***Microservice Technologies***

## Distributed Tracing

## 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.

## 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.

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**.

## Zipkin

Url: <http://localhost:9411/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.

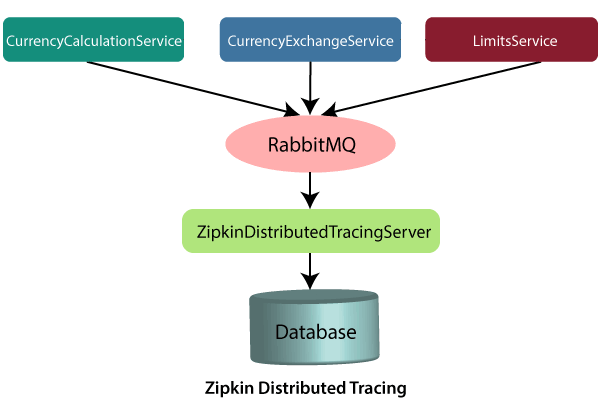
With the help of the Zipkin server, we can put all the logs of all the components in the **MQ** (RabbitMQ). We send the logs to the Zipkin server where the logs consolidate. After doing this, we can monitor different requests. We can also find what is happening to a specific request?

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 be 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.

## 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.

### AMQPdefines:

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

## 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.

# Understanding the need for Spring Cloud Bus

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.