# What is Spring Cloud?

Spring Cloud is a framework for building robust cloud applications. Spring Cloud provides a solution to the commonly encountered patterns when developing a distributed system.

## Why is Spring Cloud used?

Spring Cloud framework provides tools for developers to build a robust cloud application quickly. We can also build the microservice-based applications, for example, **configuration management, service discovery, circuit breakers, intelligent routing, cluster state, micro-proxy, a control bus, one time tokens, etc**. Using Spring Cloud, a developer can quickly develop services and applications that implement the design patterns. These patterns work well in any distributed environment, including the **bear metal data centers, developer's laptop,** and managed platform such as **Cloud Foundry**.

**Bear metal data centers:** It is a physical server dedicated to a single-tenant (a person who occupies server on rent). These are not shared between the customers. The tenant can optimize the performance according to its needs for performance, security, and reliability. The bare metal data centers are also known as Single-tenant physical server or managed dedicated server. The operating system is installed directly on the bare metal server and delivers better performance.

**Cloud Foundry:** Cloud Foundry is an open-source, multi-cloud Platform as a Service (PaaS). You can deploy your application on your own computing infrastructure.

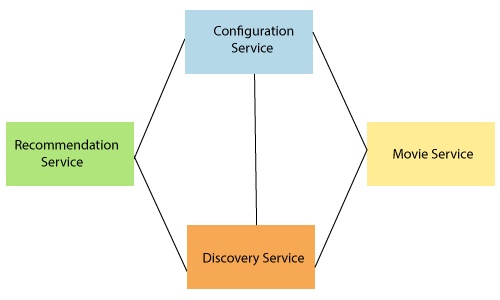
# Features of Spring Cloud

The great part about Spring Cloud is that it builds the concept of Spring Boot. Spring cloud is built upon some of the common building blocks of Spring framework which are as follows:

* Intelligent routing and service discovery
* Service-to-Service Call
* Load Balancing
* Leadership Election
* Global Locks
* Distributed Configuration
* Distributed Messaging

### **Intelligent Routing and Service discovery**

When building a microservices on Spring Cloud, there is a primary concern to deal with the first two primary microservices: **configuration service** and the **discovery service**.



The above figure represents the set of **four microservices**. The connection between each service indicates dependency. All services are dependent on each other. The configuration service lies at the top, and the discovery services at the bottom. There are two microservices in-between which are **Recommendation Service** and **Movie Service**.

### **Service-to-Service calls**

It is the process of "how a microservice communicates with other dependent microservices via service registry or Eureka server." There is a sequence which follows in the service-to-service call.

OOPs Concepts in Java

* Registering the service
* Fetching the Registry
* Finding the downstream service
* Resolving the Underlying IP address
* Call the rest Endpoint

### **Load Balancing**

Load balancing efficiently distributes network traffic to multiple backend servers or server pool. The objective of load balancing is to maximize throughput, minimize response time, increase efficiency, and optimize resource uses. It **avoids overload** of any single resource. Using multiple components with load balancing may increase **reliability** and **availability** through redundancy.

### **Leadership Election**

Leadership election allows the application to work with other application via a third-party system. The leadership election is used to provide **global state** or **global ordering** without sacrificing availability.

### **Global Locks**

Global locks are used to ensure that no two thread simultaneously access the same resource at the same time. The programmer uses a mechanism to remove such situation, is called a **lock**. Each thread first **cquires the lock, operate on the resources,** and **release the lock** for other thread.

### **Distributed Configuration**

Distributed configuration is to configure every instance of all microservices. "Spring cloud config server" provides client-side support for externalized configuration in a distributed system. With the distributed configurations, we have a central place to manage external properties for applications across all environment.

### **Distributed Messaging**

The distributed messaging system provides the benefit of reliability, scalability, and persistence. The messaging pattern follows the **Publish-Subscribe** (Pub-Sub) model. In the Pub-Sub model, the sender of the message is called publisher and receiver of the message is called subscribers. **Apache Kafka** and **RabbitMQ** are the popular high throughput messaging system.

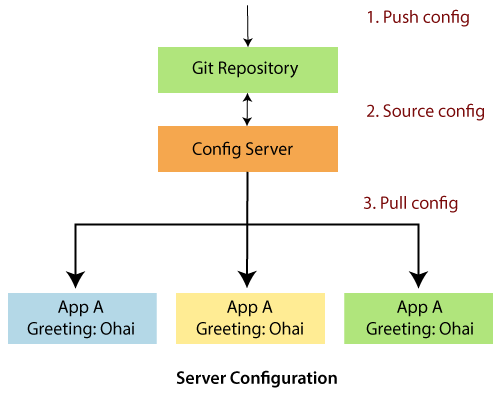
Spring Cloud Components

There are the following components:

* Configuration
* Service Discovery
* Circuit Breakers
* Routing and Messaging
* API Gateway
* Tracing
* CI Pipeline and Testing

Configuration

Spring Cloud configuration components provide server-side and client-side support for externalized configuration in a distributed system. We can manage the external properties with config server for applications across all environments. Spring Cloud config server can use Git, SVN (Apache Subversion), filesystem, and Vault to Store config. Config clients (microservice app) retrieve the configuration client from the server on startup.



Service Discovery

The service discovery is the automatic detection of devices and services over the network. In other words, service discovery is how an application and microservices connect in the distributed environment. Service discovery implementations include both:

* The **central server** that maintains a global view of the address.
* The **clients** that connect to the central server can update and retrieve the address.

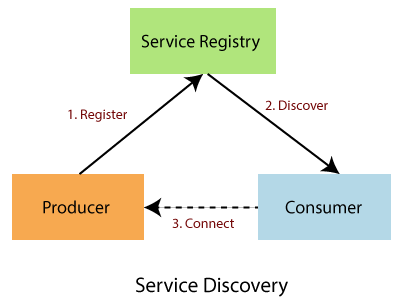
There are **two** discovery patterns: **Client-side discovery** and **Server-side discovery**.

10.6M

155

SQL CREATE TABLE

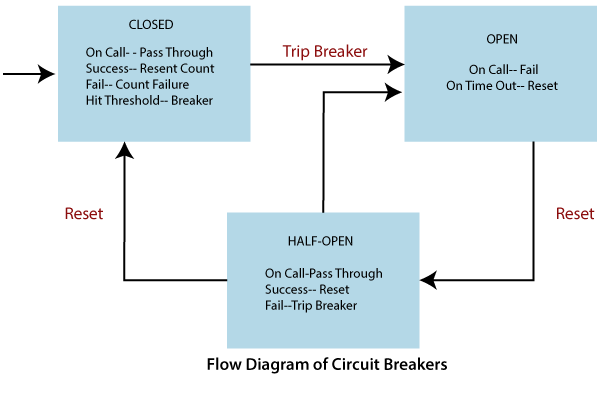
* **Client-side discovery:** In the Client-side discovery, client is responsible for determining the network location of available services. The client uses a **load-balancing algorithm** to select one of the available services and make a request. **Netflix OSS** is an example of a client-side discovery pattern.
* **Server-side discovery:** In the server-side discovery, the client makes an HTTP request to a service through a load balancer. The load balancer contacts to service registry and route each request to an available service instance. Similar to client-side discovery, service instances are registered and deregistered with the service registry. The **AWS ELB** (Elastic Load Balancer) is an example of server-side discovery. ELB balances the external traffic from the internet.



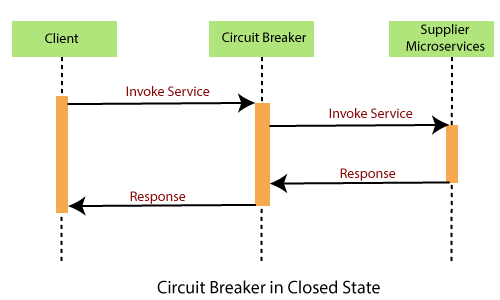
In the above figure producer is a software that sends a message to a message broker (Service Registry). A consumer is also a software that receives the message and processes it.

Circuit Breakers

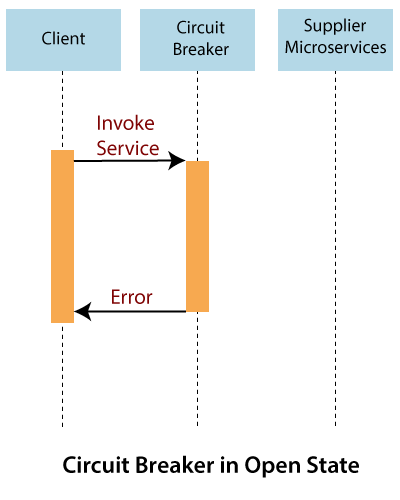
Netflix has created a library called **Hystrix**. It implements the circuit breakers pattern. Circuit breakers calculate when to open and close the circuit and what to do in case of failure. When all services fail at some point, the circuit breaker handles these failures gracefully. The circuit breakers have three states: **OPEN, CLOSED,** and **HALF-OPEN** State.



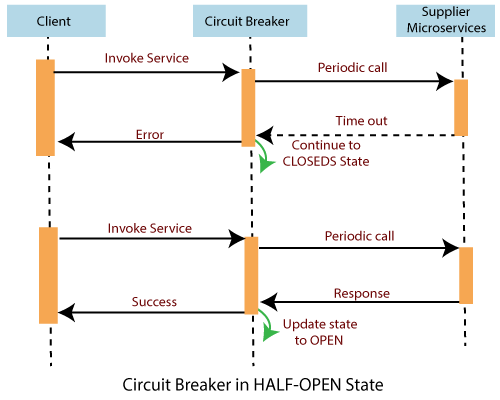
**CLOSED State:** If the Circuit breaker is in the CLOSED state and all calls pass through to the supplier microservices. It responds without any latency.



**OPEN State:** The circuit breaker returns an error calls without executing the function.

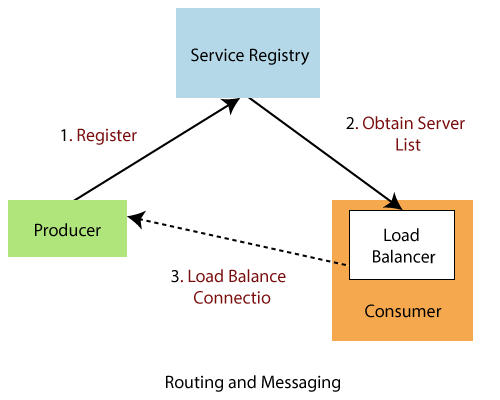


**HALF-OPEN State:** The circuit turns to HALF-OPEN state when a function execution is **timed out**. It test that underlying problem still exists or not. It is a **monitoring** and **feedback mechanism**. It makes a trial call to supplier microservices to check if it has recovered. If the call to the supplier is timed out, then the circuit remains in the **OPEN** state. If the call return success, the circuit-switched to the **CLOSED** state. The circuit breaker returns all external calls to the service with an error during the **HALF-OPEN** State.



Routing and Messaging

The cloud application made up of many microservices so the communication will be critical. Spring Cloud supports communication via messaging or HTTP request. Routing uses **Netflix Ribbon** and **Open Feign while** messaging uses Kafka or Rabbit MQ.

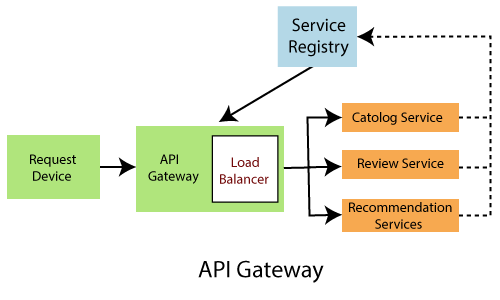


API Gateway

API Gateway allows us to route API request (external or internal) to connect services. It also provides a library for building an API gateway on the top of Spring MVC. Its aims to provide cross-cutting concerns to them, such as **security** and **monitoring**.

**Features of API Gateway**

* Built on Spring framework 5, project reactor and Spring Boot 2.0
* Able to match routes on any requested attribute
* Predicates and filters are specific to routes
* Hystrix circuit Breaker integration
* Spring Cloud Discovery Client integration
* Easy to write Predicates and filters
* Request Rate Limiting
* Path rewriting



Tracing

Spring Cloud's other functionality is **distributed tracing**. Tracing is a single request to get data from the application. Tracing results in an exponentially larger number of requests to various microservices.

We can add **Spring Cloud Sleuth** library in our project to enable tracing. Sleuth is responsible for recording **timing**, which is used for **latency analysis**. We can export this timing to Zipkin.

Zipkin is a distributed tracing tool specially designed for **analyzing latency problem** inside the microservice architecture. It exposes HTTP endpoint used for collecting input data. If we required to add tracing in our project, we should add the **spring-cloud-starter-zipkin** dependency.

In the microservices, the input traffic volume is so high, so we cannot collect an only certain amount of data. For that purpose, the Spring Cloud Sleuth provides a **sampling policy**. The sampling policy allows us how much input traffic is sent to Zipkin for analysis. To enable this feature, we have to add the **spring-cloud-sleuth-stream** dependency.

Cl Pipeline and Testing

Spring Cloud pipeline is an opinionated (self-important) pipeline for Jenkins and Concourse, which creates pipeline automatically for the application. The building, testing, and deploying in various services is critical to having a successful cloud-native application.

The Jenkins pipeline provides a set of the tool designed for modeling simple and more advanced delivery pipeline as code. The definition of a pipeline is written into a text file called Jenkinsfile.

The pipeline has **two** syntaxes: **Declarative** and **Scripted** pipeline. These syntaxes are divided into two parts: Steps, and Stages. **Steps** are the fundamental part of the pipeline as they tell the Jenkins server what to do. **Stages** are the major part of a pipeline. Stages logically group a couple of steps, which displayed on the pipeline's result screen.

# Main projects of Spring Cloud

**Spring Cloud Connectors:** It provides simple abstraction. It provides out-of-box support for discovering common services on Heroku and Cloud Foundry cloud platform. Heroku is a cloud platform as a service that provides hosting for web services. Spring supports Heroku through its Spring Cloud Connector library.

**Sting Cloud Pipelines:** The pipeline creates a common deployment pipeline. It also speeds up the time required to deploy a feature to production.

**Spring Cloud Contract:** It is a verifier tool for Consumer Driven Contract (CDC) development of JVM-based application. It deploys all microservices to end-to-end test. It creates mock microservices in the unit and integration testing.

**Spring Cloud Function:** It supports the implementation of business logic via functions. It can run standalone locally or in PaaS (Platform as a Service).

History of Java

**Spring Cloud OpenFeign:** Spring Cloud OpenFeign is a Java to HTTP binder. It reduces the complexity of binding.

**Spring Cloud AWS:** It is a part of Spring Cloud Umbrella project. It provides easy integration with Amazon Web Services. The developer can build their project around the AWS without having to care about maintenance.

**Spring Cloud Task:** It allows us to develop short-lived microservices using Spring Cloud and run them locally in the cloud or even in Spring Cloud Data Flow.

**Spring Cloud Stream:** It is used for building a highly saleable event-driven microservices.

**Spring Cloud Cluster:** It provides tools for building a cluster feature in a distributed system. For example, global locks and leadership election.

**Spring Cloud Data Flow:** It provides tools to create complex topologies for streaming batch data and pipeline. It supports data processing use cases.

**Spring Cloud Config:** It provides client and server-side support for externalized configuration in a distributed environment. We get a central place to manage external properties for applications across the distributed environment.

**Spring Cloud Netflix:** It provides integration with various Netflix OSS components like Eureka, Zuul, Hystrix, etc.

**Spring Cloud CloudFoundry:** It integrates the application with pivotal cloud foundry. It also provides service discovery and easy implementation of SSO (Single-Sign-On) and OAuth2 (OAuth2 is the method of authenticating access to the API) protected resources. It allows authentication without the external application getting the user email-address or password.

**Spring Cloud CLI:** It provides a command-line feature for sparing cloud. We can launch services like Edureka, Zipkin, Config server conveniently from the CLI.

**Spring Cloud Starter:** Spring Cloud Starters eases the curated set of dependency management for the consumers of Spring Cloud.

**Spring Cloud Bus:** It is a lightweight message broker. It can be used to broadcast state changes or other management instructions.

**Spring Cloud Sleuth:** It implements a distributing tracing solution for spring cloud.

**Spring Cloud Stream App Starters:** It provides out-of-box Spring Cloud Stream utility applications that can work independently or with Spring Cloud Data Flow. It includes connectors for the various middleware technologies and adapters for various network protocols.

**Spring Cloud Task App Starters:** It can be used with Spring Cloud Data Flow to create, deploy, and orchestrate short-lived microservices. These are the standalone executable that can be used for on-demand use-cases. For example, machine learning, database migration, and scheduled operations.

**Spring Cloud Zookeepers:** It provides configuration management and service discovery with apache zookeeper.

Difference between Spring Cloud and Spring Boot

Spring Cloud

Spring Cloud is a framework that provides facilities to use **cloud services** in your application. When it is used with **Eureka**, it acts as a **container orchestration tool**. The framework that provides an enterprise-level framework for integrating and managing container at scale is called container orchestration tool. It gives a developer-friendly environment for developing and deploying microservices.

**Advantages of Spring Cloud**

There are following advantages of Spring Cloud:

* Cloud-native based development
* Microservice-based architecture
* Inter-service communication
* Follows the Spring Boot model
* It is cloud-agnostic

Spring Boot

Spring Boot is a Rapid Application Development platform. It uses various components of Spring framework. It is a framework to develop **Restful API** and **microservices quickly**. It is similar to Spring framework but comes with some features like **auto-configurators, starters, cli,** etc.

Features of Java - Javatpoint

**Advantages of Spring Boot**

* We can quickly develop and run standalone web applications and microservices at very less time.
* It automatic configures Spring functionality whenever required.
* The beans are initialized, configured and wired automatically.
* Embedded server avoid complexity in deployment.
* It provides no XML based configuration.
* It provides opinionated 'starter' POMs to simplify your Maven configuration.
* It provides production ready features such as metrics, health status, and externalized.
* Embed Tomcat, Jetty directly. There is no need to deploy WAR files.

# 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**: "Microservices are the small services that work together."

"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."

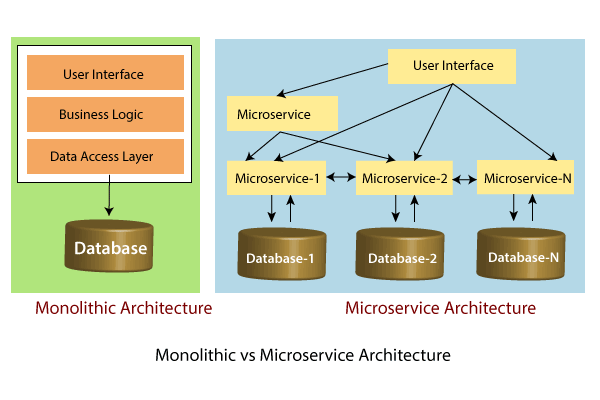
There is a bare minimum of centralized management of these services, which may be written in different programming language and use different data storage technologies.

## 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.

Exception Handling in Java - Javatpoint

### **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.

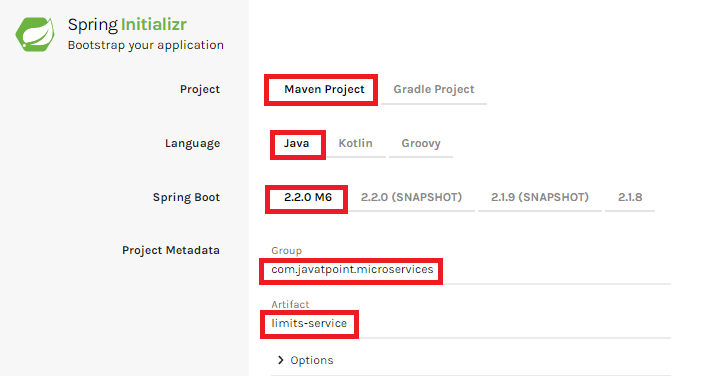
**Step 3**: Provide the **Group** name. In our case **om.javatpoint**

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

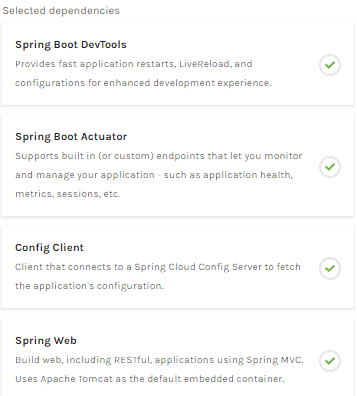
24M

497

Java Try Catch



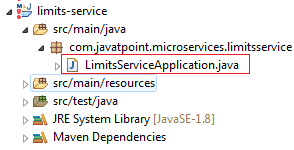
**Step 5**: Add the following dependencies: **Spring Web, Spring Boot DevTools, Spring Boot Actuator, Config Client**.



**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:

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.web.bind.annotation.GetMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. **import** com.javatpoint.microservices.limitsservice.bean.LimitConfiguration;
5. @RestController
6. **public** **class** LimitsConfigurationController
7. {
8. @GetMapping("/limits")
9. **public** LimitConfiguration retriveLimitsFromConfigurations()
10. {
11. **return** **new** LimitConfiguration(1000, 1);
12. }
13. }

**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:

1. **package** com.javatpoint.microservices.limitsservice.bean;
2. **public** **class** LimitConfiguration
3. {
4. **private** **int** maximum;
5. **private** **int** minimum;
6. //no-argument constructor
7. **protected** LimitConfiguration()
8. {
9. }
10. //generating getters
11. **public** **int** getMaximum()
12. {
13. **return** maximum;
14. }
15. **public** **int** getMinimum()
16. {
17. **return** minimum;
18. }
19. //genetrating constructor using fields
20. **public** LimitConfiguration(**int** maximum, **int** minimum)
21. {
22. **super**();
23. **this**.maximum = maximum;
24. **this**.minimum = minimum;
25. }
26. }

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.

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.boot.context.properties.ConfigurationProperties;
3. **import** org.springframework.stereotype.Component;
4. @Component
5. @ConfigurationProperties("limits-service")
6. **public** **class** Configuration
7. {
8. **private** **int** maximum;
9. **private** **int** minimum;
10. **public** **void** setMaximum(**int** maximum)
11. {
12. **this**.maximum = maximum;
13. }
14. **public** **void** setMinimum(**int** minimum)
15. {
16. **this**.minimum = minimum;
17. }
18. **public** **int** getMaximum()
19. {
20. **return** maximum;
21. }
22. **public** **int** getMinimum()
23. {
24. **return** minimum;
25. }
26. }

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

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.web.bind.annotation.GetMapping;
4. **import** org.springframework.web.bind.annotation.RestController;
5. **import** com.javatpoint.microservices.limitsservice.bean.LimitConfiguration;
6. @RestController
7. **public** **class** LimitsConfigurationController
8. {
9. @Autowired
10. **private** Configuration configuration;
11. @GetMapping("/limits")
12. **public** LimitConfiguration retriveLimitsFromConfigurations()
13. {
14. //getting values from the properties file
15. **return** **new** LimitConfiguration(configuration.getMaximum(), configuration.getMinimum());
16. }
17. }

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.

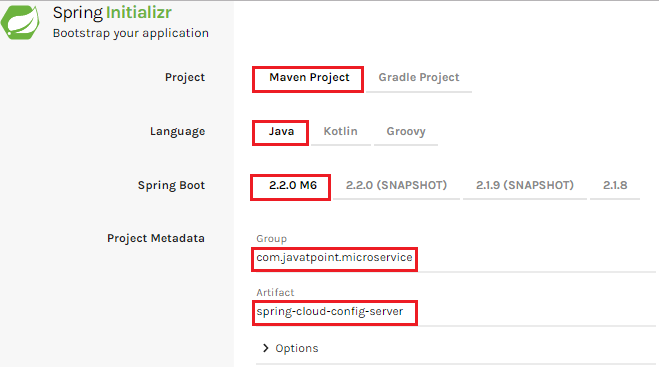
**Step** **3:** Provide the **Group** name. In our case, **com.javatpoint.microservices.**

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

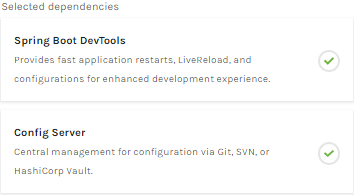
20.4M

387

Prime Ministers of India | List of Prime Minister of India (1947-2020)



**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/

Initializing a new Git repository:

1. 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:

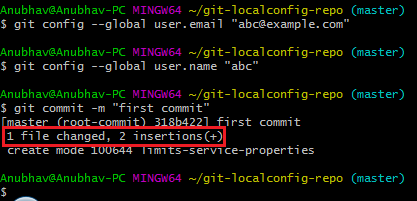
1. git config -global user.email abc@example.com
2. git config -global user.name "abc"

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:

Competitive questions on Structures in Hindi

Keep Watching

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

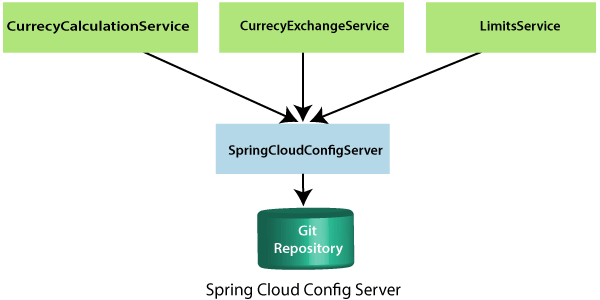
**Output**

1. {
2. name: "limits-service",
3. -profiles: [
4. "default"
5. ],
6. label: **null**,
7. version:"0898c54ae1deb62733728e37e4c7962f529ee9ad",
8. state: **null**,
9. -propertySources: [
10. - {
11. name: C:\Users\Anubhav\git-localconfig-repo\limits-service.properties",
12. -source: {
13. limits-service-minimum: "8",
14. limits-service-maximum: "88"
15. }
16. }
17. ]
18. }

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

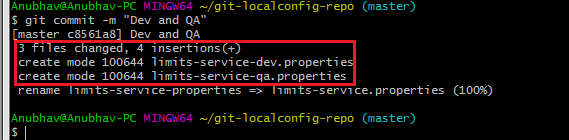
Now check the status of the files that have to be committed.

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. }

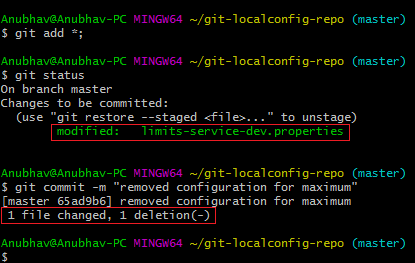
If we look at the limits-service-dev.properties file, the values are fetching from there.

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,

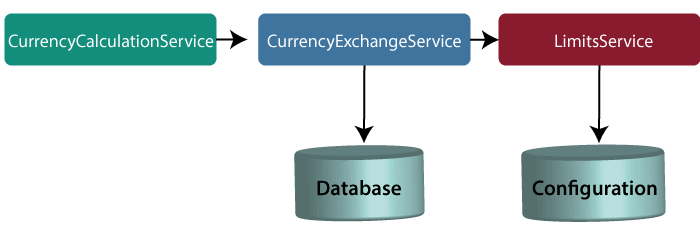
minimum: 2

}

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**.

10.6M

155

SQL CREATE TABLE

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

1. {
2. id: 101,
3. from: "USD",
4. to: "INR",
5. conversionMultiple: 72,
6. port: 8000
7. }

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**

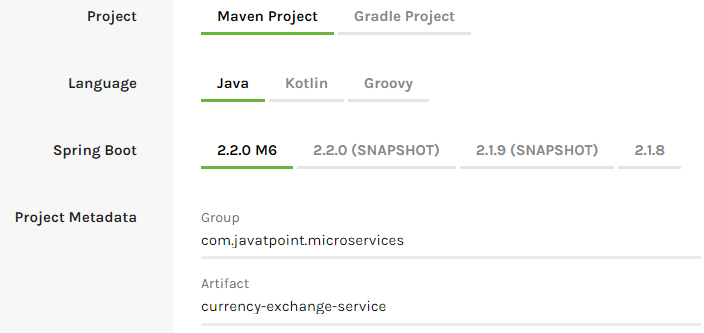
1. {
2. Id: 101,
3. from: "USD",
4. to: "INR",
5. conversionMultiple: 72,
6. quantity: 100
7. totalCalculatedAmount: 7200,
8. port: 8000
9. }

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**.

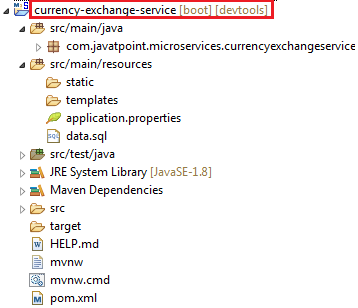
**Step 4:**Click on the **Generate Project** button. It will download the **zip** file of the project.

**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**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **import** org.springframework.boot.SpringApplication;
4. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
5. **import** org.springframework.web.bind.annotation.GetMapping;
6. **import** org.springframework.web.bind.annotation.PathVariable;
7. **import** org.springframework.web.bind.annotation.RestController;
8. @SpringBootApplication
9. @RestController
10. **public** **class** CurrencyExchangeController
11. {
12. @GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable
13. **public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)  //from map to USD and to map to INR
14. {
15. **return** **new**  ExchangeValue(1000L, from, to, BigDecimal.valueOf(65));
16. }
17. }

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

**ExchangeValue.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **public** **class** ExchangeValue
4. {
5. **private** Long id;
6. **private** String from;
7. **private** String to;
8. **private** BigDecimal conversionMultiple;
10. **public** ExchangeValue()
11. {
12. }
13. //generating constructor using fields
14. **public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple) {
15. **super**();
16. **this**.id = id;
17. **this**.from = from;
18. **this**.to = to;
19. **this**.conversionMultiple = conversionMultiple;
20. }
21. //generating getters
22. **public** Long getId()
23. {
24. **return** id;
25. }
26. **public** String getFrom()
27. {
28. **return** from;
29. }
30. **public** String getTo()
31. {
32. **return** to;
33. }
34. **public** BigDecimal getConversionMultiple()
35. {
36. **return** conversionMultiple;
37. }
38. }

**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:

1. {
2. id: 101,
3. from: "USD",
4. to: "INR",
5. conversionMultiple: 72,
6. port: 8000
7. }

## 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**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **public** **class** ExchangeValue
4. {
5. **private** Long id;
6. **private** String from;
7. **private** String to;
8. **private** BigDecimal conversionMultiple;
9. **private** **int** port;
10. **public** ExchangeValue()
11. {
12. }
13. //generating constructor using fields
14. **public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple) {
15. **super**();
16. **this**.id = id;
17. **this**.from = from;
18. **this**.to = to;
19. **this**.conversionMultiple = conversionMultiple;
20. }
21. //generating getters
22. **public** **int** getPort() {
23. **return** port;
24. }
25. **public** **void** setPort(**int** port) {
26. **this**.port = port;
27. }
28. **public** Long getId()
29. {
30. **return** id;
31. }
32. **public** String getFrom()
33. {
34. **return** from;
35. }
36. **public** String getTo()
37. {
38. **return** to;
39. }
40. **public** BigDecimal getConversionMultiple()
41. {
42. **return** conversionMultiple;
43. }
44. }

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.**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
5. **import** org.springframework.core.env.Environment;
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. @SpringBootApplication
10. @RestController
11. **public** **class** CurrencyExchangeController
12. {
13. @Autowired
14. **private** Environment environment;
15. @GetMapping("/currency-exchange/from/{from}/to/{to}") //where {from} and {to} are path variable
16. **public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)  //from map to USD and to map to INR
17. {
18. //taking the exchange value
19. ExchangeValue exchangeValue= **new** ExchangeValue (1000L, from, to, BigDecimal.valueOf(65));
20. //picking port from the environment
21. exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));
22. **return** exchangeValue;
23. }
24. }

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

1. {
2. id: 1000,
3. from: "USD",
4. to: "INR"
5. conversionMultiple: 65,
6. port: 8000
7. }

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

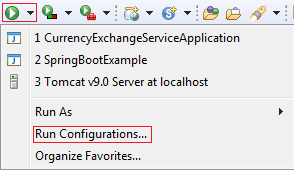
Now we will run **CurrencyExchangeServiceApplication** on a different port number. For this, we have to change the port in the **application.properties**file from 8000 to 8001, 8002, etc. whichever we want.

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

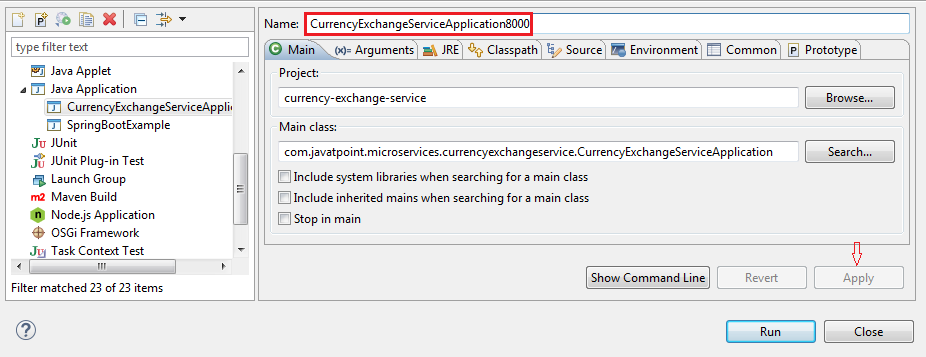
Let's create an instance of the **CurrencyExchangeServiceApplication**that runs on the port **8001**.

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

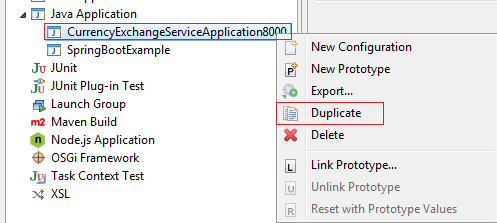
Or click on the highlighted symbol -> Run Configurations.



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



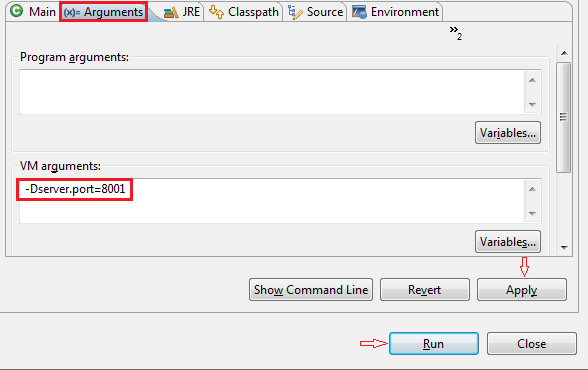
**Step 3:**Right-click on the **CurrencyExchangeServiceApplication8000**-> Duplicate.



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.

1. <dependency>
2. <groupId>com.h2database</groupId>
3. <artifactId>h2</artifactId>
4. <version>1.4.197</version>
5. <scope>test</scope>
6. </dependency>
7. <dependency>
8. <groupId>org.springframework.boot</groupId>
9. <artifactId>spring-boot-starter-data-jpa</artifactId>
10. <version>2.1.3.RELEASE</version>
11. </dependency>

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

Competitive questions on Structures in Hindi

Keep Watching

**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**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** java.math.BigDecimal;
3. **import** javax.persistence.Column;
4. **import** javax.persistence.Entity;
5. **import** javax.persistence.Id;
6. **import** javax.persistence.Table;
7. @Entity
8. @Table(name="Exchange\_Value")
9. **public** **class** ExchangeValue
10. {
11. @Id
12. @Column(name="id")
13. **private** Long id;
14. @Column(name="currency\_from")
15. **private** String from;
16. @Column(name="currency\_to")
17. **private** String to;
18. @Column(name="conversion\_multiple")
19. **private** BigDecimal conversionMultiple;
20. @Column(name="port")
21. **private** **int** port;
22. //default conatructor
23. **public** ExchangeValue()
24. {
25. }
26. //generating constructor using fields
27. **public** ExchangeValue(Long id, String from, String to, BigDecimal conversionMultiple)
28. {
29. **super**();
30. **this**.id = id;
31. **this**.from = from;
32. **this**.to = to;
33. **this**.conversionMultiple = conversionMultiple;
34. }
35. //generating getters and setters
36. **public** **int** getPort()
37. {
38. **return** port;
39. }
40. **public** **void** setPort(**int** port)
41. {
42. **this**.port = port;
43. }
44. **public** Long getId()
45. {
46. **return** id;
47. }
48. **public** String getFrom()
49. {
50. **return** from;
51. }
52. **public** String getTo()
53. {
54. **return** to;
55. }
56. **public** BigDecimal getConversionMultiple()
57. {
58. **return** conversionMultiple;
59. }
60. }

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.

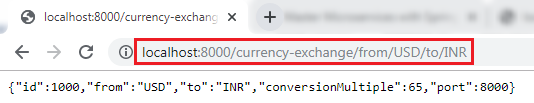
We have specified JDBS URL: **jdbc:h2:mem:javatpoint**

**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

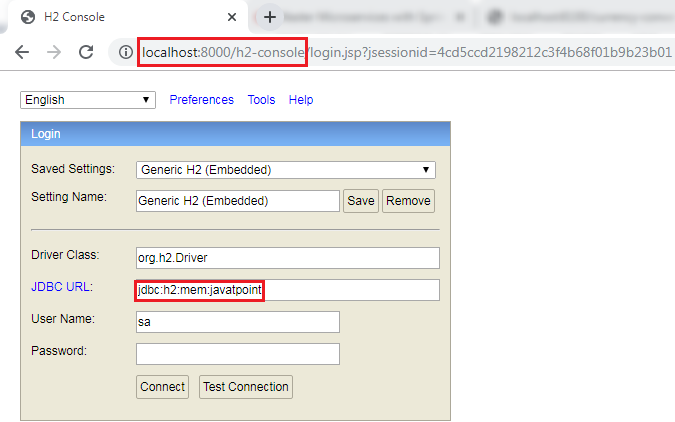
**Step 6:** Restart the application.

**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:



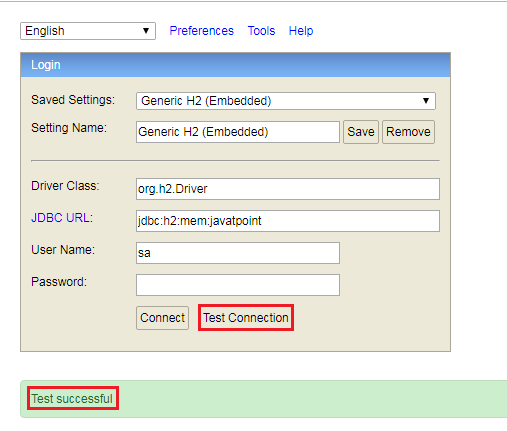
We can also see the data in the database which we have inserted in the **data.sql** file. To open the H2 Console, we have to do the following:

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

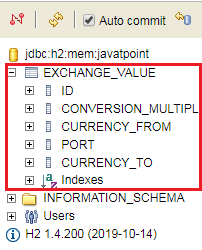


**Remember:** The **JDBC URL** must be the same as you have specified in the **application.properties** file. Do not write anything in the **User Name** and **Password** field. The default User Name is **sa.**

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



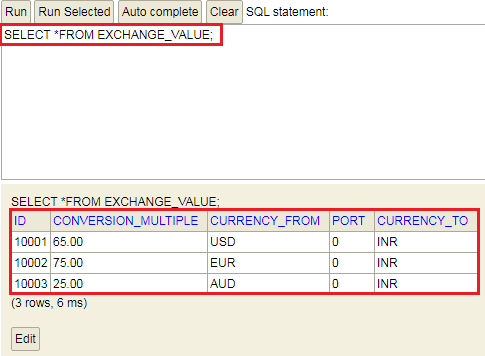
**Step 8:** Click on the **Connect** button. It shows the created table on the left-hand side of the page.



**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.

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

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

1. @Autowired
2. **private** ExchangeValueRepository repository;

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

27.3M

645

OOPs Concepts in Java

1. 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:

1. ExchangeValue findByFrom (String from);

**ExcahngeValueRepository.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** org.springframework.data.jpa.repository.JpaRepository;
3. **public** **interface** ExchangeValueRepository **extends** JpaRepository<ExchangeValue, Long>
4. {
5. //creating query method
6. ExchangeValue findByFromAndTo(String from, String to);
7. }

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

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

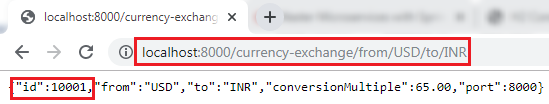
Instead of using the following statement:

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

**CurrencyExchangeController.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.core.env.Environment;
5. **import** org.springframework.web.bind.annotation.GetMapping;
6. **import** org.springframework.web.bind.annotation.PathVariable;
7. **import** org.springframework.web.bind.annotation.RestController;
8. @SpringBootApplication
9. @RestController
10. **public** **class** CurrencyExchangeController
11. {
12. @Autowired
13. **private** Environment environment;
14. @Autowired
15. **private** ExchangeValueRepository repository;
16. @GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable
17. **public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)   //from map to USD and to map to INR
18. {
19. ExchangeValue exchangeValue = repository.findByFromAndTo(from, to);
20. //setting the port
21. exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));
22. **return** exchangeValue;
23. }
24. }

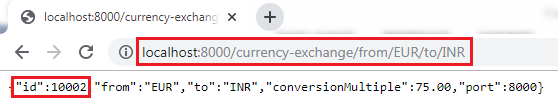
**Step 5:**Restart the application to pick up the changes. Open the browser and type the URI **http://localhost:8000/currency-exchange/from/USD/to/INR**. It returns the following response:



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.

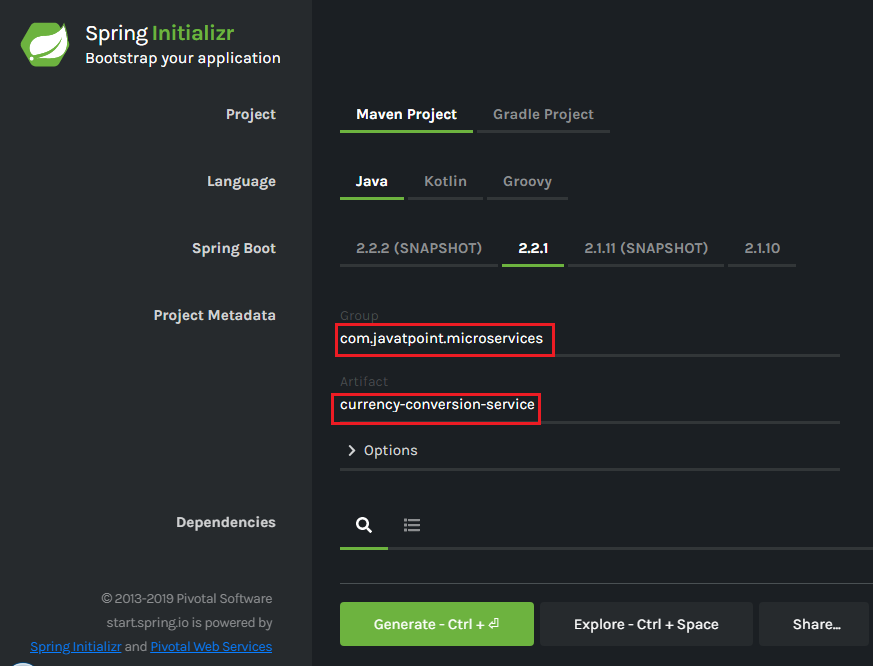
1. Hibernate: select exchangeva0\_.id as id1\_0\_, exchangeva0\_.conversion\_multiple as conversi2\_0\_, exchange.

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.

30.2M

678

Exception Handling in Java - Javatpoint

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**

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

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

Setting 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**

1. **package** com.javatpoint.microservices.currencyconversionservice;
2. **import** java.math.BigDecimal;
3. **import** org.springframework.web.bind.annotation.GetMapping;
4. **import** org.springframework.web.bind.annotation.PathVariable;
5. **import** org.springframework.web.bind.annotation.RestController;
6. @RestController
7. **public** **class** CurrencyConversionController
8. {
9. @GetMapping("/currency-converter/from/{from}/to/{to}/ quantity/{quantity}") //where {from} and {to} represents the column
10. //return a bean back
11. **public** CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity)
12. {
13. **return** **new** CurrencyConversionBean(1L, from,to,BigDecimal.ONE, quantity,quantity,0 );
14. }
15. }

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

1. **private** Long id;
2. **private** String from;
3. **private** String to;
4. **private** BigDecimal ConversionMultiple;
5. **private** BigDecimal quantity;
6. **private** BigDecimal totalCalculatedAmount;
7. **private** **int** port;

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

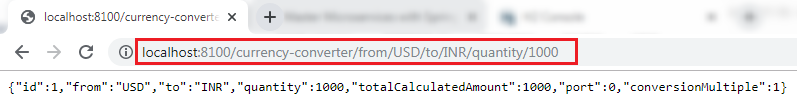
**Step 6:** Generate **constructor** and also create a **default** constructor.

**CurrencyConversionBean.java**

1. **package** com.javatpoint.microservices.currencyconversionservice;
2. **import** java.math.BigDecimal;
3. **public** **class** CurrencyConversionBean
4. {
5. **private** Long id;
6. **private** String from;
7. **private** String to;
8. **private** BigDecimal ConversionMultiple;
9. **private** BigDecimal quantity;
10. **private** BigDecimal totalCalculatedAmount;
11. **private** **int** port;
12. //default constructor
13. **public** CurrencyConversionBean()
14. {
15. }
16. //creating constructor
17. **public** CurrencyConversionBean(Long id, String from, String to, BigDecimal conversionMultiple, BigDecimal quantity, BigDecimal totalCalculatedAmount, **int** port)
18. {
19. **super**();
20. **this**.id = id;
21. **this**.from = from;
22. **this**.to = to;
23. ConversionMultiple = conversionMultiple;
24. **this**.quantity = quantity;
25. **this**.totalCalculatedAmount = totalCalculatedAmount;
26. **this**.port = port;
27. }
28. //creating setters and getters
29. **public** Long getId()
30. {
31. **return** id;
32. }
33. **public** **void** setId(Long id)
34. {
35. **this**.id = id;
36. }
37. **public** String getFrom()
38. {
39. **return** from;
40. }
41. **public** **void** setFrom(String from)
42. {
43. **this**.from = from;
44. }
45. **public** String getTo()
46. {
47. **return** to;
48. }
49. **public** **void** setTo(String to)
50. {
51. **this**.to = to;
52. }
53. **public** BigDecimal getConversionMultiple()
54. {
55. **return** ConversionMultiple;
56. }
57. **public** **void** setConversionMultiple(BigDecimal conversionMultiple)
58. {
59. ConversionMultiple = conversionMultiple;
60. }
61. **public** BigDecimal getQuantity()
62. {
63. **return** quantity;
64. }
65. **public** **void** setQuantity(BigDecimal quantity)
66. {
67. **this**.quantity = quantity;
68. }
69. **public** BigDecimal getTotalCalculatedAmount()
70. {
71. **return** totalCalculatedAmount;
72. }
73. **public** **void** setTotalCalculatedAmount(BigDecimal totalCalculatedAmount)
74. {
75. **this**.totalCalculatedAmount = totalCalculatedAmount;
76. }
77. **public** **int** getPort()
78. {
79. **return** port;
80. }
81. **public** **void** setPort(**int** port)
82. {
83. **this**.port = port;
84. }
85. }

**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.

29.9M

580

Difference between JDK, JRE, and JVM

**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" \t "_blank)

. 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.

1. Map<String, String>uriVariables=**new** HashMap<>();
2. uriVariables.put("from", from);
3. uriVariables.put("to", to);

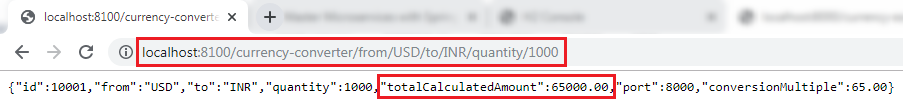
**Step 6:**The response type that we are expecting back is **CurrencyConversionBean,** so store the response in the CurrencyConversionBean.

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.

24M

497

Java Try Catch

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**

1. **package** com.javatpoint.microservices.currencyconversionservice;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.openfeign.EnableFeignClients;
5. @SpringBootApplication
6. @EnableFeignClients("com.javatpoint.microservices.currencyconversionservice")
7. **public** **class** CurrencyConversionServiceApplication
8. {
9. **public** **static** **void** main(String[] args)
10. {
11. SpringApplication.run(CurrencyConversionServiceApplication.**class**, args);
12. }
13. }

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](http://localhost:8000/currency-exchange/from/USD/to/INR" \t "_blank)

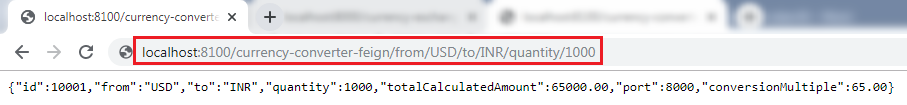
after that run the **currency-conversion-service** by using the URL [http://localhost:8100/currency-converter/from/USD/to/INR/quantity/1000](http://localhost:8100/currency-converter/from/USD/to/INR/quantity/1000" \t "_blank)

.

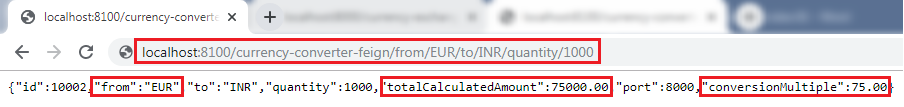
If we do not run the services in the order, the currency-conversion-service shows **Whitelabel Error Page**. It is because the currency-conversion-service uses the conversionMultiple of currency-exchange-service.

**Step 11:**Execute the feign service by using the URL [http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000](http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000" \t "_blank)

. 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.

24M

497

Java Try Catch

**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**

1. **package** com.javatpoint.microservices.currencyconversionservice;
2. **import** org.springframework.cloud.netflix.ribbon.RibbonClient;
3. **import** org.springframework.cloud.openfeign.FeignClient;
4. **import** org.springframework.web.bind.annotation.GetMapping;
5. **import** org.springframework.web.bind.annotation.PathVariable;
6. //@FeignClient(name="currency-exchange-service", url="localhost:8000")
7. //Enabling feign
8. @FeignClient(name="currency-exchange-service")
9. //enabling ribbon
10. @RibbonClient(name="currency-exchange-service")
11. **public** **interface** CurrencyExchangeServiceProxy
12. {
13. @GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable
14. **public** CurrencyConversionBean retrieveExchangeValue(@PathVariable("from") String from, @PathVariable("to") String to); //from map to USD and to map to INR
15. }

**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**

1. spring.application.name=currency-conversion-service
2. server.port=8100
3. currency-exchange-service.ribbon.listOfServers=http://localhost:8000, http://localhost:8001

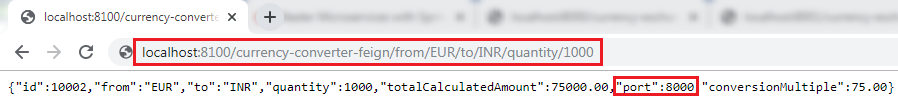
Running Client Side Load Balancing with Ribbon

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

Client-Side Load Balancing with Ribbon

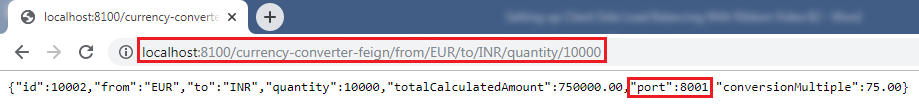
First, run the CurrencyExchangeServiceApplication on port **8000** and then run the CurrencyExchangeServiceApplication on port **8001**.

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.



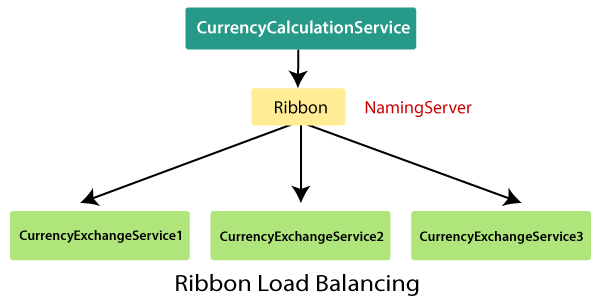
In the above image, the port 8000 represents that the currency-exchange-service is running on port 8000 and handling the current request.

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.

2.2M

Competitive questions on Structures

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

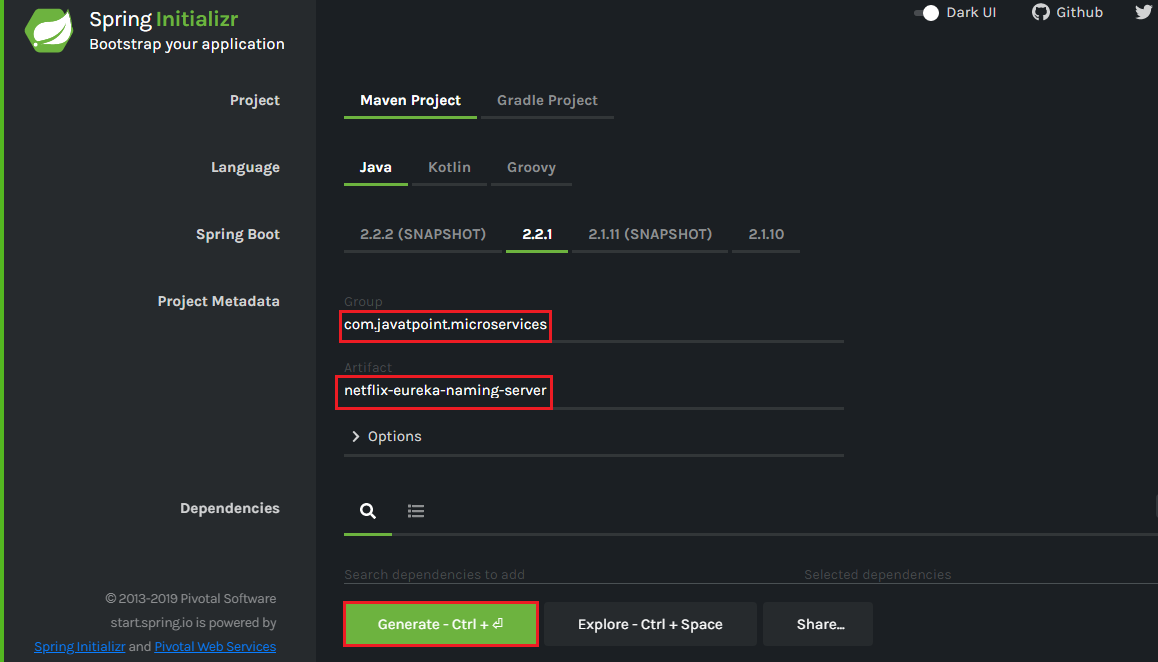
Once the instances of currency-exchange-service are registered with the Eureka naming server, then we will use the Ribbon to find the detail from the naming server. Let’s follow the steps specified above:

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

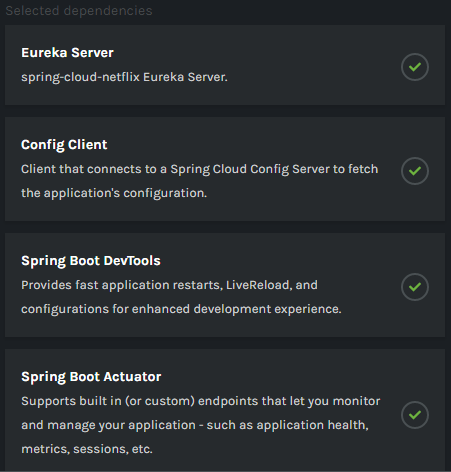
**Step 1:**Open Spring Initializr [https://start.spring.io](https://start.spring.io/).

**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.**



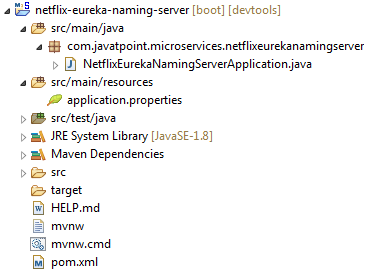
**Step 4:**Add the following dependencies: **Eureka Server, Config Client, Actuator,**and**DevTools.**



**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. }

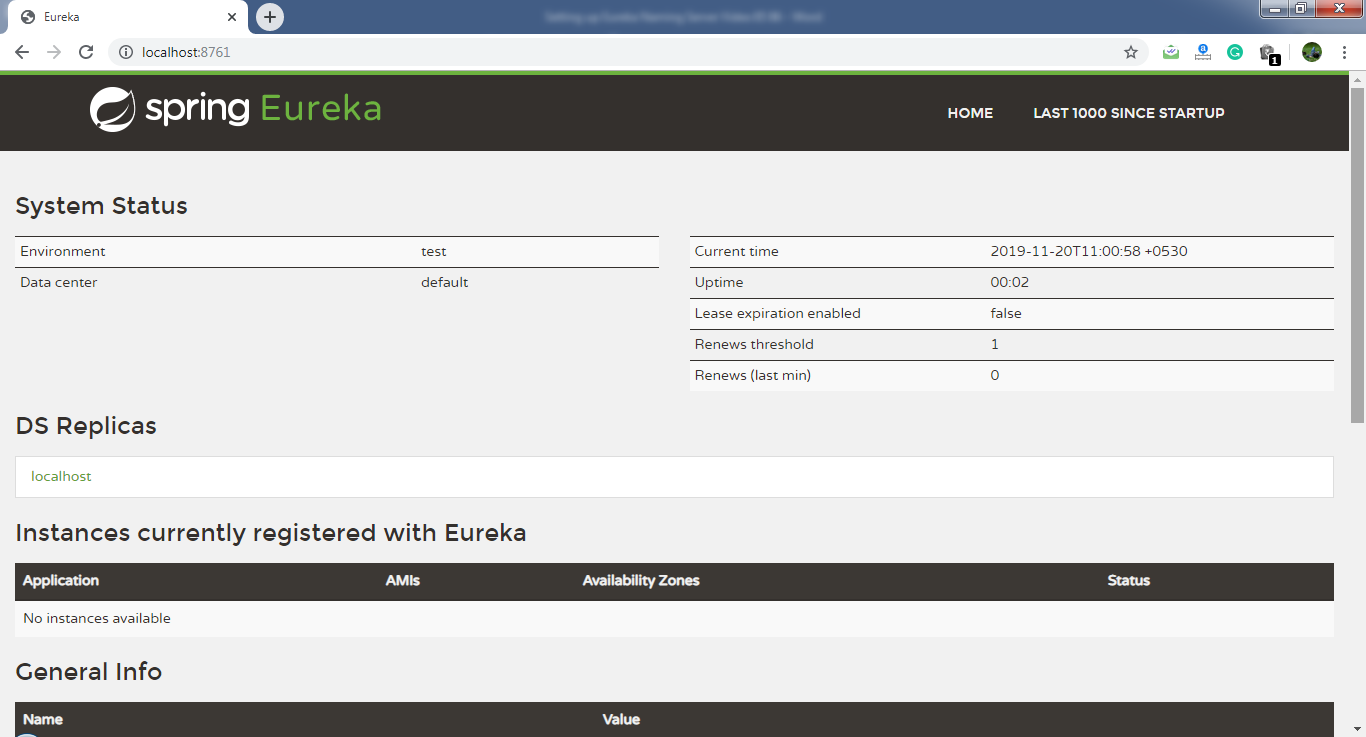
**Step 8:**Open the **application.properties** file and configure the **application name, port,**and**eureka server**.

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**.**

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

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

**CurrencyConversionServiceApplication.java**

1. **package** com.javatpoint.microservices.currencyconversionservice;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;
5. **import** org.springframework.cloud.openfeign.EnableFeignClients;
6. @SpringBootApplication
7. @EnableFeignClients("com.javatpoint.microservices.currencyconversionservice")
8. @EnableDiscoveryClient
9. **public** **class** CurrencyConversionServiceApplication
10. {
11. **public** **static** **void** main(String[] args)
12. {
13. SpringApplication.run(CurrencyConversionServiceApplication.**class**, args);
14. }
15. }

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**

1. spring.application.name=currency-conversion-service
2. server.port=8100
3. eureka.client.service-url.**default**-zone=http://localhost:8761/eureka
4. 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**.**

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

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

**CurrencyExchangeServiceApplication.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;
5. @SpringBootApplication
6. @EnableDiscoveryClient
7. **public** **class** CurrencyExchangeServiceApplication
8. {
9. **public** **static** **void** main(String[] args)
10. {
11. SpringApplication.run(CurrencyExchangeServiceApplication.**class**, args);
12. }
13. }

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

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

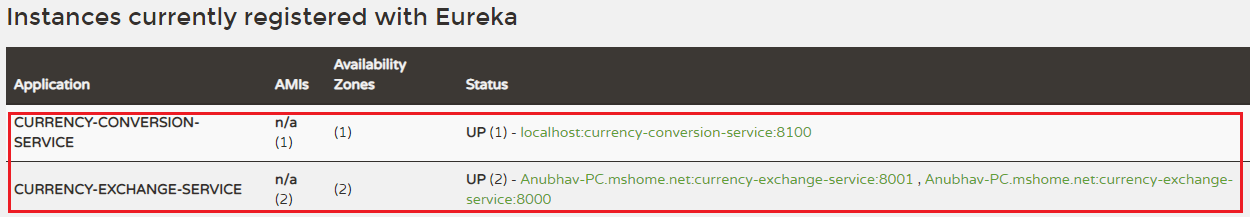
**Step 5:**Run the two instances of **CurrencyExchangeServiceApplication.java.** First instance on port **8000** and the second instance on port **8001**.

**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**.

Competitive questions on Structures in Hindi

Keep Watching

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**

1. spring.application.name=currency-conversion-service
2. server.port=8100
3. eureka.client.service-url.**default**-zone=http://localhost:8761/eureka
4. #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**.

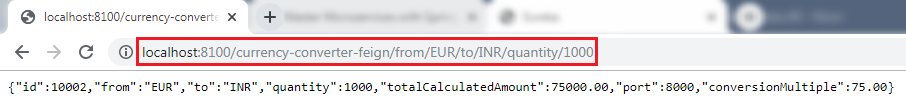
**Step 4:** Open the browser and refresh the Eureka server.

**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.

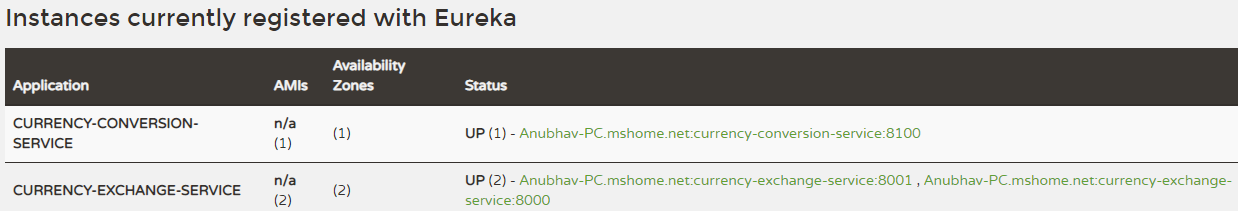
**Step 8:** Open the browser and type the URL <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000>.



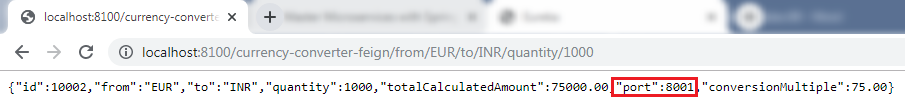
When we refresh the Eureka server, it always returns the port 8000 in the response.

**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**.

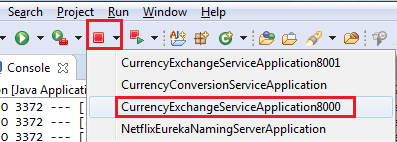


**Step 11:**Refresh the URL <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/1000>.



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:

29.6M

678

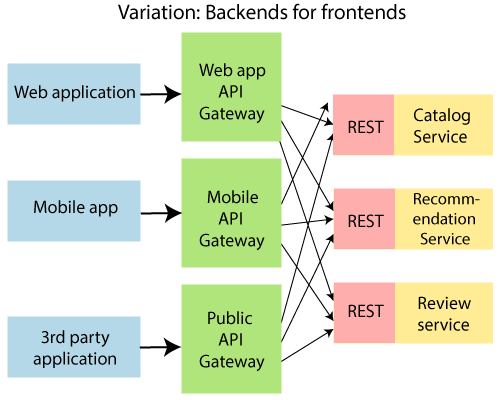
Features of Java - Javatpoint

* 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.

24M

498

Java Try Catch

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

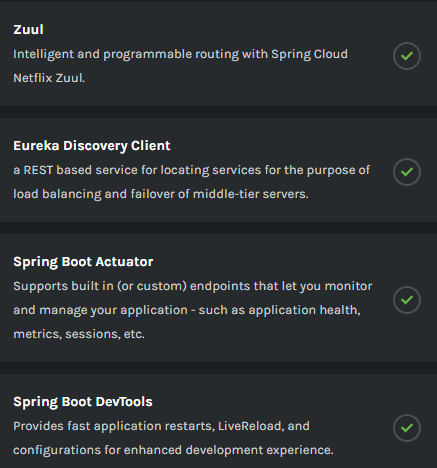
Follow the steps to set up the Zuul API Gateway server.

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

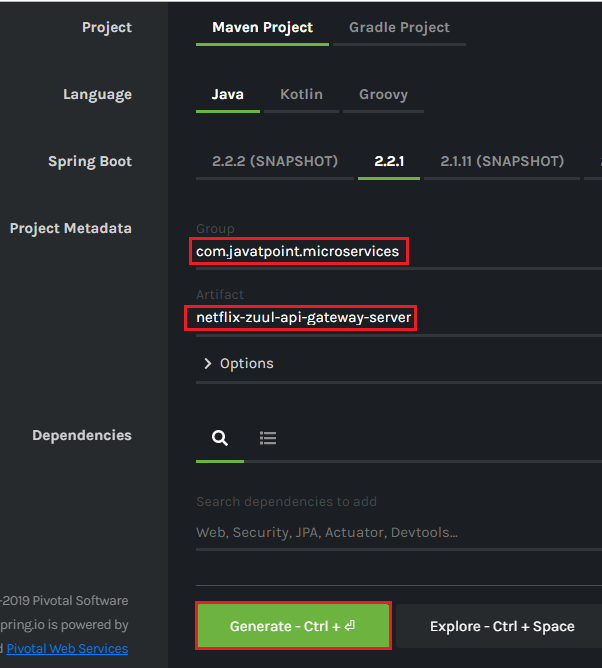
**Step 2:**Provide the **Group** name. We have provided**com.javatpoint.microservices.**

**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**

1. **package** com.javatpoint.microservices.netflixzuulapigatewayserver;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;
5. **import** org.springframework.cloud.netflix.zuul.EnableZuulProxy;
6. @EnableZuulProxy
7. @EnableDiscoveryClient
8. @SpringBootApplication
9. **public** **class** NetflixZuulApiGatewayServerApplication
10. {
11. **public** **static** **void** main(String[] args)
12. {
13. SpringApplication.run(NetflixZuulApiGatewayServerApplication.**class**, args);
14. }
15. }

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

**application.properties**

1. spring.application.name=netflix-zuul-api-gateway-server
2. server.port=8765
3. 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.**

10 Sec

10.6M

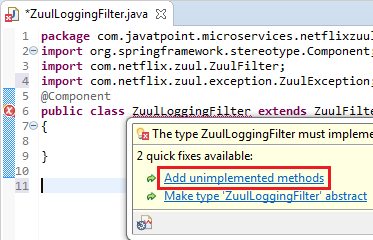
155

SQL CREATE TABLE

**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.

1. **public** Object run() **throws** ZuulException
2. {
3. //getting the current HTTP request that is to be handle
4. HttpServletRequest request=RequestContext.getCurrentContext().getRequest();
5. //printing the detail of the request
6. logger.info("request -> {} request uri-> {}", request, request.getRequestURI());
7. **return** **null**;
8. }

**ZuulLoggingFilter.java**

1. **package** com.javatpoint.microservices.netflixzuulapigatewayserver;
2. **import** javax.servlet.http.HttpServletRequest;
3. **import** org.slf4j.Logger;
4. **import** org.slf4j.LoggerFactory;
5. **import** org.springframework.stereotype.Component;
6. **import** com.netflix.zuul.ZuulFilter;
7. **import** com.netflix.zuul.context.RequestContext;
8. **import** com.netflix.zuul.exception.ZuulException;
9. @Component
10. **public** **class** ZuulLoggingFilter **extends** ZuulFilter
11. {
12. //creating Logger object
13. **private** Logger logger=LoggerFactory.getLogger(**this**.getClass());
14. @Override
15. **public** booleanshouldFilter()
16. {
17. **return** **true**; //executing filter for every request
18. }
19. //log the content of the request
20. @Override
21. **public** Object run() **throws** ZuulException
22. {
23. //getting the current HTTP request that is to be handle
24. HttpServletRequest request=RequestContext.getCurrentContext().getRequest();
25. //prints the detail of the requestin the log
26. logger.info("request -> {} request uri-> {}", request, request.getRequestURI());
27. **return** **null**;
28. }
29. @Override
30. **public** String filterType()
31. {
32. **return** "pre"; //intercept all the request before execution
33. }
34. @Override
35. **public** intfilterOrder()
36. {
37. **return** 1; //setting filter order to 1
38. }
39. }

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.**

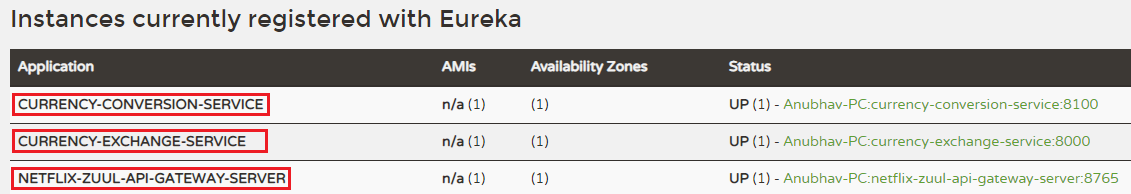
20.4M

387

Prime Ministers of India | List of Prime Minister of India (1947-2020)

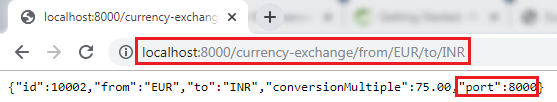
**Step 5:**Open the browser and invoke the URL [http://localhost:8761](http://localhost:8761/" \t "_blank)

. 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](http://localhost:8000/currency-exchange/from/EUR/to/INR" \t "_blank)

. We get the response, but the request does not go through the Zuul API Gateway.



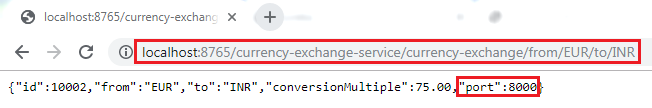
Let's invoke the request through the Zuul API Gateway. We use the following URL: [http://localhost:8765/{application-name}/{uri}](http://localhost:8765/%7bapplication-name%7d/%7buri%7d)

. 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](http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR" \t "_blank)

**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.

Executing 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](http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR" \t "_blank)

, 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}")

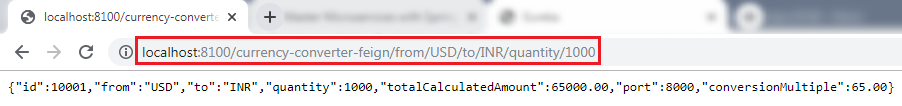
**Remember:** Remove or comment the mapping for currency-exchange-service.

**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.

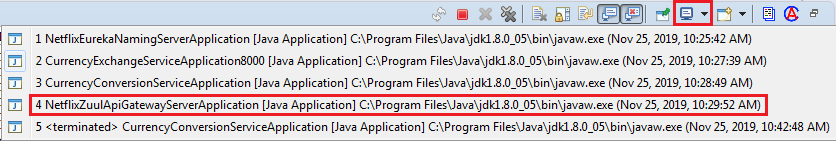
**Step 6:**Open the browser and invoke the URL [http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000](http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000" \t "_blank)

. It returns the following response:



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

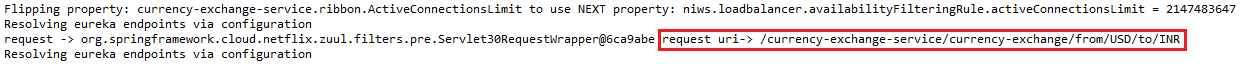
**Step 7:**Click on the arrow that is beside the console icon and select the **NetflixZullApiGatewayServerApplication.**



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

Executing 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/" \t "_blank)

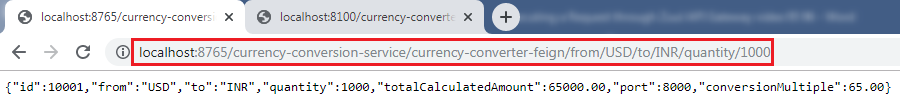
through the API Gateway. The URI will be **/{application-name}/{uri}.** The complete URL will look like the following:

[http://localhost:8765/currency-conversion-service/currency-converter-feign/from/USD/to/INR/quantity/100](http://localhost:8765/currency-conversion-service/currency-converter-feign/from/USD/to/INR/quantity/100" \t "_blank)

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](http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000" \t "_blank)

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**.

Executing 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.

24.5M

530

How to find Nth Highest Salary in SQL

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.

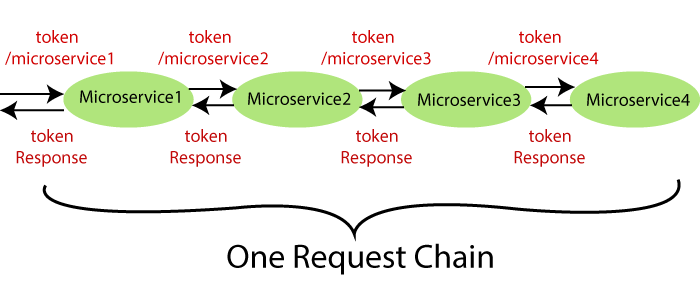
Zipkin

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

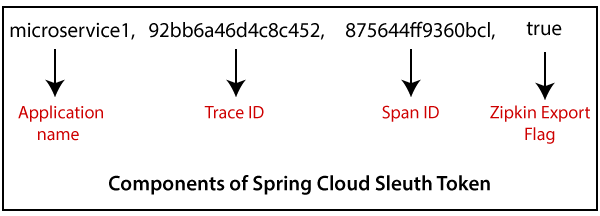
In this step, we will add Spring Cloud Sleuth for all the microservices. It adds a unique Id to all the requests. It is used to generate and attach the **trace Id**, **span Id** to the logs so that tools (Zipkin) can use these ids.



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.**

1. @Bean
2. **public** Sampler defaultSampler()
3. {
4. **return** Sampler.ALWAYS\_SAMPLE;
5. }

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

**NetflixZuulApiGatewayServerApplication.java**

1. **package** com.javatpoint.microservices.netflixzuulapigatewayserver;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;
5. **import** org.springframework.cloud.netflix.zuul.EnableZuulProxy;
6. **import** org.springframework.context.annotation.Bean;
7. **import** brave.sampler.Sampler;
8. @SpringBootApplication
9. @EnableDiscoveryClient
10. @EnableZuulProxy
11. **public** **class** NetflixZuulApiGatewayServerApplication
12. {
13. **public** **static** **void** main(String[] args)
14. {
15. SpringApplication.run(NetflixZuulApiGatewayServerApplication.**class**, args);
16. }
17. //creating a bean
18. @Bean
19. //creating a sampler called
20. **public** Sampler defaultSampler()
21. {
22. **return** Sampler.ALWAYS\_SAMPLE;
23. }
24. }

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.

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

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

**CurrencyExchangeServiceApplication.java**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;
5. **import** org.springframework.context.annotation.Bean;
6. **import** brave.sampler.Sampler;
7. @SpringBootApplication
8. @EnableDiscoveryClient
9. **public** **class** CurrencyExchangeServiceApplication
10. {
11. **public** **static** **void** main(String[] args)
12. {
13. SpringApplication.run(CurrencyExchangeServiceApplication.**class**, args);
14. }
15. @Bean
16. //creating a sampler called always sampler
17. **public** Sampler defaultSampler()
18. {
19. **return** Sampler.ALWAYS\_SAMPLE;
20. }
21. }

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

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

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

**CurrencyConversionServiceApplication.java**

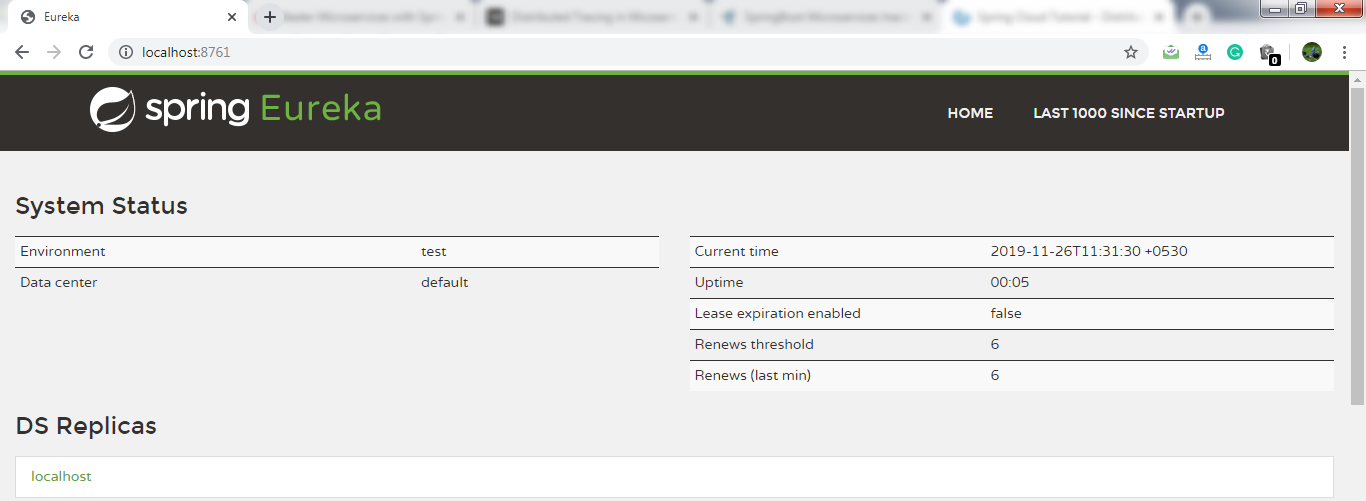
1. **package** com.javatpoint.microservices.currencyconversionservice;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.cloud.client.discovery.EnableDiscoveryClient;
5. **import** org.springframework.cloud.openfeign.EnableFeignClients;
6. **import** org.springframework.context.annotation.Bean;
7. **import** brave.sampler.Sampler;
8. @SpringBootApplication
9. @EnableFeignClients("com.javatpoint.microservices.currencyconversionservice")
10. @EnableDiscoveryClient
11. **public** **class** CurrencyConversionServiceApplication
12. {
13. **public** **static** **void** main(String[] args)
14. {
15. SpringApplication.run(CurrencyConversionServiceApplication.**class**, args);
16. }
17. @Bean
18. //creating a sampler called always sampler
19. **public** Sampler defaultSampler()
20. {
21. **return** Sampler.ALWAYS\_SAMPLE;
22. }
23. }

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.



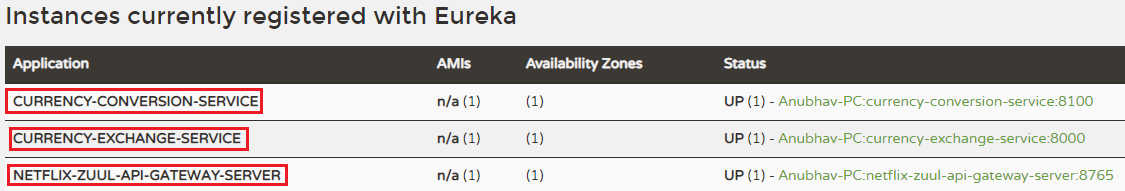
**currency-exchange-service**(on port 8000)

**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**

1. **package** com.javatpoint.microservices.currencyexchangeservice;
2. **import** org.slf4j.Logger;
3. **import** org.slf4j.LoggerFactory;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
6. **import** org.springframework.core.env.Environment;
7. **import** org.springframework.web.bind.annotation.GetMapping;
8. **import** org.springframework.web.bind.annotation.PathVariable;
9. **import** org.springframework.web.bind.annotation.RestController;
10. @SpringBootApplication
11. @RestController
12. **public** **class** CurrencyExchangeController
13. {
14. **private** Logger logger=LoggerFactory.getLogger(**this**.getClass());
15. @Autowired
16. **private** Environment environment;
17. @Autowired
18. **private** ExchangeValueRepository repository;
19. @GetMapping("/currency-exchange/from/{from}/to/{to}")       //where {from} and {to} are path variable
20. **public** ExchangeValue retrieveExchangeValue(@PathVariable String from, @PathVariable String to)   //from map to USD and to map to INR
21. {
22. ExchangeValue exchangeValue = repository.findByFromAndTo(from, to);
23. //setting the port
24. exchangeValue.setPort(Integer.parseInt(environment.getProperty("local.server.port")));
25. logger.info("{}", exchangeValue);
26. **return** exchangeValue;
27. }
28. }

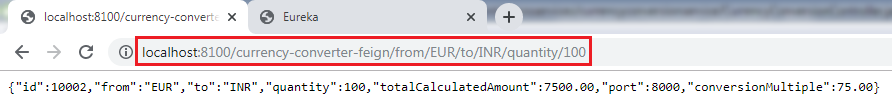
Similarly, we will add logger into CurrencyConversionContoller.

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

**CurrencyConversionContoller**.**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.slf4j.Logger;
6. **import** org.slf4j.LoggerFactory;
7. **import** org.springframework.beans.factory.annotation.Autowired;
8. **import** org.springframework.http.ResponseEntity;
9. **import** org.springframework.web.bind.annotation.GetMapping;
10. **import** org.springframework.web.bind.annotation.PathVariable;
11. **import** org.springframework.web.bind.annotation.RestController;
12. **import** org.springframework.web.client.RestTemplate;
13. @RestController
14. **public** **class** CurrencyConversionController
15. {
16. **private** Logger logger=LoggerFactory.getLogger(**this**.getClass());
17. @Autowired
18. **private** CurrencyExchangeServiceProxy proxy;
19. @GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}") //where {from} and {to} represents the column
20. //returns a bean back
21. **public** CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity)
22. {
23. //setting variables to currency exchange service
24. Map<String, String> uriVariables=**new** HashMap<>();
25. uriVariables.put("from", from);
26. uriVariables.put("to", to);
27. //calling the currency exchange service
28. ResponseEntity<CurrencyConversionBean> responseEntity=**new** RestTemplate().getForEntity("http://localhost:8000/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.**class**, uriVariables);
29. CurrencyConversionBean response=responseEntity.getBody();
30. //creating a new response bean and getting the response back and taking it into Bean
31. **return** **new** CurrencyConversionBean(response.getId(), from, to, response.getConversionMultiple(), quantity, quantity.multiply(response.getConversionMultiple()), response.getPort());
32. }
33. //mapping for currency-converter-feign service
34. @GetMapping("/currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}") //where {from} and {to} represents the column
35. //returns a bean
36. **public** CurrencyConversionBean convertCurrencyFeign(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity)
37. {
38. CurrencyConversionBean response=proxy.retrieveExchangeValue(from, to);
39. logger.info("{}", response);
40. //creating a new response bean
41. //getting the response back and taking it into Bean
42. **return** **new** CurrencyConversionBean(response.getId(), from, to, response.getConversionMultiple(), quantity, quantity.multiply(response.getConversionMultiple()), response.getPort());
43. }
44. }

**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:

Introduction to Distributed Tracing

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

Introduction to Distributed Tracing

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

Introduction 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

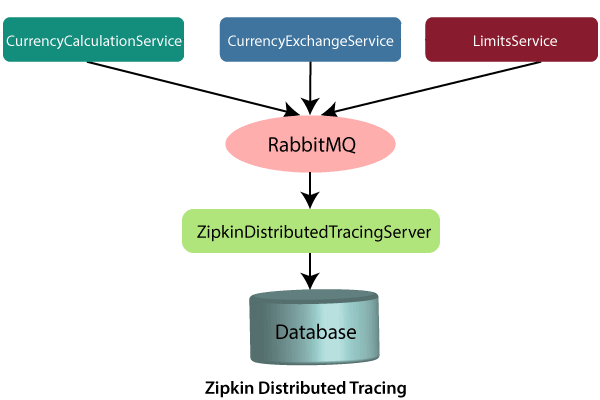
We have installed Spring Cloud Sleuth dependency in currency-conversion-service, currency-exchange-service, and netflix-zuul-api-gateway-server.  We have seen that a unique Id was being assigned to each request. We use these Id to trace the request across the logs of these multiple services.

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.

24M

498

Java Try Catch

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](https://erlang.org/download/otp_win64_22.1.exe" \t "_blank)

.

**Step 2:** Download and install **RabbitMQ** from [https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.8.1/rabbitmq-server-3.8.1.exe](https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.8.1/rabbitmq-server-3.8.1.exe" \t "_blank)

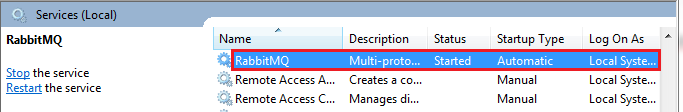
.

**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

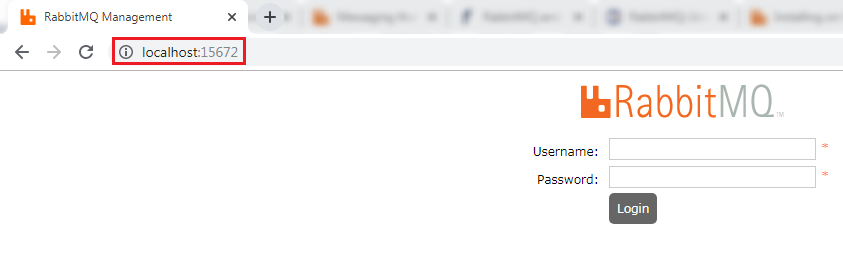
**Step 4:** Press the Windows key and type **services**or press **Windows key+R**and type **services.msc**.

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



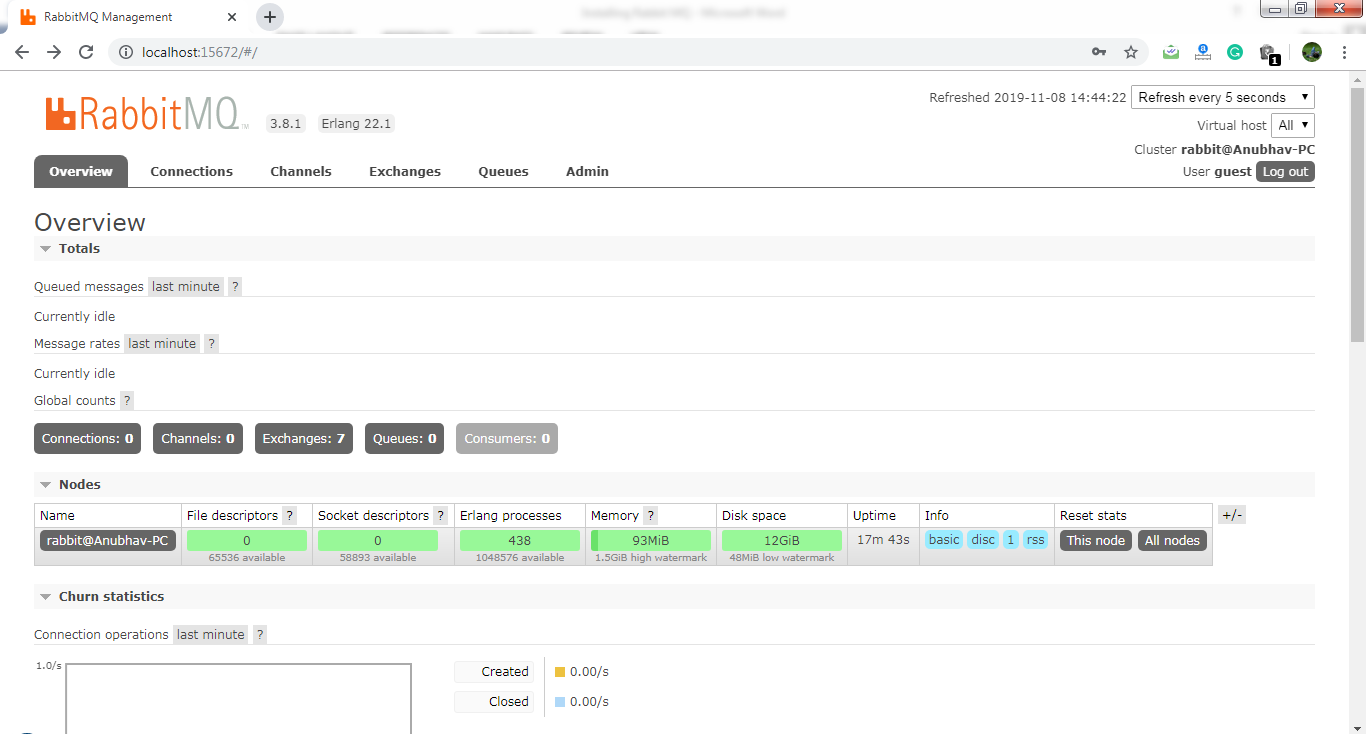
**Step 6:** Open the browser and type [http://localhost:15672](http://localhost:15672/)

. 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.



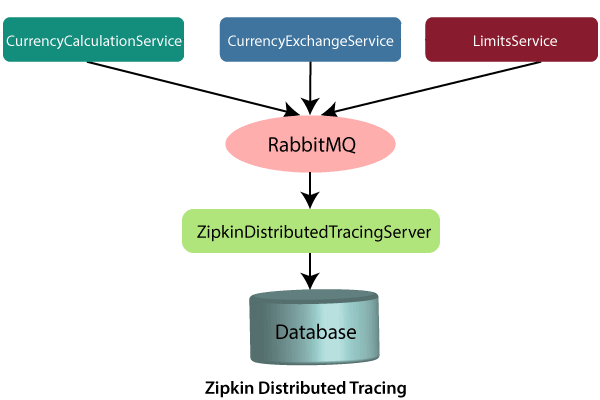
Distributed tracing using Zipkin

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.

30.2M

678

Exception Handling in Java - Javatpoint

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.

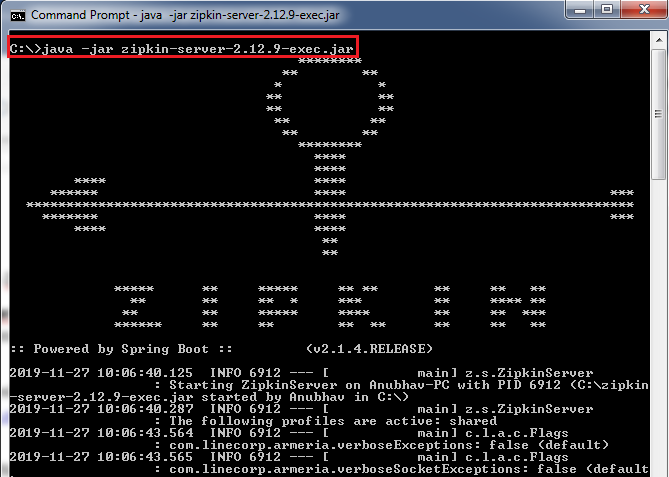
**Step 2:**Download the **zipkin-server-2.12.9-exec.jar** file from <https://search.maven.org/remote_content?g=io.zipkin&a=zipkinserver&v=LATEST&c=exec>.

**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:

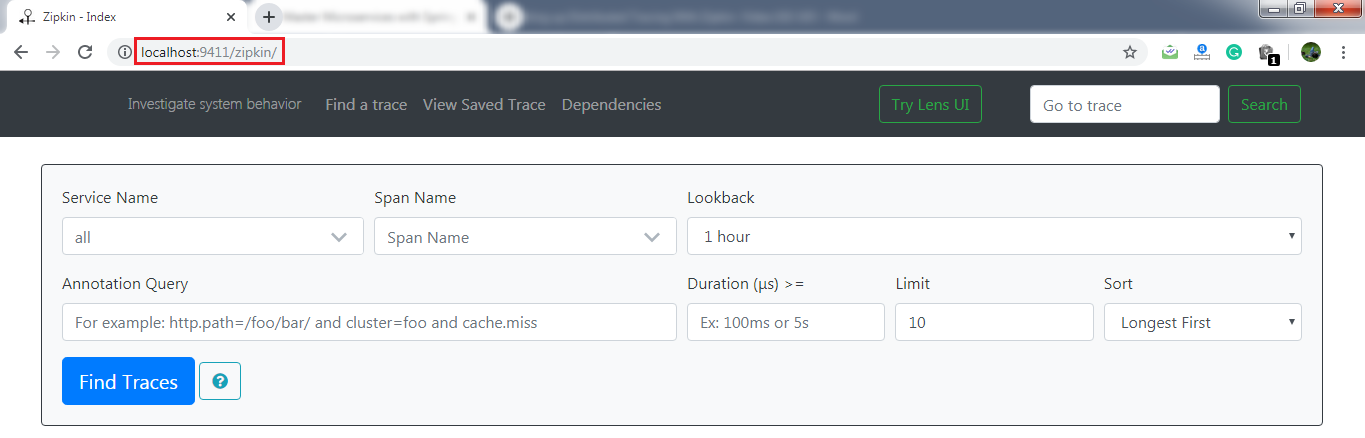
1. C:\> java -jar zipkin-server-2.12.9-exec.jar

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.



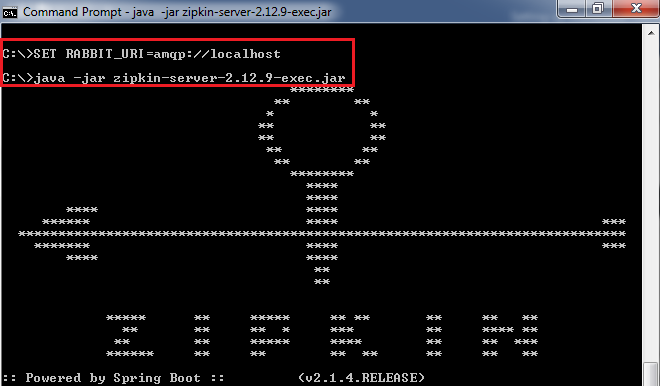
Here, the most important thing is that the Zipkin server must listen over the RabbitMQ server. So we have to start the RabbitMQ server in the background.

**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.

10 Sec

Triggers in SQL (Hindi)



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.

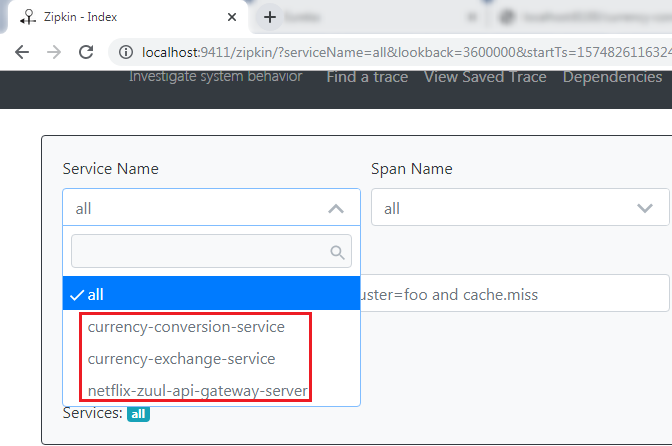
* **NetflixEurekaNamingServerApplication.java**
* **ZipkinDistributedTracingServerApplication**(run from the Command Prompt)
* **CurrencyExchangeServiceApplication.java**(on port 8000)
* **CurrencyConversionServiceApplication.java**(on port 8100)
* **NetflixZuulApiGatewayServerApplication.java**

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

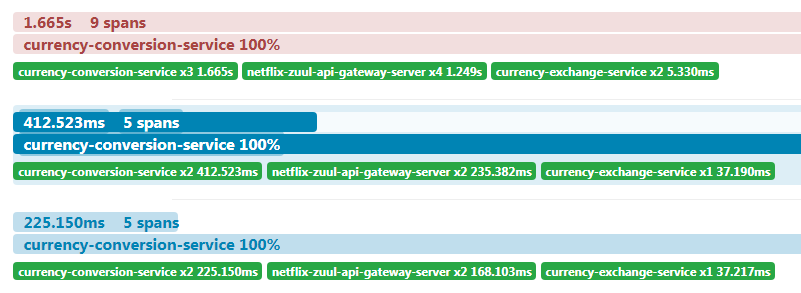
**Step 2:**Invoke the request <http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/100>. It returns the response properly.

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

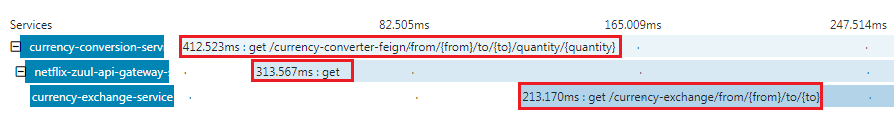
**Step 3:**Open the Zipkin UI. It shows all the three services that we have connected to Zipkin.



**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.**

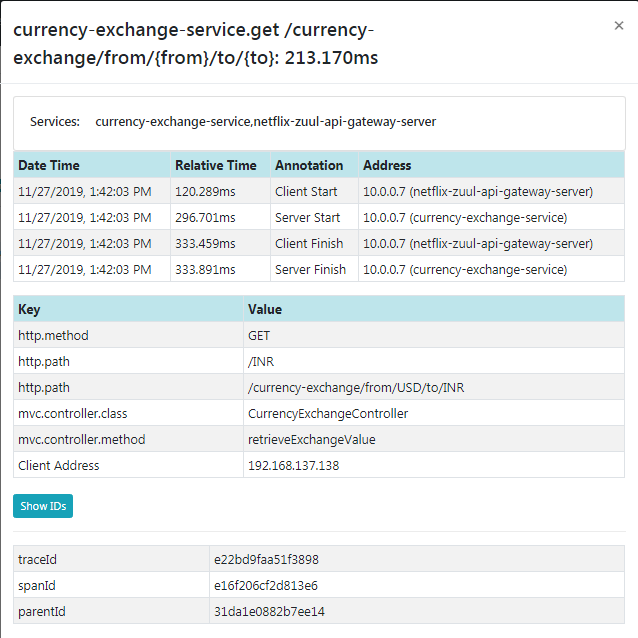


**Step 5:** Select any one of them. It shows the entire trace of a service.



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}

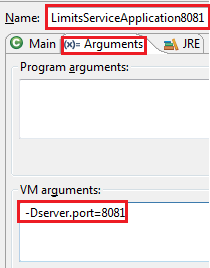
We are getting these values from the **limits-service-qa.properties** file, because we have configured this file into the **bootstrap.properties** file of the **limits-service**.

Hello Java Program for Beginners

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**.

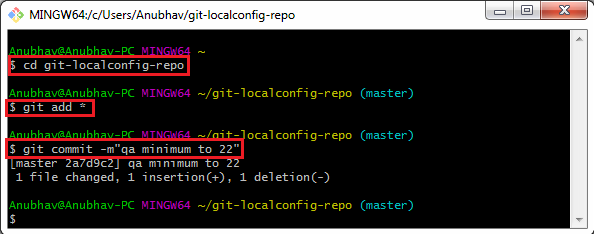
**limits-service-qa.properties**

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.**

History of Java

**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

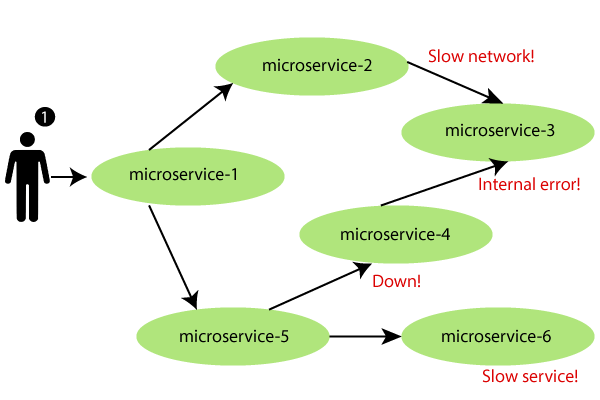
Microservices must be extremely reliable because they depend on each other. The microservice architecture contains a large number of small microservices. These microservices communicate with each other in order to fulfill their requirements.

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**.



**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.

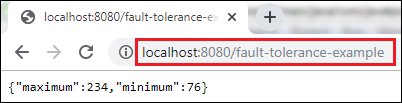
In the above method, we have created a Get mapping for fault tolerance. In the next line, we have used an annotation **@HystrixCommand** to configure the **fallback** method. We have defined a method with the name **fallbackRetrieveConfigurations()** that returns the default value if any fault occurs.

**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.



# Most Asked Microservices Interview Questions

Following is the list of most frequently asked Microservices Interview questions and their best possible answers.

### **1) What are Microservices? / What do you understand by Microservices?**

Microservices are an architectural approach or style that is used to build applications. The microservice architecture provides a rapid, frequent and reliable delivery of large and complex applications. It is distributed and loosely coupled, so it won't break the entire app if you make changes in one team.

Microservices are also known as the microservice architecture, a variant of the service-oriented architecture (SOA) structural style and used to structure an application as a collection of services that have the following features:

* Distributed and loosely coupled
* Highly maintainable and testable
* Independently deployable
* Organized around business capabilities
* Owned by a small team

### **2) What are the most significant benefits of using microservices?**

The most significant benefit of using microservices is that it builds an application to collect small autonomous services developed for a business domain. So, if the business needs to change constantly, the development teams can rapidly build new apps components to meet the requirement.

Each microservice runs a unique process and communicates through a well-defined, lightweight mechanism, such as a container, to serve a business goal. It also makes an organization capable of evolving its technology stack.

**3) What are the three commonly used tools for Microservices?**

Following are the three commonly used tools for Microservices:

* Wiremock
* Docker
* Hysrix

### **4) What are the main components of Microservices?**

Following is the list of main components of Microservices or Microservice architecture:

* Containers, Clustering, and Orchestration
* IaC (Infrastructure as Code Conception)
* Cloud Infrastructure
* API Gateway
* Enterprise Service Bus
* Service Delivery

### **5) How does a Microservice architecture work?**

The Microservice architecture of an application can be simplified into multiple modules that independently perform the single precise standalone task. Let's see how Microservice architecture works:

* An application is fragmented into loosely coupled various modules, each of which performs a distinct function.
* It is distributed across clouds and data centers.
* Each application module is an independent service/process that can be replaced, updated, or deleted without disrupting the rest of the application.
* Under microservice architecture, an application can grow along with its requirements.

### **6) What are the main advantages of using Microservices?**

Following is a list of some most important advantages of using Microservices:

* Microservices provide great technology diversity. You can mix it easily with other frameworks, libraries, and databases.
* Microservices support fault isolation as it collects small autonomous services or processes, so a process failure should not bring the whole system down.
* It provides excellent support for the minor and parallel team.
* It reduces the deployment time significantly.
* Independent deployment

### **7) What do you understand by Monolithic Architecture?**

Monolithic architecture is like a big container that contains all the software components of an application. These applications are clubbed inside a single package within the application.

### **8) What are the biggest challenges in Microservice deployment?**

We can specify the biggest challenges in Microservice deployment in two ways, i.e., technical and functional.

**The main challenges from the business point of view:**

* Microservices require a heavy investment.
* It requires a heavy infrastructure setup also.
* We need excessive planning for managing operations overhead.
* It cost a lot in staff selection and maintenance.

**The main challenges from a technical point of view:**

* The components in microservices always rely on each other, so; it requires communication between them in the application.
* There are also a lot of challenges in deployment.
* Testing and Debugging are very challenging.
* It requires complete component automation and application maintenance.
* It receives heavy operations overhead.
* It requires skilled professionals to support heterogeneously distributed microservices.

### **9) What do you understand by Spring Cloud?**

Spring cloud is an Integration software used to integrate with external systems. It allows a microservices framework to build applications that perform restricted amounts of data processing.

### **10) What are the main differences between Microservices and Monolithic Architecture?**

The main differences between Microservices and Monolithic Architecture:

|  |  |
| --- | --- |
| **Microservices** | **Monolithic Architecture** |
| The service startup is fast in Microservices. | The service startup takes time as it is slow in Monolithic Architecture. |
| It is a loosely coupled architecture. | It is primarily a tightly coupled architecture. |
| In Microservices, if you make changes in a single data model, it does not affect others. | In Monolithic Architecture, any changes in the data model affect the entire database. |
| It mainly focuses on products, not projects. | It mainly focuses on the whole project. |

### **11) In which cases microservice architecture is best suited?**

The microservice architecture is best suited for all tech devices such as desktop, web, mobile devices, Smart TVs, Wearable devices, etc.

### **12) What are the most significant advantages and disadvantages of using Microservices?**

Following is the list of the most significant advantages and disadvantages of using Microservices:

**Advantages of Microservices**

* Provide improved scalability
* Increased Agility
* Localized Complexity
* Provide fault isolation
* Debugging & Maintenance are easy and simplified.
* Communication between developers with business users is accessible and better.
* Smaller development teams
* You can easily upgrade the technology.

**Disadvantages of Microservices**

* As a whole project, it isn't easy because it uses multiple components in the application.
* It requires accurate pre-planning before use.
* It uses modular dependencies that are hard to calculate.
* The third-party applications are hard to control.
* Modular interdependencies are challenging to track.
* More opportunities for malicious intrusions.
* Complete end-to-end testing is complex.
* Deployment Challenges.

### **13) Which are some famous companies that are using Microservice architecture?**

Most large-scale software companies and websites such as Twitter, Netflix, Amazon are using microservices architecture instead of monolithic architecture.

### **14) What do you understand by RESTful?**

REST or RESTful stands for Representational State Transfer. The RESTful web service is an architectural style that helps computer systems to communicate over the internet. These web services make microservices easier to understand and implement.

### **15) What are the different strategies used in Microservices deployment?**

Following strategies are used in Microservices deployment:

* **Multiple Service Instance per Host:** It is used to run single or multiple service instances of the application on a single or multiple physical/virtual hosts.
* **Service Instance per Host:** It is used to run a service instance per host.
* Service Instance per Container: It is used to run each service instance in its respective container.
* **Serverless Deployment:** It packages the service as a ZIP file and uploads it to the Lambda function. The Lambda function is a stateless service that automatically runs enough micro-services to handle all requests.

### **16) What are the three types of tests used in Microservices?**

We can categorize the tests used in Microservice architecture into three main categories:

* **Bottom Level Test:** The bottom-level tests perform general tests such as performance tests and unit tests. These kinds of tests are entirely automated.
* **Middle-Level Tests:** The middle-level tests are used to perform exploratory tests such as the stress test and usability test.
* **Top Level Tests:** The top-level tests are used to conduct acceptance tests, mostly fewer in numbers. These types of tests make stakeholders know about different software features.

### **17) What is the main difference between SOA and the Microservices Architecture?**

SOA stands for Service Oriented Architecture. It is a collection of services used to communicate with each other through simple data passing or activity coordination. On the other hand, the Microservices Architecture is a collection of small functional modules that are independently deployable, scalable, target specific business goals, and communicate over standard protocols.

### **18) What is a Client certificate? What is its usage?**

A client certificate is a digital certificate used to make authenticated requests to a remote server. A certificate is generated for each microservice.

### **19) What do you understand by Domain-Driven Design?**

Domain-Driven Design is an architectural style based on Object-Oriented Analysis Design concepts and principles. It is used to develop a complex system by connecting the related components of the software system into a continuously evolving system. Domain-Driven Design is based on three principles:

* Focus on the core domain and domain logic.
* Base complex designs on models of the domain.
* Collaborate with the domain experts to improve the application model and resolve any emerging domain-related issues regularly.

### **20) What is the use of PACT in Microservices architecture?**

PACT is an open-source tool used for testing interactions between service providers and consumers. It increases the reliability of the Microservices applications.

### **21) What do you understand by OAuth?**

OAuth stands for Open Authorization protocol. This protocol allows you to access the client applications on HTTP for third-party providers GitHub, Facebook, etc. It also facilitates us to share resources stored on one site with another site without their credentials.

### **22) What is Spring Boot? Why is it used?**

Spring Boot is an open-source, Java-based framework that provides developers an excellent platform for developing a stand-alone and production-grade spring application. It is easy to understand, reduces development time, and increases productivity. It automatically configures a claim based on the added dependencies of an application.

### **23) What is the method to override a Spring Boot project's default properties?**

We can do it by specifying the properties in application.properties. The Spring MVC applications need the suffix and the prefix to be specified. This can be done by:

* **For suffix:**mvc.view.suffix: .jsp
* **For prefix:**mvc.view.prefix: /WEB-INF/

### **24) What do you understand by end-to-end Microservices testing?**

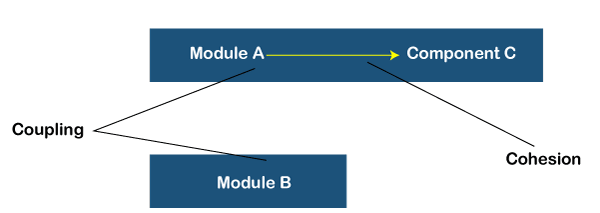
End-to-end testing is used to validate that every process in the workflow is functioning correctly. It also ensures that the system works together as a whole and fulfills all the requirements.

### **25) What is the difference between Coupling and Cohesion?**

**Difference between Coupling and Cohesion**

**Coupling:** Coupling is specified as a relationship between module A and another module B. There are mainly three types of coupling. Any module can be highly coupled (highly dependent), loosely coupled, and uncoupled with other modules. The best coupling is the loose coupling which can be achieved through interfaces.

**Cohesion:** Cohesion is the relationship between 2 or more parts within a module. The high cohesion within a module specifies that the module can perform a specific task with maximum efficiency on its own, without the need to communicate with other modules. High cohesion increases the functional strength of the module.



### **26) What is the use of containers in Microservices?**

Containers are the easiest and effective method to manage microservice-based applications. They are like a software development platform. They also help us to develop and deploy individually. A Docker is an example of a container. It is an open-source software development platform that allows us to encapsulate our microservice in a container image along with its dependencies. Microservice can use these elements without additional efforts.

### **27) What is Spring Cloud? What problems can be solved by using Spring Cloud?**

Spring Cloud is a collection of tools used by developers to quickly build some of the common patterns in distributed systems such as configuration management, circuit breakers, service discovery, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state, etc.

We can use spring cloud to solve the following problems:

* We can solve network issues, latency overhead, bandwidth issues, security issues, and other issues occurred in distributed systems.
* We can also solve redundancy issues that occur in distributed systems.
* We can balance the distribution of load between resources like network links, CPU, clusters, etc.
* We can solve the performance issues that occurred because of operational overheads.
* We can resolve the service discovery issues to make smooth communication possible between services in a cluster.

### **28) What do you understand by semantic monitoring in Microservices architecture?**

Semantic monitoring is used to combine the automated tests by monitoring the application. It is used to find out the reasons why your business is not getting more profits.

### **29) What do you understand by the Distributed Transaction?**

A distribution transaction is a type of transaction that has two or more engaged network hosts. In this transaction, a transaction manager takes care of developing and handling transactions. If the transaction involves more than one peer, the transaction managers of each peer communicate with each other using subordinate or superior relationships. In the same way, the resource manager handles the resources and coordinates with the distributed transaction coordinator for transaction atomicity and isolation.

### **30) What is the full form of CDC? What is its usage?**

The full form of CDC is a Consumer-Driven Contract. It is a pattern used for developing Microservices so that the external systems can use them efficiently.

### **31) What is Reactive Extension in Microservices?**

Reactive Extension is a design pattern that allows collecting results by calling multiple services and then compiles a combined response. It is also called Rx. Rx is a popular tool in distributed systems that works opposite to legacy flows.

### **32) How can you configure Spring Boot application login?**

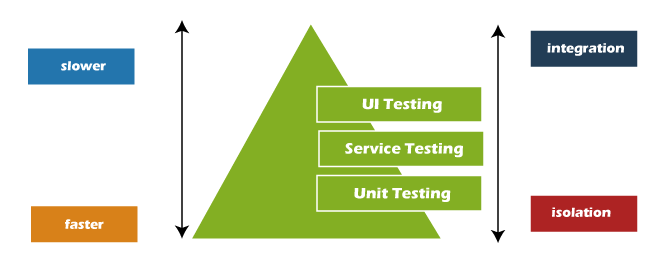
We can configure the Spring Boot application login by specifying the logging.level in the application.properties file. Generally, it is pre-configured as console output.

### **33) What do you understand by the term 'Continuous Monitoring'?**

The term continuous monitoring is used to specify a method used to search compliance and risk issues associated with a company's operational and financial environment. This method contains human processes and working systems that support efficient and actual operations.

### **34) What is Mike Cohn's Test Pyramid?**

Mike Cohn's Test Pyramid is used to maximize automation at all levels of testing, such as unit testing, service level testing, UI testing, etc. This pyramid specifies that while unit tests are faster and more isolated, UI tests, which are at the highest level, take time and focus on integration.



### **35) How independent micro-services communicate with each other?**

We can make our micro-services communicate with each other according to our project needs. In most cases, developers use HTTP/REST with JSON or Binary protocol while using any communication protocol.

### **36) How can you implement a Spring Security in a Spring Boot Application?**

We can quickly implement a Spring Security in a Spring Boot Application by using the following method:

* By adding the spring-boot-starter-security in the file pom.xml
* By creating a Spring config class that will override the required method while extending the WebSecurityConfigurerAdapter to achieve security in the application

### **37) What do you understand by ubiquitous language?**

Ubiquitous Language or UL is a common language used by developers and users of a specific domain to explain that domain easily. The ubiquitous language has to bring all the team members on the same page and be translated so that a machine can understand.

### **38) What is the difference between Rest and Microservices?**

There are multiple ways to implement microservices. REST over HTTP is one of them. REST is also used in other applications such as web apps, API design, and MVC applications to serve business data.

On the other hand, in microservices architecture, all the system components are put into individual components, which can be built, deployed, and scaled individually. Microservices provide certain principles and best practices that help in building a resilient application. So, we can say that REST is a medium to build Microservices.

### **39) What do you understand by Idempotence? Where is it used?**

Idempotence is a property that facilitates us to do something twice so that the result will remain the same always in such a way that if it had been done once only. Usage of Idempotence: Idempotence is used at the remote service or data source so that, when it receives the instruction more than once, it only processes the instruction once.

### **40) What is an Actuator in Microservices? Why is it used?**

Actuator is a sub-project of Spring Boot. It brings in production-ready features into an application and is mainly used to expose operational information about the running application's health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to interact with it.

### **41) What is the use of Bounded Context in Domain-Driven Design?**

The Bounded Context is a central pattern in Domain-Driven Design. It is the core of Domain-Driven Design's strategic design section, which deals with large models and teams. It is used to divide the large models into different Bounded Contexts and being explicit about their inter-relationships.

### **42) What is PACT in Microservices Architecture?**

The contract between a consumer application and a provider application is known as a PACT. Each PACT is a collection of interactions. It is an open-source tool that can be used to implement the Consumer-Driven Contract in Microservices.

### **43) What do you understand by Two Factor Authentication? What are the different types of credentials used in Two Factor Authentication?**

Two-factor authentication enables the users to fulfill the second level of authentication to an account login process. If a user has to enter only a username and password at the time of login, it would be considered a single-factor authentication. But in Two-factor authentication, the user has to enter more information than the login password.

**Mainly three types of credentials are used in Two-factor authentication:**

* **Something you know:** In this authentication step, you have to enter a PIN, password, or a pattern.
* **Something you have:** This authentication step requires an ATM card, phone, or OTP.
* **Something you are:** In this authentication step, you have to enter your biometric fingerprint or voice print.

### **44) What is the need for Reports and Dashboards in Microservices?**

Reports and dashboards are mainly used to monitor microservices. There are multiple tools used for this purpose.

**Following is the list of some usages of Reports and dashboards in microservices:**

* Reports and dashboards are used to find out which microservices expose what resources.
* It is also used to specify the services which are impacted whenever we make changes in a component.
* It provides an easy point to access whenever documentation is required.
* It specifies the versions of the components which are deployed.
* It is also used to obtain a sense of maturity and compliance from the components.

### **45) What do you understand by Canary Releasing?**

Canary releasing is a technique used to introduce new software versions by rolling out the updated version or new code/features to a subset of users as an initial test before making the entire infrastructure available to everybody. This technique is called canary release because it is based on canary releases in coal mines to alert miners when the toxic gases reach dangerous levels.

### **46) Why do many developers hesitate in using Microservices? / What are the biggest cons of using Microservices?**

There are some cons of Microservices that can make developers hesitate in using Microservices:

* **Microservices require heavy investment:** It requires a great deal of collaboration. Since your teams are working independently, they should be able to synchronize well at all times.
* **The architecture setup cost is high:** The Microservices system is distributed, so the architecture is heavily involved and costly.
* **Handling operations overhead is complex:** Using Microservices may lead to overhead so, you need to be ready for operations overhead if you are planning to use Microservices architecture.
* **Autonomous staff selection:** It requires skilled professionals to support Microservices that are distributed heterogeneously.

### **47) What are non-deterministic tests? What is the process to eliminate them?**

Non-deterministic tests or NDT are unreliable tests that sometimes pass and sometimes fail. When these tests fail, they are re-run again. We can use the following ways to eliminate non-determinism from Non-Deterministic tests.

* Quarantine
* Asynchronous
* Remote Services
* Isolation
* Time
* Resource Leaks

### **48) What is the usage of WebMvcTest annotation in Spring MVC applications?**

WebMvcTest annotation is used for unit testing in Spring MVC Applications in cases where the test objective is to focus on Spring MVC Components.

**See the following code:**

@WebMvcTest(value =ToTestController.class, secure = false):

Here, we want to launch only the ToTestController. All other controllers and mappings will not be launched until this unit test is executed.

### **49) What is Eureka in Microservices?**

Eureka or Eureka Server is an application that holds the information about the client-service applications. Microservices have to register into the Eureka server, and the Eureka server knows all the client applications running on each port and IP address. Eureka Server is also alternatively known as the Netflix Service Discovery Server. It uses Spring Cloud and is not heavy on the application development process.

### **50) What is the full form of DRY? What is its usage in Microservices architecture?**

The full form of DRY is Don't Repeat Yourself. It is used to promote the concept of reusing the code. This makes things easy in developing and sharing the libraries, which in turn result in tight coupling.

### **51) How can you balance the server-side load by utilizing Spring Cloud?**

We can use the Netflix Zuul to balance the server-side load by utilizing Spring Cloud. It is also known as a JVM-based router.

### **52) What do you understand by Cross-functional testing?**

Cross-functional testing is the verification of non-functional requirements, i.e., the requirements that we cannot implement like a standard feature.

### **53) What is the use of Netflix Hystrix?**

Hystrix is an error tolerance and latency library. It is mainly used to isolate the access points. It also ensures that all 3rd party libraries and services are restricted. So, the application runs efficiently and avoids the kind of failures that occur in distributed systems.

### **54) What do you understand by Tasklet in Microservices?**

The Tasklet is a simple interface with a method to execute. We can use it to perform single tasks like running queries, deleting files, etc. In Spring Batch, the Tasklet is an interface used to perform unique tasks like clean or set up resources before or after any step execution.