**Microservices**

**Monolithic Architecture:**

* **Monolithic architecture is like a big container wherein all the software components of an application is assembled and tightly packaged.**
* **Basically, all the components of the monolithic application are interconnected and inter depended**
* **Let say a classic example of an e-commerce website assume that there is a customer service, product service, and payment service. In case of monolithic architecture this three will deployed as a single application.**

**Advantages: Very sim ple to develop and straight forward,**

**Disadvantages:**

* **Modification of the application is complex. So development will slow down,**
* **Deployment wise: in order to update a small thing in one component we have to deploy the entire application.**
* **In scaling , we cannot scale the each component independent.**
* **And it’s pretty unreliable as well because of the tightly coupled components, if one of them goes down the entire application will fail to run.**
* **To overcome this Micro services architecture introduced.**

**What is Micro services:**

* **Microservices is an architectural style that structures an application as a collection of services that are highly maintainable and testable, loosely couple, independently deployable and modeled around a business domain.**
* **So large applications using this architectural pattern can be broken into small multiple micro services.**
* **Which together act as a one large application. But behind the scene it’s a microservices. These microservices communicating each other through Api’s**

**Advantages:**

* **Each service can only focus on one single business capability.**
* **It is possible to change or upgrade each service individually rather than upgrading in the entire application.**
* **Less dependency, and easy to test. Faster release cycle.**

**Disadvantages:**

* **Microservices has all the 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.**
* **Complex testing because of a distributed environment.**
* **When more services interact with each other, the possibility of failure also increases.**

**Monitoring: The traditional way of monitoring is not suitable here. because we have multiple services. When an error arises in the application, finding the root cause can be challenging.**

**Components Of MS:**

**Ref git url : https://github.com/shabbirdwd53/Springboot-Microservice**

* **Spring Cloud Config Server**
* **Service registry (Eureka Naming Server)**
* **API Gateway Server(Netflix Zuul)**
* **Distributed Tracing Server(Zipkin)**
* **Circuit breaker**
* **Hystrix Server**
* **Load Balancer(Ribbon)**

1. **Spring Cloud Config Server**

* **Spring Cloud Config is an idea to storing and serving distributed configurations across multiple applications and environments using GIT.**
* **This server provides an API for external configuration (name-value pairs or equivalent YAML content).**
* **The server is embeddable in a Spring Boot application, by using the @EnableConfigServer annotation.**
* **Default port :8888**

**Steps to create Cloud config server**

1. **Create spring boot application with spring cloud config server dependency org.springframework.cloud:spring-cloud-config-server'**
2. **Add @EnableConfigServer Annotation in main class**
3. **Create the configuration repository in git and add required application.properties/yml files . File name should be <application - name>-<profile>.yml/properties.**
4. **Add spring.cloud.config.server.git.uri =”above repo url” and Server.port = 8888 , server.application.name=”app name”in application.properties file**
5. **Run the cloud config server by invoking http://localhost:8888/<application –namr>/env**

**Eg:** [**http://localhost:8888/SpringCloudDemo/local**](http://localhost:8888/SpringCloudDemo/local)

**Sample project**

**Tutorial-https://github.com/Bhuvaneswari-Vajravel/SpringCloudConfigServer**

**Steps to create Spring cloud config client**

**1. In client application add spring-cloud-starter-config starter**

**org.springframework.cloud:spring-cloud-starter-config in build tool.**

**2.Add spring.config.import=configserver:http://localhost:8888 in application.properties.**

**3. Create controller class and get mapping to fetch cloud config daya by @value annotation. Hit the url in postman and get response.**

**Sample proj :** [**https://github.com/Bhuvaneswari-**](https://github.com/Bhuvaneswari-) **Vajravel/SpringCloudConfigClient**

1. **Service registry/ discovery server**

* **Eureka Server is an application that holds the information about all client-service applications.**
* **Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address.**
* **The registration of microservice with the naming server is called Service Registration.**
* **Eureka Server is also known as Discovery Server**
* **Whenever a service wants to talk with another service it should first talk with eureka naming server.**
* **The naming server provides the instances of particular service that are currently running. The process of providing instances to other services is called Service Discovery.**

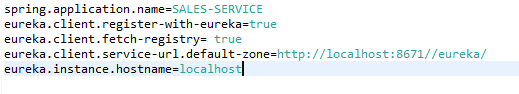
**Steps to create Service Registry:**

* **Create spring boot application with EurekaServer starter**
* **Add @EnableEurekaServer annotation in main class**
* **Set the port number as 8761 In application.properties file**
* **Add below properties to avoid this application registered as client in eureka**

**eureka.client.register-with-eureka=false**

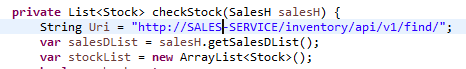
**eureka.client.fetch-registry=false**

* **Add eureka client starters in client application**
* **Add @EnableEurekaClient annotation in main class**
* **Add application name in application.properties**
* **Add eureka service URL, other properties and register the application in Eureka server**

****

* **While communicate other microservice first we can need to contact service registry with application name which will provide the instance of required service**
* **While creating rest template bean we need to mention it should be load balanced by using @LoadBalance annotation,**
* **if we have multiple services are registered to eureka server then it should be load balance the request.**

**This application name we can use like below**

****

1. **API Gateway**

* **The API Gateway is a server.**
* **It is a single-entry point into a system.**
* **API Gateway encapsulates the internal system architecture**
* **All the requests made by the client/internet go through the API Gateway.**
* **After that, the API Gateway routes requests to the appropriate microservice based on url pattern.**
* **It also has an inbuilt load balancer to load the balance of all incoming request from the client.**
* **Default port: 8765**

**Advantages of API Gateway**

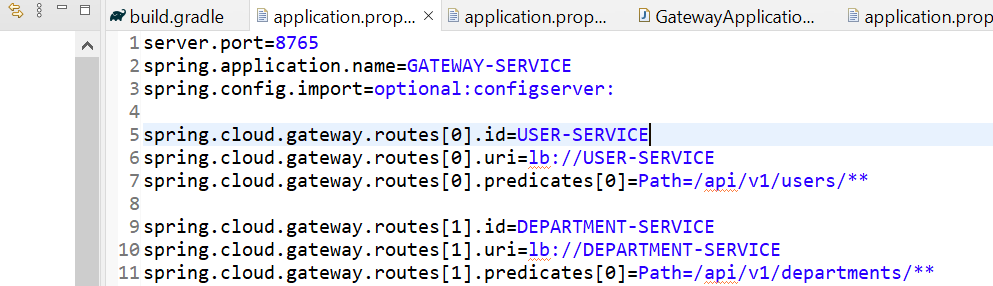
* **The most important advantage of API Gateway is that it encapsulates the internal structure of the application.**
* **Rather than invoking the specific service, the client directly talks to the API Gateway.**

**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.**

**Steps to create API Gateway**

1. **Create spring boot application with spring-boot-starter-actuator , spring-cloud-starter-gateway, spring-cloud-starter-gateway starters**
2. **Add enableEurekaClient annotation in main class. Add port number, application name, uri routes in application.properties**

****

1. **Now we can use** [**http://localhost:8765/api/v1/departments/**](http://localhost:8765/api/v1/departments/) **to add departments**

**4. Circuit Breaker / Fallback method**

* **Circuit breaker is used to identify which of the services is not running .**
* **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 rejects calls until it becomes healthy again**
* **Till now it will run the fall-back method available**

**Fault Tolerance:**

* **Fault tolerance is the individual service that does not bring down the overall system.**
* **Without fault tolerance, a single failure in the system may cause a total breakdown.**
* **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 is called fault tolerance.**
* **Fault tolerance can be achieved with the help of a circuit breaker.**
* **The circuit breaker is a pattern that wraps the request to external service and detects when they are faulty.**
* **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.**

1. **Hystrix Server**

* **Hystrix is a library. Using this we can implement the dashboard there we can identify which are the services running and which are not running.**

**LOAD BALANCER:**

* **Load balancing is nothing but efficient distribution of network or application traffic across multiple servers.**
* **Each load balancer sits between client devices and backend servers, receiving and then distributing incoming requests to any available server capable of fulfilling them.**
* **We can use netflix ribbon as a load balancer. It provides the client-side balancing algorithm. It uses a Round Robin Load Balancing**

**There are two types of load balancing available:**

* **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.**

**Configuration needs to be added**

**Distributed Tracing: (implementation pending)**

* **We use distributed tracing to follow the request or transaction as it travels through the application.**
* **This is use to pinpoint bugs, or other issues that impact the application’s performance.**
* **We can use Zipkin and sleuth**
* **If anything goes we cant go and check each microservice , spring cloud sleuth and zipkin will solve the issue**
* **Once we added spring cloud sleuth and zipkin, sleuth will generate metadata for each request. Metadata consist of component name,trace id, span id.**
* **Trace id unique for each request throughout the microservice**
* **Span id is unique per microservices**
* Zipkin is an application that monitors and manages the Spring Cloud Sleuth logs of your Spring Boot application. T

**CONFIGURATION**

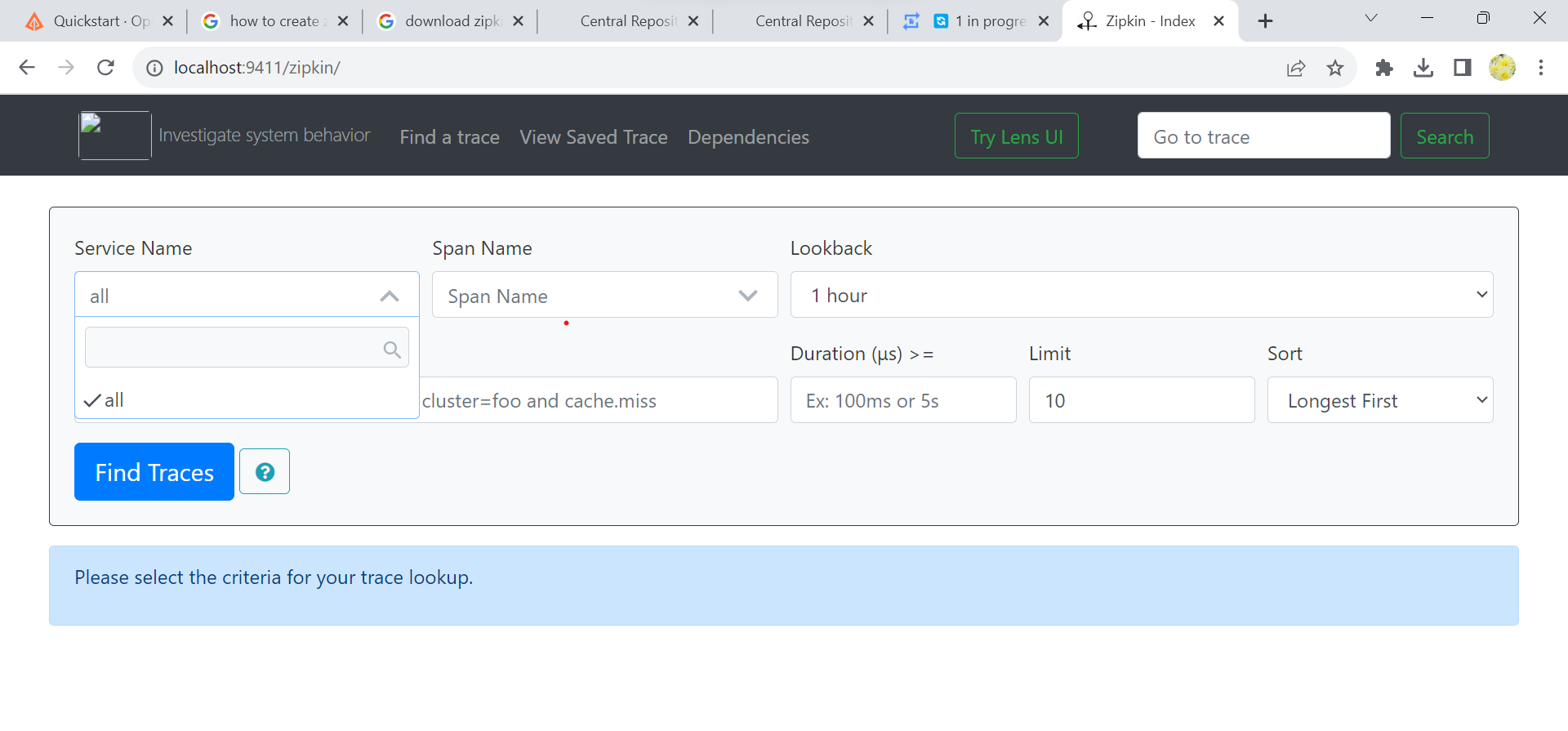
**1. download the zipkin server exec jar from maven repository https://repo1.maven.org/maven2/io/zipkin/java/zipkin-server/2.12.9/**

**2.start the zipkin server by running “java –jar zipkin-server-2.12.9-exec” in cmd.**

**Download zipkin-server-exec not zipkin-server.**

**3. it will start the zipkin server on port 9411**

**4. open** [**http://127.0.0.1:9411/**](http://127.0.0.1:9411/) **or** [**http://localhost:9411/**](http://localhost:9411/)**, no service displayed here because no application using zipkin server**

****

**5. add spring cloud sleuth zipkin jar in build tool.**

implementation 'org.springframework.cloud:spring-cloud-sleuth-zipkin'

6.and zipkin **base url** [**https://localhost:9411**](https://localhost:9411) **in each microservice application.properties , so that all microservice added to zipkin dashboard,**

**5. now we can see the trace id / span id for each microservices**

**6. in zipkin dashboard we can see the microservice span,and request flow , request url, time taken for each request and when it is triggered how long it takes etc.**

**Feign Client:**

**The Feign is a declarative web service (HTTP client) developed by Netflix. ... 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**

**QA**

**https://www.guru99.com/microservices-interview-questions.html**

**Performance improvement**

1. **Cache mechanism**
2. **load balancing**
3. **indexing tables**

**Redis Cache**

**Messaging que**

**Spring boot/Microservice design pattern**