Microservices communication:

Synchronous Way

There could be two ways:

1. **REST Template**
2. **Feign**

Asynchronous Way

1. WebClient
2. Using a message broker

Using RestTemplate:

```
@SpringBootApplication
public class UserServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

Inside service implementation:

```
@Service
@AllArgsConstructor
public class UserServiceImpl implements UserService {

    private UserRepository userRepository;
    private RestTemplate restTemplate;
```

```
ResponseEntity<DepartmentDto> responseEntity = restTemplate
        .getForEntity("http://localhost:8080/api/departments/" + user.getDepartmentId(),
        DepartmentDto.class);

DepartmentDto departmentDto = responseEntity.getBody();

System.out.println(responseEntity.getStatusCode());
```

Using WebClient:

WebClient is a non-blocking, reactive client to perform HTTP requests, exposing a fluent, reactive API over underlying HTTP client libraries such as Reactor Netty.

To use *WebClient* in our Spring boot project, we have to add *Spring WebFlux* dependency to the classpath.

Step 1: Add Spring WebFlux Dependency

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Step 2: Configure WebClient as Spring Bean

```
@SpringBootApplication
public class UserServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }

    @Bean
    public WebClient webClient(){
        return WebClient.builder().build();
    }
}
```

Step 3: Inject and Use WebClient to Call the REST API

```
@Service
@AllArgsConstructor
public class UserServiceImpl implements UserService {

    private UserRepository userRepository;
    private RestTemplate restTemplate;
    private WebClient webClient;

    private APIClient apiClient;
```

```
DepartmentDto departmentDto = webClient.get()
         .uri("http://localhost:8080/api/departments/" + user.getDepartmentId())
                 .retrieve()
                         .bodyToMono(DepartmentDto.class)
                                 .block();
responseDto.setUser(userDto);
```

Using Spring Cloud Open Feign:

Feign Client is a declarative web service client developed by Netflix.

One great thing about using Feign is that we don't have to write any code for calling the service, other than an interface definition.

Step 1: Add Spring cloud open feign Maven dependency to User-Service

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Step 2: Enable Feign Client using @EnableFeignClients

```
@SpringBootApplication
@EnableFeignClients
public class UserServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}
```

Step 3: Create feign API client

```
@FeignClient(value = "DEPARTMENT-SERVICE", url = "http://localhost:8080")
public interface APIClient {
    @GetMapping(value = "/api/departments/{id}")
    DepartmentDto getDepartmentById(@PathVariable("id") Long departmentId);
}
```

Step 4: Change the getUser method to call APIClient

```
@Service
@AllArgsConstructor
public class UserServiceImpl implements UserService {

    private UserRepository userRepository;

    private APIClient apiClient;
```

```
@Override
public ResponseDto getUser(Long userId) {

    ResponseDto responseDto = new ResponseDto();
    User user = userRepository.findById(userId).get();
    UserDto userDto = mapToUser(user);

    DepartmentDto departmentDto = apiClient.getDepartmentById(user.getDepartmentId());
    responseDto.setUser(userDto);
    responseDto.setDepartment(departmentDto);

    return responseDto;
}
```

what Is Load Balancing?

Load balancing is the process of distributing traffic among different instances of the same application.

```
@RestController
@RequestMapping("/book")
public class BookRestController {

    @Autowired
    Environment environment;
```

```
    @GetMapping("/data")
    public String getBookData() {
      return "data of BOOK-SERVICE, Running on port: "
        +environment.getProperty("local.server.port");
    }
```

What Is an Actuator?

The actuator mainly exposes **operational information about the running application** including health status, application metrics, environment information, and more.

It uses HTTP endpoints to enable us to interact with it.

Maven Dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Default Configuration

Actuator comes with most endpoints disabled.

Thus, the only two available by default are */health* and */info*.

If we want to enable all of them, we could set

```
application.properties

management.endpoints.web.exposure.include=*

# To expose only selected endpoints
#management.endpoints.jmx.exposure.include=health,info,env,beans
```

Enabling Only Specific Endpoints

If we prefer to disable all endpoints, by default, and only expose certain endpoints then we can set the property *management.endpoints.enabled-by-default* to *false* and then use the only required endpoints that the application needs to expose using the pattern management.endpoint.<id>.enabled.

```
management.endpoints.enabled-by-default=false

management.endpoint.info.enabled=true
management.endpoint.health.enabled=true
management.endpoint.loggers.enabled=true
```

Important Actuator Endpoints

`/beans`	Returns a complete list of all the Spring beans in your application.
`/mappings`	Displays a collated list of all `@RequestMapping` paths.
`/env`	Exposes the application's environment properties and configuration. It's useful for inspecting the configuration values of your application.
`/info`	Allows us to expose arbitrary application information. We can configure custom properties to be displayed here, providing metadata about the application.
`/health`	Returns application health information. It checks various indicators, such as database connectivity, disk space, and custom health indicators, and presents the overall health status as "UP" or "DOWN."

Fault Tolerance in Microservices

In a context of Microservices, Fault Tolerance is a technique of tolerating a fault.

A Microservice that tolerates the fault is known as Fault Tolerant.

Moreover, a Microservice should be a fault tolerant in such a way that the entire application runs smoothly.

Resilience4j

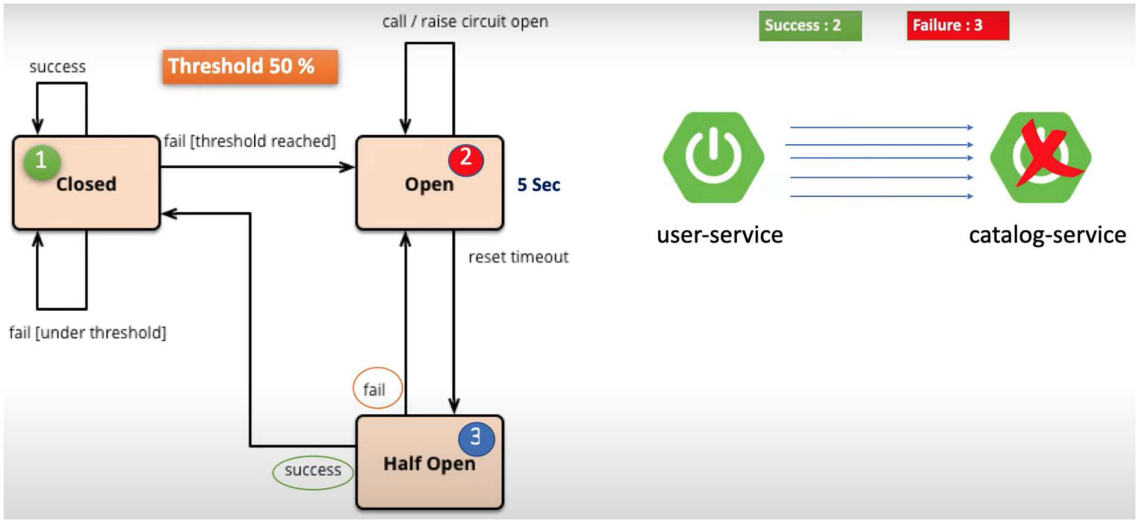
Resilience4j is a fault tolerance library for Java

Core modules of Resilience4j

- Circuit Breaker (fault tolerance)
- Rate Limiter (block too frequent requests)
- Time Limiter (Set a time limit when calling remote operation)
- Retry mechanism (Automatically retry a failed remote operation)
- Bulkhead (Avoid too many concurrent requests)
- Cache (Store results of costly remote operations)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

1. CircuitBreaker




```
@GetMapping("/courses")
@CircuitBreaker(name = "courseClient", fallbackMethod = "getCourseFallback")
public List<Course> getCourses() {
    List<Course> courses = restTemplate.getForObject("http://localhost:7072/course/all", ArrayList.class);
    return courses;
}

public List<Course> getCourseFallback(Exception e) {

    return Stream.of(
            new Course(1, "course1", 1000, 30), new Course(2, "course2", 1000, 30),
            new Course(3, "course3", 1000, 30), new Course(4, "course4", 1000, 30),
            new Course(5, "course5", 1000, 30), new Course(6, "course6", 1000, 30))
            .collect(Collectors.toList());
}
```

```
management.endpoints.web.exposure.include = health
management.health.circuitbreakers.enabled= true
management.endpoint.health.show-details= always
```

```
resilience4j.circuitbreaker.instances.courseClient.register-health-indicator = true
resilience4j.circuitbreaker.instances.courseClient.event-consumer-buffer-size = 10
resilience4j.circuitbreaker.instances.courseClient.failure-rate-threshold = 50
resilience4j.circuitbreaker.instances.courseClient.minimum-number-of-calls = 5
resilience4j.circuitbreaker.instances.courseClient.automatic-transition-from-open-to-half-open-enabled=true
resilience4j.circuitbreaker.instances.courseClient.wait-duration-in-open-state = 5s
resilience4j.circuitbreaker.instances.courseClient.permitted-number-of-calls-in-half-open-state=3
resilience4j.circuitbreaker.instances.courseClient.sliding-window-size=10
resilience4j.circuitbreaker.instances.courseClient.sliding-window-type=COUNT_BASED
```

1) '**failure-rate-threshold=50**' indicates that if 50% of requests are getting failed, open the circuit ie. Make the Circuit Breaker state as Open.

2) '**sliding-window-size=10**' indicates that if 50% of requests out of 10 (it means 5) are failing, open the circuit.

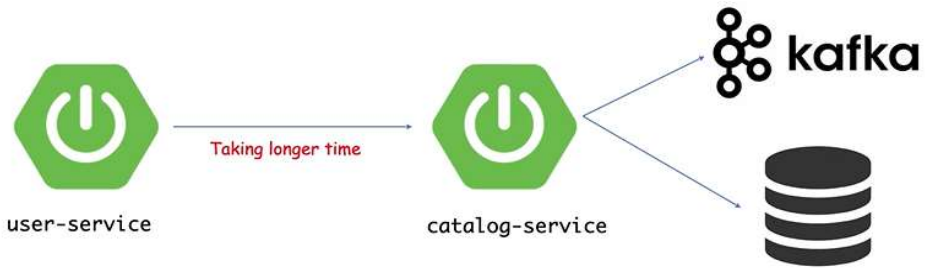
3) '**sliding-window-type=COUNT_BASED**' indicates that we are using COUNT_BASED sliding window. Another type is TIME_BASED.

4) '**minimum-number-of-calls=5**' indicates that we need at least 5 calls to calculate the failure rate threshold.

5) '**automatic-transition-from-open-to-half-open-enabled=true**' indicates that don't switch directly from the open state to the closed state, consider the half-open state also.

6) '**permitted-number-of-calls-in-half-open-state=3**' indicates that when on half-open state, consider sending 3 requests. If 50% of them are failing, switch circuit breaker to open state.

7) **'wait-duration-in-open-state=1s'** indicates the waiting time interval while switching from the open state to the closed state.



- Lag in Kafka
- Database running datacenter may be down
- Temporary unavailability of dependencies
- Deployment environment failure


```
resilience4j:
  retry:
    instances:
      userService:
        maxRetryAttempts: 3
        waitDuration: 5s
```

```
@GetMapping("/courses")
@Retry(name = "courseClient", fallbackMethod = "getCourseFallback")
public List<Course> getCourses() {
    List<Course> courses = restTemplate.getForObject("http://localhost:7072/course/all", ArrayList.class);
    System.out.println("Retry method called "+attempt++ +" times at "+ new java.util.Date());
    return courses;
}

public List<Course> getCourseFallback(Exception e) {
    return Stream.of(
            new Course(1, "course1", 1000, 30), new Course(2, "course2", 1000, 30),
            new Course(3, "course3", 1000, 30), new Course(4, "course4", 1000, 30),
            new Course(5, "course5", 1000, 30), new Course(6, "course6", 1000, 30))
            .collect(Collectors.toList());
}
```

```
resilience4j.retry.instances.courseClient.max-attempts=5
resilience4j.retry.instances.courseClient.wait-duration=10s
```

By default, the retry mechanism makes 3 attempts if the service fails for the first time. But here we have configured for 5 attempts, each after 10 seconds interval.

For each 10 seconds its retry to call the api with max attempts 5

Spring Cloud Config Server

In a Microservices architecture-based Java application, we have multiple microservices in the form of multiple Spring Boot applications. Each Spring Boot application will have its separate configuration file.

Updating configuration properties and restarting all those instances manually or even with automated scripts may not be feasible.

Here **Spring Cloud Config Server** comes into the picture and makes a developer's life easy by offering a common place for all common configurations.

We can create a Spring Cloud Config Server which provides the configuration values for all of our microservices. We can use git, svn, database, or Consul as a backend to store the configuration parameters.

we can configure the location of the Spring Cloud Config server in our microservice so that it will load all the properties when we start the application.

In addition to that, whenever we update the properties we can invoke the `actuator/refresh` the REST endpoint in our microservice so that it will reload the configuration changes without requiring us to restart the application.

Set up spring cloud config server

Create Microservice #1(Config Server)

Step #1: Create a Spring Boot Project and the below dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Step #2: Apply Annotation @EnableConfigServer at the main class

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

Step #3: Modify application.properties file

If you are using git local repository:

```
spring.application.name=config-server

server.port = 8888

spring.cloud.config.server.git.uri = D:\\trainings\\wipro\\batch2\\config-repo
spring.cloud.config.server.git.default-label=master

management.endpoints.web.exposure.include=*
```

If you are using GitHub cloud repository:

```
# Repository Location in Github
spring.cloud.config.server.git.uri=https://github.com/username/config_repo.git
# Github username
spring.cloud.config.server.git.username=yourUserName
# Github Password
spring.cloud.config.server.git.password=yourPassword

# Github default branch
spring.cloud.config.server.git.default-label=main

#spring.cloud.config.server.git.default-label=master
```

Create application.properties file and commit into git local repo.

application.properties:

message=Hello, MS how r you?

http://localhost:8888/application/default


```
{
    "name": "application",
    "profiles": [
        "default"
    ],
    "label": null,
    "version": "f03e3c16ce010431327536e382b99c2ac935ded1",
    "state": null,
    "propertySources": [
        {
            "name": "D:\\trainings\\wipro\\batch2\\config-repo/application.properties",
            "source": {
                "message": "Hello, Rhoit how r you?"
            }
        }
    ]
}
```


Create Microservice #2(Microservice as a Config Client)

Step #1: Create a Spring Boot Project and the below dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

Step #2: Modify application.properties file

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

management.endpoints.web.exposure.include=*
```

Create a controller:

```java
@RestController
public class DemoController {

    @Value("${message:Hello default}")
    private String message;

    @GetMapping("/message")
    String getMessage() {
        System.out.println("Inside controller");
        return this.message;
    }
}
```

http://localhost:7072/message

you will see the message from application.properties file

to get the client specific values:

create <application-name>.properties file and commit into git local repo

for e.g if the client application name is config-client,

```
spring.application.name=config-client

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

management.endpoints.web.exposure.include=*
```

Create config-client.properties file and commit it.

```
spring.application.name=config-client

server.port = 7072

spring.datasource.url = jdbc:mysql://localhost:3306/coursedb
spring.datasource.username = root
spring.datasource.password = root123

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto= update

message= message from config-client specific service
```

If you send the request to config-client application end point.

http://localhost:7072/message

you will see the message from config-client.properties file

Also you can test this url:

localhost:8888/config-client/default


```
{
    "name": "config-client",
    "profiles": [
        "default"
    ],
    "label": null,
    "version": "f03e3c16ce010431327536e382b99c2ac935ded1",
    "state": null,
    "propertySources": [
        {
            "name": "D:\\trainings\\wipro\\batch2\\config-repo/config-client.properties",
            "source": {
                "spring.application.name": "config-client",
                "server.port": "7072",
                "spring.datasource.url": "jdbc:mysql://localhost:3306/coursedb",
                "spring.datasource.username": "root",
                "spring.datasource.password": "root123",
                "spring.jpa.show-sql": "true",
                "spring.jpa.hibernate.ddl-auto": "update",
                "message": "message from config-client specific service"
            }
        },
        {
            "name": "D:\\trainings\\wipro\\batch2\\config-repo/application.properties",
            "source": {
                "message": "Hello, Rhoit how r you?"
            }
        }
    ]
}
```


How can we get updated values from Config Server in microservices without restarting the application?

What is RefreshScope?

Spring Cloud Configuration Server lets developers to load the new configuration properties without restarting the application and without any downtime.

By default, the configuration values are read on the client's startup and not again. You can force a bean to refresh its configuration (that is, to pull updated values from the Config Server) by annotating the RestController with the Spring Cloud Config *@RefreshScope* and then triggering a refresh event.

```java
@RefreshScope
@RestController
public class DemoController {

    @Value("${message:Hello default}")
    private String message;

    @GetMapping("/message")
    String getMessage() {
        System.out.println("Inside controller");
        return this.message;
    }
}
```

How to test Config Server with Refresh Scope Enabled Application?

Step#1: Open Postman tool
Step#2: Change the value of property in GitHub and commit the changes.
Step#3: In the Postman tool, select method as 'POST', enter URL 'http://localhost:7072/actuator/refresh' and click on send button. You will receive 200 status code with some result as shown in the screen below.
Step#4: Open your browser and hit the actual URL(http://localhost:7072/message). If the browser with actual URL id already opened, just refresh the browser. You will see the updated value.