SPRING CLOUD

DOCUMENTATION

Table of Contents

Spring Cloud

1. Introduction to Micro Services

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

1.1 Monolithic vs Micro Services

A monolithic architecture is built as one large system and is usually one code-base. A monolith is often deployed all at once, both front and end code together, regardless of what was changed. A micro services architecture however is where an app is built as a suite of small services, each with their own code-base. These services are built around specific capabilities and are usually independently deployable.

Micro Service Architecture	Monolithic Architecture
Flexible (easy to upgrade technology stack)	Not Flexible (Technology adoption is difficult)
Reliable (As it will not fail entire system since services are small we can fix easily and deploy in less time)	Not Reliable (if one feature goes down entire app might go down)
Development Speed is high	Development speed is slow and difficult to build complex apps as tech adoption is slow
Scalability	Scaling is difficult
Continuous deployment is easy	Continuous deployment is difficult
Initial cost is high but reduces while operating	Initial cost is low but increases while operating
Response time is slow (API call)	Response time is fast (process call)
Increases productivity and release cycles	Release cycles are slow compared to micro services

2. Advantages in Micro Services

- 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.

3. Challenges in Micro Services

- Bounded Context
- Dynamic Scale up and Scale Down
- Monitoring
- Fault Tolerance
- Cyclic dependencies
- 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.
- 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.

4. Spring Cloud

Spring Cloud is a framework for building robust cloud applications. Spring Cloud provides tools for developers to build some of the **common patterns** in distributed systems quickly. For example, configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, a control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state.

5. 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

6. Components of spring cloud

There are the following components:

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

6.1 Centralized 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, filesystem, and Vault to Store config. Config clients (microservice app) retrieve the configuration client from the server on startup.

6.2 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**.

- **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.

6.3 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.

6.4 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.

6.5 Routing and Messaging

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

6.6 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.

6.7 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.

7. Implementation of service discovery

7.1 Eureka Server

To make a service among micro services as eureka server in pom.xml **spring-cloud-starter-netflix-eureka-server** dependency and spring-boot-starter-actuator dependencies are required and the main class should be decorated with annotation @EnableEurekaServer and application. Properties file add values as below

eureka.client.registerWithEureka = false
eureka.client.fetchRegistry = false

By default each service acts like client so to disable the client registration our service acts as server. Server will always searches for the client's health endpoints by listening to their heart beats and it provides UI to look into the clients register for that particular server by accessing context path with /eureka

7.2 Eureka Client

To make a service among micro services as eureka server in pom.xml **spring-cloud-starter-netflix-eureka-client** dependency and spring-boot-starter-actuator dependencies are

required and the main class should be decorated with annotation @EnableEurekaClient and application. Properties file add values as below

spring.application.name=ServiceOne ->CLIENT SERVICE NAME
client.serviceUrl.defaultZone=http://localhost:8761/ -> SERVER URL

8.Implementation of HYSTRIX

Hystrix is used to enable the service when it has failed it makes service to be fault tolerant and dependency required is spring-cloud-starter-hystrix and add @EnableCircuitBreaker annotation on main class and to the resource or method where endpoint is exposed to that method @HystrixCommand Annotation is added and fallback method for that method is defined so when client hits the method with URL and method fails instead of fail the service it will response with executed fallback method