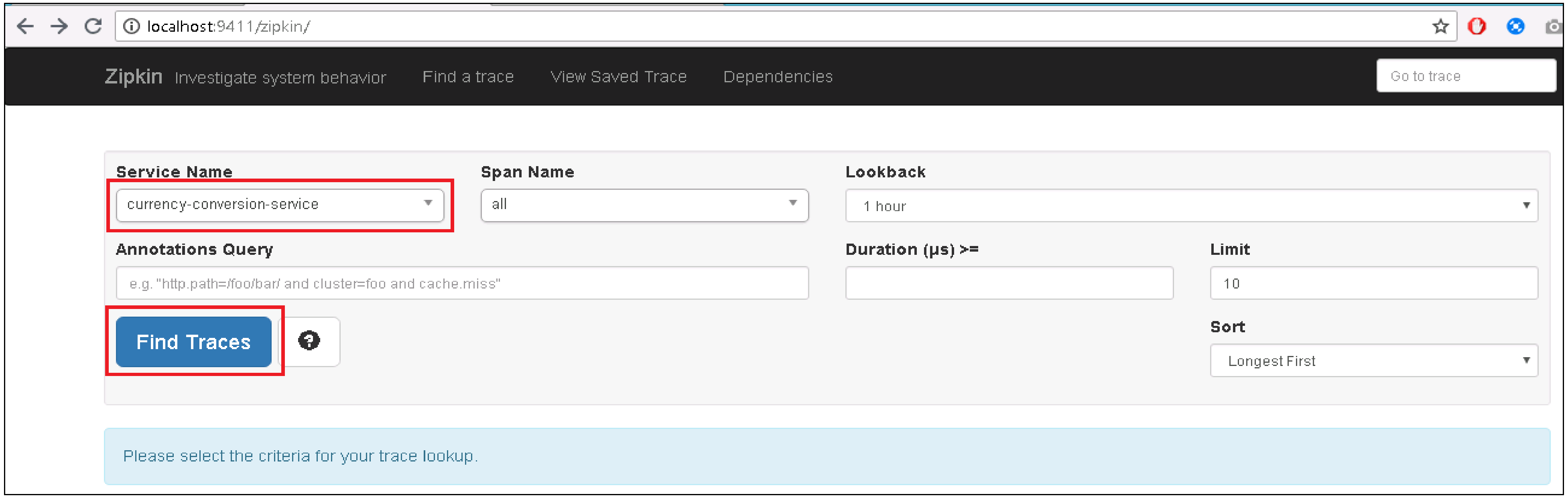
Please select the criteria for your trace lookup.

Now, fire a request http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/5000.



```
{
    "id": 10002,
    "from": "EUR",
    "to": "INR",
    "conversionMultiple": 75,
    "quantity": 5000,
    "totalCalculatedAmount": 375000,
    "port": 8000
}
```

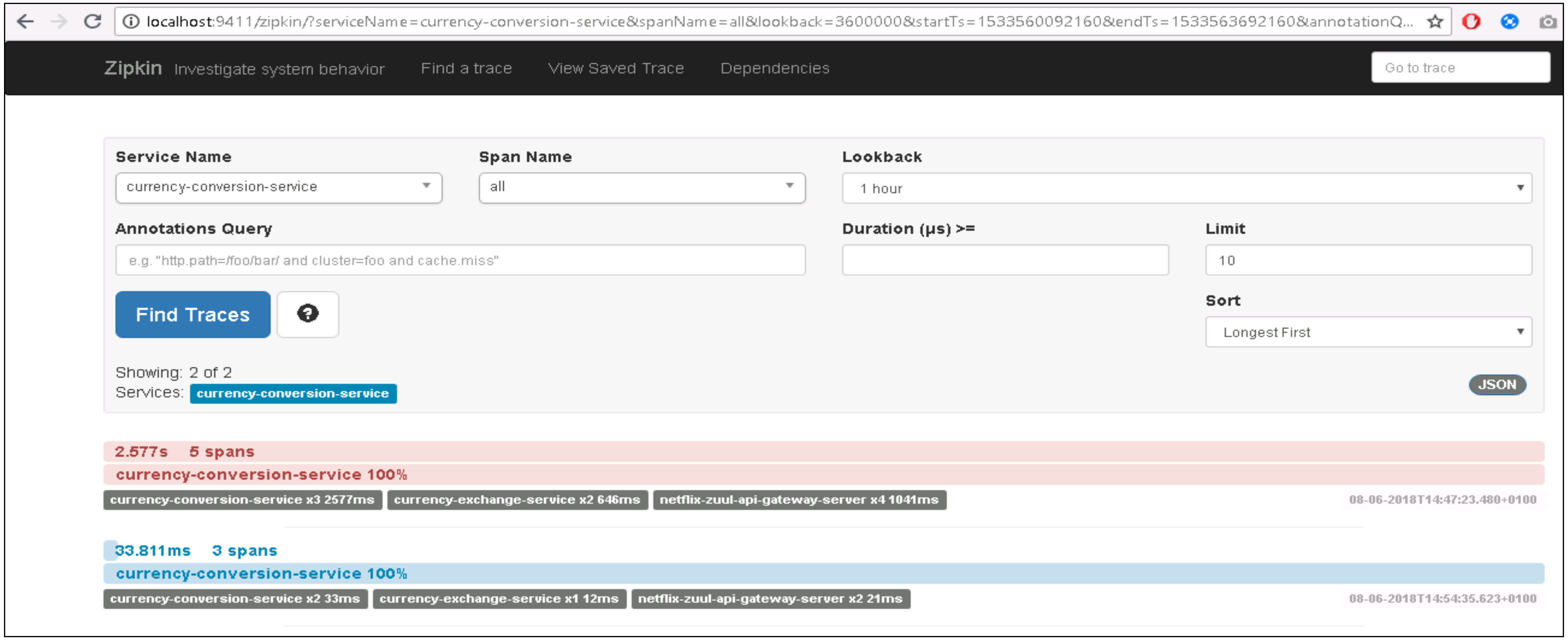
To find trace, select the mcroservice in zipkin UI and click "**Find Traces**".


localhost:9411/zipkin/

Service Name: currency-conversion-service
Span Name: all
Lookback: 1 hour
Annotations Query: e.g. "http.path=/foo/bar/ and cluster=foo and cache.miss"
Duration (μs) >=
Limit: 10
Sort: Longest First

Please select the criteria for your trace lookup.

You get the below trace:



localhost:9411/zipkin/?serviceName=currency-conversion-service&spanName=all&lookback=3600000&startTs=1533560092160&endTs=1533563692160&annotationQ...

Zipkin Investigate system behavior | Find a trace | View Saved Trace | Dependencies | Go to trace

Service Name: currency-conversion-service
Span Name: all
Lookback: 1 hour
Annotations Query: e.g. "http.path=/foo/bar/ and cluster=foo and cache.miss"
Duration (μs) >=
Limit: 10
Sort: Longest First

Find Traces

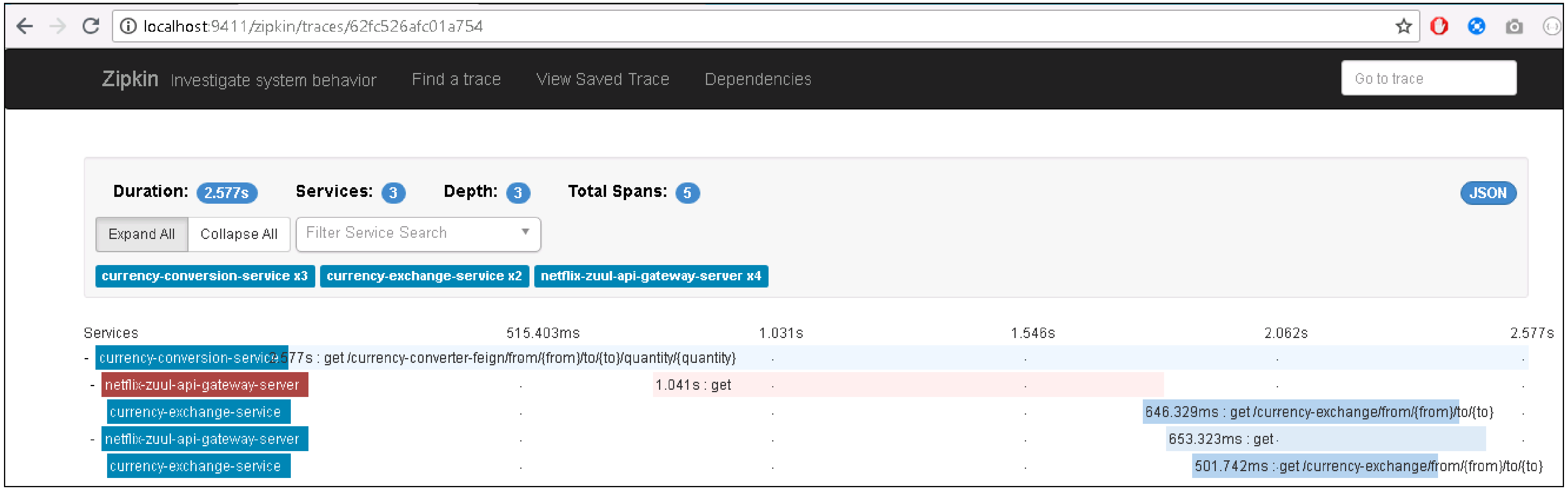
Showing: 2 of 2
Services: currency-conversion-service

JSON

2.577s 5 spans
currency-conversion-service 100%
currency-conversion-service x3 2577ms | currency-exchange-service x2 646ms | netflix-zuul-api-gateway-server x4 1041ms | 08-06-2018T14:47:23.480+0100

33.811ms 3 spans
currency-conversion-service 100%
currency-conversion-service x2 33ms | currency-exchange-service x1 12ms | netflix-zuul-api-gateway-server x2 21ms | 08-06-2018T14:54:35.623+0100

Among the listed executions above, you can select anyone and see the details.


localhost:9411/zipkin/traces/62fc526afc01a754

Zipkin Investigate system behavior | Find a trace | View Saved Trace | Dependencies | Go to trace

Duration: 2.577s Services: 3 Depth: 3 Total Spans: 5 JSON

Expand All | Collapse All | Filter Service Search

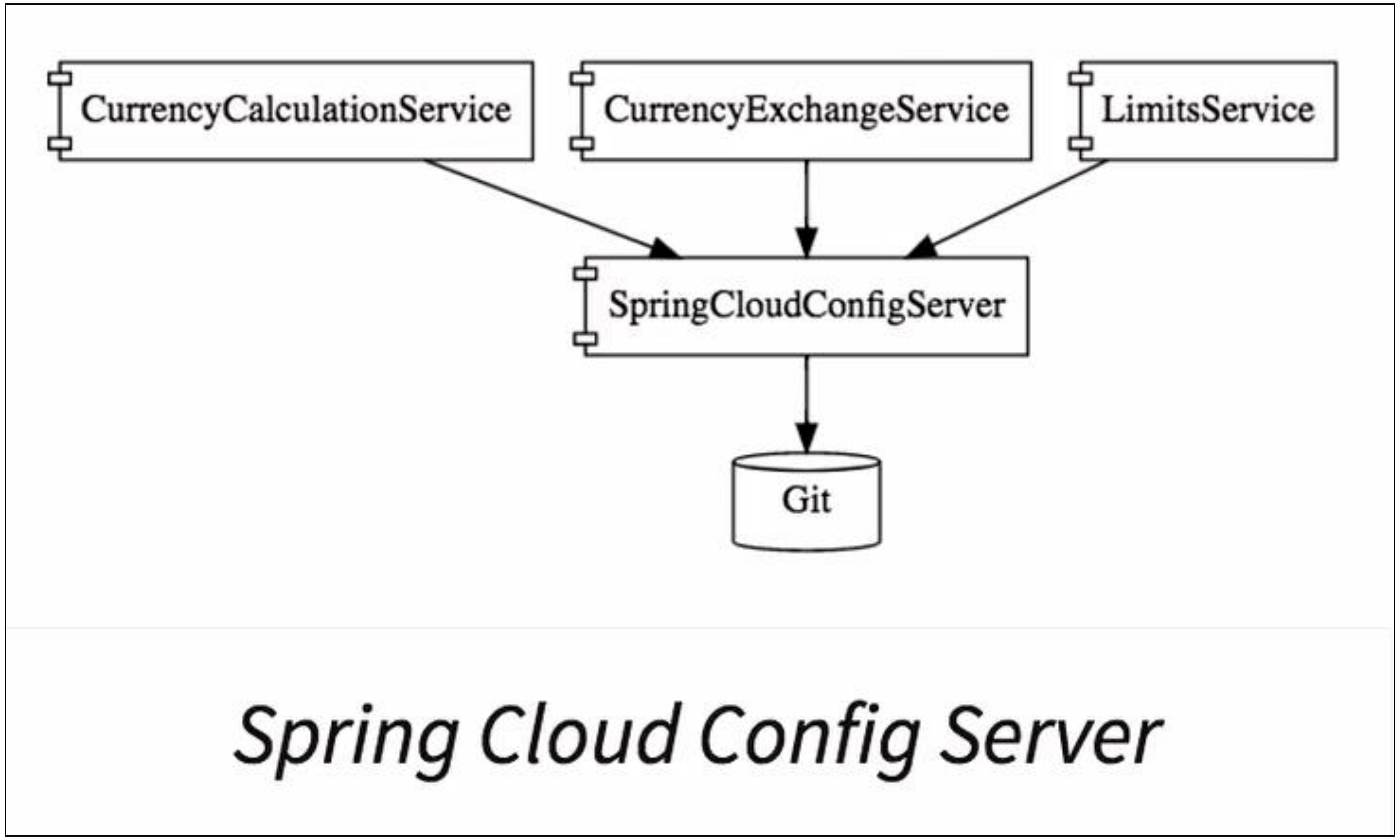
currency-conversion-service x3 | currency-exchange-service x2 | netflix-zuul-api-gateway-server x4

```
Services                          515.403ms       1.031s        1.546s        2.062s        2.577s
- currency-conversion-service     2.577s : get /currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}
 - netflix-zuul-api-gateway-server              1.041s : get
    currency-exchange-service                                          646.329ms : get /currency-exchange/from/{from}/to/{to}
 - netflix-zuul-api-gateway-server                                      653.323ms : get
    currency-exchange-service                                             501.742ms : get /currency-exchange/from/{from}/to/{to}
```

One of the typical problems (bcz of many microservices) with microservices is finding out what is happening in the background. This issue us solved by zipkindistributedtracingserver.

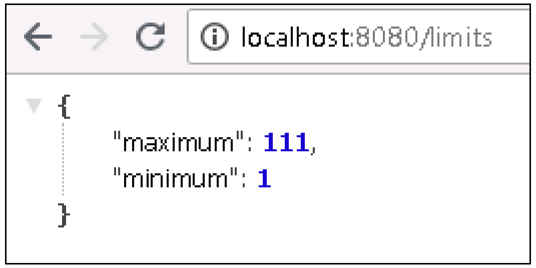
----

51. Step 42 - Understanding the need for Spring Cloud Bus



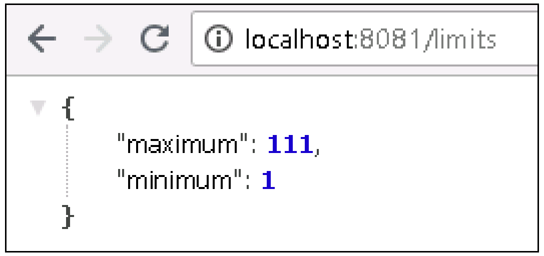
We connected **limits-service** to spring-cloud-config-server. This stored configuration for different environments for the limits-service in the git repository and we were able to connect limits-service to spring-cloud-config-server to retrieve the configuration. However there is one problem left unsolved.

Start spring-cloud-config-server and then limits-service.



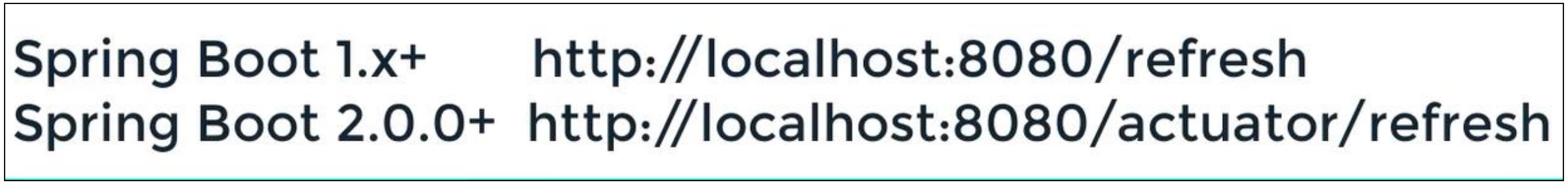
```
{
    "maximum": 111,
    "minimum": 1
}
```

Now, launch one more instance (port 8081) of the limits-service.

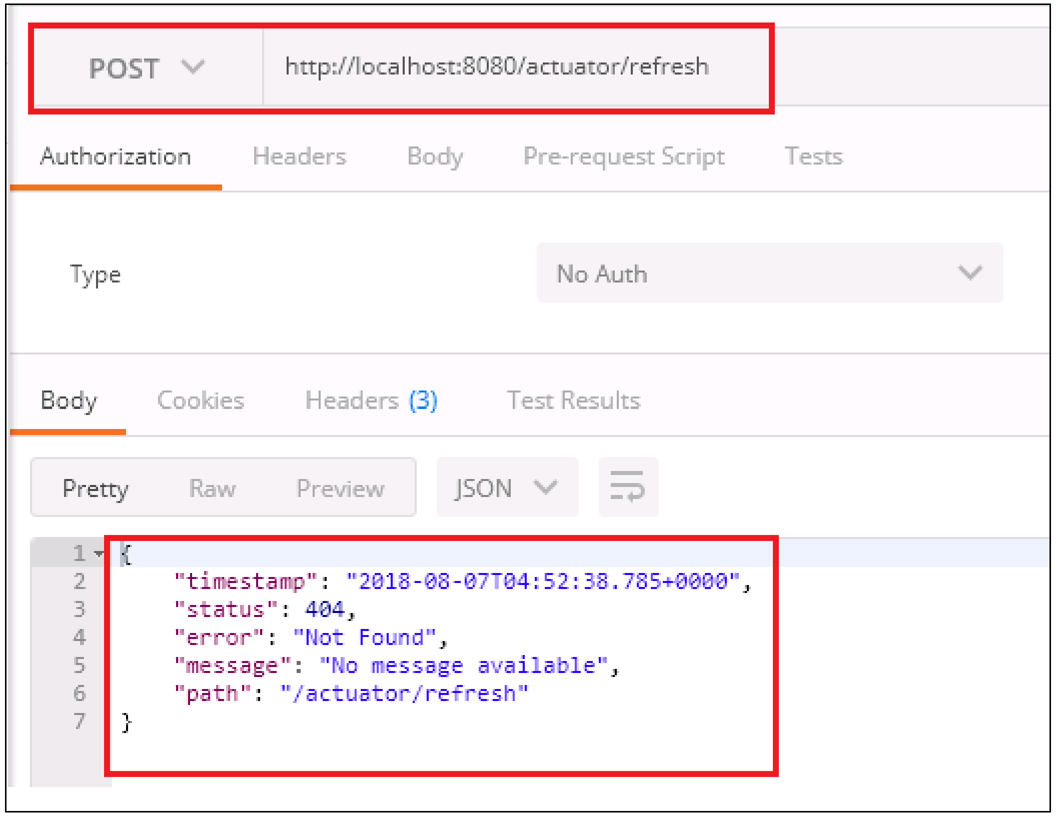


```
{
    "maximum": 111,
    "minimum": 1
}
```

Now let us change the config values in git-localconfig-repo and commit and see if the change reflects in the url for limits service. The change does not reflect when we access the url for limits-service. How can you see the changes? By executing a simple post request using postman:

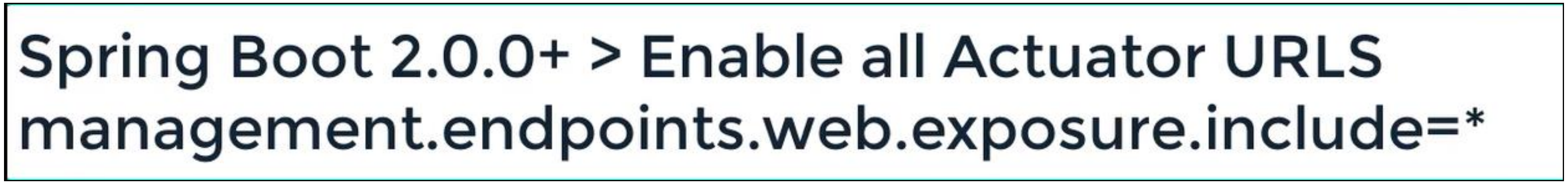


Spring Boot 1.x+	http://localhost:8080/refresh
Spring Boot 2.0.0+	http://localhost:8080/actuator/refresh



```
{
    "timestamp": "2018-08-07T04:52:38.785+0000",
    "status": 404,
    "error": "Not Found",
    "message": "No message available",
    "path": "/actuator/refresh"
}
```

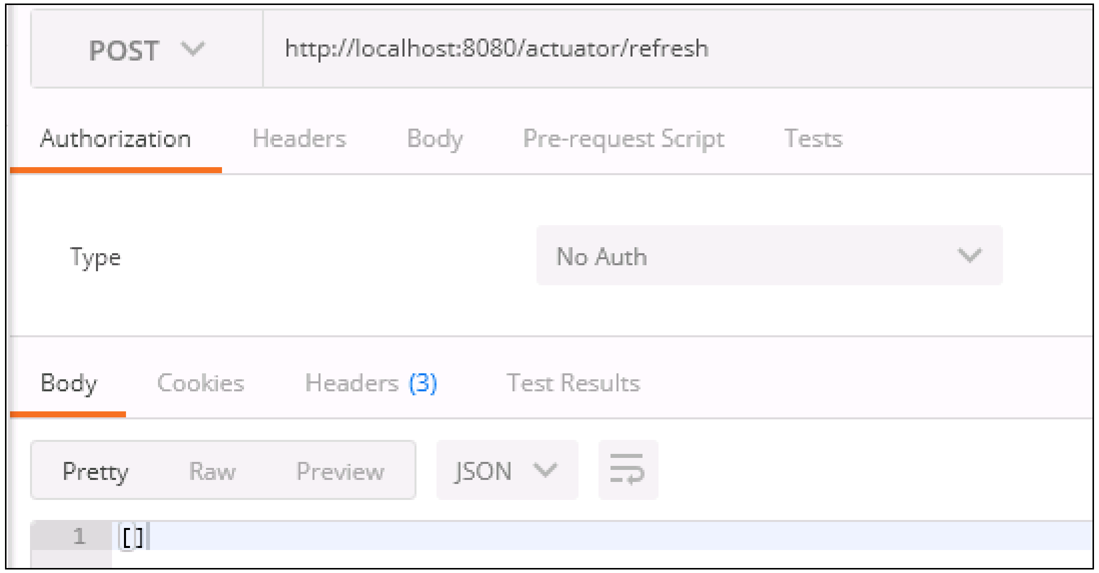
The result above gives us an error (Not Found). Add the below config in bootstrap.properties of limits-service.



Spring Boot 2.0.0+ > Enable all Actuator URLS

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

And fire the post request again. This time you get a 200 response.

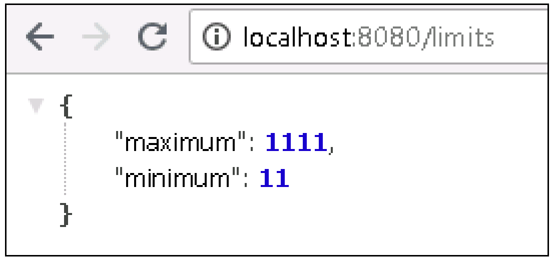


http://localhost:8080/actuator/refresh

Type: No Auth

```
[]
```

And the changes made in git-localconfig-repo is reflected in the limits-service.



localhost:8080/limits

```
{
    "maximum": 1111,
    "minimum": 11
}
```

Limits-service is running as 2 instances so we can fire a post request for each of them for the changed config in git-localconfig-repo to reflect into limits-service. But if there are 100 instances then it will be very difficult to fire 100 requests. Spring Cloud Bus provides a solution here. We can have one url for all instances, once you hit that url, all the instances would be updated with the latest values from the git configuration.

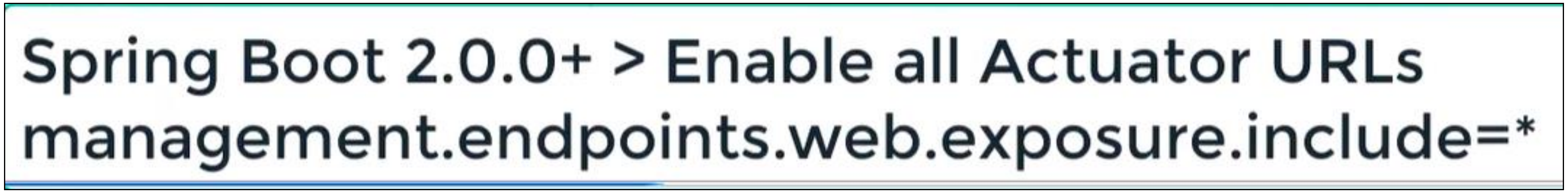
----

52. Step 43 - Implementing Spring Cloud Bus

We will use rabbit MQ. Amqp protocol that rabbit MQ uses. Connect spring-cloud-config-server and limits-service to spring cloud bus. Add the below dependency to pom.xml of both the modules.

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

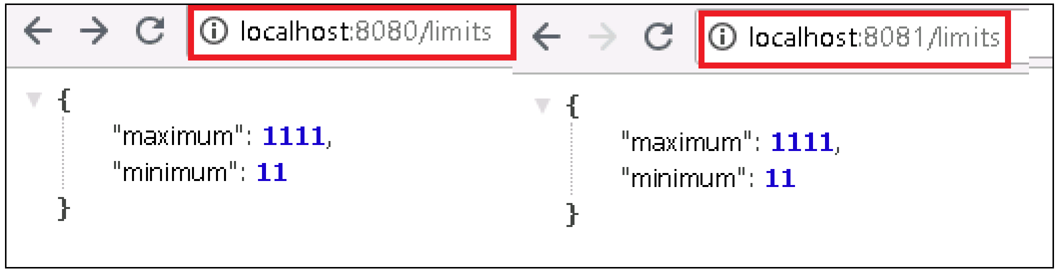
Add the below config in bootstrap.properties of limits-service.



Spring Boot 2.0.0+ > Enable all Actuator URLs

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

The current o/p of limits-service url:



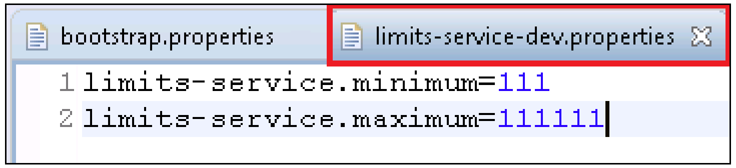
localhost:8080/limits

```
{
    "maximum": 1111,
    "minimum": 11
}
```

localhost:8081/limits

```
{
    "maximum": 1111,
    "minimum": 11
}
```

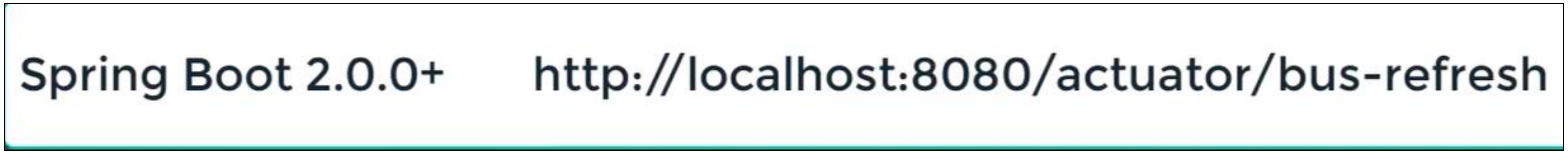
Let us now change the config in git repo and see if it reflects automatically in limits-service url.



bootstrap.properties | limits-service-dev.properties

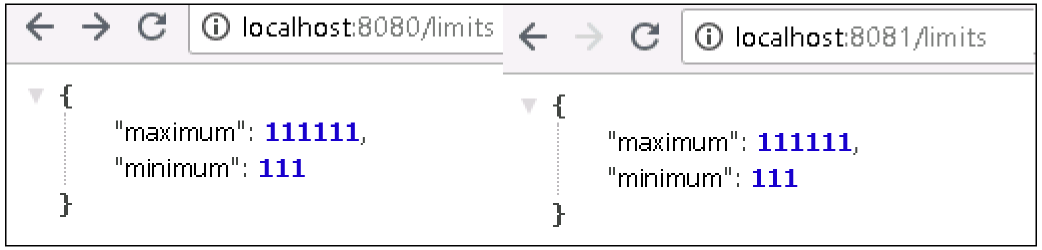
```
limits-service.minimum=111
limits-service.maximum=111111
```

Go to the postman and fire the below post request.



Spring Boot 2.0.0+ http://localhost:8080/actuator/bus-refresh

The above url will refresh the configurations for all the instances of the microservices.



localhost:8080/limits

```
{
    "maximum": 111111,
    "minimum": 111
}
```

localhost:8081/limits

```
{
    "maximum": 111111,
    "minimum": 111
}
```

So with one url invocation, we are able to update the configuration for all the instances of the microservice. The way the spring cloud bus works is, at application start up all the microservice instances register with the cloud bus. If limits-service has 3 different instances, then all of them register with the bus. When there is any change in configuration, when refresh is called on any of these instances, the microservice instance would send an event over to the spring cloud bus and the spring cloud bus would propagate that event to all the microservice instances that are registered with it. That is how all the instances would upgrade their configurations from the spring-cloud-config-server. As soon as we add right dependencies, everything is auto-configured for us. We have RabbitMQ running in the background, spring boot detects that and it sees that there is an amqp dependency in the classpath, it would automatically configure a connection to RabbitMQ.

----

53. Step 44 - Fault Tolerance with Hystrix

Microservices architectures consist of a no of components. Instead of having one big monolith application we have no of microservices interacting with each other. It is possible that a couple of microservices might be down somewhere in the entire architecture. If any of these microservices is down then they can pull down the entire chain of microservices that are dependent on them. This is one of the risks of microservices architecture. This is where fault tolerance of microservices comes into picture. One of the things that you should always consider when microservice is being created is what if some functionality does not work as expected. Let us say limits-service depends on some service that is not available, can limits-service provide some default graceful behaviour? The functionality might not be perfect but at least limits-service does not go down, it at least gives well enough response to currency-exchange-service so that it can go ahead and do the work it needs to do. It would prevent entire chain from going down.

Hystrix framework helps us to build fault tolerant microservices. How to achieve this?

Add dependency for Hystrix in pom.xml of limits-service.

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

Enable Hystrix by going to LimitsServiceApplication`.java`

```
package com.in28minutes.microservices.limitsservice;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.hystrix.EnableHystrix;

@SpringBootApplication
@EnableHystrix
public class LimitsServiceApplication {

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

@EnableHystrix → This will enable Hystrix fault tolerance on all the controllers. On all the controller methods we can add below annotation:

If this method throws an exception what is the fallback method. You can specify this on all the services inside your application.

@HystrixCommand(fallbackMethod="fallbackRetrieveConfiguration")

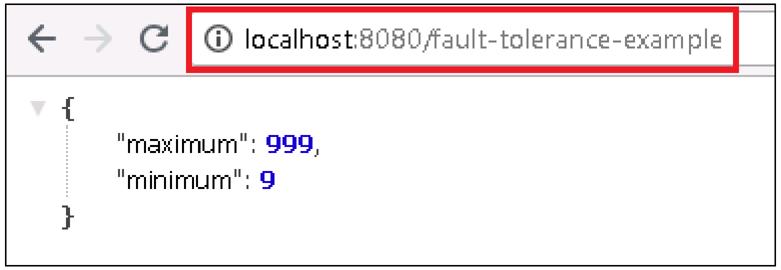
Let us create a method to illustrate Hystrix.

LimitsConfigurationController.java

```
package com.in28minutes.microservices.limitsservice;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import com.in28minutes.microservices.limitsservice.bean.LimitConfiguration;
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

@RestController
public class LimitsConfigurationController {

	@Autowired
	private Configuration configuration;

	@GetMapping("/limits")
	public LimitConfiguration retrieveLimitsFromConfigurations() {

		return new LimitConfiguration(configuration.getMaximum(), configuration.getMinimum());
	}
	@GetMapping("/fault-tolerance-example")
	@HystrixCommand(fallbackMethod="fallbackRetrieveConfiguration")
	public LimitConfiguration retrieveConfiguration() {
```

```
                throw new RuntimeException("Not Available");
        }

        public LimitConfiguration fallbackRetrieveConfiguration() {
                return new LimitConfiguration(999, 9);
        }
}
```

This is very useful in microservice architecture as there is a ripple effect if any of the microservice fails.



localhost:8080/fault-tolerance-example

```
{
    "maximum": 999,
    "minimum": 9
}
```

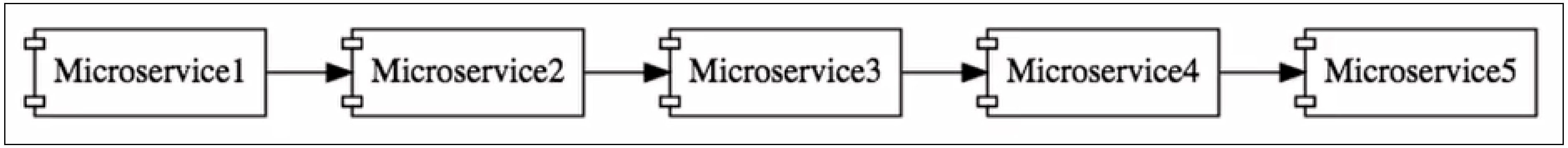
----

54. FAQ 01 - Microservices Characteristics

Some of the common features of microservices:

1) Small
   Microservices are small and have very specific functionality.

2) Independent deployment
   Each of the microservices is independently deployable.



The advantage with this is that we would need to only deploy microservice3 is there is change in that particular component. All the other microservices are unaffected.

3) Simple communication
Very simple communication between microservices. HTTP is the typical protocol used. Most of the microservices offer restful services. This enables microservices based on different technologies to be able to talk to each other.

4) Stateless
Microservices do not have any state. We want all the microservices to be dynamically scaled up/down. This will not be possible if microservices hold any state. So in typical microservices architecture all the microservices are stateless.

5) Automated Build and Deployment
Build, deployment, automation testing are automated for all the microservices.

----

55. FAQ 02 - What do you do next

- Design and Governance of Microservices
  - https://martinfowler.com/microservices/
- 12 Factor App
  - https://12factor.net/
  - https://dzone.com/articles/the-12-factor-app-a-java-developers-perspective
- Spring Cloud
  - http://projects.spring.io/spring-cloud/