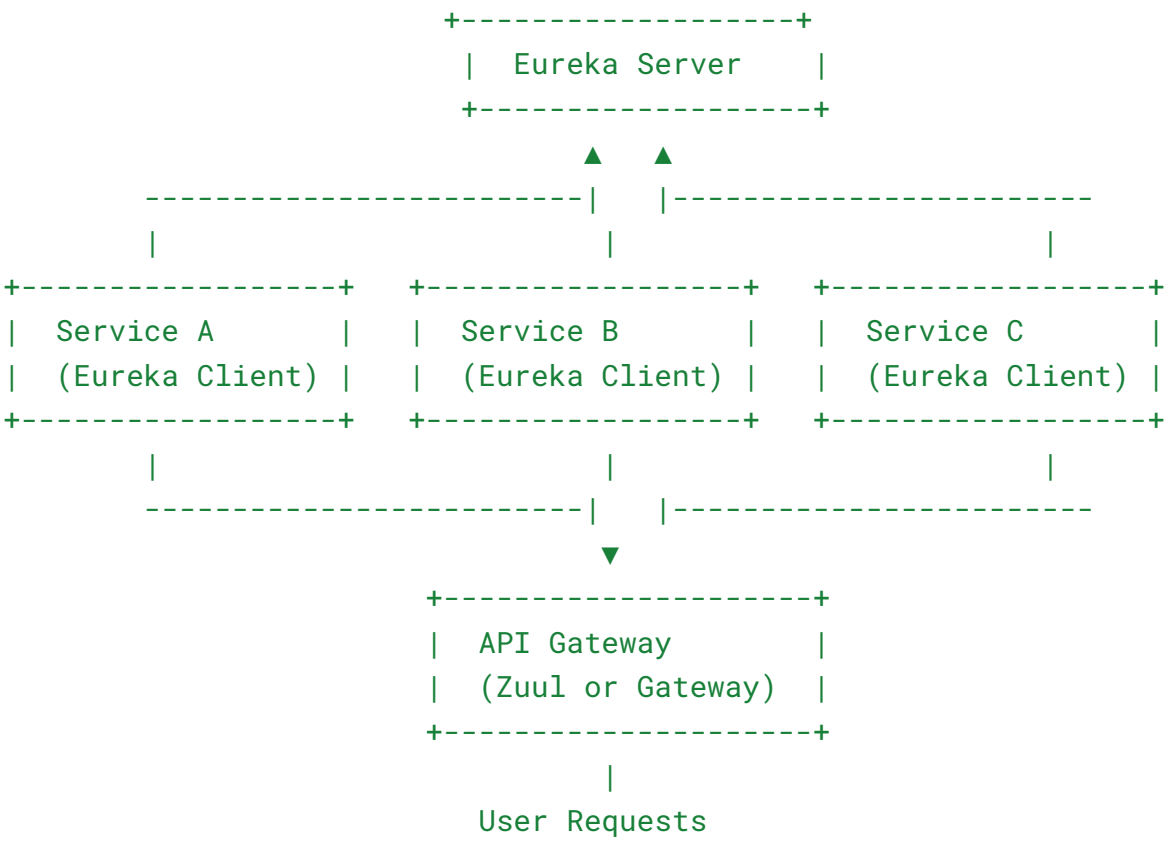
## 1. Overview of Components

1. **Spring Cloud Eureka** (Service Discovery)
    - Eureka Server acts as a registry where services register themselves.
    - Eureka Clients (microservices) fetch information about other services dynamically.
2. **Spring Cloud API Gateway** (Routing & Load Balancing)
    - It acts as a reverse proxy.
    - Routes requests to appropriate services dynamically.
    - Provides features like load balancing, rate limiting, security, etc.
3. **Spring Cloud Hystrix** (Circuit Breaker)
    - Monitors microservices calls.
    - If a service fails, it prevents cascading failures by falling back to alternative logic.
    - Helps in fault tolerance.

---

## 2. Architecture Diagram

pgsql
CopyEdit



---

## 3. Implementation Steps

### Step 1: Setup Eureka Server

Create a Spring Boot project for Eureka Server.

1. **Add Dependencies (`pom.xml`)**

xml
CopyEdit

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>
</dependencies>
```

2. **Enable Eureka Server (`EurekaServerApplication.java`)**

java
CopyEdit

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

3. **Eureka Server Configuration (`application.yml`)**

yaml
CopyEdit

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
```

```
    register-with-eureka: false
    fetch-registry: false
```

---

**Step 2: Setup Eureka Clients (Microservices)**

Create two microservices (`service-a`, `service-b`) and register them with Eureka.

1. **Add Dependencies (`pom.xml`)**

xml
CopyEdit

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

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

2. **Enable Eureka Client (`ServiceAApplication.java`)**

java
CopyEdit

```
@SpringBootApplication
@EnableEurekaClient
@RestController
public class ServiceAApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServiceAApplication.class, args);
    }

    @GetMapping("/message")
    public String message() {
        return "Hello from Service A";
    }
}
```

3. **Service Configuration (`application.yml`)**

yaml
CopyEdit

```
server:
  port: 8081
```

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

Repeat the above steps for `service-b` but change the port to `8082`.

---

**Step 3: Setup API Gateway**

Create a new Spring Boot project for API Gateway.

1. **Add Dependencies (`pom.xml`)**

xml
CopyEdit

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

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

2. **Enable API Gateway (`ApiGatewayApplication.java`)**

java
CopyEdit

```
@SpringBootApplication
@EnableEurekaClient
public class ApiGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
}
```

3. **API Gateway Configuration (`application.yml`)**

yaml

CopyEdit

```
server:
  port: 9000

spring:
  cloud:
    gateway:
      routes:
        - id: service-a
          uri: lb://SERVICE-A
          predicates:
            - Path=/service-a/**
        - id: service-b
          uri: lb://SERVICE-B
          predicates:
            - Path=/service-b/**

eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
```

- `lb://SERVICE-A` means it will fetch the service instance from Eureka dynamically.

Now, requests to:

- `http://localhost:9000/service-a/message` will be routed to `service-a`
- `http://localhost:9000/service-b/message` will be routed to `service-b`

---

**Step 4: Implement Hystrix Circuit Breaker**

Modify `ServiceAApplication` to use Hystrix.

1. **Add Hystrix Dependency (`pom.xml`)**

xml
CopyEdit

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

2. **Enable Hystrix (`ServiceAApplication.java`)**

java
CopyEdit

```java
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
@RestController
public class ServiceAApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServiceAApplication.class, args);
    }

    @GetMapping("/message")
    @HystrixCommand(fallbackMethod = "fallbackMessage")
    public String message() {
        throw new RuntimeException("Service A failed");
    }

    public String fallbackMessage() {
        return "Fallback response: Service A is down!";
    }
}
```

3. **Enable Hystrix in API Gateway (`application.yml`)**

yaml
CopyEdit

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: service-a
          uri: lb://SERVICE-A
          predicates:
            - Path=/service-a/**
          filters:
            - name: Hystrix
              args:
                name: fallback
                fallbackUri: forward:/fallback
```

```
server:
  port: 9000
```

4. **Create a Fallback Controller (`ApiGatewayApplication.java`)**

java
CopyEdit

```
@RestController
public class FallbackController {
    @GetMapping("/fallback")
    public String fallback() {
        return "API Gateway Fallback: Service is temporarily
unavailable!";
    }
}
```

---

### 4. Testing the Setup

1. **Start Eureka Server (`mvn spring-boot:run`)**
2. **Start `service-a` and `service-b`**
3. **Start API Gateway**
4. **Test Endpoints:**
    - `http://localhost:9000/service-a/message`
    - `http://localhost:9000/service-b/message`
    - Stop `service-a` and check the fallback response.

---

### 5. Conclusion

- **Eureka Server** registers services dynamically.
- **API Gateway** routes requests and balances load.
- **Hystrix** provides circuit-breaking for fault tolerance.

## Centralized Logging with ELK Stack & Distributed Log Tracing with Spring Cloud Sleuth & Zipkin

In a microservices architecture, logging is a crucial aspect for monitoring and debugging. Here, we'll explore:

1. **Centralized Logging using ELK (Elasticsearch, Logstash, Kibana) Stack**
2. **Distributed Tracing using Spring Cloud Sleuth & Zipkin**

---

# 1. Centralized Logging with ELK Stack

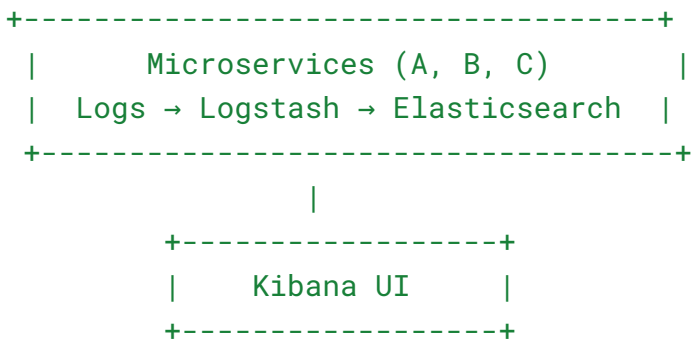
### Overview of ELK Stack

- **Elasticsearch**: Stores and indexes logs.
- **Logstash**: Collects, processes, and forwards logs to Elasticsearch.
- **Kibana**: Visualizes logs in dashboards.

---

## Architecture Diagram

lua
CopyEdit




---

## Steps to Set Up ELK

### Step 1: Setup Elasticsearch

**Download & Install Elasticsearch**
bash
CopyEdit

```
docker run --name elasticsearch -p 9200:9200 -e
"discovery.type=single-node"
docker.elastic.co/elasticsearch/elasticsearch:8.5.1
```

1.
2. **Verify Elasticsearch**
   Open browser: http://localhost:9200

---

## Step 2: Setup Logstash

**Create a `logstash.conf` file**

bash
CopyEdit

```
input {
    beats {
        port => 5044
    }
}

filter {
    json {
        source => "message"
    }
}

output {
    elasticsearch {
        hosts => ["http://localhost:9200"]
        index => "microservice-logs"
    }
    stdout { codec => rubydebug }
}
```

1.

**Run Logstash**

bash
CopyEdit

```
docker run --name logstash -p 5044:5044 -v
/path/to/logstash.conf:/usr/share/logstash/pipeline/logstash.conf
docker.elastic.co/logstash/logstash:8.5.1
```

2.

---

## Step 3: Setup Kibana

**Run Kibana**

bash
CopyEdit

```
docker run --name kibana -p 5601:5601
docker.elastic.co/kibana/kibana:8.5.1
```

1.
2. **Access Kibana**
   Open browser: http://localhost:5601
3. **Configure Index Pattern**
   - Go to **Stack Management** → **Index Patterns**
   - Create an index pattern: `microservice-logs-*`

---

## Step 4: Send Logs from Spring Boot Application

### Add Dependencies (`pom.xml`)

xml
CopyEdit

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>6.6</version>
</dependency>
```

### Configure Logback (`logback-spring.xml`)

xml
CopyEdit

```
<configuration>
    <appender name="logstash"
class="net.logstash.logback.appender.LogstashTcpSocketAppender">
        <destination>localhost:5044</destination>
        <encoder
class="net.logstash.logback.encoder.LogstashEncoder"/>
    </appender>

    <root level="info">
        <appender-ref ref="logstash"/>
    </root>
</configuration>
```

- Now, logs from microservices will be collected and indexed in **Elasticsearch**, and you can visualize them in **Kibana**.

---

# 2. Distributed Tracing using Spring Cloud Sleuth & Zipkin

### Why Use Sleuth & Zipkin?

- Tracks requests across multiple microservices.
- Helps identify performance bottlenecks.

---

## Architecture Diagram

less
CopyEdit

```
User → API Gateway → Service A → Service B → Database
        |              |
        |            Zipkin
        |
      Kibana (ELK)
```

---

## Steps to Implement Sleuth & Zipkin

### Step 1: Start Zipkin

Run Zipkin using Docker:

bash
CopyEdit

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

Access Zipkin UI: http://localhost:9411

---

### Step 2: Add Sleuth & Zipkin in Microservices

#### Add Dependencies (`pom.xml`)

xml
CopyEdit

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

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

---

### Step 3: Configure Sleuth & Zipkin (`application.yml`)

yaml
CopyEdit

```
spring:
  zipkin:
    base-url: http://localhost:9411
  sleuth:
    sampler:
      probability: 1.0  # 100% sampling for all requests
```

- Every request is now **traced** and sent to Zipkin.

---

### Step 4: Add Tracing to Microservice Controller

java
CopyEdit

```
@RestController
@RequestMapping("/service-a")
public class ServiceAController {

    private final RestTemplate restTemplate;

    public ServiceAController(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @GetMapping("/call-b")
    public String callServiceB() {
        return
restTemplate.getForObject("http://service-b/service-b/data",
String.class);
    }
```

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

- Every request now has a **trace ID**.
- These logs will be visible in **Zipkin & Kibana**.

---

## Final Testing

### Test API Calls & View Tracing

**Make a request**
bash
CopyEdit
```
curl http://localhost:8081/service-a/call-b
```

1. 
2. **View Trace in Zipkin UI**
    - Open: http://localhost:9411
    - See how requests flow across microservices.

---

# Conclusion

✅ **Centralized Logging with ELK** → Collect, store & visualize logs.
✅ **Distributed Tracing with Sleuth & Zipkin** → Monitor microservice request flows.

Would you like to add security (e.g., Log masking, JWT tracing)? 🚀

## Managing Spring Boot Applications Locally with Trampoline, Spring Boot & Spring Cloud Config

### What is Trampoline?

**Trampoline** is a tool for managing and running multiple **Spring Boot microservices** locally. It helps developers to:

- Start, stop, and monitor multiple microservices easily.

- Manage microservices dependencies.
- View logs and statuses in one place.

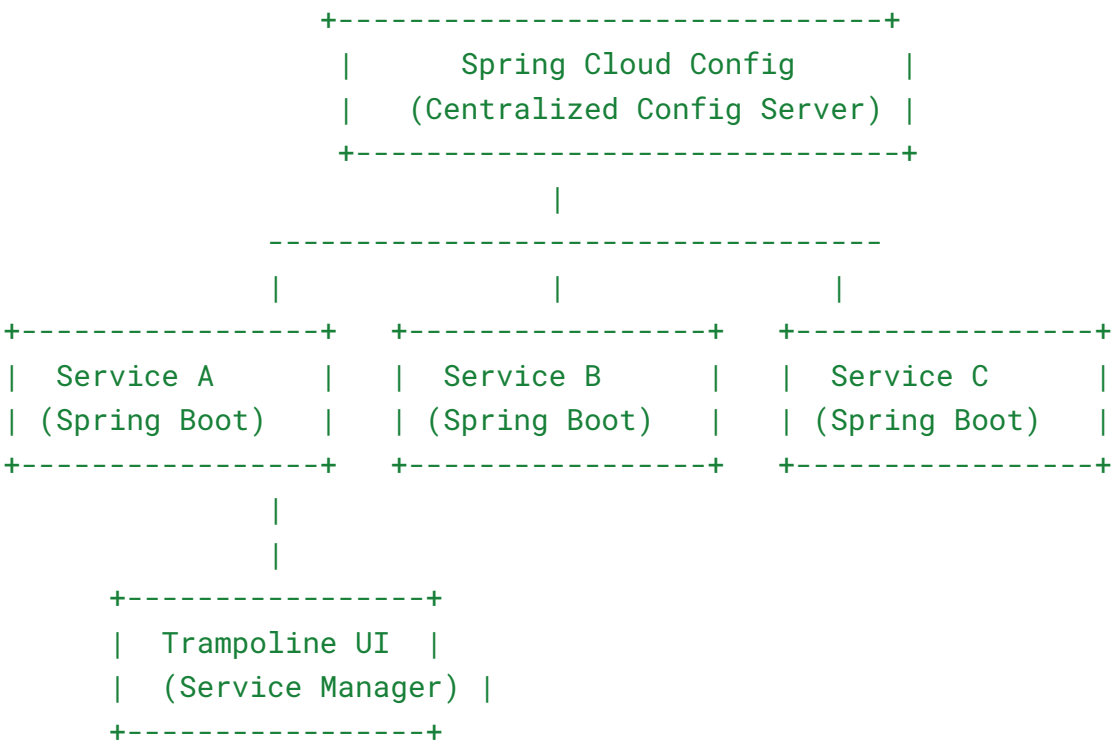
### What is Spring Cloud Config?

**Spring Cloud Config** provides a **centralized configuration** for microservices. Instead of managing individual `application.yml` files in each service, all configurations are stored in a **Config Server** (which can use Git, database, or local storage as the source).

---

## Architecture Diagram

pgsql
CopyEdit




---

# Implementation Steps

## 1. Setting Up Trampoline

Trampoline helps in managing local microservices efficiently.

### Step 1.1: Clone Trampoline

bash
CopyEdit
```
git clone https://github.com/ines/trampoline.git
cd trampoline
```

### Step 1.2: Run Trampoline

bash
CopyEdit
```
./mvnw spring-boot:run
```

> Trampoline will start and provide a **UI to manage microservices**.

---

## 2. Setting Up Spring Cloud Config Server

Spring Cloud Config Server will provide centralized configuration.

### Step 2.1: Create a New Spring Boot Application (`config-server`)

1. **Add Dependencies (`pom.xml`)**

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

2. **Enable Config Server (`ConfigServerApplication.java`)**

java
CopyEdit
```
@SpringBootApplication
@EnableConfigServer
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
```

```
    }
}
```

3. **Configure `application.yml`**

yaml
CopyEdit

```
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-repo/config-repo # Change to
your Git repo
```

4. **Run the Config Server**

bash
CopyEdit

```
mvn spring-boot:run
```

The **Config Server** is now running on `http://localhost:8888`.

---

# 3. Setting Up a Sample Microservice (`service-a`)

## Step 3.1: Create a New Spring Boot Application (`service-a`)

1. **Add Dependencies (`pom.xml`)**

xml
CopyEdit

```
<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>
```

2. **Use `bootstrap.yml` to Load Configurations from Config Server**

yaml
CopyEdit

```
spring:
  application:
    name: service-a
  cloud:
    config:
      uri: http://localhost:8888
```

This tells `service-a` to fetch its configuration from the **Config Server**.

3. **Add Controller (`ServiceAController.java`)**

java
CopyEdit

```
@RestController
@RequestMapping("/service-a")
@RefreshScope
public class ServiceAController {

    @Value("${message: Default Message}")
    private String message;

    @GetMapping("/config")
    public String getConfig() {
        return "Message from Config Server: " + message;
    }
}
```

`@RefreshScope` ensures that configuration updates take effect **without restarting** the service.

---

# 4. Storing Configurations in Git

Create a repository like **`config-repo`** with the following structure:

css
CopyEdit

```
config-repo/
├── service-a.yml
├── service-b.yml
├── application.yml
```

### Sample `service-a.yml`

yaml
CopyEdit

```yaml
message: "Hello from Config Server!"
```

---

## 5. Running Services with Trampoline

1. **Modify Trampoline Config (`trampoline.yml`)**

yaml
CopyEdit

```yaml
services:
  - name: config-server
    path: ./config-server
    command: mvn spring-boot:run

  - name: service-a
    path: ./service-a
    command: mvn spring-boot:run
```

2. **Start All Services**

bash
CopyEdit

```
./mvnw spring-boot:run
```

> Trampoline will now **start and manage all services**.

---

# 6. Testing the Setup

**Check if Config Server is Running**

bash
CopyEdit
```
curl http://localhost:8888/service-a/default
```
Output:

json
CopyEdit
```
{
   "propertySources": [
      {
         "source": {
            "message": "Hello from Config Server!"
         }
      }
   ]
}
```

1.

**Test `service-a` Endpoint**

bash
CopyEdit
```
curl http://localhost:8081/service-a/config
```
Output:

pgsql
CopyEdit
```
Message from Config Server: Hello from Config Server!
```

2.

**Update Configuration in Git (`service-a.yml`)**

yaml
CopyEdit
```
message: "Updated message from Config Server!"
```

3. Commit & Push the changes.

**Refresh Configuration Without Restarting**

bash
CopyEdit
```
curl -X POST http://localhost:8081/actuator/refresh
```
Now, **re-run** the test:

bash
CopyEdit
```
curl http://localhost:8081/service-a/config
```
Output:

pgsql
CopyEdit
```
Message from Config Server: Updated message from Config Server!
```

4.

---

# Final Thoughts

✅ **Trampoline** → Manages multiple microservices easily.
✅ **Spring Cloud Config** → Centralized Configuration for Microservices.
✅ **Live Updates** → No need to restart services for config changes.

# Resilience4J Circuit Breaker Implementation in Spring Boot

## What is Resilience4J?

**Resilience4J** is a lightweight, Java 8+ library designed for building **fault-tolerant applications**. It provides various resilience patterns, such as:
✅ **Circuit Breaker**
✅ **Retry**
✅ **Rate Limiter**
✅ **Bulkhead**
✅ **Time Limiter**

---

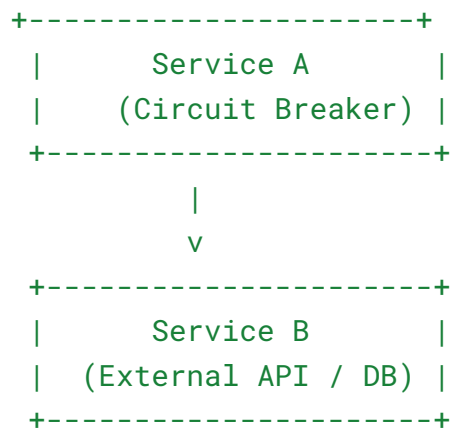
## What is a Circuit Breaker?

A **Circuit Breaker** prevents a system from **making requests to a failing service**, allowing time for recovery.
It has three states:

1. **Closed** → Requests pass normally.
2. **Open** → Requests fail immediately to prevent overload.
3. **Half-Open** → Some requests are tested to check if the service is back.

**Diagram:**

sql
CopyEdit




---

# Implementation Steps

## 1. Create a Spring Boot Application

Generate a Spring Boot project using **Spring Initializr** with dependencies: ✅ **Spring Web**
✅ **Spring Boot Actuator**
✅ **Resilience4j Spring Boot**

---

## 2. Add Dependencies (`pom.xml`)

xml
CopyEdit

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

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

<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifa
ctId>
</dependency>

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

---

## 3. Implement Circuit Breaker in a Service

### Create a REST Client (`ExternalService.java`)

java
CopyEdit

```
@Service
public class ExternalService {

    private final RestTemplate restTemplate;

    public ExternalService(RestTemplateBuilder builder) {
        this.restTemplate = builder.build();
    }

    @CircuitBreaker(name = "externalService", fallbackMethod =
"fallbackResponse")
    public String callExternalService() {
        return restTemplate.getForObject("http://slow-api.com/data",
String.class);
    }

    public String fallbackResponse(Exception e) {
        return "Service is currently unavailable. Please try again
later.";
    }
}
```

`@CircuitBreaker` will break the circuit if `http://slow-api.com/data` fails multiple times.

---

## 4. Create a REST Controller (`MyController.java`)

java
CopyEdit

```
@RestController
@RequestMapping("/api")
public class MyController {

    @Autowired
    private ExternalService externalService;

    @GetMapping("/fetch")
    public String fetchData() {
        return externalService.callExternalService();
    }
}
```

Calling `/api/fetch` will invoke the **circuit breaker**.

---

## 5. Configure Resilience4J (`application.yml`)

yaml
CopyEdit

```
resilience4j:
  circuitbreaker:
    instances:
      externalService:
        failureRateThreshold: 50  # Circuit opens if 50% of requests
fail
        waitDurationInOpenState: 5000ms  # 5 sec wait before trying
again
        permittedNumberOfCallsInHalfOpenState: 3  # Allow 3 test
calls
        slidingWindowSize: 10  # Monitor last 10 calls
```

**If 50% of requests fail, the circuit breaker will open and block further calls for 5 seconds.**

---

## 6. Enable Actuator Endpoints

Modify `application.yml`:

yaml
CopyEdit

```yaml
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

**Monitor Circuit Breaker:**

bash
CopyEdit

```bash
GET http://localhost:8080/actuator/circuitbreakers
```

**Response Example:**

json
CopyEdit

```json
{
    "externalService": {
        "state": "CLOSED",
        "failureRate": 0.0
    }
}
```

---

# 7. Testing the Circuit Breaker

### Case 1: Service is Available

bash

CopyEdit

```
GET http://localhost:8080/api/fetch
Response: "Data from Service B"
```

### Case 2: Service Fails Multiple Times

bash
CopyEdit

```
GET http://localhost:8080/api/fetch
Response: "Service is currently unavailable. Please try again
later."
```

Check Circuit Breaker state:

bash
CopyEdit

```
GET http://localhost:8080/actuator/circuitbreakers/externalService
Response: { "state": "OPEN" }
```

---

# Conclusion

✅ **Resilience4J Circuit Breaker** prevents excessive failures.
✅ **Fallback Method** ensures graceful degradation.
✅ **Actuator Monitoring** helps track circuit state.

# Resilience4J Retry & Rate Limiter Implementation in Spring Boot

Now, let's extend the **Circuit Breaker** with **Retry** and **Rate Limiter** mechanisms using **Resilience4J**.

---

# 1️⃣ What is Resilience4J Retry?

The **Retry** module automatically retries failed requests before giving up.

- **Example:** If an API request fails, it will **retry** a few times before returning an error.

**Retry Flow:**

sql
CopyEdit

```
          +--------------------+
          |   First Attempt    |
          +--------------------+
                    ↓   (Fail)
          +--------------------+
          |   Retry #1         |
          +--------------------+
                    ↓   (Fail)
          +--------------------+
          |   Retry #2         |
          +--------------------+
                    ↓   (Fail)
      Circuit Breaker Takes Over
```

---

# 2️⃣ What is Resilience4J Rate Limiter?

The **Rate Limiter** ensures that a service **does not get overloaded** by limiting the number of requests per second.

- **Example:** If a user makes too many requests in a short time, the system will reject additional requests.

**Rate Limiter Flow:**

scss
CopyEdit

```
User Requests → Allowed? (Yes) → Process Request
              → Allowed? (No)  → Reject Request (HTTP 429)
```

---

# 3️⃣ Implementation Steps

We'll extend our **Spring Boot application** from the **Circuit Breaker** example and add **Retry** and **Rate Limiter**.

---

## 🔷 Step 1: Add Dependencies (`pom.xml`)

xml
CopyEdit

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

<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifa
ctId>
</dependency>
```

> Already included in the **Circuit Breaker** setup.

---

## 🔷 Step 2: Implement Retry and Rate Limiter in Service

Modify the existing `ExternalService.java` to add `@Retry` and `@RateLimiter`.

java
CopyEdit

```
@Service
public class ExternalService {

    private final RestTemplate restTemplate;

    public ExternalService(RestTemplateBuilder builder) {
        this.restTemplate = builder.build();
    }

    // Circuit Breaker + Retry + Rate Limiter
    @CircuitBreaker(name = "externalService", fallbackMethod =
"fallbackResponse")
    @Retry(name = "externalService", fallbackMethod =
"fallbackResponse")
    @RateLimiter(name = "externalService")
    public String callExternalService() {
        System.out.println("Calling external service...");
        return restTemplate.getForObject("http://slow-api.com/data",
String.class);
    }
```

```
    public String fallbackResponse(Exception e) {
        return "Service is temporarily unavailable. Please try again
later.";
    }
}
```

---

## 🔷 Step 3: Configure Retry & Rate Limiter (`application.yml`)

yaml
CopyEdit

```
resilience4j:
  circuitbreaker:
    instances:
      externalService:
        failureRateThreshold: 50  # Circuit opens if 50% of requests
fail
        waitDurationInOpenState: 5000ms  # Wait 5s before retrying
        permittedNumberOfCallsInHalfOpenState: 3  # Allow 3 test
calls
        slidingWindowSize: 10  # Monitor last 10 calls

  retry:
    instances:
      externalService:
        maxAttempts: 3  # Retry 3 times before failing
        waitDuration: 2000ms  # Wait 2 sec between retries

  ratelimiter:
    instances:
      externalService:
        limitForPeriod: 5  # Max 5 calls in a period
        limitRefreshPeriod: 10s  # Reset limit every 10 seconds
        timeoutDuration: 500ms  # Wait 500ms before rejecting
```

---

## 🔷 Step 4: Create Controller to Test Retry & Rate Limiter

Modify `MyController.java`:

java
CopyEdit

```java
@RestController
@RequestMapping("/api")
public class MyController {

    @Autowired
    private ExternalService externalService;

    @GetMapping("/fetch")
    public String fetchData() {
        return externalService.callExternalService();
    }
}
```

---

# 4️⃣ Testing the Features

## ✅ Test Retry

**Start the service:**

bash
CopyEdit

```bash
mvn spring-boot:run
```

1.

**Call the API:**

bash
CopyEdit

```bash
curl http://localhost:8080/api/fetch
```

2.
    - If the first call fails, **it will retry up to 3 times** before falling back.

Check logs to see retry attempts:
kotlin
CopyEdit

```kotlin
Calling external service...
Calling external service...
Calling external service...
Service is temporarily unavailable. Please try again later.
```

- 

---

## ✅ Test Rate Limiter

**Make 6 requests quickly:**

bash
CopyEdit

```
for i in {1..6}; do curl -s http://localhost:8080/api/fetch & done
```

1. 
    - **First 5 calls will pass** (as per the limit).
    - **6th request will fail** with HTTP **429 Too Many Requests**.

---

## ✅ Check Circuit Breaker & Retry Metrics

**Get Circuit Breaker Status**

bash
CopyEdit

```
curl http://localhost:8080/actuator/circuitbreakers/externalService
```

Example response:

json
CopyEdit

```
{
   "state": "OPEN",
   "failureRate": 60.0
}
```

1. 

**Check Retry Metrics**

bash
CopyEdit

```
curl http://localhost:8080/actuator/retries/externalService
```

Example response:

json
CopyEdit

```
{
   "retryMetrics": {
      "name": "externalService",
      "numberOfFailedCallsWithRetryAttempt": 2,
```

```
        "numberOfSuccessfulCallsWithoutRetryAttempt": 3
    }
}
```

2.

---

# 🎯 Conclusion

✅ **Circuit Breaker** protects against failing services.
✅ **Retry** automatically retries failed requests.
✅ **Rate Limiter** prevents excessive API requests.
✅ **Actuator Monitoring** helps track system performance.

# Securing Microservices with API Key-Based Authentication using Spring Cloud Gateway

In this guide, we will implement **API Key-based authentication** in **Spring Cloud Gateway** to secure **microservices**.

---

## 1️⃣ What is API Key Authentication?

API Key authentication is a simple way to **restrict access to APIs** by requiring clients to include a unique **API Key** in their requests.

- API Keys are **generated** and **shared** with trusted clients.
- Every incoming request must contain a **valid API Key** in the **header**.
- If the API Key is missing or invalid, the request is **rejected**.

---

## 2️⃣ Architecture Diagram

pgsql
CopyEdit

```
  Client (Postman, Frontend App)
              |
              |  (API Key in Header)
```

```
          |
          v
  +----------------------+
  |  Spring Cloud Gateway  |
  | (Validates API Key)  |
  +----------------------+
          |
     +------------+
     | Microservices |
     +------------+
```

---

# 3️⃣ Implementation Steps

We'll configure **Spring Cloud Gateway** to: ✅ **Intercept API requests**
✅ **Check for an API Key**
✅ **Forward requests only if the API Key is valid**

---

## 🔷 Step 1: Create a Spring Cloud Gateway Project

Use **Spring Initializr** and add dependencies:

- **Spring Boot Starter Web**
- **Spring Cloud Gateway**
- **Spring Boot Starter Security**

Add these dependencies in `pom.xml`:

xml
CopyEdit

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

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

---

### 🔷 Step 2: Define Routes in `application.yml`

Configure **Spring Cloud Gateway** to forward requests.

yaml
CopyEdit

```yaml
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8081/
          predicates:
            - Path=/users/**
          filters:
            - name: ApiKeyAuthFilter
```

Requests to `/users/**` will be validated using **API Key authentication**.

---

### 🔷 Step 3: Implement API Key Filter

Create a custom filter `ApiKeyAuthFilter.java` to validate API keys.

java
CopyEdit

```java
package com.example.gateway.filter;

import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
public class ApiKeyAuthFilter implements GlobalFilter {
```

```
    private static final String API_KEY_HEADER = "X-API-KEY";
    private static final String VALID_API_KEY = "my-secret-api-key";

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        HttpHeaders headers = exchange.getRequest().getHeaders();

        if (!headers.containsKey(API_KEY_HEADER)) {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        String apiKey = headers.getFirst(API_KEY_HEADER);
        if (!VALID_API_KEY.equals(apiKey)) {

exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
            return exchange.getResponse().setComplete();
        }

        return chain.filter(exchange);
    }
}
```

✅ **API Key is validated** before forwarding the request.
✅ If the key is **missing or invalid**, return **401 Unauthorized** or **403 Forbidden**.

---

### 🔷 Step 4: Create a Sample Microservice

Create a **User Service (`user-service`)** that **Spring Cloud Gateway** will forward requests to.

**`UserController.java`**
java
CopyEdit

```
@RestController
@RequestMapping("/users")
public class UserController {
```

```
    @GetMapping("/profile")
    public ResponseEntity<String> getUserProfile() {
        return ResponseEntity.ok("User profile data");
    }
}
```

✅ **This service runs on `localhost:8081`** and responds to `/users/profile`.

---

# 5️⃣ Testing API Key Authentication

## ✅ Test with Correct API Key

bash
CopyEdit
```
curl -H "X-API-KEY: my-secret-api-key"
http://localhost:8080/users/profile
```

**Response:**

kotlin
CopyEdit
```
User profile data
```

## ❌ Test with Missing API Key

bash
CopyEdit
```
curl http://localhost:8080/users/profile
```

**Response:**

CopyEdit
```
401 Unauthorized
```

## ❌ Test with Invalid API Key

bash
CopyEdit
```
curl -H "X-API-KEY: wrong-key" http://localhost:8080/users/profile
```

**Response:**

CopyEdit

```
403 Forbidden
```

---

## 6️⃣ Conclusion

✅ **Spring Cloud Gateway** filters API requests.
✅ **API Key authentication** prevents unauthorized access.
✅ **Requests without a valid API Key are rejected**.
✅ **Microservices remain secure behind the API Gateway**.

# Microservices Security Using JWT with Spring Cloud Gateway

In this guide, we will enhance our **Spring Cloud Gateway** security by using **JWT (JSON Web Token)** for authentication and authorization.

---

## 1️⃣ What is JWT Authentication?

JWT (JSON Web Token) is a **secure, compact token** used to:

- **Authenticate users** in a microservices architecture.
- **Authorize API requests** by including a signed JWT in the request headers.
- **Verify user roles & permissions** across microservices.

### 🔷 JWT Flow in Microservices

pgsql
CopyEdit

```
1. User logs in and sends credentials → Auth Service
2. Auth Service validates user & generates JWT token
3. User includes JWT in Authorization header → API Gateway
4. API Gateway verifies JWT & forwards request → Microservices
```

---

## 2️⃣ Architecture Diagram

sql

CopyEdit

```
  Client (React, Postman, Mobile App)
              |
        (JWT Token)
              |
              v
   +----------------------+
   |  Spring Cloud Gateway |
   | (Validates JWT)       |
   +----------------------+
              |
      +------------+   +------------+
      | User Service|   | Order Service |
      +------------+   +------------+
```

---

# 3️⃣ Implementation Steps

## 🔷 Step 1: Add Dependencies (`pom.xml`)

Add JWT & Security dependencies:

xml
CopyEdit

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

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.11.5</version>
</dependency>

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

---

### 🔷 Step 2: Configure API Gateway (`application.yml`)

yaml
CopyEdit

```yaml
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8081/
          predicates:
            - Path=/users/**
          filters:
            - AuthenticationFilter
```

✅ **API Gateway routes requests to User Service (`/users/**`) only if JWT is valid.**

---

### 🔷 Step 3: Implement JWT Utility Class

Create `JwtUtil.java` to **generate and validate JWT tokens**.

java
CopyEdit

```java
package com.example.gateway.util;

import io.jsonwebtoken.*;
import io.jsonwebtoken.security.Keys;
import org.springframework.stereotype.Component;

import java.security.Key;
import java.util.Date;

@Component
public class JwtUtil {

    private static final String SECRET_KEY =
"MySuperSecretKeyForJwtMySuperSecretKey";
```

```
    private static final long EXPIRATION_TIME = 1000 * 60 * 60; // 1
hour

    private Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes());

    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME))
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }

    public boolean validateToken(String token) {
        try {

Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token
);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    public String extractUsername(String token) {
        return Jwts.parserBuilder().setSigningKey(key).build()
                .parseClaimsJws(token).getBody().getSubject();
    }
}
```

✅ **Generates & validates JWT tokens**.
✅ **Extracts the username from the token**.

---

## 🔷 Step 4: Implement JWT Authentication Filter

Create `AuthenticationFilter.java` to **validate JWT in API Gateway**.

java

CopyEdit

```
package com.example.gateway.filter;

import com.example.gateway.util.JwtUtil;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
public class AuthenticationFilter implements GlobalFilter {

    @Autowired
    private JwtUtil jwtUtil;

    private static final String AUTH_HEADER = "Authorization";
    private static final String TOKEN_PREFIX = "Bearer ";

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        HttpHeaders headers = exchange.getRequest().getHeaders();

        if (!headers.containsKey(AUTH_HEADER)) {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        String token = headers.getFirst(AUTH_HEADER);
        if (token == null || !token.startsWith(TOKEN_PREFIX)) {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        token = token.replace(TOKEN_PREFIX, "");
```

```
        if (!jwtUtil.validateToken(token)) {

exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
            return exchange.getResponse().setComplete();
        }

        return chain.filter(exchange);
    }
}
```

✅ **Extracts JWT from the Authorization header**.
✅ **Validates JWT token before forwarding the request**.
✅ **Returns `401 Unauthorized` if the token is missing or invalid**.

---

## 🔷 Step 5: Implement Authentication Service

Create an `AuthController.java` in a separate **Auth Service** to generate JWT.

java
CopyEdit

```
@RestController
@RequestMapping("/auth")
public class AuthController {

    @Autowired
    private JwtUtil jwtUtil;

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestParam String
username) {
        String token = jwtUtil.generateToken(username);
        return ResponseEntity.ok(token);
    }
}
```

✅ **User sends a username, and the system generates a JWT**.

---

## 🔷 Step 6: Protect Microservices

Modify **User Service (`user-service`)** to extract JWT.

java
CopyEdit

```
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/profile")
    public ResponseEntity<String>
getUserProfile(@RequestHeader("Authorization") String token) {
        String username = new
JwtUtil().extractUsername(token.replace("Bearer ", ""));
        return ResponseEntity.ok("User Profile Data for: " +
username);
    }
}
```

✅ **Extracts username from JWT before serving the request**.

---

# 7️⃣ Testing JWT Authentication

## ✅ 1. Get JWT Token

bash
CopyEdit

```
curl -X POST "http://localhost:8080/auth/login?username=john"
```

Response:

CopyEdit

```
eyJhbGciOiJIUzI1NiIsInR5cCI...
```

---

## ✅ 2. Use JWT Token for API Request

bash
CopyEdit

```
curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI..."
http://localhost:8080/users/profile
```

Response:

sql
CopyEdit
```
User Profile Data for: john
```

---

### ❌ 3. Missing JWT

bash
CopyEdit
```
curl http://localhost:8080/users/profile
```

Response:

CopyEdit
```
401 Unauthorized
```

---

## 8️⃣ Conclusion

✅ **Spring Cloud Gateway validates JWT before forwarding requests**
✅ **Microservices are protected from unauthorized access**
✅ **JWT-based authentication ensures scalability & security**

# Spring Cloud API Gateway | JWT Security | Passing User Details to Microservices

In this guide, we will:
✅ Implement **JWT Authentication** in **Spring Cloud API Gateway**
✅ **Validate JWT** in API Gateway
✅ **Pass UserDetails** (username, roles) to Microservices

---

## 1️⃣ Architecture Overview

markdown
CopyEdit
```
1. User logs in → Auth Service generates JWT
```

```
2. User sends JWT → API Gateway validates it
3. API Gateway extracts UserDetails → Passes to Microservices
4. Microservice extracts UserDetails from request headers
```

### Architecture Diagram

sql
CopyEdit

```
Client (Postman, React, Mobile App)
             |
      (JWT Token)
             |
             v
   +----------------------+
   |  Spring Cloud Gateway |
   | (Validates JWT)       |
   | (Passes UserDetails)  |
   +----------------------+
             |
      +------------+   +------------+
      | User Service|   | Order Service |
      +------------+   +------------+
```

---

## 2️⃣ Implementation Steps

### 🔷 Step 1: Add Dependencies (`pom.xml`)

xml
CopyEdit

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

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.11.5</version>
</dependency>

<dependency>
```

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

---

### 🔷 Step 2: Configure API Gateway (`application.yml`)

yaml
CopyEdit

```
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8081/
          predicates:
            - Path=/users/**
          filters:
            - AuthenticationFilter
```

✅ **API Gateway validates JWT and passes UserDetails to Microservices**.

---

### 🔷 Step 3: Create JWT Utility Class

Create `JwtUtil.java` to **generate, validate JWT & extract UserDetails**.

java
CopyEdit

```
package com.example.gateway.util;

import io.jsonwebtoken.*;
import io.jsonwebtoken.security.Keys;
import org.springframework.stereotype.Component;

import java.security.Key;
import java.util.Date;
import java.util.List;
import java.util.stream.Collectors;
```

```
@Component
public class JwtUtil {

    private static final String SECRET_KEY =
"MySuperSecretKeyForJwtMySuperSecretKey";
    private static final long EXPIRATION_TIME = 1000 * 60 * 60; // 1
hour
    private Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes());

    public String generateToken(String username, List<String> roles)
{
        return Jwts.builder()
                .setSubject(username)
                .claim("roles", roles)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() +
EXPIRATION_TIME))
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }

    public boolean validateToken(String token) {
        try {

Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token
);
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    public String extractUsername(String token) {
        return Jwts.parserBuilder().setSigningKey(key).build()
                .parseClaimsJws(token).getBody().getSubject();
    }

    public List<String> extractRoles(String token) {
        return ((List<?>)
Jwts.parserBuilder().setSigningKey(key).build()
```

```
.parseClaimsJws(token).getBody().get("roles")).stream()
                .map(String::valueOf)
                .collect(Collectors.toList());
    }
}
```

✅ **Stores username & roles in JWT**.
✅ **Extracts UserDetails from JWT**.

---

## 🔷 Step 4: Implement Authentication Filter in API Gateway

Modify `AuthenticationFilter.java` to **pass UserDetails to Microservices**.

java
CopyEdit

```
package com.example.gateway.filter;

import com.example.gateway.util.JwtUtil;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

import java.util.List;

@Component
public class AuthenticationFilter implements GlobalFilter {

    @Autowired
    private JwtUtil jwtUtil;

    private static final String AUTH_HEADER = "Authorization";
    private static final String TOKEN_PREFIX = "Bearer ";
```

```
    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        HttpHeaders headers = exchange.getRequest().getHeaders();

        if (!headers.containsKey(AUTH_HEADER)) {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        String token = headers.getFirst(AUTH_HEADER);
        if (token == null || !token.startsWith(TOKEN_PREFIX)) {

exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        token = token.replace(TOKEN_PREFIX, "");
        if (!jwtUtil.validateToken(token)) {

exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
            return exchange.getResponse().setComplete();
        }

        // Extract UserDetails
        String username = jwtUtil.extractUsername(token);
        List<String> roles = jwtUtil.extractRoles(token);

        // Add UserDetails to headers
        exchange.getRequest().mutate()
                .header("X-User-Name", username)
                .header("X-User-Roles", String.join(",", roles))
                .build();

        return chain.filter(exchange);
    }
}
```

✅ **Extracts username & roles from JWT**.
✅ **Adds UserDetails to request headers** before forwarding.

---

### 🔷 Step 5: Create Authentication Service (`AuthController.java`)

java
CopyEdit

```
@RestController
@RequestMapping("/auth")
public class AuthController {

    @Autowired
    private JwtUtil jwtUtil;

    @PostMapping("/login")
    public ResponseEntity<String> login(@RequestParam String
username) {
        List<String> roles = List.of("ROLE_USER");
        String token = jwtUtil.generateToken(username, roles);
        return ResponseEntity.ok(token);
    }
}
```

✅ **Generates JWT with username & roles**.

---

### 🔷 Step 6: Extract UserDetails in Microservices

Modify `UserController.java` in **User Service (`user-service`)**.

java
CopyEdit

```
@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/profile")
    public ResponseEntity<String> getUserProfile(
            @RequestHeader("X-User-Name") String username,
            @RequestHeader("X-User-Roles") String roles) {

        return ResponseEntity.ok("User Profile for " + username + "
with roles: " + roles);
```

```
    }
}
```

✅ **Extracts UserDetails from headers**.

---

## 7️⃣ Testing JWT Authentication & Passing UserDetails

### ✅ 1. Get JWT Token

bash
CopyEdit

```
curl -X POST "http://localhost:8080/auth/login?username=john"
```

Response:

CopyEdit

```
eyJhbGciOiJIUzI1NiIsInR5cCI...
```

---

### ✅ 2. Use JWT Token for API Request

bash
CopyEdit

```
curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI..."
http://localhost:8080/users/profile
```

Response:

sql
CopyEdit

```
User Profile for john with roles: ROLE_USER
```

---

### ❌ 3. Missing JWT

bash
CopyEdit

```
curl http://localhost:8080/users/profile
```

Response:

CopyEdit

```
401 Unauthorized
```

---

## 8️⃣ Conclusion

✅ **API Gateway validates JWT & extracts UserDetails**
✅ **UserDetails (username, roles) are passed to Microservices**
✅ **Microservices extract UserDetails for authorization**

# Secure Microservices with Keycloak & Spring Cloud Gateway

In this guide, we will:
✅ **Integrate Keycloak** for centralized authentication
✅ **Secure Spring Cloud Gateway** with Keycloak
✅ **Pass UserDetails (username, roles) to Microservices**

---

## 1️⃣ What is Keycloak?

**Keycloak** is an open-source **Identity & Access Management (IAM)** tool that provides:
✅ **User authentication** using OAuth2 / OpenID Connect
✅ **Role-based access control (RBAC)**
✅ **Token-based security (JWT)**

### 🔷 Keycloak Flow with Microservices

markdown
CopyEdit

```
1. User logs in → Keycloak issues JWT
2. User sends JWT → API Gateway validates token
3. API Gateway extracts UserDetails → Passes to Microservices
4. Microservices use UserDetails for authorization
```

---

## 2️⃣ Architecture Diagram

sql
CopyEdit

```
Client (React, Postman, Mobile App)
```

```
          |
    (JWT Token)
          |
          v
  +----------------------+
  |  Spring Cloud Gateway |
  | (Validates Keycloak JWT) |
  | (Passes UserDetails)  |
  +----------------------+
          |
     +------------+   +------------+
     | User Service|   | Order Service |
     +------------+   +------------+
```

---

# 3️⃣ Setting Up Keycloak

## 🔷 Step 1: Run Keycloak using Docker

bash
CopyEdit

```
docker run -d -p 8081:8080 --name keycloak \
  -e KEYCLOAK_ADMIN=admin \
  -e KEYCLOAK_ADMIN_PASSWORD=admin \
  quay.io/keycloak/keycloak:latest start-dev
```

✅ **Runs Keycloak on `http://localhost:8081`**
✅ **Admin Login → `admin / admin`**

---

## 🔷 Step 2: Configure Keycloak Realm & Client

1️⃣ **Login to Keycloak Admin Console** `http://localhost:8081/admin/`
2️⃣ **Create Realm:** `myrealm`
3️⃣ **Create Client:** `api-gateway`

- **Client Type:** OpenID Connect
- **Access Type:** Public
- **Valid Redirect URIs:** `http://localhost:8080/*`

4️⃣ **Create Roles:** `ROLE_USER`, `ROLE_ADMIN`
5️⃣ **Create User:**

- Username: `john`
- Password: `password`
- Assign role: `ROLE_USER`

---

## 4️⃣ Secure Spring Cloud API Gateway with Keycloak

### 🔷 Step 1: Add Dependencies (`pom.xml`)

xml
CopyEdit

```
<dependency>
    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>

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

---

### 🔷 Step 2: Configure Keycloak in API Gateway (`application.yml`)

yaml
CopyEdit

```
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8082/
          predicates:
            - Path=/users/**
          filters:
            - RemoveRequestHeader=Authorization

spring:
```

```
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/myrealm
```

✅ **API Gateway validates JWT from Keycloak**
✅ **Removes Authorization header before forwarding**

---

### 🔷 Step 3: Extract UserDetails & Pass to Microservices

Modify `AuthenticationFilter.java` in **API Gateway**.

java
CopyEdit

```
package com.example.gateway.filter;

import org.springframework.http.HttpHeaders;
import org.springframework.security.oauth2.jwt.Jwt;
import
org.springframework.security.oauth2.server.resource.authentication.J
wtAuthenticationToken;
import org.springframework.stereotype.Component;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

import java.util.List;

@Component
public class AuthenticationFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        return exchange.getPrincipal()
                .filter(principal -> principal instanceof
JwtAuthenticationToken)
                .cast(JwtAuthenticationToken.class)
                .flatMap(token -> {
```

```
                    Jwt jwt = token.getToken();
                    String username =
jwt.getClaimAsString("preferred_username");
                    List<String> roles =
jwt.getClaimAsStringList("roles");

                    exchange.getRequest().mutate()
                            .header("X-User-Name", username)
                            .header("X-User-Roles", String.join(",",
roles))
                            .build();

                    return chain.filter(exchange);
                });
    }
}
```

✅ **Extracts `username` & `roles` from Keycloak JWT**
✅ **Passes UserDetails to Microservices** via headers

---

# 5️⃣ Secure Microservices & Extract UserDetails

## 🔷 Step 1: Add Dependencies in `User-Service` (`pom.xml`)

xml
CopyEdit

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

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-resource-server</artifactId>
</dependency>
```

---

## 🔷 Step 2: Configure Security (`application.yml`)

yaml

CopyEdit

```
server:
  port: 8082

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/myrealm
```

✅ **User Service validates Keycloak JWT**

---

### 🔷 Step 3: Extract UserDetails in `UserController.java`

java
CopyEdit

```
package com.example.userservice.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/profile")
    public String getUserProfile(
            @RequestHeader("X-User-Name") String username,
            @RequestHeader("X-User-Roles") String roles) {

        return "User Profile for: " + username + " with roles: " +
roles;
    }
}
```

✅ **Extracts `username` & `roles` from request headers**

---

# 6️⃣ Testing Keycloak Authentication

## ✅ 1. Get JWT Token from Keycloak

bash
CopyEdit

```
export TOKEN=$(curl -X POST
"http://localhost:8081/realms/myrealm/protocol/openid-connect/token"
\
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "client_id=api-gateway" \
  -d "username=john" \
  -d "password=password" \
  -d "grant_type=password" | jq -r '.access_token')
```

✅ **Retrieves JWT Token for `john` from Keycloak**

---

## ✅ 2. Use JWT Token for API Request

bash
CopyEdit

```
curl -H "Authorization: Bearer $TOKEN"
http://localhost:8080/users/profile
```

Response:

sql
CopyEdit

```
User Profile for: john with roles: ROLE_USER
```

✅ **JWT is validated, and UserDetails are passed to Microservices**

---

## ❌ 3. Missing JWT

bash
CopyEdit

```
curl http://localhost:8080/users/profile
```

Response:

CopyEdit
```
401 Unauthorized
```

✅ **Access denied without a valid JWT**

---

## 7️⃣ Conclusion

✅ **Spring Cloud Gateway validates JWT from Keycloak**
✅ **Extracted UserDetails (username, roles) are passed to Microservices**
✅ **Microservices use UserDetails for authorization**

# Role-Based Access Control (RBAC) with Keycloak & Spring Cloud Gateway

In this guide, we will:
✅ **Implement Role-Based Access Control (RBAC)**
✅ **Assign Roles in Keycloak**
✅ **Enforce Role-Based Authorization in Spring Cloud Gateway and Microservices**

---

## 1️⃣ What is Role-Based Access Control (RBAC)?

**Role-Based Access Control (RBAC)** restricts access to resources based on the roles assigned to users. In this case:

- **Keycloak** will assign roles to users.
- **Spring Cloud Gateway** will validate the roles in the incoming JWT and enforce the appropriate access control.
- **Microservices** will check if the user has the necessary roles before processing the request.

---

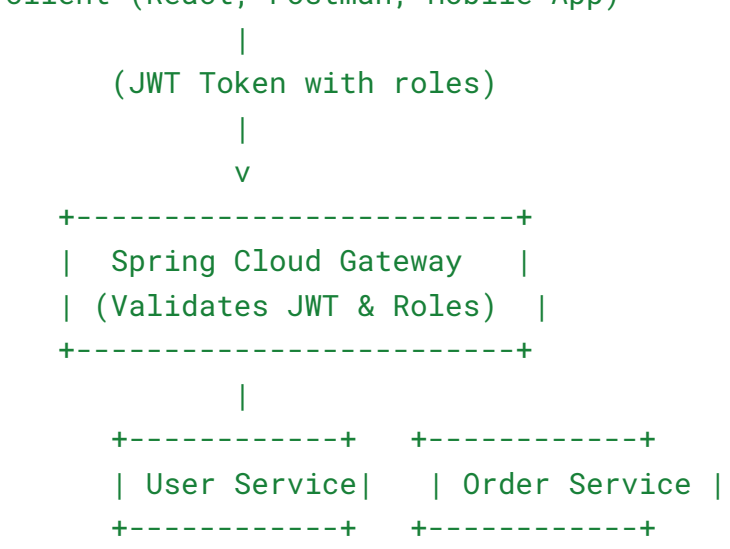
## 2️⃣ Architecture Overview with RBAC

pgsql
CopyEdit
```
1. User logs in → Keycloak generates JWT with roles
```

2. User sends JWT → API Gateway validates token and roles
3. API Gateway checks if the user has the required roles
4. Microservices enforce role-based authorization using JWT roles

### Architecture Diagram

sql
CopyEdit




---

## 3️⃣ Setting Up RBAC in Keycloak

### 🔷 Step 1: Assign Roles to Users in Keycloak

1. **Login to Keycloak Admin Console** (`http://localhost:8081/admin/`)
2. **Create Roles:**
   Go to `myrealm` → `Roles` → `Add Role`
   - `ROLE_USER`
   - `ROLE_ADMIN`
3. **Assign Roles to Users:**
   - Go to `Users` → Select a user (e.g., `john`)
   - Under the `Role Mappings` tab, assign roles (`ROLE_USER` or `ROLE_ADMIN`) to the user.

---

## 4️⃣ Secure Spring Cloud Gateway with RBAC

### 🔷 Step 1: Update Dependencies (`pom.xml`)

xml
CopyEdit

```
<dependency>
    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>

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

---

### 🔷 Step 2: Configure Role-Based Access Control in Gateway (`application.yml`)

yaml
CopyEdit

```
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8082/
          predicates:
            - Path=/users/**
          filters:
            - RemoveRequestHeader=Authorization
            - JwtRoleBasedFilter

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/myrealm
```

In this configuration:

- `JwtRoleBasedFilter`: This filter will check if the JWT contains the required roles.

---

### 🔷 Step 3: Implement Role-Based Authorization Filter in Gateway

Create a custom filter to validate the roles.

java
CopyEdit

```
package com.example.gateway.filter;

import org.springframework.security.oauth2.jwt.Jwt;
import
org.springframework.security.oauth2.server.resource.authentication.J
wtAuthenticationToken;
import org.springframework.stereotype.Component;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
public class JwtRoleBasedFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        return exchange.getPrincipal()
                .filter(principal -> principal instanceof
JwtAuthenticationToken)
                .cast(JwtAuthenticationToken.class)
                .flatMap(token -> {
                    Jwt jwt = token.getToken();
                    // Extract roles from the JWT
                    var roles = jwt.getClaimAsStringList("roles");

                    // Example: Only allow access if the user has
'ROLE_USER' or 'ROLE_ADMIN'
                    if (roles.contains("ROLE_USER") ||
roles.contains("ROLE_ADMIN")) {
```

```
                    return chain.filter(exchange);
                } else {

exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
                    return exchange.getResponse().setComplete();
                }
            });
    }
}
```

✅ **The filter ensures only users with the appropriate roles (`ROLE_USER` or `ROLE_ADMIN`) can access the microservices.**

---

## 5️⃣ Secure Microservices with RBAC

### 🔷 Step 1: Add Dependencies in `User-Service` (`pom.xml`)

xml
CopyEdit

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

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-resource-server</artifactId>
</dependency>
```

---

### 🔷 Step 2: Configure Security in Microservices (`application.yml`)

yaml
CopyEdit

```
server:
  port: 8082

spring:
  security:
    oauth2:
```

```
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/myrealm
```

---

### 🔷 Step 3: Role-Based Authorization in Microservices (`UserController.java`)

java
CopyEdit

```
package com.example.userservice.controller;

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/profile")
    @PreAuthorize("hasRole('ROLE_USER')")
    public String getUserProfile(
            @RequestHeader("X-User-Name") String username,
            @RequestHeader("X-User-Roles") String roles) {

        return "User Profile for: " + username + " with roles: " +
roles;
    }

    @GetMapping("/admin")
    @PreAuthorize("hasRole('ROLE_ADMIN')")
    public String getAdminProfile() {
        return "Admin Profile - Access restricted to admins only.";
    }
}
```

✅ **Uses `@PreAuthorize` to enforce role-based access**

- `ROLE_USER` can access `/users/profile`
- `ROLE_ADMIN` can access `/users/admin`

---

## 6️⃣ Testing Role-Based Access

### ✅ 1. Get JWT Token from Keycloak for `john` (ROLE_USER)

bash
CopyEdit

```
export TOKEN=$(curl -X POST
"http://localhost:8081/realms/myrealm/protocol/openid-connect/token"
\
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "client_id=api-gateway" \
  -d "username=john" \
  -d "password=password" \
  -d "grant_type=password" | jq -r '.access_token')
```

---

### ✅ 2. Use JWT Token to Access `/users/profile`

bash
CopyEdit

```
curl -H "Authorization: Bearer $TOKEN"
http://localhost:8080/users/profile
```

Response:

sql
CopyEdit

```
User Profile for: john with roles: ROLE_USER
```

---

### ✅ 3. Use JWT Token to Access `/users/admin` (User with ROLE_USER)

bash
CopyEdit

```
curl -H "Authorization: Bearer $TOKEN"
http://localhost:8080/users/admin
```

Response:

CopyEdit

```
403 Forbidden
```

---

### ✅ 4. Get JWT Token from Keycloak for `admin` (ROLE_ADMIN)

bash
CopyEdit

```
export TOKEN_ADMIN=$(curl -X POST
"http://localhost:8081/realms/myrealm/protocol/openid-connect/token"
\
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "client_id=api-gateway" \
  -d "username=admin" \
  -d "password=password" \
  -d "grant_type=password" | jq -r '.access_token')
```

---

### ✅ 5. Use Admin Token to Access `/users/admin`

bash
CopyEdit

```
curl -H "Authorization: Bearer $TOKEN_ADMIN"
http://localhost:8080/users/admin
```

Response:

pgsql
CopyEdit

```
Admin Profile - Access restricted to admins only.
```

---

## 7️ Conclusion

✅ **Keycloak assigns roles (`ROLE_USER`, `ROLE_ADMIN`)**
✅ **Spring Cloud Gateway enforces role-based access**
✅ **Microservices use `@PreAuthorize` to enforce role-based authorization**

This setup ensures that only users with the correct roles can access specific microservices or endpoints.

# Fine-Grained Access Control with Keycloak Policies

In this guide, we will:
✅ **Implement Fine-Grained Access Control using Keycloak**
✅ **Create Policies in Keycloak for finer control over permissions**
✅ **Enforce these policies in Spring Cloud Gateway and Microservices**

---

## 1️⃣ What is Fine-Grained Access Control?

Fine-grained access control refers to managing access to resources at a very detailed level based on user roles, attributes, and other conditions. Keycloak provides the ability to define **Policies** for users that allow access based on various conditions like:

- **Roles**
- **Attributes**
- **IP addresses**
- **Time-based conditions**

With this, we can control access in a more **dynamic** and **conditional** manner.

---

## 2️⃣ Keycloak Policies

### 🔹 Types of Policies in Keycloak:

1. **Role-Based Policies:** Allow access based on user roles.
2. **Attribute-Based Policies:** Allow access based on specific user attributes (e.g., department, job title).
3. **Time-Based Policies:** Control access based on the time of day.
4. **IP-Based Policies:** Restrict access to certain IP addresses or IP ranges.

### 🔹 Policy Flow:

1. **User Logs in**: Keycloak generates a **JWT** token containing user roles, attributes, and other claims.
2. **Keycloak evaluates Policies**: Based on user roles, attributes, or other conditions, Keycloak determines whether the user is allowed to access the requested resource.
3. **API Gateway / Microservices**: The API Gateway and Microservices will enforce these policies by checking the JWT for specific roles or claims.

---

# 3️⃣ Setting Up Fine-Grained Policies in Keycloak

## 🔷 Step 1: Enable Authorization in Keycloak

1. **Login to Keycloak Admin Console** (`http://localhost:8081/admin/`)
2. **Go to Realm Settings → Client →** **`api-gateway`**.
3. In the **Client Settings**, enable **Authorization** under the **Authorization Settings** tab.
4. **Save the changes**.

---

## 🔷 Step 2: Create Authorization Policies

1. **Create a Policy** for **Role-Based Access**:
    - Go to the `Authorization` tab for the `api-gateway` client.
    - Click **Create Policy**.
    - Choose **Role-Based**.
    - Create a policy called **"AdminPolicy"** and assign the role `ROLE_ADMIN`.
2. **Create a Policy for Attribute-Based Access**:
    - Click **Create Policy**.
    - Choose **Attribute-Based**.
    - For example, create a policy called **"DepartmentPolicy"** that checks if the user has the attribute `department=finance`.
3. **Create a Policy for Time-Based Access**:
    - Click **Create Policy**.
    - Choose **Time-Based**.
    - Set a time window (e.g., access allowed between 9 AM to 5 PM).

---

## 🔷 Step 3: Create Resource and Scope

1. **Create a Resource** for the endpoint you want to protect (e.g., `/admin`).
    - Go to the `Authorization` tab and create a **Resource** for `/admin`.
2. **Create a Scope**:
    - You can create a scope for specific actions (e.g., `read`, `write`, `delete`).
3. **Assign Policies to Resources**:
    - In the **Resource Permissions** tab, assign the policies created above (e.g., `AdminPolicy`, `DepartmentPolicy`) to specific resources or scopes.

---

# 4️⃣ Secure Spring Cloud API Gateway with Fine-Grained Policies

## 🔷 Step 1: Add Dependencies

Ensure that your API Gateway includes the necessary dependencies for Keycloak and OAuth2 Resource Server.

xml
CopyEdit

```
<dependency>
    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>

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

---

## 🔷 Step 2: Configure the API Gateway (`application.yml`)

Update your **`application.yml`** to enable resource server capabilities and to enforce policies via JWT.

yaml
CopyEdit

```
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://localhost:8082/
          predicates:
            - Path=/users/**
```

```
          filters:
            - RemoveRequestHeader=Authorization

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/myrealm
          jwk-set-uri:
http://localhost:8081/realms/myrealm/protocol/openid-connect/certs
```

---

### 🔷 Step 3: Enforce Fine-Grained Access Control in the Gateway

To enforce fine-grained access control, you need to create a custom **Global Filter** that checks the **JWT** for specific claims or roles and validates against Keycloak's policies.

java
CopyEdit

```
package com.example.gateway.filter;

import org.springframework.http.HttpStatus;
import org.springframework.security.oauth2.jwt.Jwt;
import
org.springframework.security.oauth2.server.resource.authentication.J
wtAuthenticationToken;
import org.springframework.stereotype.Component;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
public class PolicyEnforcementFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        return exchange.getPrincipal()
                .filter(principal -> principal instanceof
JwtAuthenticationToken)
```

```
                .cast(JwtAuthenticationToken.class)
                .flatMap(token -> {
                    Jwt jwt = token.getToken();
                    // Check if user has the necessary role or
attributes based on Keycloak policies
                    if
(jwt.getClaimAsStringList("roles").contains("ROLE_ADMIN")) {
                        return chain.filter(exchange);
                    } else {

exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
                        return exchange.getResponse().setComplete();
                    }
                });
    }
}
```

This filter can be modified to enforce multiple policies:

- **Role-based** access (`ROLE_ADMIN`, `ROLE_USER`).
- **Attribute-based** access (check user attributes like `department`).
- **Time-based** access (check the current time).

---

# 5️⃣ Secure Microservices with Fine-Grained Access Control

## 🔷 Step 1: Add Dependencies

Make sure your microservice (e.g., `user-service`) includes the necessary dependencies to work with JWT and Keycloak.

xml
CopyEdit

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

<dependency>
    <groupId>org.springframework.security</groupId>
```

```
    <artifactId>spring-security-oauth2-resource-server</artifactId>
</dependency>
```

---

### 🔷 Step 2: Configure Security in Microservice (`application.yml`)

yaml
CopyEdit

```
server:
  port: 8082

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/myrealm
```

---

### 🔷 Step 3: Role-Based and Attribute-Based Access in Microservice

You can apply `@PreAuthorize` to enforce role-based or attribute-based access to specific endpoints.

java
CopyEdit

```
package com.example.userservice.controller;

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class UserController {

    @GetMapping("/profile")
    @PreAuthorize("hasRole('ROLE_USER')")
    public String getUserProfile(
            @RequestHeader("X-User-Name") String username,
```

```
            @RequestHeader("X-User-Roles") String roles) {

        return "User Profile for: " + username + " with roles: " +
roles;
    }

    @GetMapping("/admin")
    @PreAuthorize("hasRole('ROLE_ADMIN')")
    public String getAdminProfile() {
        return "Admin Profile - Access restricted to admins only.";
    }
}
```

- **Role-based Access** is handled by the `@PreAuthorize` annotation. Only users with specific roles (`ROLE_USER`, `ROLE_ADMIN`) can access the endpoints.

---

## 6️⃣ Testing Fine-Grained Access Control

### ✅ 1. Get JWT Token for User (`ROLE_USER`)

bash
CopyEdit

```
export TOKEN=$(curl -X POST
"http://localhost:8081/realms/myrealm/protocol/openid-connect/token"
\
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "client_id=api-gateway" \
  -d "username=john" \
  -d "password=password" \
  -d "grant_type=password" | jq -r '.access_token')
```

---

### ✅ 2. Access `/users/profile` with User JWT

bash
CopyEdit

```
curl -H "Authorization: Bearer $TOKEN"
http://localhost:8080/users/profile
```

Response:

sql
CopyEdit
```
User Profile for: john with roles: ROLE_USER
```

---

## ✅ 3. Access `/users/admin` with User JWT

bash
CopyEdit
```
curl -H "Authorization: Bearer $TOKEN"
http://localhost:8080/users/admin
```

Response:

CopyEdit
```
403 Forbidden
```

---

## ✅ 4. Get JWT Token for Admin (`ROLE_ADMIN`)

bash
CopyEdit
```
export TOKEN_ADMIN=$(curl -X POST
"http://localhost:8081/realms/myrealm/protocol/openid-connect/token"
\
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "client_id=api-gateway" \
  -d "username=admin" \
  -d "password=password" \
  -d "grant_type=password" | jq -r '.access_token')
```

---

## ✅ 5. Access `/users/admin` with Admin JWT

bash
CopyEdit
```
curl -H "Authorization: Bearer $TOKEN_ADMIN"
http://localhost:8080/users/admin
```

Response:

pgsql

CopyEdit

```
Admin Profile - Access restricted to admins only.
```

---

# 7️⃣ Conclusion

✅ **Fine-Grained Access Control** using Keycloak policies allows you to manage user access dynamically.
✅ **Role-based**, **attribute-based**, and **time-based policies** ensure that only the right users can access certain resources.
✅ **Spring Cloud Gateway** and **Microservices** enforce these policies through JWT validation and custom filters.

Let's implement a more **complex policy** by combining **Role-Based Access Control (RBAC)** and **Attribute-Based Access Control (ABAC)** in Keycloak.

### Scenario:

We will implement a policy that grants access to a particular resource not only based on the user's **role** (e.g., `ROLE_USER` or `ROLE_ADMIN`), but also based on an **attribute** such as the user's **department**.
For example:

- **Only users with `ROLE_USER` and department `finance`** can access the `/finance` endpoint.

---

# 1️⃣ Setup Keycloak for RBAC + ABAC

## 🔷 Step 1: Add User Attributes in Keycloak

1. **Go to Keycloak Admin Console** (`http://localhost:8081/admin/`)
2. **Select the Realm** (`myrealm`).
3. **Create Users with Attributes**:
    - Go to `Users` → `Add User`.
    - **User 1** (`john`): Assign `ROLE_USER` and **set the attribute** `department=finance`.
    - **User 2** (`mary`): Assign `ROLE_USER` and **set the attribute** `department=sales`.

---

## 🔷 Step 2: Create a Role-Based Policy in Keycloak

1. **Go to the `Authorization` tab** for the `api-gateway` client.
2. **Create a Policy** for `ROLE_USER`:
    - Click **Create Policy** → **Role-based**.
    - Create a policy called `RolePolicy` with **Role**: `ROLE_USER`.

---

### 🔷 Step 3: Create an Attribute-Based Policy in Keycloak

1. **Create a Policy** for `department=finance`:
    - Click **Create Policy** → **Attribute-based**.
    - Create a policy called `DepartmentPolicy`.
    - **Add condition**: `department=finance`.

---

### 🔷 Step 4: Combine Both Policies into a Permission

1. **Create a Resource** for `/finance` endpoint.
    - Go to `Authorization` → `Resources` → **Create Resource**.
    - Name the resource `/finance`.
2. **Create a Permission** and **Assign Both Policies**:
    - Go to `Permissions` → **Create Permission**.
    - Assign `RolePolicy` and `DepartmentPolicy` to the `/finance` resource.
    - This ensures that both conditions must be met: `ROLE_USER` **and** `department=finance`.

---

## 2️⃣ Implement Role-Based + Attribute-Based Access Control in API Gateway

### 🔷 Step 1: Add Dependencies in API Gateway (`pom.xml`)

xml
CopyEdit

```
<dependency>
    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>

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

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

---

### 🔷 Step 2: Configure API Gateway for JWT Validation (`application.yml`)

Update your `application.yml` to configure JWT validation with Keycloak.

yaml
CopyEdit

```yaml
server:
  port: 8080

spring:
  cloud:
    gateway:
      routes:
        - id: finance-service
          uri: http://localhost:8082/
          predicates:
            - Path=/finance/**
          filters:
            - RemoveRequestHeader=Authorization

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8081/realms/myrealm
```

---

### 🔷 Step 3: Implement a Custom Filter in API Gateway to Enforce Role and Attribute Policies

Now, we need a filter in the API Gateway that checks both the role and the user's department. If the user is not `ROLE_USER` and their department is not `finance`, they will be denied access.

java
CopyEdit

```
package com.example.gateway.filter;

import org.springframework.http.HttpStatus;
import org.springframework.security.oauth2.jwt.Jwt;
import
org.springframework.security.oauth2.server.resource.authentication.J
wtAuthenticationToken;
import org.springframework.stereotype.Component;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Component
public class RoleAndAttributeFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        return exchange.getPrincipal()
                .filter(principal -> principal instanceof
JwtAuthenticationToken)
                .cast(JwtAuthenticationToken.class)
                .flatMap(token -> {
                    Jwt jwt = token.getToken();

                    // Extract roles and attributes
                    var roles = jwt.getClaimAsStringList("roles");
                    var department =
jwt.getClaimAsString("department");

                    // Enforce Role and Attribute-Based Access
Control
                    if (roles.contains("ROLE_USER") &&
"finance".equals(department)) {
                        return chain.filter(exchange);
                    } else {

exchange.getResponse().setStatusCode(HttpStatus.FORBIDDEN);
                        return exchange.getResponse().setComplete();
                    }
```

```
                });
    }
}
```

- The filter checks if the JWT contains:
  - `ROLE_USER`
  - `department=finance`

If both conditions are met, the request proceeds; otherwise, the user is forbidden.

---

# 3️⃣ Secure Microservices with RBAC + ABAC

## 🔷 Step 1: Add Dependencies in Microservice (`pom.xml`)

Make sure your microservices have the necessary dependencies to validate JWT.

xml
CopyEdit

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

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-resource-server</artifact-id>
</dependency>
```

---

## 🔷 Step 2: Configure Microservices to Accept JWT (`application.yml`)

yaml
CopyEdit

```
server:
  port: 8082

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
```

```
issuer-uri: http://localhost:8081/realms/myrealm
```

---

### 🔷 Step 3: Implement Role and Attribute Checks in Microservices (`FinanceController.java`)

java
CopyEdit

```
package com.example.finance.controller;

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FinanceController {

    @GetMapping("/finance")
    @PreAuthorize("hasRole('ROLE_USER') and
@userService.hasDepartment('finance')")
    public String getFinanceData() {
        return "Finance Data - Access granted to finance users
only!";
    }
}
```

In this example:

- **`@PreAuthorize`** ensures that only users with the role `ROLE_USER` and department `finance` can access the `/finance` endpoint.
- **`@userService.hasDepartment('finance')`** is a custom service method to check the department.

---

## 4️⃣ Testing the RBAC + ABAC Configuration

### Step 1: Get JWT Token for `john` (ROLE_USER, department=finance)

bash
CopyEdit

```
export TOKEN=$(curl -X POST
"http://localhost:8081/realms/myrealm/protocol/openid-connect/token"
\
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "client_id=api-gateway" \
  -d "username=john" \
  -d "password=password" \
  -d "grant_type=password" | jq -r '.access_token')
```

---

### Step 2: Access `/finance` with `john`'s JWT

bash
CopyEdit

```
curl -H "Authorization: Bearer $TOKEN" http://localhost:8080/finance
```

Response:

pgsql
CopyEdit

```
Finance Data - Access granted to finance users only!
```

---

### Step 3: Get JWT Token for `mary` (ROLE_USER, department=sales)

bash
CopyEdit

```
export TOKEN_MARY=$(curl -X POST
"http://localhost:8081/realms/myrealm/protocol/openid-connect/token"
\
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "client_id=api-gateway" \
  -d "username=mary" \
  -d "password=password" \
  -d "grant_type=password" | jq -r '.access_token')
```

---

### Step 4: Try Accessing `/finance` with `mary`'s JWT

bash
CopyEdit

```
curl -H "Authorization: Bearer $TOKEN_MARY"
http://localhost:8080/finance
```

Response:

CopyEdit
```
403 Forbidden
```

---

## 5️⃣ Conclusion

- **RBAC**: Users with the correct roles (`ROLE_USER`) can access certain resources.
- **ABAC**: Access is further restricted by user attributes (e.g., department).
- **Keycloak**: Handles policy enforcement and generates JWTs with roles and attributes.
- **API Gateway & Microservices**: Validate the JWT and enforce access control based on both roles and attributes.

This setup ensures **fine-grained control** over who can access resources, ensuring your system remains both flexible and secure.

### Seamless Exception Handling in Microservices with Propagation to API Gateway

In a **microservices architecture**, it's crucial to have a robust **exception handling mechanism**. This ensures that errors are properly handled at the service level and that the API Gateway and other downstream consumers receive meaningful and consistent responses. Here, we will discuss a structured approach for handling exceptions in microservices and how these exceptions can be propagated to the **API Gateway** for a **fault-tolerant** system.

---

## 1️⃣ Why is Exception Handling Important in Microservices?

In a distributed architecture with multiple microservices, proper exception handling is important for:

1. **Consistency**: Uniform error response across all microservices.
2. **Resilience**: Gracefully handle unexpected failures and provide meaningful feedback to clients.
3. **Security**: Prevent leaking internal system details to the end-users.

4. **Transparency**: Ensure that clients are aware of errors in a standardized format for easy troubleshooting.

---

# 2️⃣ Design Considerations for Exception Handling in Microservices

The core idea behind **exception handling** in microservices is to:

1. **Handle errors locally** in each microservice.
2. **Propagate errors consistently** across services.
3. **Ensure the API Gateway handles errors gracefully** and provides uniform responses.

---

## Steps to Implement Seamless Exception Handling in Microservices

---

## 3️⃣ Centralized Exception Handling in Microservices

In **Spring Boot**, we can use `@ControllerAdvice` for global exception handling. Here's how you can set it up:

### Step 1: Create Custom Exception Classes

Define custom exception classes that represent different error scenarios:

java
CopyEdit

```
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}

public class UnauthorizedException extends RuntimeException {
    public UnauthorizedException(String message) {
        super(message);
    }
}
```

---

**Step 2: Global Exception Handler with `@ControllerAdvice`**

Use `@ControllerAdvice` to globally handle exceptions across all your controllers.

java
CopyEdit

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse>
handleResourceNotFound(ResourceNotFoundException ex) {
        ErrorResponse errorResponse = new ErrorResponse("NOT_FOUND",
ex.getMessage());
        return new ResponseEntity<>(errorResponse,
HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(UnauthorizedException.class)
    public ResponseEntity<ErrorResponse>
handleUnauthorized(UnauthorizedException ex) {
        ErrorResponse errorResponse = new
ErrorResponse("UNAUTHORIZED", ex.getMessage());
        return new ResponseEntity<>(errorResponse,
HttpStatus.UNAUTHORIZED);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse>
handleAllExceptions(Exception ex) {
        ErrorResponse errorResponse = new
ErrorResponse("INTERNAL_SERVER_ERROR", "An unexpected error
occurred");
        return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

In this handler:

- **`ResourceNotFoundException`**: Returns a `404` status with an appropriate error message.
- **`UnauthorizedException`**: Returns a `401` status indicating unauthorized access.
- **Generic Exception**: Catches all other exceptions and returns a `500` status.

---

#### Step 3: Error Response Object

Define a custom error response object to standardize the error format:

java
CopyEdit

```
public class ErrorResponse {
    private String errorCode;
    private String message;

    // Constructor, Getters, Setters
}
```

### Result: If a service encounters an error, it will return a consistent JSON error response like:

json
CopyEdit

```
{
  "errorCode": "NOT_FOUND",
  "message": "The requested resource could not be found"
}
```

---

## 4️⃣ Propagating Errors to the API Gateway

The **API Gateway** (e.g., Spring Cloud Gateway) is the entry point for clients to interact with multiple microservices. It plays a crucial role in **error propagation** and must handle errors from downstream microservices properly.

#### Step 1: Configure Global Filters in API Gateway

In Spring Cloud Gateway, you can define **filters** that intercept requests and handle errors centrally.

Here’s how to propagate exceptions from downstream services to the API Gateway:

java
CopyEdit

```
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.web.server.ServerWebExchange;
import org.springframework.stereotype.Component;
import org.springframework.http.HttpStatus;
import reactor.core.publisher.Mono;

@Component
public class ErrorHandlingFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        return chain.filter(exchange)
                .onErrorResume(e -> {
                    // Log the error for debugging
                    System.out.println("Error occurred: " +
e.getMessage());

                    // Set the response status and return a custom
error message

exchange.getResponse().setStatusCode(HttpStatus.INTERNAL_SERVER_ERRO
R);

exchange.getResponse().getHeaders().add("Content-Type",
"application/json");

                    String errorMessage = "{ \"errorCode\":
\"INTERNAL_SERVER_ERROR\", \"message\": \"An unexpected error
occurred at the gateway.\" }";
                    return
exchange.getResponse().writeWith(Mono.just(exchange.getResponse().bu
fferFactory().wrap(errorMessage.getBytes())));
                });
    }
}
```

This **global filter** catches all exceptions occurring within the gateway and forwards a consistent error message with a **500 Internal Server Error**. You can customize the error response further based on the type of exception.

---

**Step 2: Handle Specific Errors from Microservices**

If a downstream microservice throws an exception (like `ResourceNotFoundException`), you can map that exception to an appropriate HTTP status code in the API Gateway:

java
CopyEdit

```
@Component
public class SpecificErrorHandlingFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        return chain.filter(exchange)
                .onErrorResume(e -> {
                    if (e instanceof ResourceNotFoundException) {

exchange.getResponse().setStatusCode(HttpStatus.NOT_FOUND);
                        String errorMessage = "{ \"errorCode\":
\"NOT_FOUND\", \"message\": \"Resource not found\" }";
                        return
exchange.getResponse().writeWith(Mono.just(exchange.getResponse().bu
fferFactory().wrap(errorMessage.getBytes())));
                    } else {
                        return Mono.error(e);  // Propagate other
errors
                    }
                });
    }
}
```

---

## 5️⃣ Example of Fault-Tolerant Handling with Retry and Circuit Breaker

In microservices, we often face temporary failures that might be resolved by retrying the request. Additionally, for prolonged failures, we can use a **circuit breaker** to prevent the system from making requests to a failing service.

**Step 1: Enable Circuit Breaker and Retry**

Using **Resilience4j** for Circuit Breaker and Retry in Spring Boot:

Add the dependencies to `pom.xml`:

xml
CopyEdit

```xml
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

**Step 2: Configure Circuit Breaker and Retry in `application.yml`**

yaml
CopyEdit

```yaml
resilience4j:
  retry:
    instances:
      myRetry:
        maxAttempts: 3
        waitDuration: 1000ms

  circuitbreaker:
    instances:
      myCircuitBreaker:
        registerHealthIndicator: true
        failureRateThreshold: 50
        slidingWindowSize: 100
```

This configuration:

- Configures the **retry mechanism** to retry failed requests up to 3 times with a 1-second wait between attempts.
- Configures the **circuit breaker** to open if 50% of the requests fail in the last 100 requests.

---

**Step 3: Implement Circuit Breaker and Retry in Microservices**

Use `@Retry` and `@CircuitBreaker` annotations from Resilience4j to apply these mechanisms:

java
CopyEdit

```
import io.github.resilience4j.retry.annotation.Retry;
import
io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    @Retry(name = "myRetry")
    @CircuitBreaker(name = "myCircuitBreaker", fallbackMethod =
"fallback")
    public String getData() {
        // Simulate a service call that might fail
        if (Math.random() > 0.7) {
            throw new RuntimeException("Service failed");
        }
        return "Data fetched successfully";
    }

    public String fallback(Throwable t) {
        return "Fallback response: Service unavailable";
    }
}
```

---

## 6️⃣ Propagating Exception and Handling in the API Gateway

When a microservice throws an exception, the **API Gateway** handles it using the custom filters defined above and returns the appropriate HTTP status code and error message to the client.

**Example Request Flow:**

1. **Client Request → API Gateway**:
    - The API Gateway receives the request and routes it to the appropriate microservice.
2. **Microservice Processing**:
    - The microservice processes the request but throws an exception (e.g., `ResourceNotFoundException`).
3. **Exception Propagation**:

    - The exception is caught by the microservice's global exception handler and converted into a standard error response.
4. **API Gateway Handles Error**:
    - The error is propagated back to the API Gateway, which formats the response into a consistent error format and sends it to the client.

**Example Response:**
json
CopyEdit

```
{
  "errorCode": "NOT_FOUND",
  "message": "The requested resource could not be found"
}
```

---

## 7️⃣ Conclusion

- **Microservice Exception Handling**: Centralized exception handling ensures consistent error responses across all services.
- **API Gateway Role**: The API Gateway is key in propagating these exceptions and handling them in a user-friendly way.
- **Resilience**: Circuit Breakers and Retries ensure fault tolerance, helping to prevent cascading failures and improving system stability.
- **Custom Filters in API Gateway**: Filters allow the API Gateway to handle exceptions from microservices and return meaningful responses to clients.

By combining these techniques, you create a more **robust**, **resilient**, and **user-friendly** system that gracefully handles failures and communicates them effectively to clients.