# Microservices

**Microservice Architecture** is a Service Oriented Architecture. In the microservice architecture, there are a large number of **microservices**. By combining all the microservices, it constructs a big service. In the microservice architecture, all the services communicate with each other.

In the **Microservices** tutorial, we will understand how to implement microservices using **Spring Cloud**. We will learn how to establish communication between microservices, **enable** **load balancing**, **scaling up and down of microservices**. We will also learn to **centralize the configuration of microservices**with **Spring Cloud Config Server**. We will implement **Eureka Naming Server** and **Distributed tracing** with **Spring Cloud Sleuth** and **Zipkin**. We will create fault tolerance microservices with **Zipkin**.

Our **microservices** tutorial discusses the basic functionalities of **Microservice Architecture**along with relevant examples for easy understanding.

## **What are Microservices**

**Definition**: According to **Sam Newman**, "Microservices are the small services that work together."

According to **James Lewis and Martin Fowler**, "The microservice architectural style is an approach to develop a single application as a suite of small services. Each microservice runs its process and communicates with lightweight mechanisms. These services are built around business capabilities and independently developed by fully automated deployment machinery."

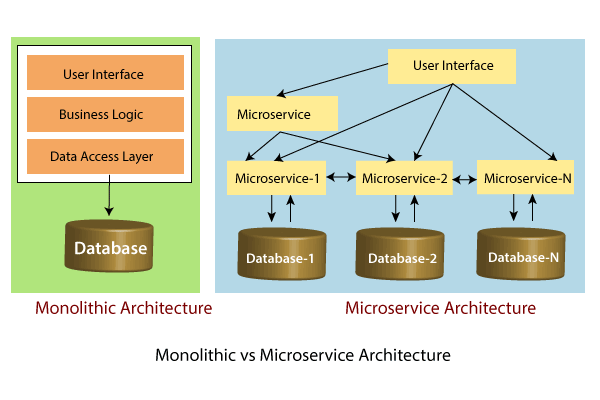
There is a bare minimum of centralized management of these services, which may be written in different programming language and use different data storage technologies.

## **Points to remember**

* These are the services which are exposed by REST.
* These are small well-chosen deployable units.
* The services must be cloud-enabled.

The microservice defines an approach to the architecture that divides an application into a pool of loosely coupled services that implements business requirements. It is next to **Service-Oriented Architecture (SOA)**. The most important feature of the microservice-based architecture is that it can perform **continuous delivery** of a large and complex application.

Microservice helps in breaking the application and build a logically independent smaller applications. For example, we can build a cloud application with the help of Amazon AWS with minimum efforts.



In the above figure, each microservice has its own business layer and database. If we change in one microservice, it does not affect the other services. These services communicate with each other by using lightweight protocols such as HTTP or REST or messaging protocols.

## **Principles of Microservices**

There are the following principles of Microservices:

* Single Responsibility principle
* Modelled around business domain
* Isolate Failure
* Infrastructure automation
* Deploy independently

### **Single Responsibility Principle**

The single responsibility principle states that a class or a module in a program should have only one responsibility. Any microservice cannot serve more than one responsibility, at a time.

### **Modeled around business domain**

Microservice never restrict itself from accepting appropriate technology stack or database. The stack or database is most suitable for solving the business purpose.

### **Isolated Failure**

The large application can remain mostly unaffected by the failure of a single module. It is possible that a service can fail at any time. So, it is important to detect failure quickly, if possible, automatically restore failure.

### **Infrastructure Automation**

The infrastructure automation is the process of scripting environments. With the help of scripting environment, we can apply the same configuration to a single node or thousands of nodes. It is also known as configuration management, scripted infrastructures, and system configuration management.

### **Deploy independently**

Microservices are platform agnostic. It means we can design and deploy them independently without affecting the other services.

Advantages of Microservices

* Microservices are self-contained, independent deployment module.
* The cost of scaling is comparatively less than the monolithic architecture.
* Microservices are independently manageable services. It can enable more and more services as the need arises. It minimizes the impact on existing service.
* It is possible to change or upgrade each service individually rather than upgrading in the entire application.
* Microservices allows us to develop an application which is organic (an application which latterly upgrades by adding more functions or modules) in nature.
* It enables event streaming technology to enable easy integration in comparison to heavyweight interposes communication.
* Microservices follows the single responsibility principle.
* The demanding service can be deployed on multiple servers to enhance performance.
* Less dependency and easy to test.
* Dynamic scaling.
* Faster release cycle.

## **Disadvantages of Microservices**

* Microservices has all the associated complexities of the distributed system.
* There is a higher chance of failure during communication between different services.
* Difficult to manage a large number of services.
* The developer needs to solve the problem, such as network latency and load balancing.
* Complex testing over a distributed environment.

Challenges of Microservices Architecture

Microservice architecture is more complex than the legacy system. The microservice environment becomes more complicated because the team has to manage and support many moving parts. Here are some of the top challenges that an organization face in their microservices journey:

* Bounded Context
* Dynamic Scale up and Scale Down
* Monitoring
* Fault Tolerance
* Cyclic dependencies
* DevOps Culture

**Bounded context**: The bounded context concept originated in Domain-Driven Design (DDD) circles. It promotes the Object model first approach to service, defining a data model that service is responsible for and is bound to. A bounded context clarifies, encapsulates, and defines the specific responsibility to the model. It ensures that the domain will not be distracted from the outside. Each model must have a context implicitly defined within a sub-domain, and every context defines boundaries.

In other words, the service owns its data and is responsible for its integrity and mutability. It supports the most important feature of microservices, which is independence and decoupling.

**Dynamic scale up and scale down**: The loads on the different microservices may be at a different instance of the type. As well as auto-scaling up your microservice should auto-scale down. It reduces the cost of the microservices. We can distribute the load dynamically.

**Monitoring**: The traditional way of monitoring will not align well with microservices because we have multiple services making up the same functionality previously supported by a single application. When an error arises in the application, finding the root cause can be challenging.

**Fault Tolerance**: Fault tolerance is the individual service that does not bring down the overall system. The application can operate at a certain degree of satisfaction when the failure occurs. Without fault tolerance, a single failure in the system may cause a total breakdown. The circuit breaker can achieve fault tolerance. The circuit breaker is a pattern that wraps the request to external service and detects when they are faulty. Microservices need to tolerate both internal and external failure.

**Cyclic Dependency**: Dependency management across different services, and its functionality is very important. The cyclic dependency can create a problem, if not identified and resolved promptly.

**DevOps Culture**: Microservices fits perfectly into the DevOps. It provides faster delivery service, visibility across data, and cost-effective data. It can extend their use of containerization switch from Service-Oriented-Architecture (SOA) to Microservice Architecture (MSA).

## **Other challenges of microservices**

* As we add more microservices, we have to be sure they can scale together. More granularity means more moving parts, which increase complexity.
* The traditional logging is ineffective because microservices are stateless, distributed, and independent. The logging must be able to correlate events across several platforms.
* When more services interact with each other, the possibility of failure also increases.

Difference between Microservices Architecture (MSA) and Services-Oriented Architecture (SOA)

|  |  |
| --- | --- |
| **Microservice Based Architecture (MSA)** | **Service-Oriented Architecture (SOA)** |
| Microservices uses **lightweight protocols** such as **REST**, and **HTTP**, etc. | SOA supports **multi-message protocols**. |
| It focuses on **decoupling**. | It focuses on application service **reusability**. |
| It uses a **simple messaging system** for communication. | It uses **Enterprise Service Bus** (ESB) for communication. |
| Microservices follows "**share as little as possible**" architecture approach. | SOA follows "**share as much as possible architecture**" approach. |
| Microservices are much better in **fault tolerance** in comparison to SOA. | SOA is not better in fault tolerance in comparison to MSA. |
| Each microservice have an **independent** database. | SOA services share the **whole** data storage. |
| MSA used **modern** relational databases. | SOA used **traditional** relational databases. |
| MSA tries to **minimize** sharing through bounded context (the coupling of components and its data as a single unit with minimal dependencies). | SOA **enhances** component sharing. |
| It is better suited for the **smaller** and **well portioned**, web-based system. | It is better for a **large** and **complex** business application environment. |

# Microservices Monitoring

Monitoring is the control system of the microservices. As the microservices are more complex and harder to understand its performance and troubleshoot the problems. Given the vivid changes to software delivery, it is required to monitor the service. There are **five** principles of monitoring microservices, as follows:

* Monitor container and what's inside them.
* Alert on service performance.
* Monitor services that are elastic and multi-location.
* Monitor APIs.
* Monitor the organizational structure.

These principles allow us to address technological changes associated with the microservices and organizational changes related to them.

## **Microservices Monitoring Tool**

There are three monitoring tools are as follows:

* Hystrix dashboard
* Eureka admin dashboard
* Spring boot admin dashboard

## **Microservice Virtualization**

Microservices virtualization is the method to simulate the behavior of specific components in various component-based application like cloud-based application, SOA, and API driven architecture. Service virtualization also reduces cost and save time. By combining service virtualization, an organization can develop the application which can be delivered from various locations and dissimilar environments.

# Components of Microservices

There are the following components of microservices:

* Spring Cloud Config Server
* Netflix Eureka Naming Server
* Hystrix Server
* Netflix ZuulAPI Gateway Server
* Netflix Ribbon
* Zipkin Distributed Tracing Server

### **Spring Cloud Config Server**

Spring Cloud Config Server provides the HTTP resource-based API for external configuration in the distributed system. We can enable the Spring Cloud Config Server by using the annotation **@EnableConfigServer**.

### **Netflix Eureka Naming Server**

Netflix Eureka Server is a discovery server. It provides the REST interface to the outside for communicating with it. A microservice after coming up, register itself as a discovery client. The Eureka server also has another software module called **Eureka Client**. Eureka client interacts with the Eureka server for service discovery. The Eureka client also balances the client requests.

### **Hystrix Server**

Hystrix server acts as a fault-tolerance robust system. It is used to avoid complete failure of an application. It does this by using the **Circuit Breaker mechanism**. If the application is running without any issue, the circuit remains closed. If there is an error encountered in the application, the Hystrix Server opens the circuit. The Hystrix server stops the further request to calling service. It provides a highly robust system.

### **Netflix Zuul API Gateway Server**

Netflix Zuul Server is a gateway server from where all the client request has passed through. It acts as a unified interface to a client. It also has an inbuilt load balancer to load the balance of all incoming request from the client.

### **Netflix Ribbon**

Netflix Ribbon is the client-side Inter-Process Communication (IPC) library. It provides the client-side balancing algorithm. It uses a Round Robin Load Balancing:

* Load balancing
* Fault tolerance
* Multiple protocols(HTTP, TCP, UDP)
* Caching and Batching

### **Zipkin Distributed Server**

Zipkin is an open-source project m project. That provides a mechanism for sending, receiving, and visualization traces.

One thing you need to be focused on that is port number.

|  |  |
| --- | --- |
| **Application** | **Port** |
| Spring Cloud Config Server | 8888 |
| Netflix Eureka Naming Server | 8761 |
| Netflix Zuul API gateway Server | 8765 |
| Zipkin distributed Tracing Server | 9411 |

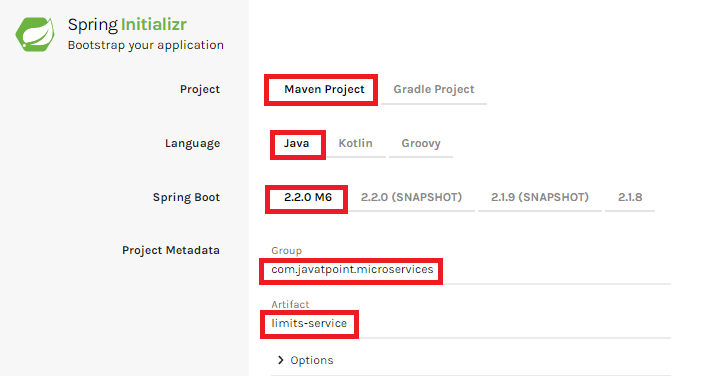
# Creating a Simple Microservice

**Step 1**: Create a Maven project using Spring Initializr <https://start.spring.io/>

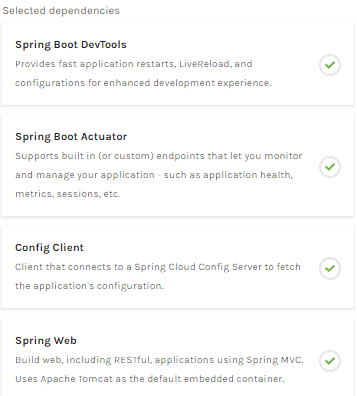
**Step 2**: Choose the Spring Boot version **2.2.0 M6** or higher version. Do not choose the snapshot version.

**Step 3**: Provide the **Group** name. In our case **om.javatpoint**

**Step 4**: Provide the **Artifact id**. We have provided **limits-service**.



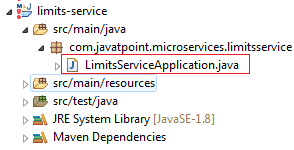
**Step 5**: Add the following dependencies: **Spring Web, Spring Boot DevTools, Spring Boot Actuator, Config Client**.



**Step 6**: Click **on Generate the project** button. A **zip** file will download, extract it into the hard disk.

**Step 7**: Now, open the **eclipse**. Import the created maven project. It takes some time to download the required files.

**Step 8**: Once the project is downloaded, go to **src/main/java**. Open the **LimitsServiceApplication**.



**Step 9**: Now run the **LimitsServiceApplication.java** as Java Application.

**It started the Tomcat on port(s) 8080 (http).**

Now we will add couple of services in the above project. For this we will have to follow the following steps:

**Step 1**: Open **application.properties** file and write the following code:

1. spring.application.name=limits-service      //name of application

**Step 2**: Create a class file with name **LimitsConfigurationController.java** in the folder src/main/java under the package **com.javatpoint.microservices.limitsservice** and write the following code:

**package** com.javatpoint.microservices.limitsservice;

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

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

**import** com.javatpoint.microservices.limitsservice.bean.LimitConfiguration;

@RestController

**public** **class** LimitsConfigurationController

{

@GetMapping("/limits")

**public** LimitConfiguration retriveLimitsFromConfigurations()

{

**return** **new** LimitConfiguration(1000, 1);

}

}

**Step 3**: Create a class file with name **LimitConfiguration.java** in the folder **src/main/java** under the package **com.javatpoint.microservices.limitservice.bean** and write the following code:

**package** com.javatpoint.microservices.limitsservice.bean;

**public** **class** LimitConfiguration

{

**private** **int** maximum;

**private** **int** minimum;

//no-argument constructor

**protected** LimitConfiguration()

{

}

//generating getters

**public** **int** getMaximum()

{

**return** maximum;

}

**public** **int** getMinimum()

{

**return** minimum;

}

//genetrating constructor using fields

**public** LimitConfiguration(**int** maximum, **int** minimum)

{

**super**();

**this**.maximum = maximum;

**this**.minimum = minimum;

}

}

Type the **localhost:8080/limits** in the browser and press enter, we get the JSON response as output.

**Output**

{

maximum: 1000,

minimum: 1

}

## **Adding services to the application.properties**

In the previous program, we will modify the code according to the requirement.

Now we call the **limits-service** from the **application.properties** file. In this file, we are configuring a couple of values.

1. limits-service.minimum=99
2. limits-service.maximum=9999

There is a better approach in Spring Boot to read values from the configuration using the annotation **@ConfigurationProperties**.

**Step 1**: Create a class with name **Configuration.java** in the folder **src/main/java** under the package **com.javatpoint.microservices.limitservice**.

**Step 2**: Add the annotations **@Component** and **@ConfigurationProperties**.

**Step 3**: Declare two variables **minimum** and **maximum**.

**Step 4**: If we are using the Configuration file, we need to generate getters and setters.

The Configuration.java file look like this.

**package** com.javatpoint.microservices.limitsservice;

**import** org.springframework.boot.context.properties.ConfigurationProperties;

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

@Component

@ConfigurationProperties("limits-service")

**public** **class** Configuration

{

**private** **int** maximum;

**private** **int** minimum;

**public** **void** setMaximum(**int** maximum)

{

**this**.maximum = maximum;

}

**public** **void** setMinimum(**int** minimum)

{

**this**.minimum = minimum;

}

**public** **int** getMaximum()

{

**return** maximum;

}

**public** **int** getMinimum()

{

**return** minimum;

}

}

**Step 5**: Now move to **LimitsConfigurationController.java** file and modify the code. In this we will use Configuration.

**package** com.javatpoint.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.javatpoint.microservices.limitsservice.bean.LimitConfiguration;

@RestController

**public** **class** LimitsConfigurationController

{

@Autowired

**private** Configuration configuration;

@GetMapping("/limits")

**public** LimitConfiguration retriveLimitsFromConfigurations()

{

//getting values from the properties file

**return** **new** LimitConfiguration(configuration.getMaximum(), configuration.getMinimum());

}

}

Now refresh the browser page. It shows the JSON format of the updated values which are configured in **application .properties** file.

**Output**

{

maximum: 999,

minimum: 99

}

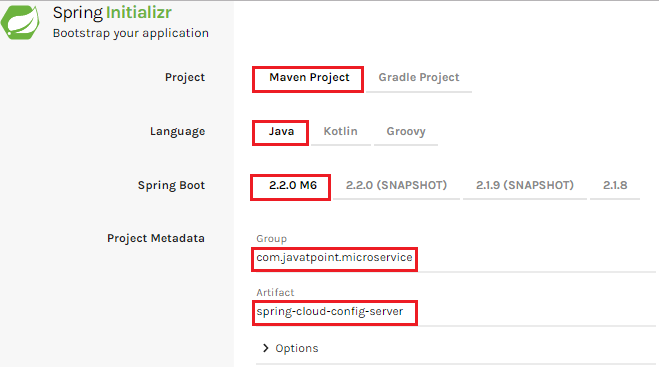
# Setting up Spring Cloud Config Server

**Step 1:** Create a Maven project using Spring Initializr <https://start.spring.io/>

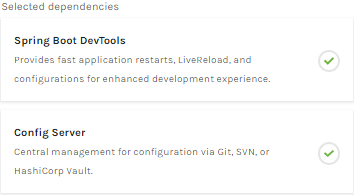
**Step 2:** Choose the Spring Boot version **2.2.0 M6** or higher version. Do not choose the snapshot version.

**Step** **3:** Provide the **Group** name. In our case, **com.javatpoint.microservices.**

**Step 4:** Provide the **Artifact id**. We have provided **spring-cloud-config-server.**



**Step 5:**Add the **Spring Boot DevTools**and**Config Server**dependencies**.**



**Step 6:** Click on**Generate the project** button. A zip file will download, extract it in the hard disk.

**Step 7:** Now, open the **eclipse.** Import the downloaded maven project. It will download the required files.

In the next step, we will create a simple Git repository and configure the spring cloud config server to pick up the values from the particular Git repository. We need to install the local Git.

## **Installing Git and creating a local repository**

**Step 1:**Download Git from <https://git-scm.com/> and install it.

**Step 2:**Create a Git repository and store the files that we want to be able to configure a limits-service. We will try to access them from the spring-cloud-config-server. Open the Git bash and type the following commands:

**Creating a new directory:**

1. mkdir git-localconfig-repo
2. cd git-localconfig-repo/
3. git init

It initializes an **empty**git repository.

**Step 3:**Now move to the **spring-cloud-config-server** project and add a link to the specific folder.

1. Right-click on the **spring-cloud-config-server**project**.**
2. Click on **Build Path**->**Configure Build Path**…
3. Select the **Source** tab.
4. Click on **Link Source** and browse the folder **git-localconfig-repo**.
5. Right click on the folder-> **New** -> **Other** -> **File** -> **Next** -> Provide the file name**: limits-service-properties**-> **Finish**.
6. Now write the following code in the properties file:
7. limits-service.minimum=8
8. limits-service.maximum=888

**Step 4:**Configure the user name and user email:

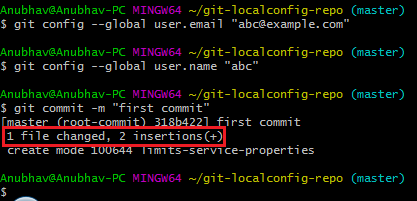
1. git config -global user.email abc@example.com
2. git config -global user.name "abc"

The command commits any file we have added with the git add command and also commits any files we have changed since then.

1. git add -A

Now execute the command to commit the changes in the repository. It records or snapshots the file permanently in the version history.

1. git commit -m "first commit"



We can see that a file is changed with two new instructions. These instructions are changed in the local repository.

# Connect Spring Cloud Config Server to Local Git Repository

In this section, we are going to learn how to connect spring-cloud-config-server to the local git repository. First, we will find the folder path.

Right-click on **git-localconfig-repo** -> **Properties** -> copy the **Location** label address and paste it into the **application.properties** file.

Add the annotation **@EnableConfigServer**in the SpringCloudConfigServerApplication.java file.

Type the following URL in the browser:

**localhost:8888/limits-service/default**

**Output**

{

name: "limits-service",

-profiles: [

"default"

],

label: **null**,

version:"0898c54ae1deb62733728e37e4c7962f529ee9ad",

state: **null**,

-propertySources: [

- {

name: C:\Users\Anubhav\git-localconfig-repo\limits-service.properties",

-source: {

limits-service-minimum: "8",

limits-service-maximum: "88"

}

}

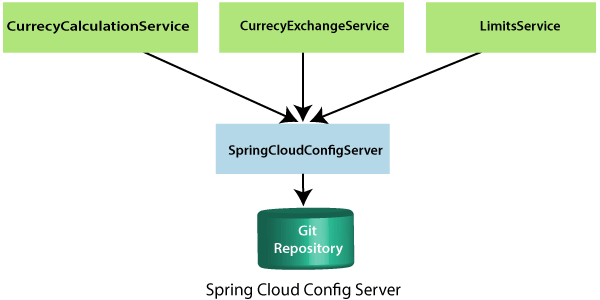
]

}

In this we have establish the connection between **SprinCloudConfigServer** and the **Git repository**.

We can see that it displays a set of property and values. It also retrieves the file name of the property file from where these values (minimum and maximum) are retrieved.

The important thing about SpringCloudConfigServer is that **it stores configuration for multiple services.**It can also store configuration for each of the services for different environments.



In the above figure, there are three services **CurrencyCalculationService**, **CurrencyExchangeService**, and **LimitsService**. The LimitsService has four environment services **Dev, QA, Stage,** and **Production**. We can configure these three services in SpringCloudConfigServer.

## **Configuration for Multiple Environment in Git Repository**

services **Dev, QA, Stage,** and **Production**. We can configure these three services in SpringCloudConfigServer.

**Configuration for Multiple Environment in Git Repository**

In the spring-cloud-config-server project, we have added a link to git-localconfig-repo, which contains the limits-service.properties file. It becomes the default configuration for the limits-service.

However, we can overwrite them for a specific environment. To overwrite these values, copy the **limits-service.properties** and paste in the folder **git-localconfig-repo**rename it with **limits-service-dev.properties**. Now update the minimum and maximum values.

1. limits-service.minimum=1
2. limits-service.maximum=111

Again copy the same file and paste it in the same folder. Rename it with **limits-service-qa.properties**. Now update the minimum and maximum values.

1. limits-service.minimum=2
2. limits-service.maximum=222

If we want to pick the default value of the maximum instead of modified value, put a **introduction-to-currency-conversion-and-currency-exchange-service** symbol at the starting of the statement. Now the second statement becomes a comment.

1. limits-service.minimum=1
2. introduction-to-currency-conversion-and-currency-exchange-servicelimits-service.maximum=111

When we execute it, it picks up the maximum value 888 from the default properties file instead of maximum value 111. Whenever we make the changes in the file, commit the changes in the local repository.

Now open the Git Bash and execute the following commands:

Create the directory in which we want to add files.

1. cd git-localconfig-repo

Add the files into the Git repository.

1. git add -A

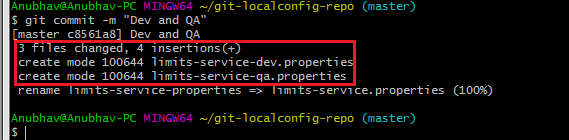
Now check the status of the files that have to be committed.

1. git status



Now commit the changes

1. git commit -m "Dev and QA"



Now we can access the properties Dev and QA.

Type the following in the address bar of the browser.

1. localhost:8888/limits-service/qa

**Output**

1. {
2. name: "limits-service",
3. -profiles: [
4. "qa"
5. ],
6. label: **null**,
7. version:"0898c54ae1deb62733728e37e4c7962f529ee9ad",
8. state: **null**,
9. -propertySources: [
10. - {
11. name: C:\Users\Anubhav\git-localconfig-repo\limits-service-qa.properties",
12. -source: {
13. limits-service-minimum: "2",
14. limits-service-maximum: "222"
15. }
16. },
17. -{
18. name: C:\Users\Anubhav\git-localconfig-repo\limits-service.properties?,
19. -source: {
20. limits-service-minimum: "8",
21. limits-service-maximum: "888"
22. }
23. }
24. ]
25. }

We can observe that it is retrieving the property sources. These list of property are in the list of priority. The heights priority is whatever values are configured in the QA file.

If there is a value that is not present in the QA file, then the value from the default file will be picked up. So whatever is in the QA file gets the highest property.

## **Connect limits-service to Spring Cloud Config Server**

In this section, we will connect limits-service to pick up the configuration from the spring-cloud-config-server. We do not need to configure values in the application.properties file. Move to the **limits-service** project and rename the **application.properties** file to **bootstrap.properties**. We do not need to configure values in the bootstrap.properties. All the configuration values picked from the spring-cloud-config-server. Specify the URI in the bootstrap.properties.

1. spring.application.name=limits-service
2. spring.cloud.config.uri=http://localhost:8888

**limits-service** is the critical path of the bootstrap.properties. Based on the application name, we are going to pick up values from the local Git repository. Now restart the **LimitsServiceApplication.java.**

1. Fetching config from the server at http://localhost:8888
2. Located environment: name=limits-service, profiles=[**default**], label= **null**,  version="0898c54ae1deb62733728e37e4c7962f529ee9ad", state=**null**,

## **Configuring profiles for Limit Service**

The point to understand here is that all the configuration for the limits-service is coming from the Git repository. We did not configure anything in the limits-service. The advantage of configuring stuff in the Git repository is that the entire configuration of limits-service is separated from the deployment of the limits-service. It will automatically pick up from the Git repository.

Now open the **bootstrap.properties** and add the **dev** profile into it.

1. spring .profile.active=dev

When we run the limits, it shows the following output:

1. {
2. maximum: 111,
3. minimum:1
4. }

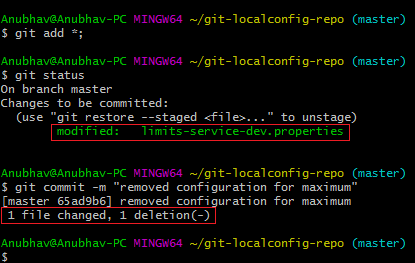
If we look at the limits-service-dev.properties file, the values are fetching from there.

Suppose we want to pick a maximum value from the **limits-service.properties** and minimum value from **limits-service-dev.properties** then remove the maximum value from the **limits-service-dev.properties**. The limits-service-dev.properties file looks like this:

1. limits-service-minimum: 1

Now commit the changes by using the following commands:

1. git add \*;
2. git status
3. git commit -m "removed configuration for maximum "



Now start the **LimitsServiceApplication.java**. When we start the LimitsServiceApplication, it picks values from the SpringCloudConfigServer. We can observe that it picks the maximum value from the limits-service.properties (default service) that is**888**and the minimum value from the **limit-service-dev.properties**that is **1.**However, we have overwritten the minimum value of the default service.

Let's see what happens when we change the profile **dev** to **qa.**Open **bootstrap.properties**and write **qa** in place of **dev**. The application will start and pick up the changes. Now execute the **limits.**

**Output**

{

maximum: 222,

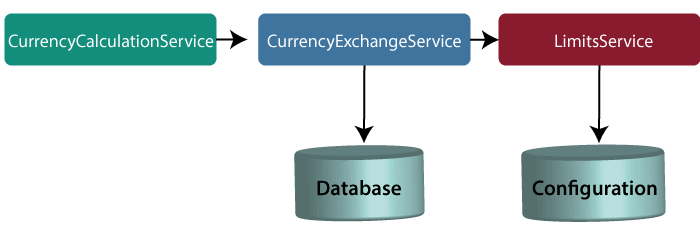
minimum: 2

}

These are the values that are coming from the qa environment configuration.

# Introduction to Currency Conversion and Currency Exchange Service

In this section, we will create a couple of microservices: **CurrencyCalculationService** and **CurrencyExchangeService**.



#### Note**: In this tutorial, we have quoted currency conversion service as a currency calculation service. Both the services have the same meaning, so don't be confused.**

Let's understand the functionality of these services.

In the above figure, the CurrencyExchangeService uses JPA to talk to the database and returns the exchange value of the specific currency. For example, USD to INR conversion.

When we invoke CurrencyExchangeService, we need to pass two parameters: **from**(convert from), and **to** (convert to). For example, if we want to convert currency from **USD** to **INR**.

Consider the URL **http://localhost:8000/currency-exchange/from/USD/to/INR**. It retunes the following response:

{

id: 101,

from: "USD",

to: "INR",

conversionMultiple: 72,

port: 8000

}

The currency exchange service will return what the conversion multiple is. The conversion multiple means **1 USD** is equal to **72 INR**. The currency converter service uses a currency exchange service. Suppose the currency converter service wants to convert 100 USD to INR. So it will call the currency exchange service and will convert the specified amount that we have provided in the path parameter. For example:

**http://localhost:8100/currency-converter/from/USD/to/INR/quantity/100**

{

Id: 101,

from: "USD",

to: "INR",

conversionMultiple: 72,

quantity: 100

totalCalculatedAmount: 7200,

port: 8000

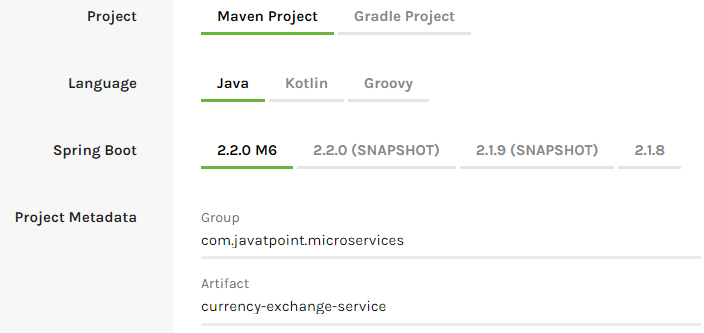
}

We will implement these two services in our example using Spring Cloud.

## **Setting up a currency-exchange-service**

**Step 1:**Open the spring initializer [http://start.spring.io](https://start.spring.io/).

**Step 2:**Select the **Project**: Maven Project, **Language:**Java, and Spring Boot version **2.2.0 M6**or above. Provide the **Group name** and **Artifact ID.**We have provided**com.javatpoint.microservices**and **currency-exchange-service,**for group name and Artifact id respectively.



**Step 3:**Add the dependencies **Web, DevTools, Actuator,**and **Config Client**.

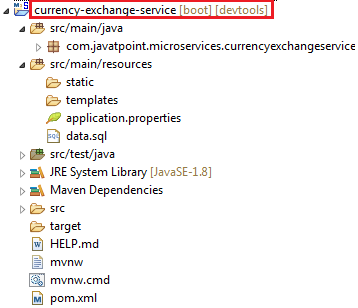
**Step 4:**Click on the **Generate Project** button. It will download the **zip** file of the project.

**Step 5: Extract** it in the local disk.

**Step 6: Import** the project.

Click on File menu-> Import -> Existing Maven Projects -> Next -> Browse ->Select the project ->Finish

It takes some time to import. When the project import is done, it shows the following project directory. Do not consider the data.sql file in the directory, because we will create it later.



**Step 7:**Open the **application.properties** file and configure the **application name** and **port** number.

**application.properties**

1. spring.application.name=currency-exchange-service.
2. server.port=8000

When we run the currency-exchange-service, it runs but does not perform any service. In the next step, we will implement code in the currency-exchange-service.

## **Hardcoded the currency-exchange-service**

Now we will create a service that converts the currency from USD to INR.

**Step 1:**Create a class file (REST Controller) with the name **CurrencyExchangeController** in the package **com.javatpoint.microservices.currencyexchangeservice.**

**CurrencyExchangeController.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

**import** java.math.BigDecimal;

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

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

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

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

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

@SpringBootApplication

@RestController

**public** **class** CurrencyExchangeController

{

@GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable

**public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)  //from map to USD and to map to INR

{

**return** **new**  ExchangeValue(1000L, from, to, BigDecimal.valueOf(65));

}

}

**Step 2:**Create a class file with the name **ExchangeValue.**

**ExchangeValue.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

**import** java.math.BigDecimal;

**public** **class** ExchangeValue

{

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal conversionMultiple;

**public** ExchangeValue()

{

}

//generating constructor using fields

**public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple) {

**super**();

**this**.id = id;

**this**.from = from;

**this**.to = to;

**this**.conversionMultiple = conversionMultiple;

}

//generating getters

**public** Long getId()

{

**return** id;

}

**public** String getFrom()

{

**return** from;

}

**public** String getTo()

{

**return** to;

}

**public** BigDecimal getConversionMultiple()

{

**return** conversionMultiple;

}

}

**Step 3:**Run the **CurrencyExchangeServiceApplication.java.**It runs on the port **8000** that we have configured in the application.properties file.

We get the following response on the browser:

{

id: 101,

from: "USD",

to: "INR",

conversionMultiple: 72,

port: 8000

}

## **Setting up Dynamic port in the Response**

The CurrencyExchangeService determines the exchange value of the currency. The CurrencyCalculationService uses the CurrencyExchangeService to determine the value of one currency in other currency. We will create multiple instances of the **CurrencyExchangeService** later in next topic.

At present, the service is running on port **8000**. Later we will run it on port **8001, 8002,** and so on.  In the next step, we will set a port to the currency-exchange-service.

**Step 1:** Open the **ExchangeValue.java** file and add a **port**variable. Generate getters and setters for the port variable only.

**ExchangeValue.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

**import** java.math.BigDecimal;

**public** **class** ExchangeValue

{

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal conversionMultiple;

**private** **int** port;

**public** ExchangeValue()

{

}

//generating constructor using fields

**public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple) {

**super**();

**this**.id = id;

**this**.from = from;

**this**.to = to;

**this**.conversionMultiple = conversionMultiple;

}

//generating getters

**public** **int** getPort() {

**return** port;

}

**public** **void** setPort(**int** port) {

**this**.port = port;

}

**public** Long getId()

{

**return** id;

}

**public** String getFrom()

{

**return** from;

}

**public** String getTo()

{

**return** to;

}

**public** BigDecimal getConversionMultiple()

{

**return** conversionMultiple;

}

}

We have already configured the application name and port number in the application.properties file, so need not to configure again.

Now pick up port number from the environment.

**Step 3**: Open the **CurrencyExchangeController.java** and get the property of the environment.

**CurrencyExchangeController.java.**

**package** com.javatpoint.microservices.currencyexchangeservice;

**import** java.math.BigDecimal;

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

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

**import** org.springframework.core.env.Environment;

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

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

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

@SpringBootApplication

@RestController

**public** **class** CurrencyExchangeController

{

@Autowired

**private** Environment environment;

@GetMapping("/currency-exchange/from/{from}/to/{to}") //where {from} and {to} are path variable

**public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)  //from map to USD and to map to INR

{

//taking the exchange value

ExchangeValue exchangeValue= **new** ExchangeValue (1000L, from, to, BigDecimal.valueOf(65));

//picking port from the environment

exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));

**return** exchangeValue;

}

}

When we refresh the browser, the URL changes to: **http://localhost:8000/currency-exchange/from/USD/to/INR**.

{

id: 1000,

from: "USD",

to: "INR"

conversionMultiple: 65,

port: 8000

}

At present **CurrencyExchangeServiceApplication** is running on port **8000**.

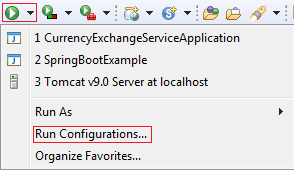
Now we will run **CurrencyExchangeServiceApplication** on a different port number. For this, we have to change the port in the **application.properties**file from 8000 to 8001, 8002, etc. whichever we want.

Suppose we want to create two instances of the **CurrencyExchangeServiceApplication**. For this, we have to set port externally.

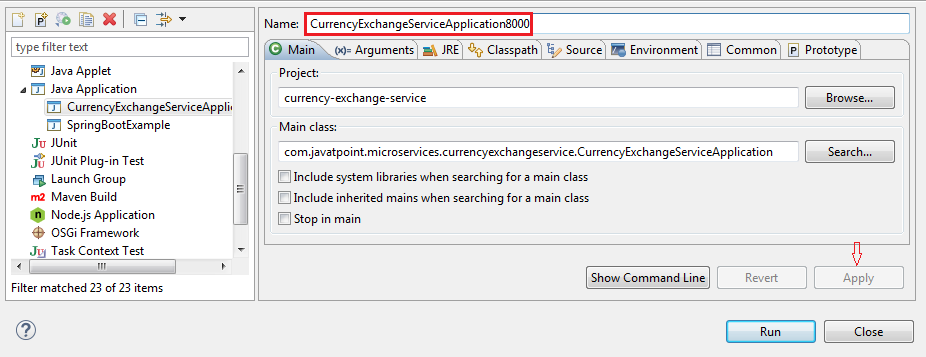
Let's create an instance of the **CurrencyExchangeServiceApplication**that runs on the port **8001**.

**Step 1:**Right-click on the project -> Run As -> Run Configurations.

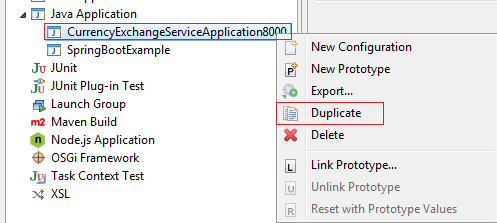
Or click on the highlighted symbol -> Run Configurations.



**Step 2: Rename**the**CurrencyExchangeServiceAppication** to**CurrencyExchangeServiceAppication8000**and click on the**Apply**button.



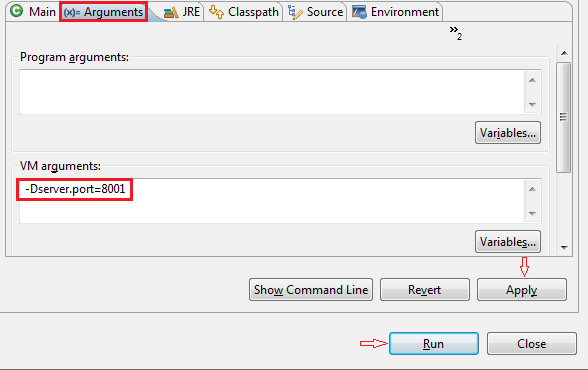
**Step 3:**Right-click on the **CurrencyExchangeServiceApplication8000**-> Duplicate.



It generates the duplicate file of **CurrencyExchangeServiceApplication8000.**We will run it on port **8001.**

**Step 4:**Click on the **Arguments** tab and write **–Dserver.port=8001**in the **VM arguments** text box. Click on the **Apply**and **Run** button, respectively.

#### Note**: Whatever value we are passing in the VM arguments, it overwrites the configuration of the application.properties file.**



After clicking on the **Run**button, it starts running on port **8001**.

**Step 5:**Change the port number in the URL **http://localhost:8001/currency-exchange/from/USD/to/INR** and press enter key. We get the following response:

1. {
2. id: 1000,
3. from: "USD",
4. to: "INR",
5. conversionMultiple: 65,
6. port: 8001
7. }

Now we have two instances of **CurrencyExchangeServiceApplication**that are running on two different ports **8000** and **8001**.

Configure JPA and Initialized Data

In the previous section, we have hardcoded the response for the exchange value. It comes from the database. In this section, we will create a connection to the in-memory database.

Let's see how to connect microservice to the H2 database. Follow the following steps to connect microservice to JPA in-memory database.

**Step 1:** Open **pom.xml** of **currency-exchange-service** and add the following two dependencies.

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.4.197</version>

<scope>test</scope>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

<version>2.1.3.RELEASE</version>

</dependency>

Once we have added the dependencies, now we have to define **entity**.

**Step 2:**Open the **ExchangeValue.java**file and do the following:

* Add **@Entity** annotation at the class level.
* Define the **table name** by using the annotation **@Table**.
* Define an **Id** for the entity by adding the annotation **@Id.**
* Define columns by adding the annotation **@Column**above each field and also specify the column name.

**ExchangeValue.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

**import** java.math.BigDecimal;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

**import** javax.persistence.Table;

@Entity

@Table(name="Exchange\_Value")

**public** **class** ExchangeValue

{

@Id

@Column(name="id")

**private** Long id;

@Column(name="currency\_from")

**private** String from;

@Column(name="currency\_to")

**private** String to;

@Column(name="conversion\_multiple")

**private** BigDecimal conversionMultiple;

@Column(name="port")

**private** **int** port;

//default conatructor

**public** ExchangeValue()

{

}

//generating constructor using fields

**public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple)

{

**super**();

**this**.id = id;

**this**.from = from;

**this**.to = to;

**this**.conversionMultiple = conversionMultiple;

}

//generating getters and setters

**public** **int** getPort()

{

**return** port;

}

**public** **void** setPort(**int** port)

{

**this**.port = port;

}

**public** Long getId()

{

**return** id;

}

**public** String getFrom()

{

**return** from;

}

**public** String getTo()

{

**return** to;

}

**public** BigDecimal getConversionMultiple()

{

**return** conversionMultiple;

}

}

We have created the entity, now we have to insert some data into the database.

**Step 3:** Create a **data.sql**file to insert data into database.

Right-click on the folder **src/main/resources** -> New -> File -> Provide the name **data.sql** -> Finish

**Step 4:**Insert the data into data.sql file. We have inserted the following data:

**data.sql**

1. insert into exchange\_value(id,currency\_from,currency\_to,conversion\_multiple,port)
2. values(10001,'USD', 'INR' ,65,0);
3. insert into exchange\_value(id,currency\_from,currency\_to,conversion\_multiple,port)
4. values(10002,'EUR', 'INR' ,75,0);
5. insert into exchange\_value(id,currency\_from,currency\_to,conversion\_multiple,port)
6. values(10003,'AUD', 'INR' ,25,0);

**Step 5:** Open **application.properties**file and enable **H2 console,**configure **URL** and **datasource**. The default JDBC URL is **testdb**. We can specify our own JDBC URL.

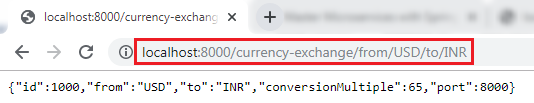
We have specified JDBS URL: **jdbc:h2:mem:javatpoint**

**application.properties**

1. spring.application.name=currency-exchange-service
2. server.port=8000
3. spring.jpa.show-sql=**true**
4. spring.h2.console.enabled=**true**
5. spring.datasource.platform=h2
6. spring.datasource.url=jdbc:h2:mem:javatpoint

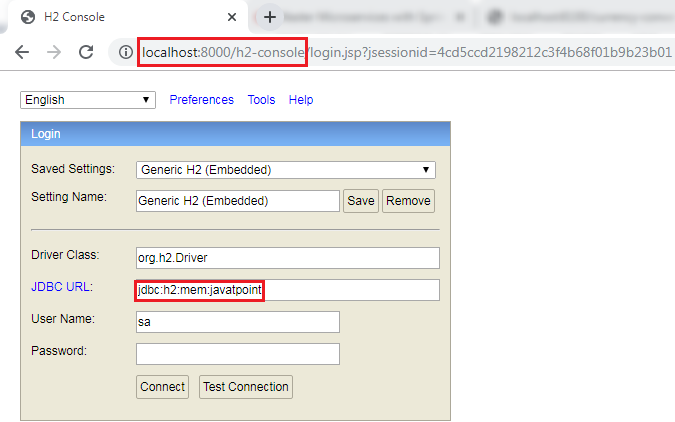
**Step 6:** Restart the application.

**Step 7:**Open the browser and type the URI **http://localhost:8000/currency-exchange/from/USD/to/INR**. It returns the response, as shown below:



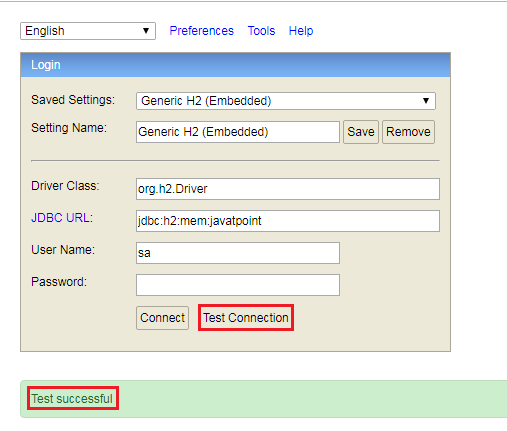
We can also see the data in the database which we have inserted in the **data.sql** file. To open the H2 Console, we have to do the following:

* In the browser type <http://localhost:8000/h2-console>. It displays the following page:

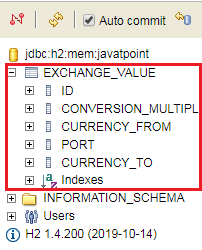


**Remember:** The **JDBC URL** must be the same as you have specified in the **application.properties** file. Do not write anything in the **User Name** and **Password** field. The default User Name is **sa.**

Now click on the **Test Connection**button**;**if the connection is successful, it shows the message **Test Successful.**



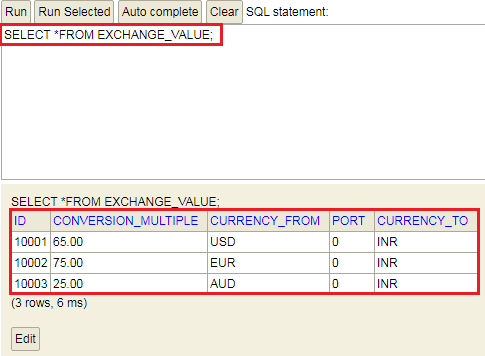
**Step 8:** Click on the **Connect** button. It shows the created table on the left-hand side of the page.



**Step 9:** Run the following query to view the data.

1. SELECT \*FROM EXCHANGE\_VALUE;

It shows the data which we have inserted in the **data.sql** file.



In this section, we have created the in-memory database and inserted some values in the database.

Creating a JPA Repository

In the previous section, we have created a table in-memory database and saw that all the data is populated correctly. In this section, we will create a repository that returns the response for the service.

**Step 1:** Create an interface with the name **ExchangeValueRepository** and extends the **JpaRepository**class. We have to pass **two** parameters: **type of the entity**that it manages and the **type of the Id**field.

**public** **interface** ExchangeValueRepository **extends** JpaRepository<ExchangeValue, Long>

**Step 2:** Open **CurrencyExchageController.java** file and autowired the **ExchageValueRepository**.

@Autowired

**private** ExchangeValueRepository repository;

**Step 3:**Create a **query method**in the **ExcahngeValueRepository.java**file.

ExchangeValue findByFromAndTo(String from, String to);

In the above statement, **ExchangeValue** is the expected response. There are **two** columns that we have to find are **from** and **to**.

If we want to find data on the basis of single column, we can pass a column name. For example:

ExchangeValue findByFrom (String from);

**ExcahngeValueRepository.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

**import** org.springframework.data.jpa.repository.JpaRepository;

**public** **interface** ExchangeValueRepository **extends** JpaRepository<ExchangeValue, Long>

{

//creating query method

ExchangeValue findByFromAndTo(String from, String to);

}

**Step 4:**In the **CurrencyExchangeController.java** use the following statement:

ExchangeValue exchangeValue=repository.findByFromAndTo(from,to);

Instead of using the following statement:

ExchangeValue exchangeValue=**new** ExchangeValue(1000L, from, to, BigDecimal.valueOf(65));

**CurrencyExchangeController.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

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

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

**import** org.springframework.core.env.Environment;

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

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

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

@SpringBootApplication

@RestController

**public** **class** CurrencyExchangeController

{

@Autowired

**private** Environment environment;

@Autowired

**private** ExchangeValueRepository repository;

@GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable

**public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)   //from map to USD and to map to INR

{

ExchangeValue exchangeValue = repository.findByFromAndTo(from, to);

//setting the port

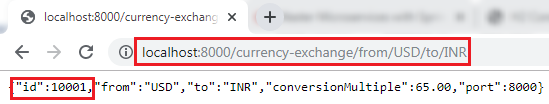
exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));

**return** exchangeValue;

}

}

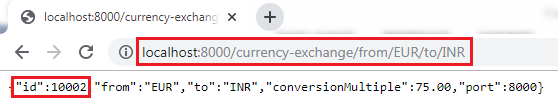
**Step 5:**Restart the application to pick up the changes. Open the browser and type the URI **http://localhost:8000/currency-exchange/from/USD/to/INR**. It returns the following response:



We can also try a different conversion by changing the currency **USD** to **EUR** in the URI.

**http://localhost:8000/currency-exchange/from/EUR/to/INR**.

It returns the following response:



In this above response, we are retrieving the values from the database.

When we pass the currency in the URI (EUR/to/INR), the query gets fired to the database. To see which query gets fired, we can see the query in the log.

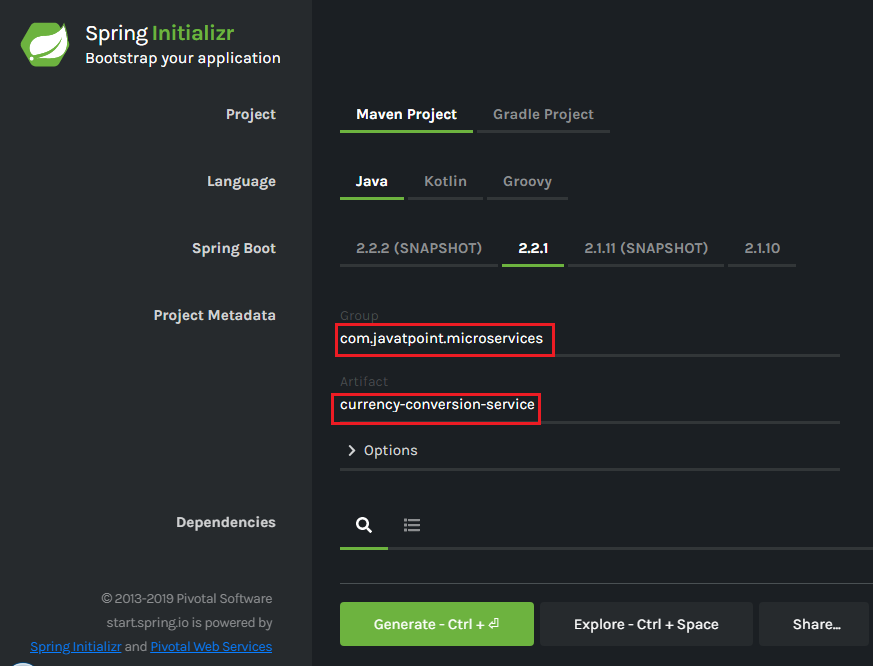
1. Hibernate: select exchangeva0\_.id as id1\_0\_, exchangeva0\_.conversion\_multiple as conversi2\_0\_, exchang

Setting up Currency Conversion Microservice

In the previous section, we have created currency-exchange-service. Now we will create a currency-conversion-service that talks to currency-exchange-service.

**Step 1:**Open the browser and type <https://start.spring.io/>.

* Provide the Group name **javatpoint.microservice** and Artifact **currency-conversion-service.**
* Add the dependencies: **Spring web, DevTools, Actuator,**and **Config Client**.
* Click on the **Generate** It downloads the created project.



**Step 2**: Import the downloaded project in **Spring Tool Suite (STS)**.

File -> Import -> Existing Maven Projects -> Next -> Browse -> Select the project -> Finish.

It takes some time to import the project.

**Step 3:** Open the **application.properties** file and configure the **application name** and **port**number.

**application.properties**

spring.application.name=currency-conversion-service

server.port=8100

The currency-conversion-service runs on port **8100**.

etting up Currency Conversion Microservice

In the next section, we will create a service that talks to the currency-exchange-service.

## **Creating a Service for currency-conversion-service**

In the previous section, we have used EUR to INR that returns what the **conversionMultiple**is. The currency-exchange-service tells what is the exchange value when we convert currency from EUR to INR.

In this section, we will create CurrencyCalculationService. It defines a lot of functionality related to conversion.

We will create a service currency-converter that accepts two path parameters "**from**" and "**to**". It also accepts the quantity (amount which we want to convert).

Let's create a currency-conversion-service.

**Step 1:** Create a class with the name **CurrencyConversionController**.

**Step 2:**Add an annotation **@RestController.**

**Step 3:**Create a **GetMapping**.

**CurrencyConversionController.java**

**package** com.javatpoint.microservices.currencyconversionservice;

**import** java.math.BigDecimal;

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

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

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

@RestController

**public** **class** CurrencyConversionController

{

@GetMapping("/currency-converter/from/{from}/to/{to}/ quantity/{quantity}") //where {from} and {to} represents the column

//return a bean back

**public** CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity)

{

**return** **new** CurrencyConversionBean(1L, from,to,BigDecimal.ONE, quantity,quantity,0 );

}

}

**Step 4:**Create a class with the name **CurrencyConversionBean** and define the following fields:

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal ConversionMultiple;

**private** BigDecimal quantity;

**private** BigDecimal totalCalculatedAmount;

**private** **int** port;

**Step 5:** Generate **Getters** and **Setters**.

**Step 6:** Generate **constructor** and also create a **default** constructor.

**CurrencyConversionBean.java**

**package** com.javatpoint.microservices.currencyconversionservice;

**import** java.math.BigDecimal;

**public** **class** CurrencyConversionBean

{

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal ConversionMultiple;

**private** BigDecimal quantity;

**private** BigDecimal totalCalculatedAmount;

**private** **int** port;

//default constructor

**public** CurrencyConversionBean()

{

}

//creating constructor

**public** CurrencyConversionBean(Long id, String from, String to, BigDecimal conversionMultiple, BigDecimal quantity, BigDecimal totalCalculatedAmount, **int** port)

{

**super**();

**this**.id = id;

**this**.from = from;

**this**.to = to;

ConversionMultiple = conversionMultiple;

**this**.quantity = quantity;

**this**.totalCalculatedAmount = totalCalculatedAmount;

**this**.port = port;

}

//creating setters and getters

**public** Long getId()

{

**return** id;

}

**public** **void** setId(Long id)

{

**this**.id = id;

}

**public** String getFrom()

{

**return** from;

}

**public** **void** setFrom(String from)

{

**this**.from = from;

}

**public** String getTo()

{

**return** to;

}

**public** **void** setTo(String to)

{

**this**.to = to;

}

**public** BigDecimal getConversionMultiple()

{

**return** ConversionMultiple;

}

**public** **void** setConversionMultiple(BigDecimal conversionMultiple)

{

ConversionMultiple = conversionMultiple;

}

**public** BigDecimal getQuantity()

{

**return** quantity;

}

**public** **void** setQuantity(BigDecimal quantity)

{

**this**.quantity = quantity;

}

**public** BigDecimal getTotalCalculatedAmount()

{

**return** totalCalculatedAmount;

}

**public** **void** setTotalCalculatedAmount(BigDecimal totalCalculatedAmount)

{

**this**.totalCalculatedAmount = totalCalculatedAmount;

}

**public** **int** getPort()

{

**return** port;

}

**public** **void** setPort(**int** port)

{

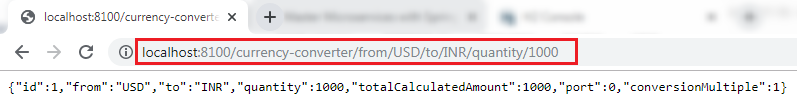
**this**.port = port;

}

}

**Step 7:** Restart the application and type the following URI in the browser:

**http://localhost:8100/currency-converter/from/USD/to/INR/quantity/1000**



In the above response, "**from,"** "**to,"** and "**quantity"** variables picked up from the path. We have hardcoded the other variables.

In the next step, from the currency-conversion-service, we will call the currency-exchange-service. We will also determine what the **conversion multiple** is, and will use that amount (**conversion multiple**) to calculate the **total** amount. We will also use the port that comes in the response.

# Invoking currency-exchange-service from currency-conversion-service

We have the currency-exchange-service ready, and we have set up a currency-calculation-service (currency-conversion-service). Now we will invoke the currency exchange service from the currency calculation service.

We use **RestTemplate()**constructor to invoke an external service. Let's create a RestTemplate and try to invoke currency-exchange-service.

**Step 1:**Select the **currency-conversion-service** project.

**Step 2:** Open the **CurrencyConversionController.java** and create a new **RestTemplate** that invokes the currency-exchange-service application.

**Step 3:**Invoke the **getForEntity()**method of RestTemplate class.

**getForEntity():** It is a method of **RestTemplate** class that retrieves an entity by using the **HTTPGET** method for the specified URL. It converts and stores the response in the ResponseEntity. It returns the **ResponseEntity**.

**Parameters:**It accepts two parameters:

* **URL:** The URL.
* **responseType:** The type of the return value.

1. ResponseEntity<CurrencyConversionBean>responseEntity=**new** RestTemplate().getForEntity("http://localhost:8000/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.**class**, uriVariables);

**Step 4:**In the URL parameter, put the URL of **currency-converter-service** that is [http://localhost:8000/currency-exchange/from/{from}/to/{to}](http://localhost:8000/currency-exchange/from/%7bfrom%7d/to/%7bto%7d). It takes values from the variable **{from}** and **{to}** from the request. Whatever comes in the request we sent it to the currency-exchange-service.

**Step 5:**In the above URL, we need to pass two values **"from"** and **"to."**For passing the values, create a **Map**for URI variables. Pass the **uriVariables** in the URI as a parameter.

Map<String, String>uriVariables=**new** HashMap<>();

uriVariables.put("from", from);

uriVariables.put("to", to);

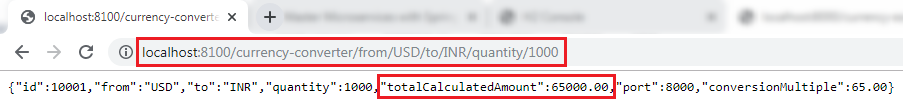
**Step 6:**The response type that we are expecting back is **CurrencyConversionBean,** so store the response in the CurrencyConversionBean.

1. CurrencyConversionBean response=responseEntity.getBody();

**CurrencyConversionController.java**

1. **package** com.javatpoint.microservices.currencyconversionservice;
2. **import** java.math.BigDecimal;
3. **import** java.util.HashMap;
4. **import** java.util.Map;
5. **import** org.springframework.http.ResponseEntity;
6. **import** org.springframework.web.bind.annotation.GetMapping;
7. **import** org.springframework.web.bind.annotation.PathVariable;
8. **import** org.springframework.web.bind.annotation.RestController;
9. **import** org.springframework.web.client.RestTemplate;
10. @RestController
11. **public** **class** CurrencyConversionController
12. {
13. @GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}") //where {from} and {to} represents the column
14. //returns a bean back
15. **public** CurrencyConversionBeanconvertCurrency(@PathVariable String from, @PathVariable String to, @PathVariableBigDecimal quantity)
16. {
17. //setting variables to currency exchange service
18. Map<String, String>uriVariables=**new** HashMap<>();
19. uriVariables.put("from", from);
20. uriVariables.put("to", to);
21. //calling the currency-exchange-service
22. ResponseEntity<CurrencyConversionBean>responseEntity=**new** RestTemplate().getForEntity("http://localhost:8000/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.**class**, uriVariables);
23. CurrencyConversionBean response=responseEntity.getBody();
24. //creating a new response bean and getting the response back and taking it into Bean
25. **return** **new** CurrencyConversionBean(response.getId(), from,to,response.getConversionMultiple(), quantity,quantity.multiply(response.getConversionMultiple()),response.getPort());
26. }
27. }

**Step 7:** Run the both services independently. When we run the currency conversion, it returns the response shown below:



The conversion Multiple is multiplied by the quantity and returns the **totalCalculatedAmount** 65000.00. It means $1000 is equal to 65000.00 INR. It also shows the port **8000** that denotes the other service (currency-exchange-service) is running on port 8000.

# Using Feign REST Client for Service Invocation

In this section, we will start with one of the popular Spring Cloud Component that is **Feign**.

## **Feign**

The Feign is a declarative web service (HTTP client) developed by **Netflix**. Its aim is to simplify the HTTP API clients. It is a Java to HTTP client binder. If you want to use Feign, create an interface, and annotate it. It provides pluggable annotation support, including Feign annotations and JAX-RS annotations.

It is a library for creating REST API clients. It makes web service clients easier. The developers can use declarative annotations to call the REST services instead of writing representative boilerplate code.

## **Spring Cloud OpenFeign**

**Spring Cloud OpenFeign** provides OpenFeign integrations for Spring Boot apps through auto-configuration and binding to the Spring Environment. Without Feign, in Spring Boot application, we use **RestTemplate** to call the User service. To use the Feign, we need to add **spring-cloud-starter-openfeign** dependency in the pom.xml file.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Let’s simplify the previously developed code using Spring Cloud OpenFeign.

In the previous section, one of the things that we had already encountered is when we were developing currency-conversion-service; how difficult it was to call the REST service. There is a lot of manuals that we have to do to call a very simple service. But still we have written a lot of code for it.

When we work with microservices, there will be a lot of calls to other services. We need not to code like the previous one. Feign is a component that solves this problem. Feign makes it easy to invoke other microservices.

The other additional thing that Feign provides is:  it integrates with the **Ribbon**(client-side load balancing framework).

Let's implement the Feign in our project and invoke other microservices using Feign.

**Step 1:** Select **currency-conversion-service**project.

**Step 2:** Open the **pom.xml** and add the **Feign**dependency. Feign inherits from the **Netflix**.

1. **<dependency>**
2. **<groupId>**org.springframework.cloud**</groupId>**
3. **<artifactId>**spring-cloud-starter-feign**</artifactId>**
4. **<version>**1.4.4.RELEASE**</version>**
5. **</dependency>**

**Step 3:** Once the dependency is added, **enable** the Feign to scan the clients by adding the annotation **@EnableFeignClients**in the**CurrencyConversionServiceApplication.java**file.

**Step 4:**Define an attribute in the **@EnableFeignClients**annotation. The attribute is the name of the package that we want to scan.

**CurrencyConversionServiceApplication.java**

**package** com.javatpoint.microservices.currencyconversionservice;

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

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

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

@SpringBootApplication

@EnableFeignClients("com.javatpoint.microservices.currencyconversionservice")

**public** **class** CurrencyConversionServiceApplication

{

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

{

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

}

}

We have enabled the Feign in our project. Now, we will use the Feign to invoke the service.

**Step 5:**Create a **Feign proxy** that enables us to talk to external microservices. Let’s create an interface with the name **CurrencyExchangeServiceProxy.**

**Step 6:**Add an annotation **@FeignClient.**Pass the attributes **name** and **URL**.

In the **name** attribute, write the name of the service that we are going to consume. In our case, we are going to consume **currency-exchange-service**. In the **URL** attribute, write the port on which the currency-exchange-service is running.

1. @FeignClient(name="currency-exchange-service", url="localhost:8000")

**Step 7:** Now, we need to define a method that talks to the **currency-exchange-controller**. Open the **Currency-ConverterController.java**file. Copy the **currency-converter** mapping and paste it in the same file.

**Step 8:** Change the mapping name to **/currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}** and the method name to **convertCurrencyFeign.**

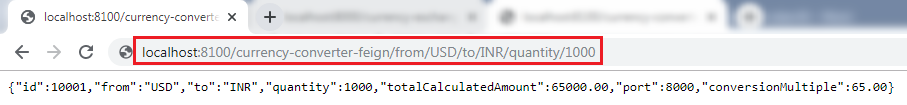
**Step 9:**Make the use of **CurrencyExchangeServiceProxy**and autowired it.

1. @Autowired
2. **private** CurrencyExchangeServiceProxy proxy;

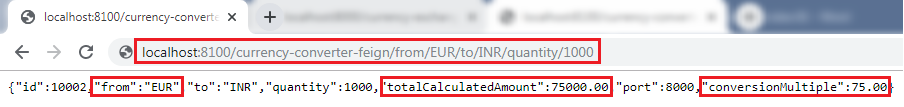
**Step 10:** First, run the **currency-exchange-service** by invoking the URL <http://localhost:8000/currency-exchange/from/USD/to/INR> after that run the **currency-conversion-service** by using the URL <http://localhost:8100/currency-converter/from/USD/to/INR/quantity/1000>.

If we do not run the services in the order, the currency-conversion-service shows **Whitelabel Error Page**. It is because the currency-conversion-service uses the conversionMultiple of currency-exchange-service.

**Step 11:**Execute the feign service by using the URL <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000>. It returns the same response as currency-converter-service.



Change the currency **USD** to **EUR** in the above URL and again invoke the same URL. It takes the variable **"from"** from the currency-exchange-service and returns the **totalCalculated Amount.**



The request we are sending uses **Feign**. Feign is a REST Service client. Feign can call the RESTful web services easily. When we use the RestTemplate to call the RESTful service, it creates **duplication** of code that talks to RESTful services.

When we define Feign, we need only to define a proxy and define a single method into it. Feign helps us to simplify client code to talk to the RESTful web services.

Consider a scenario in which currency-exchange-service offers fifteen different services. All the details related to these services must be defined in one place that is **CurrencyExchangeServiceProxy**interface.

# Client-Side Load Balancing with Ribbon

## **Netflix Ribbon**

Netflix Ribbon is a Part of **Netflix Open Source Software** (Netflix OSS). It is a cloud library that provides the **client-side load balancing**. It automatically interacts with **Netflix Service Discovery** (Eureka) because it is a member of the Netflix family.

The Ribbon mainly provides client-side load balancing algorithms. It is a client-side load balancer that provides control over the behavior of **HTTP** and **TCP** client. The important point is that when we use **Feign**, the **Ribbon** also applies.

## **Features of Ribbon**

* Load balancing
* Fault tolerance
* Multiple protocol support in Asynchronous model
* Caching and batching

## **Modules**

* **ribbon:** It is an API that integrates **load balancing, fault-tolerance, caching,** and
* **ribbon-loadbalancer:** It is a Load balancer API that can be used independently or with other modules.
* **ribbon eureka:** It uses **Eureka** client that provides a dynamic server list for the Spring Cloud.
* **ribbon-transport:** It is a transport client that supports **HTTP, TCP,** and **UDP** These protocols use **RxNetty** with load balancing capability.
* **ribbon-httpclient:** It is a REST client built on top of Apache HttpClient integrated with load balancers.
* **ribbon-core:** It is a Client Configuration API.

## **Types of Load Balancing:**

There are two types of load balancing

* **Server Side Load Balancing:** Server side load balancing is a **monolithic** It applies between the client and the server. It accepts incoming network, application traffic, and distributes the traffic across the multiple backend servers by using various methods. The middle component is responsible for distributing the client requests to the server.
* **Client-Side Load Balancing:**The client holds the list of server’s IPs so that it can deliver the requests. The client selects an IP from the list, randomly, and forwards the request to the server.

Let's configure the Ribbon server in our project.

**Step 1:** Go to the project **currency-conversion-service**.

**Step 2:** Open **pom.xml** file and add the **ribbon** dependency**.**

1. **<dependency>**
2. **<groupId>**org.springframework.cloud**</groupId>**
3. **<artifactId>**spring-cloud-starter-netflix-ribbon**</artifactId>**
4. **</dependency>**

After adding the dependency, we need to enable ribbon on the proxy.

**Step 3:** Open the **CurrencyExchangeServiceProxy.java**file. Enable **Ribbon** by adding an annotation **@RibbonClient**and specify the name of the service which we want to talk to. Ribbon client provide the declarative configuration for a client.

1. @RibbonClient(name="currency-exchange-service")

**CurrencyExchangeServiceProxy.java**

**package** com.javatpoint.microservices.currencyconversionservice;

**import** org.springframework.cloud.netflix.ribbon.RibbonClient;

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

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

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

//@FeignClient(name="currency-exchange-service", url="localhost:8000")

//Enabling feign

@FeignClient(name="currency-exchange-service")

//enabling ribbon

@RibbonClient(name="currency-exchange-service")

**public** **interface** CurrencyExchangeServiceProxy

{

@GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable

**public** CurrencyConversionBean retrieveExchangeValue(@PathVariable("from") String from, @PathVariable("to") String to); //from map to USD and to map to INR

}

**Step 4:** In the annotation **@FeignClient,** remove the attribute **URL**. Because we do not need to talk with one particular service. We will configure that URL in the **application.properties** file.

**Step 5:**Open the **application.properties** file of the project **currency-conversion-service**and configure the servers. The property that we have to configure is:

1. name-of-the-application.ribbon.listOfServers=URLs

We have configured the two instances of currency-exchange-service that we want to invoke.

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

**application.properties**

spring.application.name=currency-conversion-service

server.port=8100

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

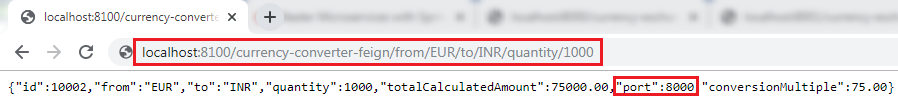
## **Running Client Side Load Balancing with Ribbon**

We have two instances of **CurrentlyExchangeServiceApplication.java,**as shown in the following image:

lient-Side Load Balancing with Ribbon

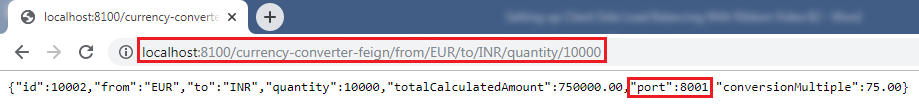
First, run the CurrencyExchangeServiceApplication on port **8000** and then run the CurrencyExchangeServiceApplication on port **8001**.

After running the CurrencyExchangeServiceApplication on both the ports, run the **CurrencyConversionServiceApplication.java**by sending the request <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/10000>. It returns the following response.



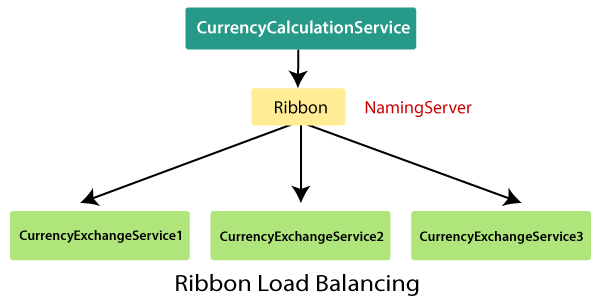
In the above image, the port 8000 represents that the currency-exchange-service is running on port 8000 and handling the current request.

Now, refresh the page. We get the same response except for the port number and quantity because we have changed the quantity in the request.



In the above image, the port 8001 represents that the currency-exchange-service is running on port 8001 and handling the current request.

Let's understand the load balancing through a figure:



In the above figure, Ribbon is distributing the load between three active CurrencyExchangeServices. The **CurrencyExchangeService1** is running on port **8000,** and **CurrencyExchangeService2**is running on port **8001,**and so on. So whatever calls are made using Ribbon through the CurrencyCalculationService, are distributed among these three services.

# Eureka Naming Server

In the previous section, we have configured the **Ribbon** and distributed the load between the two services. In this section, we will set up the **Eureka** naming Server.

## **Naming server**

The **naming server** is a computer application that implements a network service for responding to queries against a directory service.

## **Eureka naming server**

**Eureka naming server** is a REST-based server that is used in the **AWS Cloud** services for load balancing and failover of middle-tier services.

Eureka naming server is an application that holds information about all client service applications. Each microservice registers itself with the Eureka naming server. The naming server registers the client services with their **port numbers** and **IP addresses**. It is also known as **Discovery Server.**  Eureka naming server comes with the bundle of Spring Cloud. It runs on the default port **8761**. It also comes with a Java-based client component, the eureka client, which makes interactions with the service much easier.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

## **The need of naming server**

We have another load balancer application, which is known as **AWS Cloud**. Because of its inherent nature, server ups and down. There is no middle-tier load balancer. Eureka naming server fills the gap between the **client** and the **middle tier load balancer.**

Suppose that we want to start another instance of currency-exchange-service that is **CurrencyExchangeService3** and launch it on port 8002. Here a question arises, **will ribbon be able to distribute the load to it?**

If the Ribbon wants to distribute the load to the new server, we need to add it to the configuration. Based on the load, we can increase or decrease the number of instances of the services.

In this section, we will be able to increase or decrease the number of instances dynamically.

If we keep on changing in the CurrencyCalulationService, based on how many CurrenyExchangeService are active right, then, it becomes very difficult to maintain.

The Eureka naming server comes into existence when we want to make maintenance easier. All the instances of all microservices will be register with the **Eureka** naming server. Whenever a new instance of a microservice comes up, it would register itself with the Eureka naming server. The registration of microservice with the naming server is called **Service Registration.**

Whenever a service wants to talk with another service, suppose CurrencyCalculationService wants to talk to the CurrencyExchangeService. The CurrencyCalculationService first talk with the Eureka naming server. The naming server provides the instances of CurrencyExchangeService that are currently running. The process of providing instances to other services is called **Service Discovery.**

**Service registration** and **service discovery**are the two important features of the naming server. In the next step, we will set up a Eureka naming server.

## **Setting up Eureka naming server**

There are a lot of steps that are involved in setting up the Eureka naming server are as follows:

* Create a component for the Eureka naming server
* Update the CurrencyCalculationService to connect to the Eureka naming server
* Connect CurrencyExchangeService to the Eureka naming server
* Configure the Ribbon

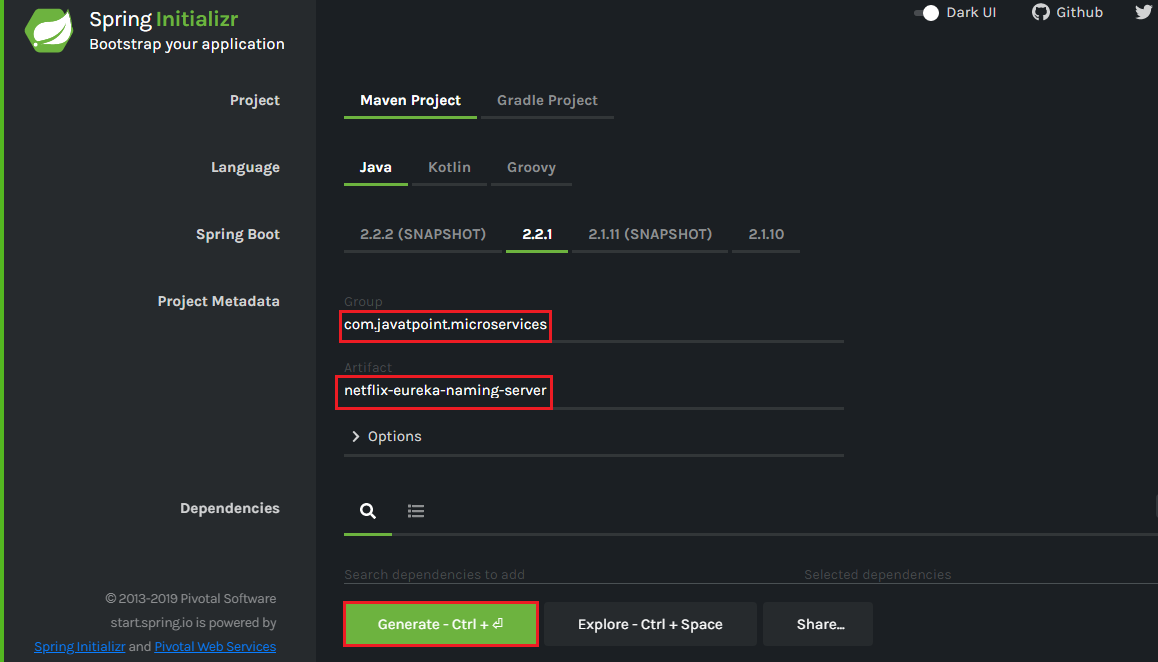
Once the instances of currency-exchange-service are registered with the Eureka naming server, then we will use the Ribbon to find the detail from the naming server. Let’s follow the steps specified above:

**Create a component for the Eureka naming server**

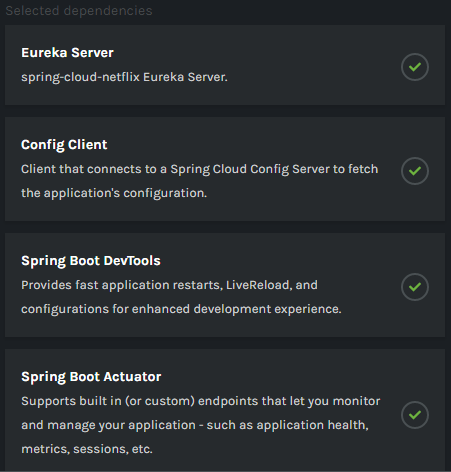
**Step 1:**Open Spring Initializr [https://start.spring.io](https://start.spring.io/).

**Step 2:**Provide the **Group** name. We have provided **com.javatpoint.microservices.**

**Step 3:**Provide the **Artifact Id**. We have provided **netflix-eureka-naming-server.**



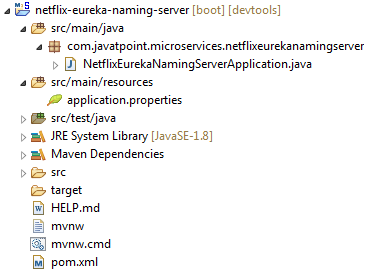
**Step 4:**Add the following dependencies: **Eureka Server, Config Client, Actuator,**and**DevTools.**



**Step 5:**Click on the **Generate** button. It downloads a **zip** file.

**Step 6:** **Extract** the zip file and paste the folder in the Spring Tool Suite (STS) workspace and then **import** it.

File -> Import -> Existing Maven Projects -> Next -> Browse -> Select the **netflix-eureka-naming-server project** -> Finish



**Step 7:**Open the **NetflixEurekaNamingServerApplication.java** file and enable Eureka naming server by using an annotation **@EnableEurekaServer**.

**NetflixEurekaNamingServerApplication.java**

1. **package** com.javatpoint.microservices.netflixeurekanamingserver;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
5. @SpringBootApplication
6. @EnableEurekaServer
7. **public** **class** NetflixEurekaNamingServerApplication
8. {
9. **public** **static** **void** main(String[] args) {
10. SpringApplication.run(NetflixEurekaNamingServerApplication.**class**, args);
11. }
12. }

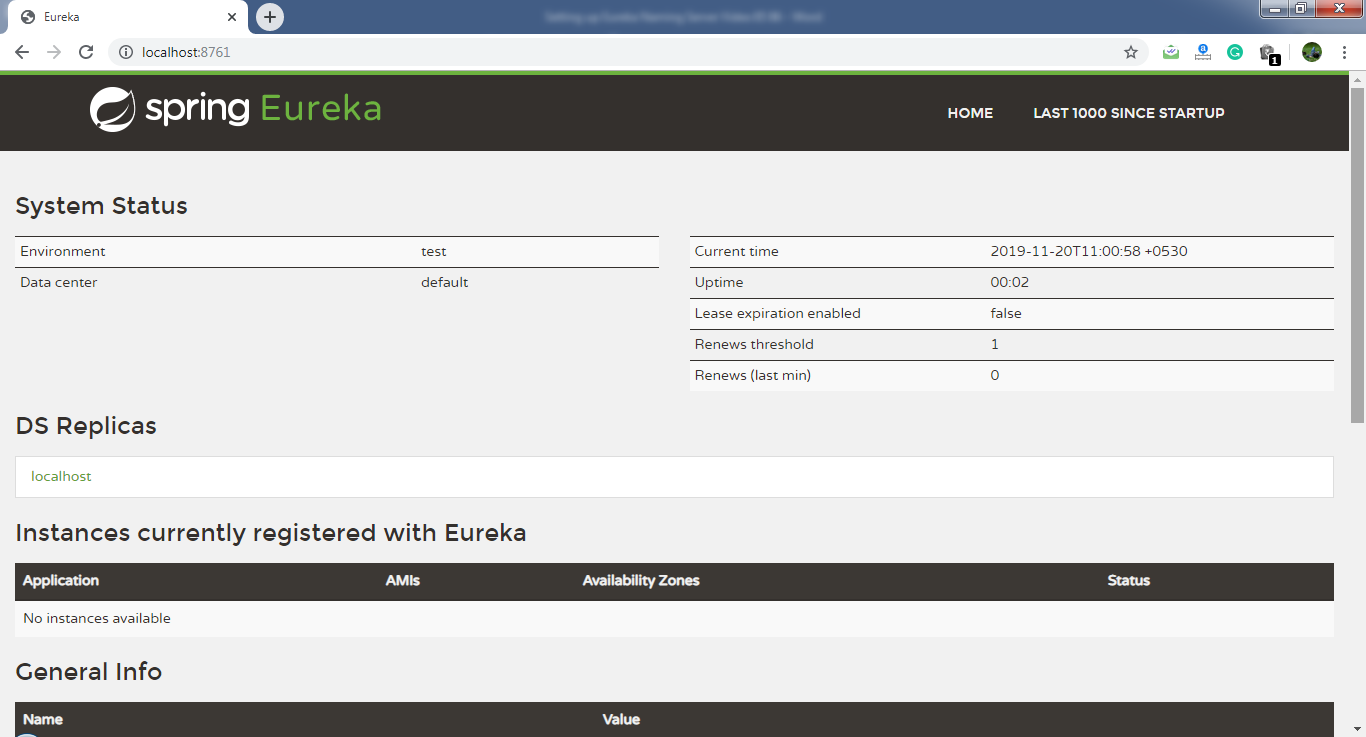
**Step 8:**Open the **application.properties** file and configure the **application name, port,**and**eureka server**.

1. spring.application.name=netflix-eureka-naming-server
2. server.port=8761
3. eureka.client.register-with-eureka=**false**
4. eureka.client.fetch-registry=**false**

Port **8761** is the default port for the Eureka naming server.

**Step 9:**Run the **NetflixEurekaNamingServerApplication.java**file as Java Application.

**Step 10:**Open the browser and type the URL [http://localhost:8761](http://localhost:8761/). It shows the Eureka server UI.



In this section, we have created a component Eureka naming server. In the next step, we will connect the microservices with the Eureka naming server.

# Connecting Microservices to Eureka naming server

In this section, we will connect the **currency-conversion-service** and **currency-exchange-service** to the **Eureka** naming server.

First, we will connect the currency-conversion-service.

**Step 1:**Select the **currency-conversion-service**project.

**Step 2:**Open the **pom.xml** file and add the **eureka-client** dependency**.**

**<dependency>**

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

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

**</dependency>**

**Step 3:**Open **CurrencyConversionServiceApplication.java** file and enable **discovery client** by using the annotation **@EnableDiscoveryClient**.

**CurrencyConversionServiceApplication.java**

**package** com.javatpoint.microservices.currencyconversionservice;

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

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

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

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

@SpringBootApplication

@EnableFeignClients("com.javatpoint.microservices.currencyconversionservice")

@EnableDiscoveryClient

**public** **class** CurrencyConversionServiceApplication

{

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

{

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

}

}

After enabling the discovery client, configure the **URL** for the Eureka naming server.

**Step 4:**Open the **application.properties** file and configure the URL for the Eureka naming server.

**application.properties**

spring.application.name=currency-conversion-service

server.port=8100

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

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

If we look at the Eureka UI, we see that there is no instance registered with the Eureka server.

**Step 5:**Run the **CurrencyConversionServiceApplication.java**file.

**Step 6:** Open the browser and **refresh** the Eureka server page. It shows the instances of registered microservices.



We see that an instance of currency-conversion-service is registered with the Eureka naming server, and running on port **8100**.

Let's connect **currency-exchange-service**with Eureka naming server.

**Step 1:** Select the **currency-exchange-service**project.

**Step 2:** Open the **pom.xml** file and add the **eureka-client**dependency**.**

**<dependency>**

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

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

**</dependency>**

**Step 3:**Open **CurrencyExchangeServiceApplication.java** file and enable the **discovery client** by using the annotation **@EnableDiscoveryClient**.

**CurrencyExchangeServiceApplication.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

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

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

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

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** CurrencyExchangeServiceApplication

{

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

{

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

}

}

**Step 4:**Open the **application.properties** file and configure the **URL** for Eureka naming server.

**application.properties**

spring.application.name=currency-exchange-service

server.port=8000

spring.jpa.show-sql=**true**

spring.h2.console.enabled=**true**

spring.datasource.platform=h2

spring.datasource.url=jdbc:h2:mem:javatpoint

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

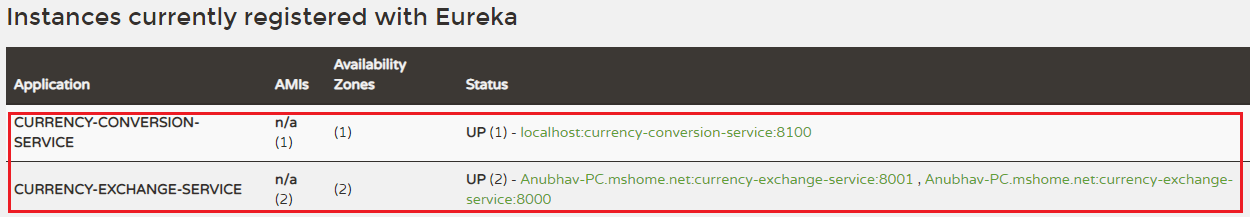
**Step 5:**Run the two instances of **CurrencyExchangeServiceApplication.java.** First instance on port **8000** and the second instance on port **8001**.

**Step 6:** Open the browser and **refresh** the Eureka server. It shows the registered microservice.



We see that the two instances of currency-exchange-service are registered with the Eureka naming server. The two instances are running on port **8001**and**8000**.

In the following image, we can see that both the microservices has registered with the Eureka naming server.



# Distributing calls using Eureka and Ribbon

In the previous section, we have registered currency-exchange-service and currency-conversion-service with the Eureka naming server.

When CurrencyCalculationService (currency-conversion-service) invokes CurrencyExchangeService, the Eureka naming server finds the details of the **currency-exchange-service**.

Instead of hardcoding the URLs for Ribbon, the Ribbon talks to the Eureka naming server and retrieve the details of all the instances of microservices.

**Step 1:** Open the file **application.properties**.

In this file, we have configured the Eureka naming server and disable the list of servers that we have configured earlier. In the currency-conversion-service, we have already configured the URL for Eureka. Now, we have configured the Eureka naming server in both the services.

Now the CurrencyExchangeService, starts talking to the Eureka naming server.

**application.properties**

spring.application.name=currency-conversion-service

server.port=8100

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

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

#### Note**: In the above code, hash (#) denotes the comment.**

Here one thing is to notice that in the **application.properties** file, we don’t have any source of **currency-exchange-service**. We did not hardcode the URLs of currency-exchange-service in the currency-conversion-service.

**Step 2:** Kill all the running applications.

**Step 3:** First, run the **NetflixEurekaNamingServerApplication.java**.

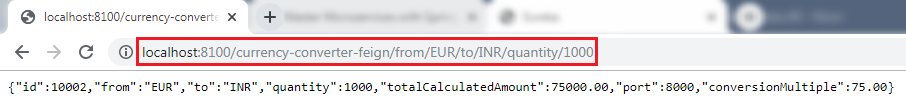
**Step 4:** Open the browser and refresh the Eureka server.

**Step 5:** Run an instance of **CurrencyExchangeServiceApplication.java** on port **8000**.

**Step 6:** Run the **CurrencyConversionServiceApplication.java**.

**Step 7:** Clear the console and let it be warm-up.

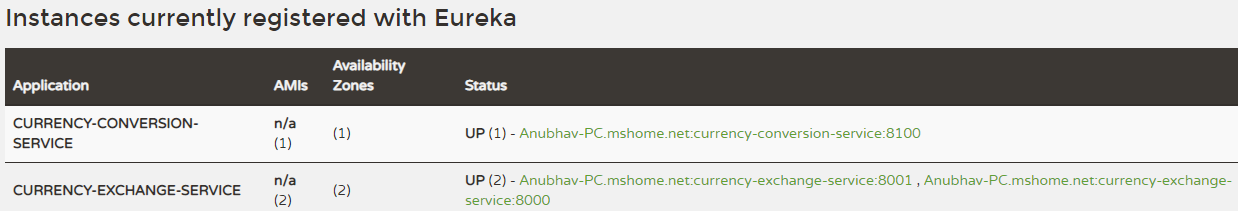
**Step 8:** Open the browser and type the URL <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000>.



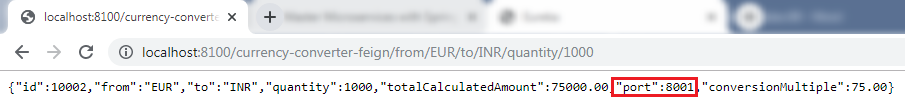
When we refresh the Eureka server, it always returns the port 8000 in the response.

**Step 9:** Run another instance of **CurrencyExchangeServiceApplication.java** on port **8001**.

**Step 10:** Again, refresh the Eureka server. We see that there are **two** instances of **currency-exchange-service**that are running on port **8000** and **8001**.

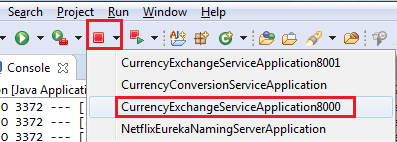


**Step 11:**Refresh the URL <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000>.



When we refresh the Eureka server, it changes the port again and again.

**Step 12:** Kill the **CurrencyExchangeServiceApplication8000**.



**Step 13:** Again, refresh the URL <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000>. It returns the port **8001** in the response.

Hence, service will never go down even if an instance of the same service is down or interrupted by other reasons.

# Introduction to API Gateways

## **What is an API Gateway?**

An API stands for Application Program Interface. It is a set of instructions, protocols, and tools for building software applications. It specifies how software components should interact.

The API Gateway is a server. It is a single entry point into a system. API Gateway encapsulates the internal system architecture. It provides an API that is tailored to each client. It also has other responsibilities such as **authentication, monitoring, load balancing, caching, request shaping and management,**and **static response handling**.

API Gateway is also responsible for **request routing, composition,** and **protocol translation**. All the requests made by the client go through the API Gateway. After that, the API Gateway routes requests to the appropriate microservice.

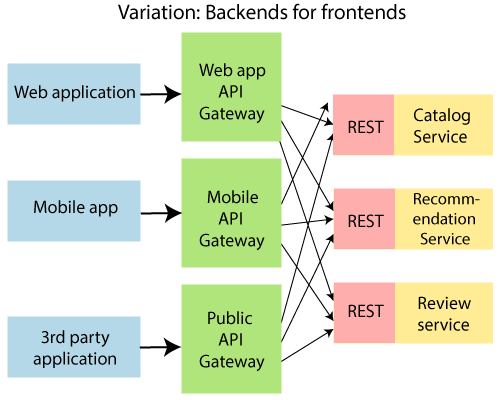
The API Gateway handles the request in one of the two ways:

* It routed or proxied the requests to the appropriate service.
* Fanning out (spread) a request to multiple services.

The API Gateway can provide each client with a custom API. It also translates between two protocols, such as **HTTP,** **WebSockets,** and **Web-Unfriendly** protocols that are used internally.

**Example**

The popular example of API Gateway is **Netflix API Gateway**. The Netflix streaming services are available on hundreds of different kinds of devices such as **televisions**, **set-top boxes, smartphones, tablets,** etc. It attempts to provide a **one-size-fits-all**API for its streaming service.



An API Gateway includes:

* Security
* Caching
* API composition and processing
* Managing access quotas
* API health monitoring
* Versioning
* Routing

## **Advantages of API Gateway**

* The most important advantage of API Gateway is that it encapsulates the internal structure of the application.
* Rather than invoking the specific service, the client directly talks to the API Gateway.
* It reduces the number of round trips between client and application.
* It simplifies the client code.
* It reduces coding efforts, makes the application more efficient, decreases errors all at the same time.
* It provides each kind of client with a specific API.

## **Disadvantages**

* It requires routing rules.
* There is a possibility of a single point of failure.
* Risk of complexity due to all the API rules are in one place.

## **Working of API Gateway**

In microservices, we route all the requests through an API. We can implement common features like **authentication, routing, rate limiting, auditing,**and **logging** in the API Gateway.

Consider a scenario in which we do not want to call a microservice more than five times by a particular client. We can do it as a part of the limit in the API Gateway. We can implement the common features across microservices in the API gateway. The**Zuul API Gateway**is a popular API Gateway implementation.

We must implement the following features in all the microservices:

* **Service Aggregation**
* **Authentication, authorization and Security**
* **Rate Limits**
* **Fault Tolerance**

Suppose there is an external consumer who wants to call **fifteen** different services as part of one process. It is better to aggregate those fifteen services and provide one service call for the external consumer. These are the kinds of features that are common across all the microservices. These features are implemented at the level of API.

Instead of allowing microservices to call each other directly, we would do all the calls through API Gateway. API Gateway will take care of common features like authentication, fault tolerance, etc. It also provides aggregation services around all microservices because all calls get routed through the API Gateway.

# Zuul API Gateway

## **What is Zuul?**

Zuul Server is an API Gateway application. It handles all the requests and performs the dynamic routing of microservice applications. It works as a front door for all the requests. It is also known as **Edge Server.**

Zuul is built to enable **dynamic routing, monitoring, resiliency,**and**security.** It can also route the requests to multiple **Amazon Auto Scaling Groups**.

For Example, **/api/products** are mapped to the **product** service and **/api/user** is mapped to the **user** service. The Zuul Server dynamically routes the requests to the respective backend application.

## **Why we use Zuul?**

The volume and variety of Netflix API traffic sometimes result in production issues that arises quickly and without warning. So we need a system that allows us to rapidly change behavior in order to react to these situations.

Zuul provides a range of different types of **filters** that allows us to quickly and nimbly apply functionality to our edge service. The filters perform the following functions:

* **Authentication and Security:**It provides authentication requirements for each resource.
* **Insights and Monitoring:**It tracks meaningful data and statistics that give us an accurate view of production.
* **Dynamic Routing:**It dynamically routes the requests to different backed clusters as needed.
* **Stress Testing:**It increases the traffic to a cluster in order to test performance.
* **Load Shedding:**It allocates capacity for each type of request and drops a request that goes over the limit.
* **Static Response Handling:**It builds some responses directly at the edge instead of forwarding them to an internal cluster.
* **Multi-region Resiliency:** It routes requests across AWS regions in order to diversify our ELB usage.

## **Zuul Components**

**Zuul 2.x components:**

* **zuul-core:** It is a library that contains the core functionality of Zuul 2.0.
* **zuul-sample:**It is a sample driver application for Zuul 2.0

**Zuul 1.x components:**

* **zuul-core:** It defines the core functionality.
* **zuul-simple-webapp:**A web app that shows a simple example of how to build an application with zuul-core.
* **zuul-netflix:**It is a library that adds other NetflixOSS components to Zuul.
* **zuul-netflix-webapp:**It is a webapp that packages zuul-core and zuul-netflix together.

## **Setting up Zuul API Gateway Server**

There are **three** steps to set up the Zuul API Gateway:

* Create a component for the Zuul API Gateway
* Decide the things that the Zuul API Gateway should do
* All the important requests are configured to pass through the Zuul API Gateway

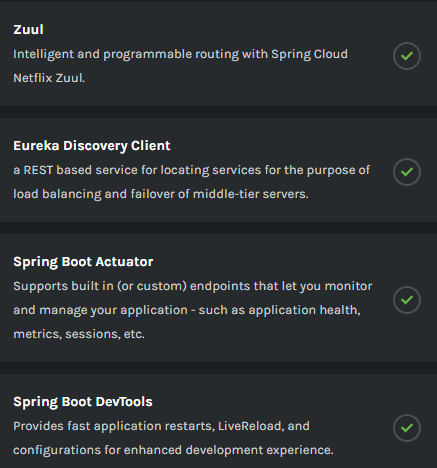
Follow the steps to set up the Zuul API Gateway server.

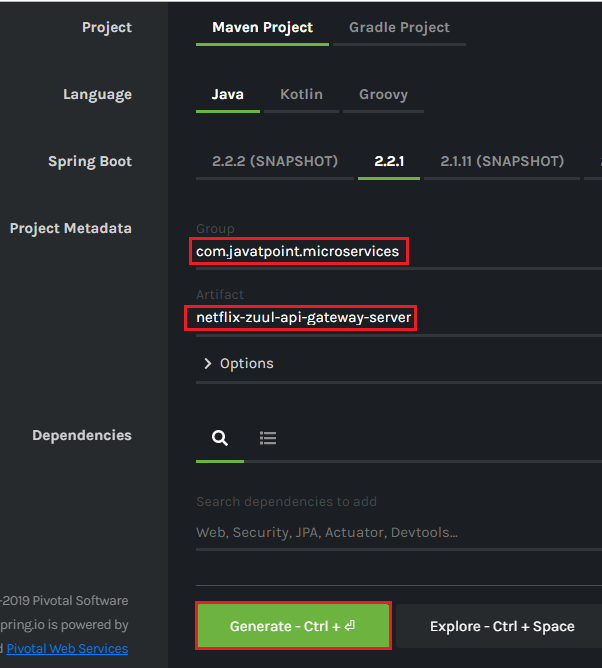
**Step 1:** Open **Spring Initializr** [https://start.spring.io](https://start.spring.io/).

**Step 2:**Provide the **Group** name. We have provided**com.javatpoint.microservices.**

**Step 3:**Provide the **Artifact.**We have provided**netflix-zuul-api-gateway-server.**

**Step 4:**Add the following dependencies: **Zuul, Eureka Discovery, Actuator,**and**DevTools.**



**Step 5:**Click on the **Generate** button. It starts packing the project into **zip** file and download it. 

**Step 6: Extract** the zip file and paste it in the Spring Tool Suite’s workspace.

**Step 7: Import** the project in the STS IDE.

File -> Import -> Existing Maven Projects -> Browse -> Select **netflix-zuul-api-gateway-server**-> Select Folder -> Finish

It takes some time to import.

**Step 8:**Open the **NetflixZuulApiGatewayServerApplication.java** file and enable zuul proxy and discovery client by using the annotations **@EnableZuulProxy**and **@EnableDiscoveryClient,**respectively**.**

**NetflixZuulApiGatewayServerApplication.java**

**package** com.javatpoint.microservices.netflixzuulapigatewayserver;

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

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

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

**import** org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@EnableZuulProxy

@EnableDiscoveryClient

@SpringBootApplication

**public** **class** NetflixZuulApiGatewayServerApplication

{

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

{

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

}

}

**Step 9:** Open **application.properties** file and configure the **application name, port,**and **eureka naming server**.

**application.properties**

spring.application.name=netflix-zuul-api-gateway-server

server.port=8765

eureka.client.service-url.**default**-zone=http://localhost:8765/eureka

# Implementing Zuul Logging Filter

In the previous section, we have discussed common functionality that is implemented in API Gateway. We have the Zuul server ready that acts as API Gateway.

In this section, we will implement the logging functionality in the Zuul API Gateway.

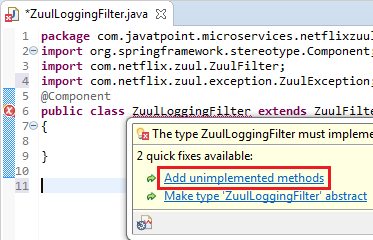
Let's implement the logging in the Zuul API Gateway.

**Step 1:**In the **netflix-zuul-api-gateway-server**project, create a new class file with the name **ZuulLoggingFilter.**

**Step 2:**Add an annotation **@Componenet**.

**Step 3:**Extend the **ZuulFilter** class. There are some abstract methods in the ZuulFilter class. These abstract methods must be implemented in the ZuulLoggingFilter class.

**Step 4:**Move your cursor over the **ZuulLoggingFilter**class**.**It suggests two quick fixes, as shown in the following image. Click on the **Add unimplemented methods.**We get the default implementation of all the abstract methods of the ZuulFilter class.



**ZuulFilter** class has four abstract methods that are listed below:

* **shouldFilter():**The shouldFilter() method checks the request and decides whether filter to be executed or not.
* **run():**The run() method invokes, if both **!isFilterDisabled()** and **shouldFilter()**methods returns **true**.
* **filterType():**The filterType() method classify a filter by type. There are four types of standard filters in Zuul: **pre** for **pre-routing filtering**, **route** for **routing to an origin**, **post** for **post-routing filters**, and **error** for **error handling**. Zuul also supports a **static** type for **static responses.** Any filter type can be created or added and run by calling the method **runFilters(type).**
* **filterOrder()**: The filter order must be defined for a filter. Filters may have the same filter order if the precedence is not important for the filters. The filter order does not need to be sequential.

**Step 5:**Create the **Logger** class object and invoke **getLogger()**method to create a logger.

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

**Remember:**Import **Logger** class of **org.slf4j** package.

**Step 6:**Implement the logic in the **run()** method.

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

{

//getting the current HTTP request that is to be handle

HttpServletRequest request=RequestContext.getCurrentContext().getRequest();

//printing the detail of the request

logger.info("request -> {} request uri-> {}", request, request.getRequestURI());

**return** **null**;

}

**ZuulLoggingFilter.java**

**package** com.javatpoint.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

**public** **class** ZuulLoggingFilter **extends** ZuulFilter

{

1. //creating Logger object

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

@Override

**public** booleanshouldFilter()

{

**return** **true**; //executing filter for every request

}

//log the content of the request

@Override

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

{

//getting the current HTTP request that is to be handle

HttpServletRequest request=RequestContext.getCurrentContext().getRequest();

//prints the detail of the requestin the log

logger.info("request -> {} request uri-> {}", request, request.getRequestURI());

**return** **null**;

}

@Override

**public** String filterType()

{

**return** "pre"; //intercept all the request before execution

}

@Override

**public** intfilterOrder()

{

**return** 1; //setting filter order to 1

}

}

In the next step, we will see how to intercept the request using Zuul.

# Executing a Request through Zuul API Gateway

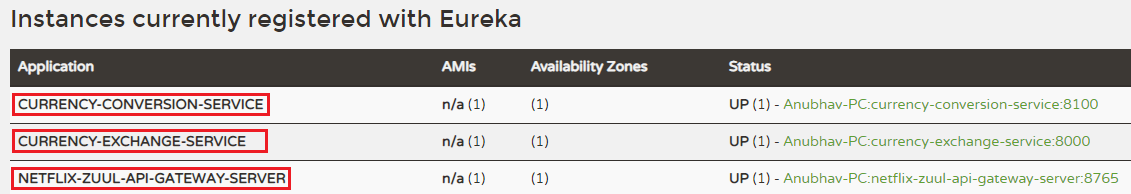
**Step 1:** Run the **netflix-eureka-naming-server.**

**Step 2:** Run the **currency-exchange-service**on port**8000**.

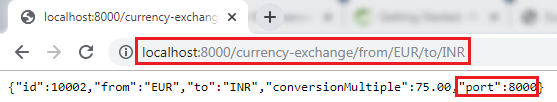
**Step 3:** Run the **currency-conversion-service** on port **8100**.

**Step 4:** Run the **netflix-zuul-api-gateway-server.**

**Step 5:**Open the browser and invoke the URL [http://localhost:8761](http://localhost:8761/). It shows all the services that are registered with the Eureka naming server.



**Step 6**: Invoke the URL <http://localhost:8000/currency-exchange/from/EUR/to/INR>. We get the response, but the request does not go through the Zuul API Gateway.

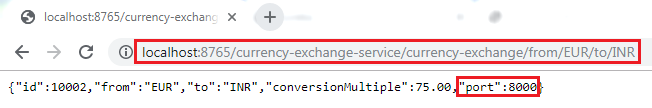


Let's invoke the request through the Zuul API Gateway. We use the following URL: [http://localhost:8765/{application-name}/{uri}](http://localhost:8765/%7bapplication-name%7d/%7buri%7d). The port **8765** is the default port for the Zuul API Gateway server.

In our case, the **application name** is **currency-exchange-service,** and the **URI** is **/currency-exchange/from/EUR/to/INR**. So the complete URL will look like the following:

<http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR>

**Step 7:**Copy the above URL and paste it in the browser. We get the same response as above, but at this time, the request is going through the **Zuul API Gateway**.



We can also see the content of the request that is printed on the Zuul API Gateway server. The request prints the request URI.

xecuting a Request through Zuul API Gateway

We have sent the request through the Zuul API Gateway, instead of directly calling the microservices.

## **Setting up Zuul API Gateway between microservices invocations**

In the previous step, we have used a direct URL to execute the currency-exchange-service through the Zuul API Gateway proxy. When we use the URL <http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR>, it uses port 8765 that is proxy for API Gateway.

In this section, we will call the currency-calculation-service (currency-conversion-service) that calls the currency-exchange-service. So far, we were calling the service directly. Now, we will call it through the Zuul API Gateway instead of directly calling the currency-exchange-service.

**Step 1:**Select the project **currency-conversion-service**.

**Step 2:** Open the **CurrencyExchangeServiceProxy.java** file.

**Step 3:** Enable the **Feign** by using the annotation **@FeignClient** with the attribute **name="netflix-zuul-api-gateway-server"**.

1. @FeignClient(name="netflix-zuul-api-gateway-server")

**Remember:** Remove or comment all other annotations @FeignClient in **CurrencyExchangeServiceProxy.java** file.

**Step 4:**Define the **mapping** for the Zuul API Gateway server.

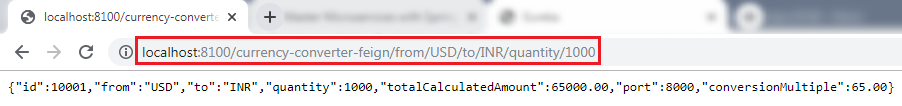
1. @GetMapping("/currency-exchange-service/currency-exchange/from/{from}/to/{to}")

**Remember:** Remove or comment the mapping for currency-exchange-service.

**Step 5:**Run the **netflix-eureka-naming-server, currency-exchange-service, currency-conversion-service,**and**netflix-zuul-api-gateway-server** in the same order in which we have written.

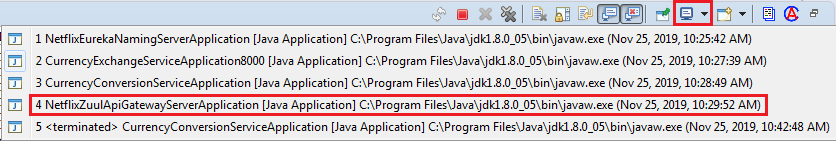
**Remember:** Ensure that all **four** services are running properly.

**Step 6:**Open the browser and invoke the URL <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000>. It returns the following response:



Let's see the log for**NetflixZullApiGatewayServerApplication.**

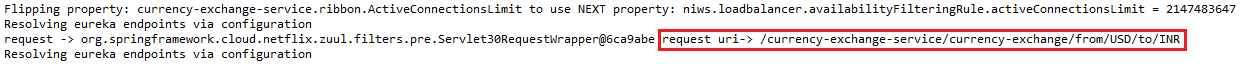
**Step 7:**Click on the arrow that is beside the console icon and select the **NetflixZullApiGatewayServerApplication.**



It shows a couple of logs, as shown in the following image.

xecuting a Request through Zuul API Gateway

**Step 8:** Refresh the URL again. It shows a single log on the console.



Whenever we call the CurrencyClaculationService (currency-converter-service) through Feign, it routed through the API Gateway server. The Gateway executes a filter called **ZuulLoggingFilter** that invokes the currency-exchange-service.

Now let's intercept the calls between **currency converter-service** and **currency-exchange-service**. It means the API Gateway executes **two** times when we invoke the URL.

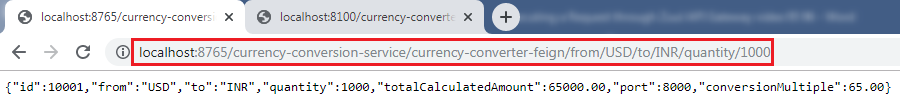
* The **first time**, API Gateway executes when we call the currency-conversion-service. It means before the execution of the currency-conversion-service. The currency-conversion-service routed through the API Gateway.
* The **second time,**API Gateway executes when the currency-conversion-service calls the currency-exchange-service. It means **after** the execution of **currency-conversion-service** and **before** the execution of **currency-exchange-service**. The currency-exchange-service also routed through the API Gateway.

Let's implement the interception in our project.

Send the request [http://localhost:8765](http://localhost:8765/) through the API Gateway. The URI will be **/{application-name}/{uri}.** The complete URL will look like the following:

<http://localhost:8765/currency-conversion-service/currency-converter-feign/from/USD/to/INR/quantity/100>0

Invoke the above URL. It returns the same response as the URL <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000> returns.



We can see in the log that the logging filter executes **two** times. The **first time** it calls the **currency-converter-service** and the **second time** when the **currency-converter-service** calls the **currency-exchange-service**.

xecuting a Request through Zuul API Gateway

In this section, we have executed both the services through the Zuul API Gateway server.

# Introduction to Distributed Tracing

## **Distributed Tracing**

Distributed tracing is a technique to **monitor** and **profile** the applications, especially those built using microservice architecture. It is also known as **distributed request tracing**. Developers use distributed tracing to **debug** and **optimize** the code.

Distribute tracing provides a place where we can see that "what is happening with a specific request?" It is important because there are a variety of components that are involved in the microservices.

If we want to solve a problem or debug a problem, we need a centralized server. So the term **distributed tracing** comes into existence.

In this section, we will use **Spring** **Cloud Sleuth** with **Zipkin**. Spring Cloud Sleuth assigns a **unique Id** to each request that we have made. We can trace all the requests based on unique Ids across all the components.

## **Spring Cloud Sleuth**

Spring Cloud Sleuth is a **Spring Cloud library** that provides the ability to track the progress of subsequent microservices by adding **trace** and **span Ids** on the appropriate HTTP request headers. The Sleuth library is based on the **MDC** (Mapped Diagnostic Context) concept, where we can easily extract values, put to context, and display them in the log.

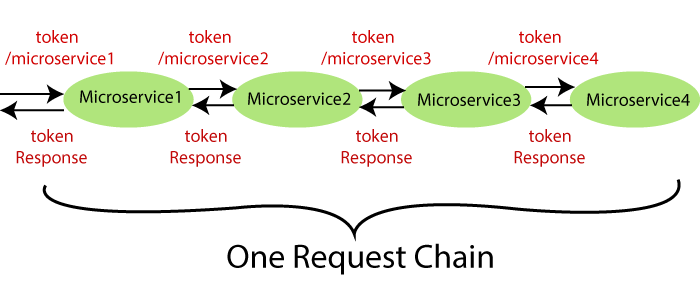
## **Zipkin**

Zipkin is an open-source, Java-based **distributed tracing system**. It has a management console that provides a mechanism for **sending, receiving, storing,** and **visualizing** traces details of the subsequent services.

With the help of the Zipkin server, we can put all the logs of all the components in the **MQ** (RabbitMQ). We send the logs to the Zipkin server where the logs consolidate. After doing this, we can monitor different requests. We can also find what is happening to a specific request?

## **Implementing distributed tracing Using Spring Cloud Sleuth**

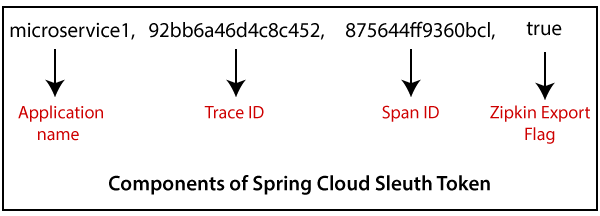
In this step, we will add Spring Cloud Sleuth for all the microservices. It adds a unique Id to all the requests. It is used to generate and attach the **trace Id**, **span Id** to the logs so that tools (Zipkin) can use these ids.



The Spring Cloud Sleuth token has the following components:

* **Application name:**The name of the application that is defined in the **properties** file.
* **Trace Id:**The Sleuth adds the Trace Id. It remains the same in all services for a given request.
* **Span Id:**The Sleuth also adds the Span Id. It remains the same in a unit of work but different for different services for a given request.
* **Zipkin Export Flag:**It indicates a boolean value. It can be either **true** or

The following figure shows the Spring Cloud Sleuth token.



Let's implement the **Spring Cloud Sleuth** in our projects.

**Step 1:** Select the project **netflix-zuul-api-gateway-server**.

**Step 2:**Open**pom.xml**and add the **Sleuth** dependency.

1. <dependency>
2. <groupId>org.springframework.cloud</groupId>
3. <artifactId>spring-cloud-starter-sleuth</artifactId>
4. </dependency>

Now we need to trace all the requests. If we want to trace all the requests, we would need to create **ALWAYS\_SAMPLE**. We can create a Sampler by using a **Bean**.

**Sampler**

Distributed tracing may have a very high volume of data, so the sampling is important in distributed tracing. Spring Cloud Sleuth provides a **Sampler** strategy. With the help of Sampler, we can implement the sampling algorithm that provides control of the algorithm. By default, we get a procedure that continuously performs the tracing if a **span** **(correlation: is an individual operation)** is already active.

But the newly created spans are always marked as **non-exportable**. If all the applications are running with the Sampler, we can see trace (end-to-end latency graph that is composed of spans) in the log, not in any remote location. By default, Spring Cloud Sleuth sets all spans to **non-exportable**.

When we export span data to the **Zipkin** or **Spring Cloud Stream**, Spring Cloud Sleuth provides **AlwaysSampler** class that exports everything to the Zipkin. It also provides a **PercentageBasedSampler** class that samples a fixed fraction of span.

**Remember:** If you are using **Spring 2.0.0** or above versions, use the following Sampler. We have used the same because we are using Spring version **2.2.1.**

@Bean

**public** Sampler defaultSampler()

{

**return** Sampler.ALWAYS\_SAMPLE;

}

**Step 3:**Open **NetflixZuulApiGatewayServerApplication.java**file and define a **Bean**.

**NetflixZuulApiGatewayServerApplication.java**

**package** com.javatpoint.microservices.netflixzuulapigatewayserver;

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

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

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

**import** org.springframework.cloud.netflix.zuul.EnableZuulProxy;

**import** org.springframework.context.annotation.Bean;

**import** brave.sampler.Sampler;

@SpringBootApplication

@EnableDiscoveryClient

@EnableZuulProxy

**public** **class** NetflixZuulApiGatewayServerApplication

{

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

{

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

}

//creating a bean

@Bean

//creating a sampler called

**public** Sampler defaultSampler()

{

**return** Sampler.ALWAYS\_SAMPLE;

}

}

In the above code, we have added Spring Cloud Sleuth to the Zuul API Gateway server.

Now we have to define Bean in **currency-exchange-service** and **currency-conversion-service**also.

**Step 4:**Open **pom.xml** of **currency-exchange-service** and add the **Sleuth** dependency.

**<dependency>**

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

**<artifactId>**spring-cloud-starter-sleuth**</artifactId>**

**</dependency>**

**Step 5:**Open **CurrencyExchangeServiceApplication.java** file and define a **Bean**.

**CurrencyExchangeServiceApplication.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

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

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

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

**import** org.springframework.context.annotation.Bean;

**import** brave.sampler.Sampler;

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** CurrencyExchangeServiceApplication

{

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

{

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

}

@Bean

//creating a sampler called always sampler

**public** Sampler defaultSampler()

{

**return** Sampler.ALWAYS\_SAMPLE;

}

}

**Step 6:** Similarly, Open the **pom.xml** of **currency-conversion-service** and add the Sleuth dependency.

**<dependency>**

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

**<artifactId>**spring-cloud-starter-sleuth**</artifactId>**

**</dependency>**

**Step 7:**Open **CurrencyConversionServiceApplication.java** file and define a **Bean**.

**CurrencyConversionServiceApplication.java**

**package** com.javatpoint.microservices.currencyconversionservice;

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

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

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

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

**import** org.springframework.context.annotation.Bean;

**import** brave.sampler.Sampler;

@SpringBootApplication

@EnableFeignClients("com.javatpoint.microservices.currencyconversionservice")

@EnableDiscoveryClient

**public** **class** CurrencyConversionServiceApplication

{

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

{

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

}

@Bean

//creating a sampler called always sampler

**public** Sampler defaultSampler()

{

**return** Sampler.ALWAYS\_SAMPLE;

}

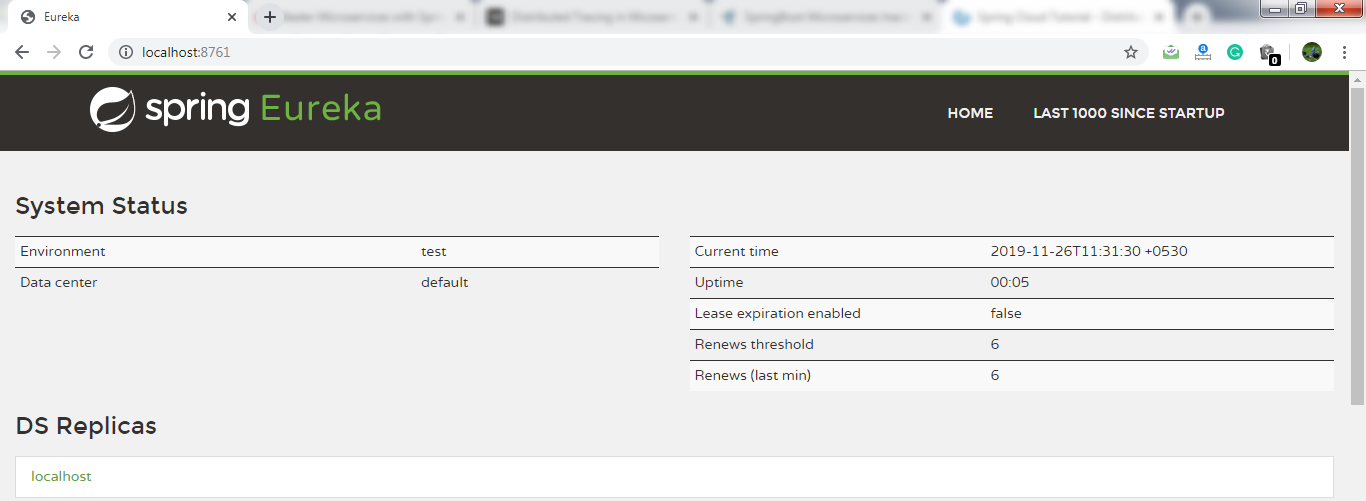
}

Now we have three applications that are connect to Spring Cloud Sleuth.

**Step 8:**Launch the applications in the following order:

**netflix-eureka-naming-server**

* Open the browser and invoke the URL [http://localhost:8761](http://localhost:8761/). It returns the Eureka interface, as shown below.



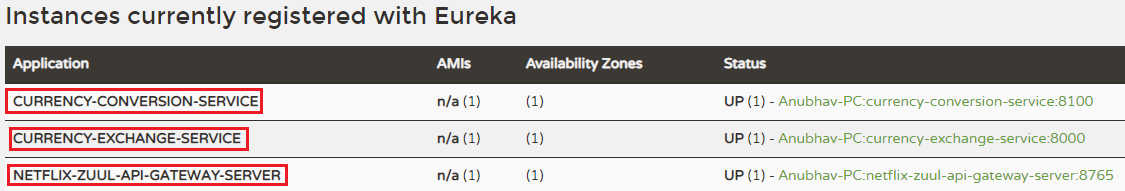
**currency-exchange-service**(on port 8000)

**currency-conversion-service**

**netflix-zuul-api-gateway-server**

**Remember:** After launching each service, refresh the **Eureka server**.

It shows all the instances currently registered with the Eureka server.



**Step 9:**Open **CurrencyExchangeController.java** file and add a **logger** into it.

**CurrencyExchangeController.java**

**package** com.javatpoint.microservices.currencyexchangeservice;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

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

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

**import** org.springframework.core.env.Environment;

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

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

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

@SpringBootApplication

@RestController

**public** **class** CurrencyExchangeController

{

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

@Autowired

**private** Environment environment;

@Autowired

**private** ExchangeValueRepository repository;

@GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable

**public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)   //from map to USD and to map to INR

{

ExchangeValue exchangeValue = repository.findByFromAndTo(from, to);

//setting the port

exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));

logger.info("{}", exchangeValue);

**return** exchangeValue;

}

}

Similarly, we will add logger into CurrencyConversionContoller.

**Step 10:** Open **CurrencyConversionContoller**.**java**file and add a **logger** into it.

**CurrencyConversionContoller**.**java**

**package** com.javatpoint.microservices.currencyconversionservice;

**import** java.math.BigDecimal;

**import** java.util.HashMap;

**import** java.util.Map;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

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

**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

{

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

@Autowired

**private** CurrencyExchangeServiceProxy proxy;

@GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}") //where {from} and {to} represents the column

//returns a bean back

**public** CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity)

{

//setting variables to currency exchange service

Map<String, String> uriVariables=**new** HashMap<>();

uriVariables.put("from", from);

uriVariables.put("to", to);

//calling the currency exchange service

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

CurrencyConversionBean response=responseEntity.getBody();

//creating a new response bean and getting the response back and taking it into Bean

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

}

//mapping for currency-converter-feign service

@GetMapping("/currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}") //where {from} and {to} represents the column

//returns a bean

**public** CurrencyConversionBean convertCurrencyFeign(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity)

{

CurrencyConversionBean response=proxy.retrieveExchangeValue(from, to);

logger.info("{}", response);

//creating a new response bean

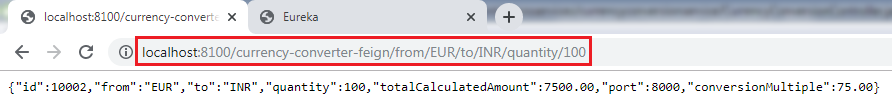
//getting the response back and taking it into Bean

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

}

}

**Step 12:**Execute the request <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/100>. It returns the following response, as shown below.



Let's see the log of **currency-conversion-service** in the console. The currency-conversion-service shows the following log:

ntroduction to Distributed Tracing

We can also see the log of **currency-exchange-service**. The currency-exchange-service shows the following log:

ntroduction to Distributed Tracing

Similarly, we can see the log for **netflix-zuul-api-gateway-server.**

ntroduction to Distributed Tracing

Let's have a close look at the above three logs for different services. We find that all three services have the same **trace Id (533f8d3966d8f4e7)**.

Spring Cloud Sleuth assigns a trace Id to the request. We can use this Id to trace the requests across multiple components. But there is a problem that this log is distributed in multiple places. We use **Zipkin** to remove this problem. With the help of Zipkin, we can centralize the logs in one place.

## **Distributed Tracing with Zipkin**

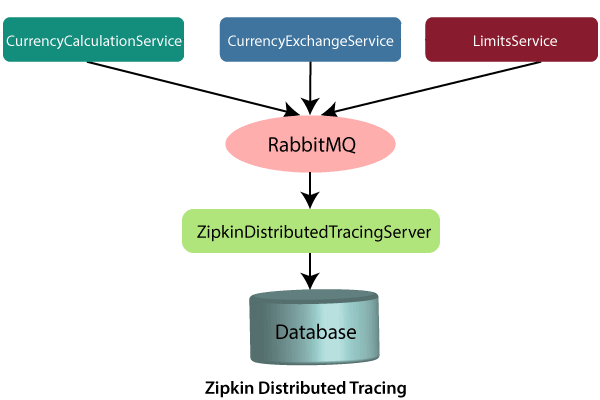
We have installed Spring Cloud Sleuth dependency in currency-conversion-service, currency-exchange-service, and netflix-zuul-api-gateway-server.  We have seen that a unique Id was being assigned to each request. We use these Id to trace the request across the logs of these multiple services.

However, we face a challenge in tracing. If we want to trace a request, we have to check the log of the individual application. The solution to this problem is called **centralized logs.**

We need to centralize all the logs from all the microservices. We can search through Id assigned by Spring Cloud Sleuth. At the centralized place, we will able to search and find out what is happening to a specific request.

There are the following solutions for centralize logging:

* **ELK Stack (Elastic Search)**
* **Kibana**
* **Zipkin**



In this distributed tracing, we will use **Zipkin distribute tracing server**. It gives us a consolidated view of all the microservices. We get all the logs messages form the individual microservices. The Zipkin server collects the log messages. All the microservices puts the log messages on the queue called **RabbitMQ,** and the Zipkin picks these log messages from the RabbitMQ. The Zipkin tracing server is connected with the database.

In our case, we use the in-memory database. We will pull log messages from the database. In the next step, we will install RabbitMQ.

# Installing RabbitMQ Server

## **RabbitMQ**

**RabbitMQ** is widely deployed open-source **message broker** software that implements **Advanced Message Queuing Protocol**(AQMP). It is lightweight and easy to deploy in the cloud. It supports multiple messaging protocols. It can be deployed in a distributed environment to meet **high-scale**and**high-availability** requirements. It is modeled on the AMQP standard. The RabbitMQ is written in the **Erlang** programming language. It is developed on the **Open Telecom Platform (OTP)** framework for clustering and failover.

RabbitMQ runs on different operating systems and cloud environments. It provides a large number of platforms like **Java, .NET, Python,** etc.

### **Advantagesof RabbitMQ**

* Fast performance
* Polyglot (using several languages)
* Easy Management
* No Erlang knowledge needed
* Great documentation

### **AMQPdefines:**

* Where to send messages **(Routing)**
* How to get there **(Delivery)**
* What goes in must come out **(Fidelity)**

### **Message broker**

A message broker sits between the machine and the distributed computing system. Instead of passing the messages directly to the receiver, the messages are first sent to the message broker (RabbitMQ). The message broker orders the messages in an optimized queue and passes them to the receiving machine when the machines are ready to process the messages.

A message might be a **command to process an order, run a specified task, a pull request**made to a database.

The machine that sends the message is called the **producer**. The machine that receives the message is called the **consumer**. The bit in the middle is called the**broker**.

Message broker does the following:

* **Decouple** the message publisher and consumer
* **Store** the message
* **Routing** the message
* **Monitoring** and **management** of message
* **Transform** message format between producer and consumer

## **Erlang**

**Erlang** is a **compiled, fault-tolerant, concurrent, dynamically typed**programming language. It is used to build a massively scalable, real-time system with requirements on high availability. It is used in **banking**, **e-commerce**, **telecom**, **computer telephony,** and **instant messaging**.

## **OTP**

OTP stands for **Open Telecom Platform**. It is a collection of **Erlang libraries** and **design principles**. It provides middleware to develop these systems. It includes its own tools such as **distributed database**, **applications to interface towards other languages, debugging** and **release handling** tools.

## **How to install RabbitMQ on Windows**

**Remember:**Before installing RabbitMQ, we need to install **Erlang**.

**Step 1:** Download and install **Erlang** from <https://erlang.org/download/otp_win64_22.1.exe>.

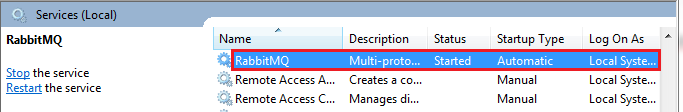
**Step 2:** Download and install **RabbitMQ** from <https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.8.1/rabbitmq-server-3.8.1.exe>.

**Step 3:**Open the **command prompt** and run the following commands one by one:

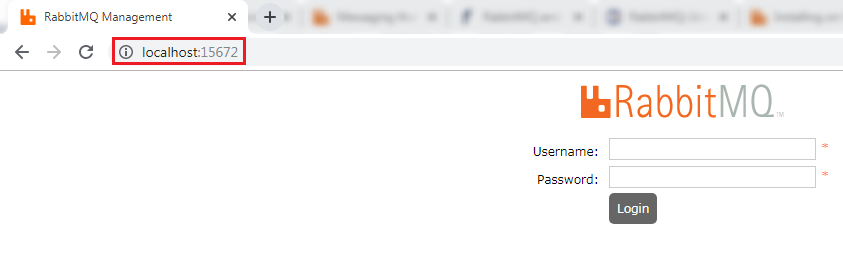
1. c:\>cd\
2. c:\>cd Program Files
3. c:\Program Files>cd RabbitMQ Server
4. c:\Program Files\RabbitMQ Server>dir
5. c:\Program Files\RabbitMQ Server>cd rabbitmq\_server-3.8.1
6. c:\Program Files\RabbitMQ Server\rabbitmq\_server-3.8.1>dir
7. c:\Program Files\RabbitMQ Server\rabbitmq\_server-3.8.1>cd sbin
8. c:\Program Files\RabbitMQ Server\rabbitmq\_server-3.8.1\sbin>dir
9. c:\Program Files\RabbitMQ Server\rabbitmq\_server-3.8.1\sbin>rabbitmq-plugins enable rabbitmq\_management

**Step 4:** Press the Windows key and type **services**or press **Windows key+R**and type **services.msc**.

**Step 5:** Select the **RabbitMQ**service->right-click -> Restart.



**Step 6:** Open the browser and type [http://localhost:15672](http://localhost:15672/). By default, the management plug-in runs on port **15672**.



**Step 7:**Provide the **Username** and **Password**and click on **Login** button**.**The default username and password is **guest**.

The following page shows the RabbitMQ user interface.

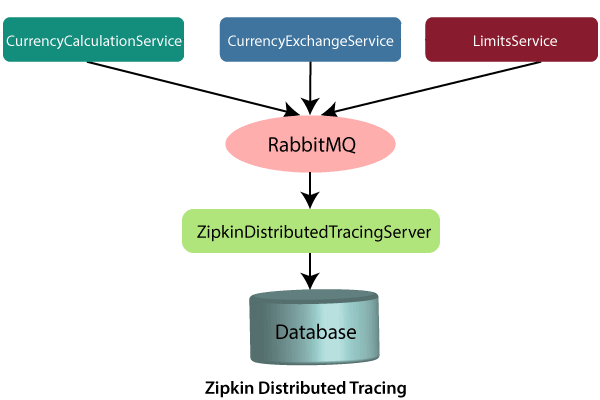
# Distributed tracing using Zipkin

## **What is distributed tracing?**

Distributed tracing is a technique used to **profile** and **monitor** applications, especially those built using the microservice architecture. Distributed tracing, also called **distributed request** **tracing**. IT and DevOps teams can use distributed tracing to monitor applications.

It identifies the **failed** microservices or the services having **performance issues** when there are many services call within a request. It is very useful when we need to track the request passing through the multiple microservices. It is also used for measuring the performance of the microservices.

In the previous section, we have installed RabbitMQ that works as middleware. In this section, we will implement the Zipkin Server for Distributed Tracing.



In the above figure, the ZipkinDistributedTracingServer connected to the in-memory database. All the microservices will put the messages in the RabbitMQ server. ZipkinDistributedTracingServer consumes the messages from the RabbitMQ server.

In this section, we will focus on installing ZipkinDistributedTracingServer and will connect it with both the RabbitMQ server and the in-memory database.

Let’s see how to connect the Zipkin server to other servers.

**Step 1:**Download the Zipkin server.

Search **Zipkin quickstart** on Google. Click on the link **Quickstart OpenZipkin**. We get the two options to quick start Zipkin, one is Docker, and the other is Java. But we will use the Java approach.

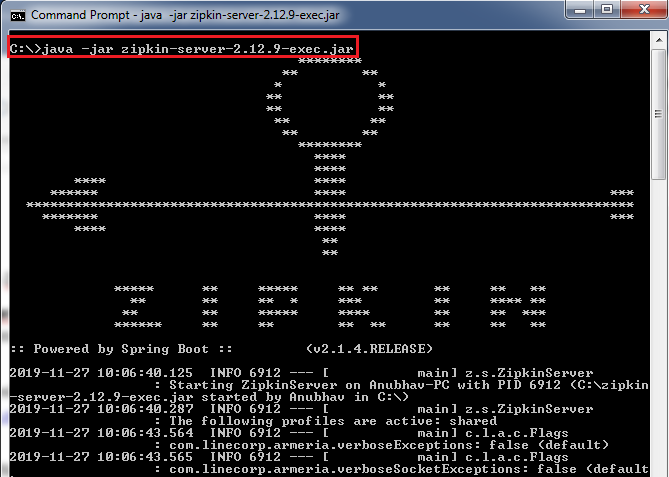
**Step 2:**Download the **zipkin-server-2.12.9-exec.jar** file from <https://search.maven.org/remote_content?g=io.zipkin&a=zipkinserver&v=LATEST&c=exec>.

**Step 3:** Copy the JAR file and paste it into any folder or drive. We have pasted the JAR file in the **C drive**directly.

**Step 4:**Open the **Command Prompt**and run the following commands:

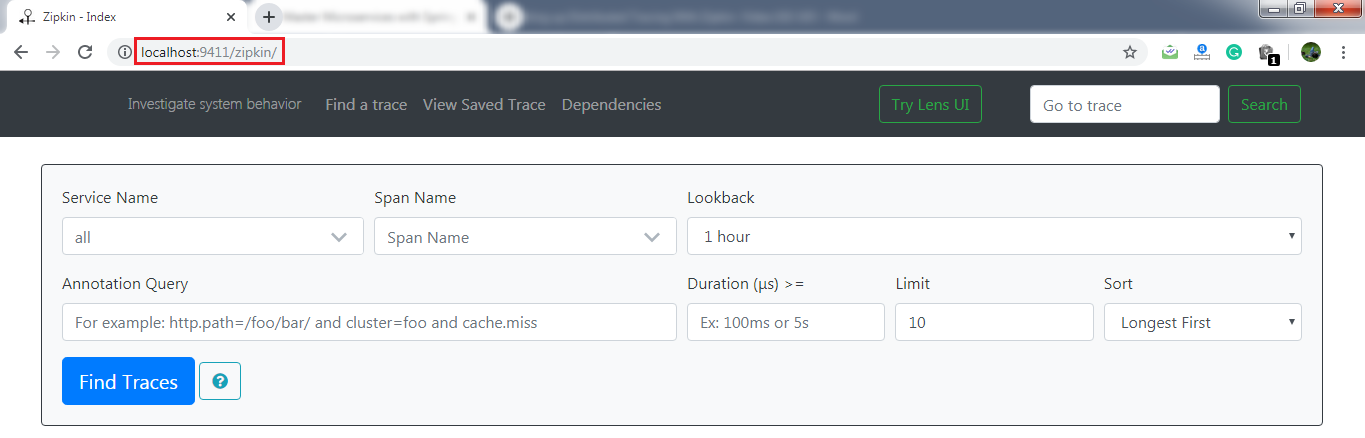
1. C:\> java -jar zipkin-server-2.12.9-exec.jar

The command launches the Zipkin server.



Let’s check the Zipkin server is running or not.

**Step 5:**Open the browser and type <http://localhost:9411/zipkin/>. It shows the dashboard of the Zipkin server. Here, the port **9411** is the default port of the Zipkin server.



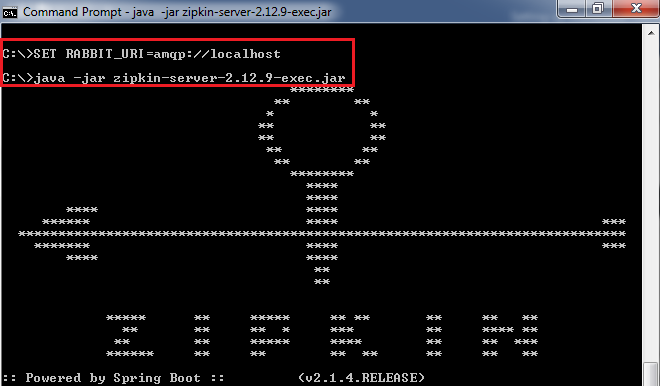
Here, the most important thing is that the Zipkin server must listen over the RabbitMQ server. So we have to start the RabbitMQ server in the background.

**Step 6:**Kill the Zipkin server by pressing the keys **Ctrl+c.**

**Step 7:**Run the following commands in the Command Prompt:

1. C:\>SET RABBIT\_URI=amqp://localhost
2. C:\> java -jar zipkin-server-2.12.9-exec.jar

The commands again start the Zipkin server along with the RabbitMQ server.



In this section, we have installed the Zipkin server. We have also started the RabbitMQ server and connect it to the Zipkin server. Now the Zipkin server is listening over the RabbitMQ server. But the microservices are not putting the trace messages in the RabbitMQ.

In the next step, we will start putting the trace messages in the RabbitMQ.

# Connecting Microservices to Zipkin

In this section, we will connect microservices to put their trace messages on the RabbitMQ. Once we place the messages on the RabbitMQ, it will be picked up by the Zipkin server.

Let’s connect the **currency-conversion-service, currency-exchange-service,** and the **netflix-zuul-api-gateway-server** to the RabbitMQ server.

**Step 1:** Open the **pom.xml**file of **currency-conversion-service, currency-exchange-service,** and the **netflix-zuul-api-gateway-server** and add the following dependencies:

We want to create a message in the format that it excepts. We need to add **Zipkin** dependency.

We will send the message to Zipkin that uses **amqp** message protocol. So we need to add the **amqp** dependency. Adding the dependency of amqp, we get the connection of RabbitMQ into **currency-conversion-service, currency-exchange-service,** and the **netflix-zuul-api-gateway-server**.

Now, we will try to run all the services together.

## **Using Zipkin UI Dashboard to trace the request**

In this step, we will fire a request. But before firing a request, make sure that the **five** services are running correctly. We must run these five services in the following order. Otherwise, we get errors or incorrect responses.

**Step 1:** Run the following services in the same order as we have listed.

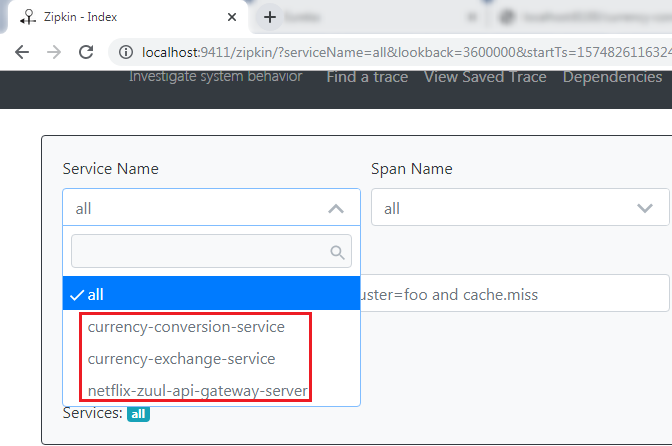
* **NetflixEurekaNamingServerApplication.java**
* **ZipkinDistributedTracingServerApplication**(run from the Command Prompt)
* **CurrencyExchangeServiceApplication.java**(on port 8000)
* **CurrencyConversionServiceApplication.java**(on port 8100)
* **NetflixZuulApiGatewayServerApplication.java**

**Remember:**Make sure that all the five applications are running correctly.

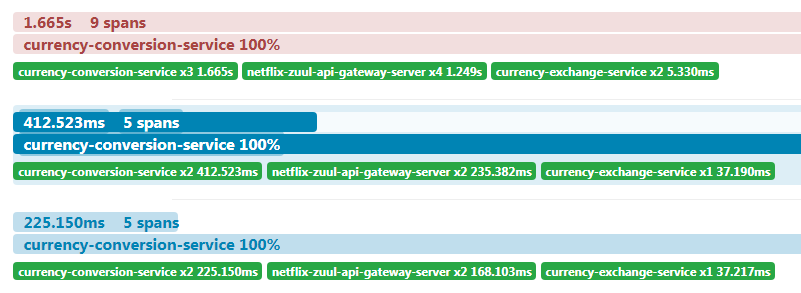
**Step 2:**Invoke the request <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/100>. It returns the response properly.

But we are interested to know what is happening in the background.

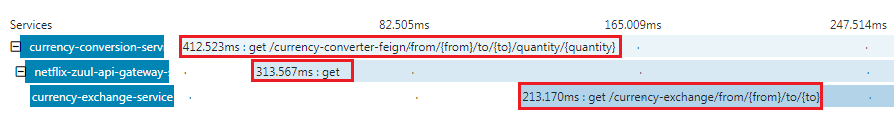
**Step 3:**Open the Zipkin UI. It shows all the three services that we have connected to Zipkin.



**Step 4:** Select any **one** service from the dropdown list and click on the **Find Traces** button. We have selected a **currency-conversion-service.**It shows the list of different execution of **currency-conversion-service.**

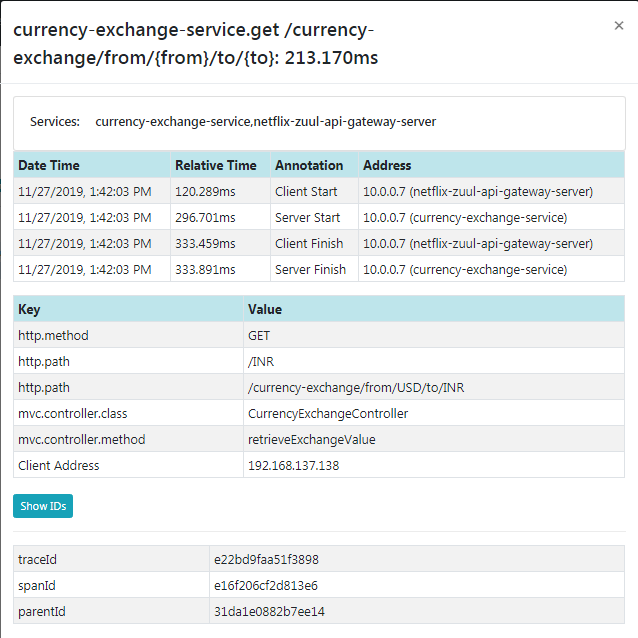


**Step 5:** Select any one of them. It shows the entire trace of a service.



In the above figure, when we invoke the **currency-converter-feign**, the request first goes to the **API Gateway**, and the API Gateway sends the request to the **currency-exchange-service.**

We can also see the detail of the services. In the following image, we have shown the details of the **currency-exchange-service.**



The most important challenge in microservices is to bring visibility (what is happening in the background). Hence, the Zipkin server enables the visibility of the services.

Understanding the need for Spring Cloud Bus

**Step 1:** Select the project **spring-cloud-config-server** and run the **SpringCloudConfigServerApplication.java**file.

**Step 2:** Select the project **limits-service**and run the **LimitsServiceApplication.java**file.

**Step 3:**Open the browser and type <https://localhost/8080/limits>. It returns the following response:

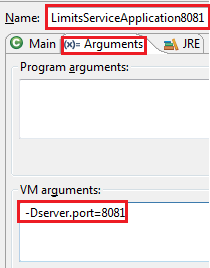
1. {"maximum":222,"minimum":2}

We are getting these values from the **limits-service-qa.properties** file, because we have configured this file into the **bootstrap.properties** file of the **limits-service**.

In the next step, we will create one more instance of **LimitsServiceApplication.**

**Step 4:**Create an instance of **LimitsServiceApplication.**

Right click on the **limits-service**project -> Run As -> Run Configurations… -> Right-click on the **LimitsServiceApplication** -> Duplicate -> Rename the application name to **LimitsServiceApplication8081**-> Click on the **Arguments** tab -> Provide the VM arguments:**-Dserver.port=8081**-> Run**.**



An instance of LimitsServiceApplication will run on port **8081**.

**Step 5:**Open the browser and invoke the URL <http://locahost:8081/limits>. It returns the same response as the original limits-service sends.

1. {"maximum":222,"minimum":2}

The two instances of limits-service up and running.

**Step 6:** Make the changes in the **limits-service-qa.properties,**we have changed the minimum value from **2** to **22**.

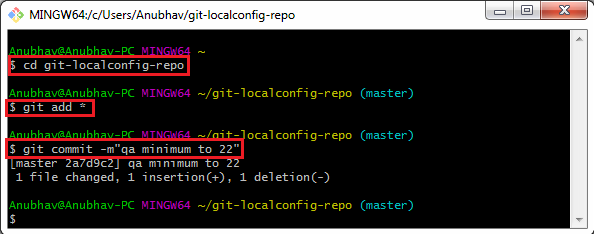
**limits-service-qa.properties**

1. limits-service.minimum=22
2. limits-service.maximum=222

We are required to **commit** the changes.

**Step 7:**Open the **Git Bash** and run the following commands:

1. $ cd git-localconfig-repo
2. $ git add \*
3. $ git commit –m "qa minimum to 22"



Again, invoke the URLs <http://localhost:8080/limits> and <http://localhost:8081/limits>. Both the URLs return the old values, while we have committed the values in the Git repository. It does not reflect the values in the **limits-service**.  To make the changes in the limits-service, we will use **Postman.**

#### Note**: Before moving to the next step, remove the security from the limits-service.**

**Step 8:** Open **limits-service.properties**file and disable the **security** by using the following statement.

1. management.security.enabled=**false**

Again, set the old values in the **limits-service-qa.properties**file and commit the changes.

**Step 9:**Open the **Postman** and send a **POST** request with the URL <http://localhost:8080/application/refresh>.

#### Note**: If you are invoking the URL http://localhost:8080/limits and it does not return the new values. So to get the new values on invoking the URL http://localhost:8081/limits, you have to do the following: Open the Postman and send a POST request with the URL http://localhost:8081/application/refresh.**

**Step 10:**  Invoke the URLs <http://localhost:8080/limits> and <http://localhost:8081/limits>. Now both the URLs return the changed value.

1. {"maximum":222,"minimum":22}

We have created the two instances of the limits-service. Suppose there are a hundred instances of the limits-services that are running in parallel. We need to invoke a hundred URLs to refresh the configuration from the Git repository.

Invoking a hundred URLs is not as easy as the number of limits-service increases. Along with this, the maintenance headache for the service also grows.

Whenever we make the changes in the configurations, it must reflect changes in the microservices. Here, the **Spring Cloud Bus**provides the solution for this, so we do not need to call hundred URLs.

Spring Cloud Bus provides a URL for all the hundred instances. When we invoke that URL, all the instances of the microservices would be updated with the latest values from the Git configuration.

In the next step, we will implement the Spring Cloud Bus.

# Implementing Spring Cloud Bus

In this section, when we make the changes in the Git repository, we have to hit multiple instances of the **limits-service** to refresh the configuration.

We will invoke one URL, and it will reflect all the hundred instances of the microservices. Here we will use **Spring Cloud Bus**. There are many options available in the Spring Cloud Bus: **Apache** **Kafka, RabbitMQ,**etc. In this section, we will use **RabbitMQ**.

**Note:** Before moving to the next step, make sure that the **RabbitMQ** server is running in the background.

Let’s implement the Spring Cloud Bus in **limits-service** and **spring-cloud-config-server.**

**Step 1:**Open the **pom.xml** file of **limits-service** and **spring-cloud-config-server**project**.**Add the **amqp**dependency and save the file to reflect the changes.

1. <dependency>
2. <groupId>org.springframework.cloud</groupId>
3. <artifactId>spring-cloud-starter-bus-amqp</artifactId>
4. </dependency>

**Step 2:** Open **bootstrap.properties** file and disable the management security by adding the following statement:

dependency and save the file to reflect the changes.

1. management.security.enabled=**false**

**Step 3:** First, run the **SpringCloudConfigServerApplication.java** file and then run the two instances of limits-service: **LimitsServiceApplication**and**LimitsServiceApplication8081.**

**Step 5:**Invoke the URL <http://localhost:8080/limits>. It returns the following response:

dependency and save the file to reflect the changes.

1. {"maximum":222,"minimum":22}

Now, invoke the URL <http://localhost:8081/limits>. It also returns the same response.

**Step 6:** Open the **limits-service-qa.properties** file and change the minimum value from **22** to **29**.

Now we have to commit the changes.

**Step 7:**Open the **Postman**and send a **POST** request with the URL <http://localhost:8080/bus/refresh>.

#### Note:**If you are using Spring Boot 2.0.0 or above versions, use the following URL: http://localhost:8080/actuatror/bus-refresh**

**Step 8:** Open the browser and invoke both instances (<http://localhost:8080/limits> and <http://localhost:8081/limits>) of limits-service. Both the URLs return the updated value that we have configured in the **limits-service-qa.properties**file**.**

1. {"maximum":222,"minimum":29}

We have seen that the minimum value changes to 29. Here, you can notice that we have not committed the changes in the Git repository manually, but the changes reflect in both instances of limits-service.

Instead of committing the Git repository, we have invoked a URL <http://localhost:8080/bus/refresh>. All this happening due to **Spring Cloud Bus**.

In the specific instance, we have run the Spring Cloud Bus over **RabbitMQ**. When we start the application, it also starts with that. After starting the Spring Cloud Bus, all the microservices registers with the **Bus**.

When we make changes in configuration, and the changes are called on any of the instances, the microservice sends an event over the **Spring Cloud Bus**. The Spring Cloud Bus propagates that event to all the microservice instances that are registered with it.

In this section, we have solved the problem of calling multiple instances of microservices.

# Fault Tolerance with Hystrix

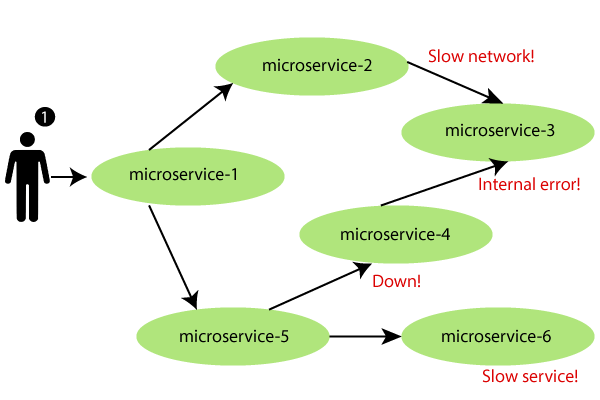
Microservices must be extremely reliable because they depend on each other. The microservice architecture contains a large number of small microservices. These microservices communicate with each other in order to fulfill their requirements.

The instances of microservices may go up and down frequently. **As the number of interactions between microservices increases, the chances of failure of the microservice also increases in the system.**

## **Fault Tolerance**

Consider a scenario in which six microservices are communicating with each other. The **microservice-5** becomes down at some point, and all the other microservices are directly or indirectly depend on it, so all other services also go down.

The solution to this problem is to use a **fallback** in case of failure of a microservice. This aspect of a microservice is called **fault tolerance**.

[[1]](#footnote-1)

**Fault tolerance** can be achieved with the help of a **circuit breaker**. It is a pattern that wraps requests to external services and detects when they fail. If a failure is detected, the circuit breaker opens. All the subsequent requests immediately return an error instead of making requests to the unhealthy service. It monitors and detects the service which is down and misbehaves with other services. It rejects calls until it becomes healthy again.

## **Hystrix**

Hystrix is a library that controls the interaction between microservices to provide latency and fault tolerance. Additionally, it makes sense to modify the UI to let the user know that something might not have worked as expected or would take more time.

**Implementing Fault Tolerance with Hystrix**

**Step 1:** Open the **pom.xml** file of **limits-service** and add the Hystrix dependency

1. <dependency>
2. <groupId>org.springframework.cloud</groupId>
3. <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
4. </dependency>

**Step 2:**Open **LimitsServicesApplication.java** file and enable **Hystrix** by using the annotation **@EnableHystrix.**

**LimitsServicesApplication.java**

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.netflix.hystrix.EnableHystrix;
5. @SpringBootApplication
6. @EnableHystrix
7. **public** **class** LimitsServiceApplication
8. {
9. **public** **static** **void** main(String[] args)
10. {
11. SpringApplication.run(LimitsServiceApplication.**class**, args);
12. }
13. }

**Step 3:**Open the **LimitsConfigurationController.java** file and create a **Get** method.

1. @GetMapping("/fault-tolerance-example")
2. //configuring a fallback method
3. @HystrixCommand(fallbackMethod="fallbackRetrieveConfigurations")
4. **public** LimitConfiguration retrieveConfigurations()
5. {
6. **throw** **new** RuntimeException("Not Available");
7. }
8. //defining the fallback method
9. **public** LimitConfiguration fallbackRetrieveConfigurations()
10. {
11. //returning the default configuration
12. **return** **new** LimitConfiguration(999, 9);
13. }

Let’s understand what is happening in the above method.

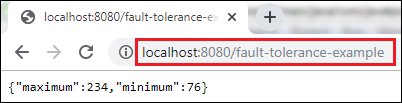
In the above method, we have created a Get mapping for fault tolerance. In the next line, we have used an annotation **@HystrixCommand** to configure the **fallback** method. We have defined a method with the name **fallbackRetrieveConfigurations()** that returns the default value if any fault occurs.

**Fallback method**

The fallback method is a method that invokes when a fault occurs. Hystrix allows us to define a fallback method for each service method. Here one question arises that if the method throws an exception, what should be returned to the consumer?

So answer is that if **retrieveConfiguraions()** fails, the method **fallbackRetrieveConfigurations()**is called. The fallback method returns the hardcoded **LimitConfiguration** instance.

**Step 4:**Open the browser and invoke the URL <http://localhost:8080/fault-tolerance-example>. It returns the values that we have returned in the **fallbackRetrieveConfigurations()**method.



1. [↑](#footnote-ref-1)