**What is spring cloud**

Used to build some of the common patterns in distributed systems.

example configuration management, service discovery, circuit breaker

configuration management: ex: having many **application.properties** file at one git repo and accessed by different microservices.

example**, when using centralized database it’s good to have it in one place so that other microservices can use it.**

**What is discovery server**:

Two types server-side and client side service discovery

**client side service discovery**: allows client applications to find services by looking into service registry that has service instances and endpoints all within the service registry

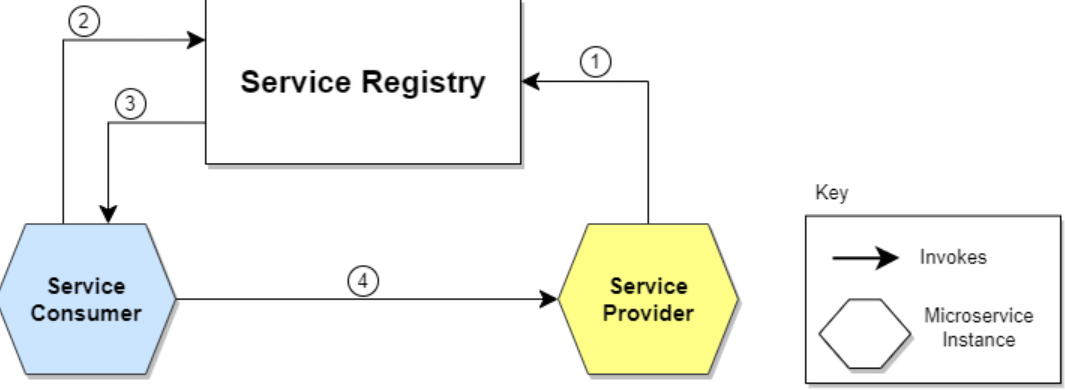
**Server-side service discovery**: allows clients applications to find services through a router or a load balancer

**why? (need):** A microservice needs to know the location (IP address and port)

of every service it communicates with

**Implementation**:

**We could wire the location or inject them directly in the configuration, but it is not recommended in modern cloud based application of this kind.**

**Also, in big or scaled up environment we constantly create or destroy the instances of different microservices hence the need of services discovery mechanism.**

How to create service discovery : create microservice (say named: eureka server) :

**Add following dependency**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

**Add annotation in main class** : @EnableEurekaServer

**Add following to application.properties**

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

eureka.client.service-url.defaultZone=http://eureka-server:${server.port}/eureka/

default port of eureka server is : 8761

**Finally to register a client with the eureka server**

**Add Following to pom.xml of client :**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

**Add Annotation in main class as : @EnableEurekaClient**

**Add following to application.properties**

eureka.client.service-url.defaultZone=http://eureka-server:8765/eureka/

**Monolith vs microservices**

The term monolithic applies to tightly integrated applications where it is hard to change one function without also recoding other parts of the application. Components in a monolithic application might be distributed among many machines, but they remain highly dependent on one another. Not only does the addition of a new feature have ripple effects throughout the code, but deploying the change requires retesting and redeploying the entire application. These upgrades can be labor-intensive and hazardous

An MOA breaks its logic into small, well-encapsulated services that are distributed over several computing devices in a loosely coupled manner. Each service lives at a distinct IP address on the network and exposes a public interface

**Monolith advantage:**

**Easy to communicate between internal modules;**

**Easy to manage and deploy.**

Simple to test (testing end to end or unit testing)

Disadvantage:

If load comes on the application, then it might go down

Example Mumbai university result site , or 10th board site

Continuous deployment is difficult: as for every small chages we will have to deploy the application again and again as we will have to test the entire application end to end again because modules could be related or dependent on each other.

Microservices:

It is an architectural style, whole application is divided into small small services that are independently managed and loosely coupled also they can communicate with each other .

**Advantages** :

Each microservice is independent.

Continuous deployment is possible.

Easy to scale.

Better fault tolerant.

Disadvantages:

Management is time consuming (as we will be deploying each ms independently)

Should be used for large scale application.

**Challenges in Microservices**

Bounded context

how to set right boundary for your microservices ie which microservice will do what

Configuration Management

100s of microservices and tons of environment to manage how do we simplify that.

Dynamically Scale up and Scale down

Dynamically scale up and scale down and dynamically distribute the load between the active instances

Visibility

We should have great visibility of whats happening between each microservices

Fault tolerance

How do I make sure that failure of one microservice will not affect other microservices

**How independent micro-services communicate with each other?**

**Synchronize and asynchronize communication;**

Synchronize : Synchonous , waits for the response . example **rest template.**

Asynchronize : Does not wait for response uses messaging service like RabbitMQ, apache kafka to initiate communication

**Principles characteristics of microservices:** (https://developers.redhat.com/articles/2022/01/11/5-design-principles-microservices#five\_design\_principles\_for\_microservices)

**Five design principles for microservices**

The five basic principles of microservice application design are:

* A microservice has a single concern. (Having a single concern means that a microservice should do one thing and one thing only. For example, if the microservice is intended to support authentication, it should do authentication only.)
* A microservice is discrete. (A microservice must have clear boundaries separating it from its environment. Another way to think about this principle is that a microservice must be well-encapsulated. This means that all logic and data relevant to a microservice's single concern must be encapsulated into a single deployment unit)
* A microservice is transportable.( A transportable microservice can be moved from one runtime environment to another with little effort. Perhaps currently, the optimal form of a transportable microservice is a Linux container image.)

A microservice carries its own data.( For example, imagine an e-commerce application. That application might have a microservice that manages customer profile information. The application has another microservice that handles purchases. When the principle that every microservice carries its own data is in force, it's quite possible that the purchases microservice might have data that is redundant with the customer profile microservice. Such data redundancy goes against the grain of developers who embrace the DRY principle (Don't Repeat Yourself).

On the other hand, developers who embrace the carry-their-own data principle understand the benefits and have adjusted accordingly. When a microservice carries its own data, any strange behavior is confined within the microservice.

When microservices try to share data, one microservice can make a change that causes a side effect in another microservice. This is fundamentally a bad way of doing business.)

* A microservice is ephemeral.( The principle that a microservice is ephemeral means that it can be created, destroyed, and replenished on demand on a given target easily, quickly, and with no side effects)

**What is config server and why do we need it? How to implement it?**

**It provides the url required and common for different mocroservices example centralized db configurations.**

**Implementation:**

**Add :**

|  |
| --- |
|  |
| <dependency> |
|  | <groupId>org.springframework.cloud</groupId> |
|  | <artifactId>spring-cloud-config-server</artifactId> |
|  | </dependency> |
|  |  |

**Add annotation :** @EnableConfigServer

This will be responsible for accessing the configuration form the central repository and config-server url will be configured in the application.properties file of different microservices that wishes to access these central configuration information from this config-server application.

Url of the central repository is mentioned in the application.properties file of config-server

spring.cloud.config.server.git.uri=file:///C:/prashant\_softwares/springboot\_projects/git-repo-microservice-configuration/git-local-configuration-repo/

**In client microservice that wishes to access the config-server**

|  |
| --- |
| <dependency> |
|  | <groupId>org.springframework.cloud</groupId> |
|  | <artifactId>spring-cloud-starter-config</artifactId> |
|  | <version>3.1.1</version> |
|  | </dependency> |

**Add following to the application.properties**

spring.config.import = optional:configserver:http://localhost:8888

**What is API gateway? Explain advantages of API gateway.**

Api gateway represents a single entry point to the application , it also provides security, composition of all the other microservices.

Implementation

Add :

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

application.properties

server:

port: 8080

spring:

application:

name: api-gateway

cloud:

gateway:

routes:

- id: admin-Service

uri: http://admin-service:8086

predicates:

- Path=/admin/\*\*

- id: jwt-authentication-service

uri: http://jwt-service:8282

predicates:

- Path=/token, /user/\*\*

main:

web-application-type: reactive

eureka:

client:

serviceUrl:

defaultZone: http://eureka-server:8765/eureka/

**Why do we need api gateway:**

**To stop the imposing internal concerns to external client**

**i.e internal uris are hidden from the external clients**

**Provides additional security to your microservices (using spring security)**

**Example zool api and Spring cloud gateway**

**Why do we need hystrix? How to implement it? How to implement Hystrix Dashboard?**

**Why ?**

Microservice architecture has many small independent services that communicate with each other to fulfill their requirement. Consider a small scenario six independent microservices are communicating with each other, out of these six one goes down then entire application will be affected, to avoid this we can use hystrix.

Hystrix is the library that contains the interaction between microservices that provides latency and fault tolerance.

Additionally it makes sense to modify the ui to let the user know that something is going wrong : (Example showing user understandable message if any error comes)

Implementation:

<**dependency**> <**groupId**>org.springframework.cloud</**groupId**> <**artifactId**>spring-cloud-starter-hystrix</**artifactId**> <**version**>1.4.7.RELEASE</**version**> </**dependency**>

Add annotation : @EnableHystrix

**Now, add the @Hystrix command and @HystrixProperty for the Rest API and define the timeout in milliseconds value.**

@RequestMapping(value = "/")

@HystrixCommand(fallbackMethod = "fallback\_hello", commandProperties = {

@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000")

})

public String hello() throws InterruptedException {

Thread.sleep(3000);

return "Welcome Hystrix";

}

private String fallback\_hello() {

return "Request fails. It takes long time to response";

}

**What are different ways to deploy microservices?**

**There could be may ways to deploy microservices.**

**We chose one of the way that is , creating docker images**

**Implementation:**

**Create Dockerfile**

**FROM openjdk:11**

**WORKDIR usr/src**

**EXPOSE 8086**

**ADD ./target/admin-service.jar /usr/src/admin-service.jar**

**ENTRYPOINT ["java","-jar","admin-service.jar"]**

**First create jar file : maven install**

**Build the Docker image**

Docker build -f Dockerfile -t application-name .

**Create repo in hub.docker.com with same or some related name .**

**Tagging the local image to the repo.**

Docker tag container-id username/reponame

**Push the docker image**

Docker push username/reponame

**What is domain driven design**

[**https://dzone.com/articles/the-concept-of-domain-driven-design-explained#:~:text=Domain%2Ddriven%20design%20is%20the,run%20independently%20from%20other%20microservices**](https://dzone.com/articles/the-concept-of-domain-driven-design-explained#:~:text=Domain%2Ddriven%20design%20is%20the,run%20independently%20from%20other%20microservices)**.**

**Distributed tracing in microservices**

## Implement distributed tracing in your microservices-based app

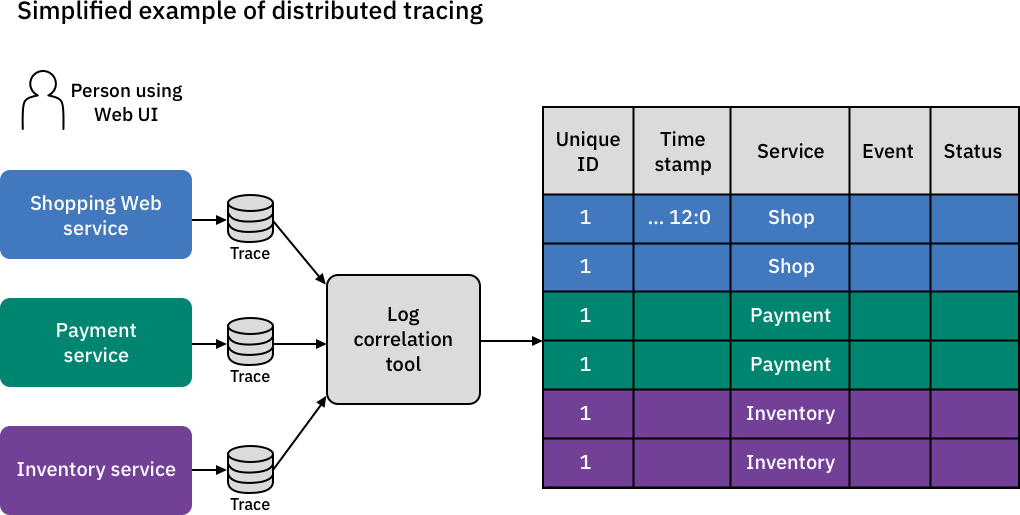
Moving your applications from a monolithic design to a microservices-oriented design introduces several advantages during development and in operations. However, that move has a price. New challenges are introduced, as traditional metrics and log information tend to be captured and recorded in a component and machine-centric way. When your components are spread across machines and physical locations and are subject to dynamic horizontal scaling over transient compute units, traditional tools to capture and analyze information become powerless.

Distributed tracing is a technique that addresses logging information in microservice-based applications. A unique transaction ID is passed through the call chain of each transaction in a distributed topology. One example of a transaction is a user interaction with a website. The unique ID is generated at the entry point of the transaction. The ID is then passed to each service that is used to finish the job and written as part of the services log information. It's equally important to include timestamps in your log messages along with the ID. The ID and timestamp are combined with the action that a service is taking and the state of that action.

Unique identifiers, such a transaction IDs and user IDs, are helpful when you gather analytics and debug. Unique IDs can point you to the exact transaction that failed. Without them, you must look at all the information that the entire application logged in the time frame when your problem occurred. After you implement the generation and usage of the unique ID in your logs, you can use the unique ID in several ways.

### **Enable log correlation**

By implementing the transaction ID and creating a timestamp of each log entry that contains that ID, you can take advantage of log-aggregating tools, such as IBM Cloud™ Log Analysis. Those tools provide a distributed stack trace of the steps that led to the failure of a specific transaction.



**Distributed tracing can be achieved using zipkin server**

 Zipkin is **a distributed tracing system**. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data. If you have a trace ID in a log file, you can jump directly to it.

Implementing zipking in spring boot

Add dependency:

**<dependency>**

**<groupId>io.zipkin.java</groupId>**

**<artifactId>zipkin-server</artifactId>**

**<version>2.11.7</version>**

**</dependency>**

**<dependency>**

**<groupId>io.zipkin.java</groupId>**

**<artifactId>zipkin-autoconfigure-ui</artifactId>**

**<version>2.11.7</version>**

**</dependency>**

**Add annotation :**

**@EnableZipkinServer**

**Add following to application.properties file**

**spring.application.name=zipkin-server**

**server.port=9411**

 url: <http://localhost:9411/zipkin>

registering client application with zipkin

Add following to application.properties file

spring.zipkin.base-url=http://localhost:9411/

Add dependency

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zipkin</artifactId>

</dependency>

How to create cluster of the eureka server