**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 mono lithic 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 simple 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)**
* **Distributed Tracing Server(Zipkin/splunk)**
* **API Gateway Server(Netflix Zuul)**
* **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**

1. **Add @EnableConfigServer Annotation in main class**
2. **Create git repository to store the properties files for different environment and application. File name should be - <application-name>-<env-name>.properties/yml**
3. **Add spring.cloud.config.server.git.uri =”above repo path ”in application.properties file eg: spring.cloud.config.server.git.uri=https://github.com/Bhuvaneswari-Vajravel/cloud-config.git**

**Run and test the application by invoking [http://localhost:8888/<application](http://localhost:8888/%3capplication) name>/environment**

**Eg:** [**http://localhost:8888/SpringCloudConfigServer/dev**](http://localhost:8888/SpringCloudConfigServer/dev)

**Sample proj:**

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

**Steps to create cloud config client**

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

**2. in application.properties add below properties**

**spring.config.import=configserver:http://localhost:8888, profile**

**3. Create rest api to fetch the cloud config details**

**Sample proj :**

**https://github.com/Bhuvaneswari-Vajravel/SpringCloudConfigClient**

1. **Service registry/ discovery server**

* **The registration of microservice with the naming server is called Service Registration.**
* **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.**
* **Eureka Server is also known as 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.**
* **Whenever a service wants to talk with another service it should first talk with eureka naming server. Eureka server will provide the instance.**

**Steps to create Service Registry Server:**

* **Create spring boot application with EurekaServer starter org.springframework.cloud:spring-cloud-starter-netflix-eureka-server**
* **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**
* [**http://localhost:8761**](http://localhost:8761/)**to view the Eureka dashboard,**
* **Sample proj :**

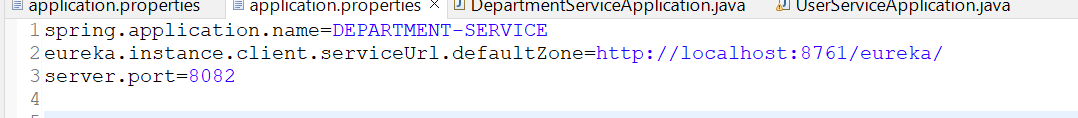
**https://github.com/Bhuvaneswari-Vajravel/EurekaServiceRegistryServer**

**Steps to create service registry client**

* **Add eureka client starters** org.springframework.cloud:spring-cloud-starter-netflix-eureka-client **in client application build tool.**
* **Add application name,port in application.properties**
* With *spring-cloud-starter-netflix-eureka-client* on the classpath, we just need to configure

*eureka.instance.client.service-url.defaultZone* property in *application.properties* to automatically register with the Eureka Server.

* When a service is registered with Eureka Server it keeps sending heartbeats for certain intervals. If the Eureka server didn’t receive a heartbeat from any service instance it will assume the service instance is down and take it out from the pool.

****

**Sample proj :**

[**https://github.com/Bhuvaneswari-Vajravel/DepartmentService**](https://github.com/Bhuvaneswari-Vajravel/DepartmentService)

**Communication between micro services:**

* **While communicate other microservice first we need to contact service registry with application name which will provide the instance of required service**
* **So that we can avoid hard coding ip address and port number for all the microservice**
* **We can communicate in 3 ways**

**1. Rest template**

**2. web client**

**3. Spring cloud open feign**

**1. Implementation using rest template:**

* **Create rest template bean.**
* **Inject rest temblate bean in service class**
* **Invoke other microservice like below**

String resp = restTemplate.getForObject("http://DepartmentService/api/department/test", String.**class**);

**2. Implement using web client**

* **Add spring-boot-starter-webflux in build tool.**
* **Create webclient bean and invoke like below**

String resp = webClient.get()

.uri("http://DepartmentService/api/department/test")

.retrieve()

.bodyToMono(String.**class**)

.block();

**Note: for rest template we should add web started and for web client we should add webflux(reactive web)**

**For both we need create bean, inject and call in the above way**

**Sample Proj : https://github.com/Bhuvaneswari-Vajravel/UserService**

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**

**The Spring Cloud Gateway has three important parts to it:**

**Route** − These are the building blocks of the gateway which contain the URL to which the request is to be forwarded to and the predicates and filters that are applied to the incoming requests.

**Predicate** − These are the set of criteria that should match for the incoming requests to be forwarded to internal microservices. For example, a path predicate will forward the request only if the incoming URL contains that path.

**Filters** − These act as the place where you can modify the incoming requests before sending the requests to the internal microservices or before responding back to the client

**Advantages of API Gateway**

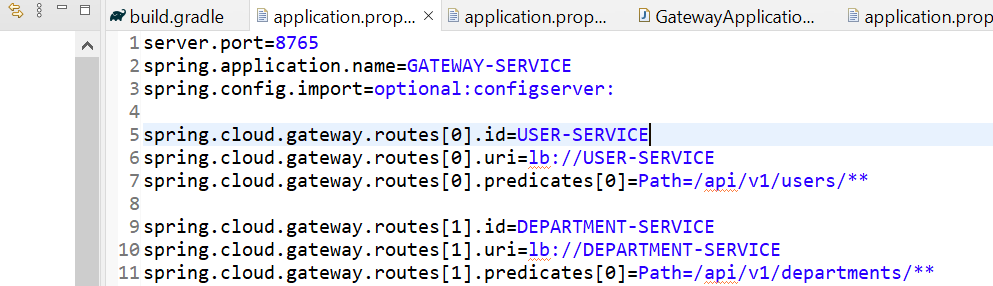
* **The most important advantage of API Gateway is that it encapsulates the internal structure of the application.**
* **Rather than invoking the specific service, the client directly talks to the API Gateway.**

**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.**

**Diff vs eureka server and hystrix server**

**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:**

* **Distributed tracing is following /track single request through across multiple service where as distributed logging is collect the logs from multiple microserivces**
* **This is use to pinpoint bugs, or other issues that impact the application’s performance.**
* **It is used to understand the performance of specific service**
* **We can use Zipkin and spring cloud sleuth or Splunk**

**CONFIGURATION**

**Zipkin Configuration**

1. **download the**  **and execute it in cmd prompt using java – jar zipkin-server-2.12.9-exec.jar**
2. **Default port for zipkin server is 9411**
3. **Add** spring-cloud-starter-zipkin and **spring cloud sleuth starter in build tool.**
4. **Add** spring.zipkin.baseUrl=http://localhost:9411/ in application.properties
5. **Once we start the server we can see client application zipkin dash board and we can see the trace id span id for each request in console logs which is generated by sleuth**
6. **In zipkin dashboard we can see the request flow for each request and time taken to execute the service and url method everything we can see**
7. **Tace id unique for across the micro services, span id will change for each micro service**

**Splunk**

**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**

**RB mq**

**Messaging que**

**Spring boot/Microservice design pattern**