1) Hystrix, [Hystrix](https://www.baeldung.com/introduction-to-hystrix) is watching methods for failing calls to related services. If there is such a failure, it will open the circuit and forward the call to a fallback method

In Client eureka class order Service

@EnableCircuitBreaker

@EnableHystrixDashboard

In the controller of the order Service-specific rest method

@HystrixCommand(fallbackMethod = "fallBackResponse")

public Object fallBackResponse(String orderId, Throwable t) {

Responses response = new Responses();

response.setStatusCode("404");

response.setMessage("Failure Message "+ t.getMessage());

return response;

}

------------------------------------------------------------------------------------------------------------

2) Zuul. API gateway

In eurekaClient

@SpringBootApplication

@EnableEurekaClient

@EnableZuulProxy

in application.yml

zuul routes order-service and product-service or we can setting up ribbon config as well connectTimeout readTimeout.

------------------------------------------------------------------------------------------------------------

3) Zipkin + Sleuth monitoring our microservice application.

Add dependency

@EnableZipkinStreamServer

@SpringBootApplication

then

Configuration bootstrap file

Config zipkin.properties with port number and other information

Config service.

------------------------------------------------------------------------------------------------------------

4) Ribbon/ Feign **dynamic load balance**

enables you to achieve fault tolerance in your applications, it balancing out the capacity needed to route application traffic. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource.

**Ribbon: Load Balancer Without Eureka**

The ribbon is a client-side load balancer, which gives you a lot of control over the behavior of HTTP and TCP clients. Ribbon's Client component offers a good set of configuration options such as connection timeouts, retries, retry algorithm (exponential, bounded back off) etc

In client orderService main class

proxy class

@RibbonClient(name = "product-service")

------------------------------------------------------------------------------------------------------------

**Feign - Load Balancer using with Eureka**

Feign is a declarative web service client or declarative REST client. It makes writing web service clients easier.

In EurekaServer class

In the application main class

@EnableEurekaClient

@EnableFeignClients

OrderProjectMain

------------------------------------------------------------------------------------------------------------

In client orderService main class

@EnableEurekaClient

@EnableFeignClients

In client orderService

In proxy class give name of the service you want to get rest client.

@FeignClient(name = "product-service")

proxy class for feign api call

------------------------------------------------------------------------------------------------------------

5) Eureka server microservice client

@EnableEurekaClient

application.yml set port and eureka client and instance config

bootstrap.yml set cloud-config url

@EnableEurekaClient

application.yml port number name of client service, eureka instance, client and healthcheck.

------------------------------------------------------------------------------------------------------------

Kafka

Producer

In the main class springbootApplication

add restTemplate

@Bean

public RestTemplate getRestClient() {

return new RestTemplate();

}

kafkaConfiguration class

@Configuration

@EnableKafka

@Bean

public KafkaTemplate<String, User> kafkaTemplate(){

return new KafkaTemplate<>(producerFactory());

}

In @service

KafkaSender class

kafkaTemplate.send

Controller class

kafkaSender.sendMessage();

------------------------------------------------------------------------------------------------------------

Consumer

@Configuration

@EnableKafka

Configuration class KafkaConfiguration

@Bean

public ConsumerFactory<String , User> consumerFactory(){

Map<String, Object> configs = new HashMap<>();

configs.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

configs.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

configs.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, JsonDeserializer.class);

configs.put(ConsumerConfig.GROUP\_ID\_CONFIG, "group\_id");

return new DefaultKafkaConsumerFactory<>(configs, new StringDeserializer(), new JsonDeserializer<>(User.class));

}

@Bean

public ConcurrentKafkaListenerContainerFactory<String, User> kafkaListenerContainerFactory(){

ConcurrentKafkaListenerContainerFactory<String, User> factory = new ConcurrentKafkaListenerContainerFactory<String, User>();

factory.setConsumerFactory(consumerFactory());

return factory;

}

@Service

kafkaConsumer class

@KafkaListener(topics="example-kafka-sender", groupId="group\_id", containerFactory = "kafkaListenerContainerFactory")

------------------------------------------------------------------------------------------------------------

Elastic search

In bean class product

@Document(indexName = "product\_it", type = "it")

@ApiModelProperty(notes = "Product Id", name = "id", required = false)

In elastic config

ElasticSearchConfig class

@Configuration

@EnableElasticsearchRepositories(basePackages = "com.example.ansh.repositorys"

private String url = "localhost:9200";

@Bean

public RestHighLevelClient client() {

ClientConfiguration clientConfiguration = ClientConfiguration.builder().connectedTo(url).build();

return RestClients.create(clientConfiguration).rest();

}

@Bean

public ElasticsearchOperations elasticsearchTemplate() {

return new ElasticsearchRestTemplate(client());

}

In controller class

@RestController

@RequestMapping(value = "/product")

@Api(value = "ProductController", description = "REST API for Search Service by ElasticSearch")

------------------------------------------------------------------------------------------------------------

Spring Actuator

What is Actuators?

To monitor performance and help of the application, actuators make any application production-ready. It provides N number of endpoints like info, auto-config, health, and beans.

How to add custom information in spring boot actuators?

1. We can go to the property file and add it into the property file.spring.info.properties we want to add. (this will be static only)
2. We have to create our own actuators class and implementing InfoContributor

How to implement an Actuator?

First add dependency of actuator,

Enabling Endpoints “management.endpoint.shutdown.enabled=true”,

Securing HTTP Endpoints. If Spring Security is present, endpoints are secured by default using Spring Security’s content-negotiation strategy. like hasRole("ACTUATOR")“/”.

we can access endpoint using <http://localhost:8080/actuator/beans>, or http://localhost:8080/actuator/env