15. 외부 API 연동

RestTemplate vs WebClient

Spring 애플리케이션에서 외부 REST API 호출을 처리할 때는 주로 **RestTemplate** 또는 **WebClient**를 사용한다.
이 두 라이브러리는 모두 **HTTP 클라이언트**로서 동작하지만, **아키텍처**와 **기능적 특성**에서 차이가 있다.

Spring 5 이후부터는 **WebClient가 RestTemplate의 후속 기술**로 권장된다.

1️⃣ 기술 개요

항목	RestTemplate	WebClient
도입 시기	Spring 3.x	Spring 5.x (WebFlux 포함)
패러다임	동기(synchronous)	비동기(asynchronous), 동기 모두 지원
기반 API	Java HttpURLConnection, Apache HttpClient 등	Reactor Netty(기본), 다른 HttpClient로 교체 가능
지원 방식	명령형(Imperative)	반응형(Reactive), 명령형 모두 지원
지원 상태	deprecated 예정 (향후 유지보수만 진행)	Spring의 공식 권장 클라이언트

2️⃣ RestTemplate 사용법 🛠️

의존성 추가 (Spring Boot Starter Web 포함 시 기본 제공)

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}
```

예시 사용

```
RestTemplate restTemplate = new RestTemplate();

String result = restTemplate.getForObject("https://api.example.com/data", String.class);

ResponseEntity<MyDto> response =
restTemplate.postForEntity("https://api.example.com/data", requestBody, MyDto.class);
```

특징

✅ 사용법이 단순하며 익숙한 동기적 API 패턴.
✅ Retrofit과 유사한 감각으로 빠르게 REST 호출 구현 가능.
❌ **비동기 처리 불가** → 대량 호출 시 블로킹 발생 가능성.
❌ Spring 5 이후에는 점진적 유지보수만 예정.

3️⃣ WebClient 사용법 🚀

WebClient는 Spring 5에서 도입된 **비동기 / 논블로킹 HTTP 클라이언트**이다.
기본적으로 **Reactor 기반 Mono/Flux API**를 사용하지만, 동기 호출도 지원한다.

의존성 추가

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

Bean 등록

```
@Bean
public WebClient webClient(WebClient.Builder builder) {
    return builder.baseUrl("https://api.example.com").build();
}
```

예시 사용

비동기 호출 (Reactive)

```
Mono<String> result = webClient.get()
    .uri("/data")
    .retrieve()
    .bodyToMono(String.class);

result.subscribe(response -> System.out.println("응답: " + response));
```

동기 호출

```
String result = webClient.get()
    .uri("/data")
    .retrieve()
    .bodyToMono(String.class)
    .block();
```

특징

✅ 비동기 / 논블로킹 지원 (대량 트래픽 처리에 적합).
✅ 동기 호출도 가능 (`block()` 사용 시).
✅ **Streaming**, **SSE(Server-Sent Events)**, **WebSocket** 연계도 지원.
✅ **Backpressure** 지원 (Reactive Streams 기반).
✅ **Connection Pool**과 Timeout 설정 유연.
✅ 현대적 API 설계.
❌ API 설계가 RestTemplate 대비 다소 복잡함 (초기 학습 필요).

---

4 주요 비교표 🗂️

항목	RestTemplate	WebClient
동기/비동기	동기	비동기(기본), 동기도 지원
성능	동기 블로킹 → 대규모 요청에 불리	논블로킹 → 고성능
프로그래밍 모델	Imperative	Reactive / Imperative 혼용 가능
Streaming 지원	미지원	지원 (Flux 기반)
SSE 지원	미지원	지원
유지보수 상태	Deprecated 예정	Spring 공식 권장 방향
학습 곡선	낮음	다소 높음 (Reactive 패턴 이해 필요)

---

5 사용 권장 가이드 🎯

상황	권장 클라이언트
기존 레거시 시스템 유지보수	RestTemplate 사용 지속 가능
새로운 프로젝트	WebClient 권장
Reactive 기반 애플리케이션 (WebFlux 사용)	WebClient 필수
고성능 API Gateway / Proxy 구성	WebClient 권장
단순 REST 호출 (적은 빈도)	RestTemplate 사용 가능

---

6 마이그레이션 전략 🚀

- RestTemplate 기반 기존 API는 유지하되, 신규 API는 WebClient로 설계 권장.
- Gradual Migration → RestTemplate → WebClient로 점진적 전환.
- 대규모 병렬 호출, Streaming이 필요한 서비스는 WebClient 우선 적용.

결론 🗂️

✅ RestTemplate → 단순 동기 호출 시 간편하지만, 미래 지향적 사용은 WebClient 권장.
✅ WebClient → 고성능, Reactive 지원, 대규모 트래픽 처리에 유리.
✅ WebFlux 기반 시스템에서는 WebClient 필수 사용.
✅ 학습 곡선은 다소 존재하나, 현대적인 애플리케이션 설계에는 WebClient가 표준.

OpenFeign 사용법

OpenFeign은 선언적(Declarative)으로 HTTP API 클라이언트를 생성할 수 있는 라이브러리이다.
Spring Cloud에서는 `spring-cloud-starter-openfeign`를 통해 쉽게 통합할 수 있으며, 인터페이스 기반으로 외부 REST API 호출을 매우 간결하게 구현할 수 있다.

OpenFeign은 다음과 같은 장점이 있다:

- **인터페이스 기반**으로 API 호출 정의
- Boilerplate 코드 제거 → 코드 간결성 ↑
- 다양한 플러그인 지원 (Load Balancer, Retry, Logging, Metrics 등)
- 통합적인 에러 처리 및 커스터마이징 용이
- Spring MVC 어노테이션과 자연스럽게 연계

1️⃣ 기본 구성

의존성 추가

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

> ⚠️ Feign은 Spring Boot 3.0+ 에서 **Spring Cloud BOM** 관리 필요.
> **spring-cloud-dependencies** BOM을 반드시 import할 것:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Hoxton.SR12</version> <!-- 버전 확인 필요 -->
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2️⃣ @EnableFeignClients

Feign Client 기능 활성화를 위해 `@EnableFeignClients` 선언 필요.

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

3️⃣ FeignClient 인터페이스 정의 🚀

가장 핵심적인 부분 → **인터페이스 기반으로 외부 API 호출 선언**.

```
@FeignClient(name = "exampleClient", url = "https://api.example.com")
public interface ExampleClient {

    @GetMapping("/data")
    String getData();

    @PostMapping("/data")
    ResponseEntity<MyDto> createData(@RequestBody MyDto dto);

    @GetMapping("/user/{id}")
    UserDto getUserById(@PathVariable("id") Long id);
}
```

주요 구성 요소

요소	설명
`@FeignClient`	Feign Client 정의, name은 Bean name으로도 사용 가능
`url`	호출할 대상 서버 URL (LoadBalancer 사용 시 생략 가능)
`@GetMapping`, `@PostMapping` 등	Spring MVC 스타일의 선언적 매핑 가능
메서드 인자	`@RequestBody`, `@PathVariable`, `@RequestParam` 지원

4️⃣ FeignClient 사용 예시

```
@Service
@RequiredArgsConstructor
public class MyService {

    private final ExampleClient exampleClient;

```

```
    public void process() {
        String data = exampleClient.getData();
        System.out.println("수신 데이터: " + data);

        UserDto user = exampleClient.getUserById(42L);
        System.out.println("User: " + user);
    }
}
```

- **DI 주입**으로 인터페이스 사용 → 내부적으로 Proxy가 주입됨.
- 코드에 직접 RestTemplate/WebClient 작성할 필요 없음 ✨.

5️⃣ 구성 흐름 🗂️

```
Application Startup → @EnableFeignClients → Feign Proxy Bean 생성
   ↓
Service Layer → FeignClient 인터페이스 사용
   ↓
Feign Proxy → HTTP 호출 수행 (RestTemplate or HttpClient 내부 사용 가능)
   ↓
응답 변환 후 결과 리턴
```

6️⃣ 고급 기능

Logging 설정

```
logging:
  level:
    com.example.client: DEBUG
    feign: DEBUG
```

Feign Client별 로그 레벨 설정 가능 → HTTP 요청/응답 로깅 확인 가능.

Timeout 설정

```
feign:
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 10000
```

- 기본 설정은 모든 FeignClient에 적용.
- 특정 Client 명으로 `config.{client-name}` 설정 가능.

Retry 설정

```
feign:
  client:
    config:
      default:
        retryer:
          period: 100
          maxPeriod: 1000
          maxAttempts: 3
```

- **내장 Retryer 사용 가능** → 재시도 전략 적용.

ErrorDecoder 커스터마이징

```
@Component
public class CustomErrorDecoder implements ErrorDecoder {

    private final ErrorDecoder defaultDecoder = new Default();

    @Override
    public Exception decode(String methodKey, Response response) {
        if (response.status() == 404) {
            return new NotFoundException("404 에러 발생");
        }
        return defaultDecoder.decode(methodKey, response);
    }
}
```

- 글로벌 ErrorDecoder 등록 시 모든 FeignClient에 적용됨.

7️⃣ RestTemplate / WebClient / OpenFeign 비교

항목	RestTemplate	WebClient	OpenFeign
사용 패턴	명령형	Reactive/명령형	선언형(인터페이스 기반)
비동기 지원	X	O	O(내부적으로 가능하나 기본은 동기)
부가 기능	거의 없음	매우 풍부	LoadBalancer, Retry, Metrics 내장
코드 간결성	중간	복잡 (Mono/Flux 필요 시)	매우 간결
코드 유지보수	보통	다소 어렵다 (Reactive는 학습 필요)	매우 쉽다 (interface 기반 유지관리 용이)
Spring Cloud 통합	보통	보통	매우 우수 (Spring Cloud 핵심 기능)

결론 🗂️

✅ OpenFeign은 선언적 HTTP 클라이언트 구성에 최적화되어 있으며, Spring Cloud 기반의 마이크로서비스 아키텍처에서 널리 사용된다.
✅ @FeignClient 인터페이스 기반으로 코드 간결성을 크게 높인다.
✅ 다양한 부가기능(LoadBalancer, Retry, Metrics, Logging 등) 내장 → 운영 환경에서 유리.
✅ RestTemplate/WebClient보다 개발 편의성이 높으나, 고성능 비동기 처리 필요 시 WebClient도 여전히 유효하다.

OAuth2 클라이언트 연동

많은 외부 API 서비스는 **OAuth2 기반 인증**을 요구한다.
Spring Boot에서 OpenFeign 사용 시, OAuth2 인증 토큰(Access Token)을 자동으로 발급받아 Feign 요청에 포함할 수 있다.

Spring Security와 Spring Cloud OAuth2 Client 기능을 이용하면 **자동화된 토큰 관리**와 **Feign Client 연계**를 간편하게 구성할 수 있다.

1️⃣ 기본 구성 흐름

```
Application Startup → OAuth2 ClientRegistration 구성
   ↓
OAuth2AuthorizedClientManager가 Access Token 발급
   ↓
Feign RequestInterceptor가 Access Token을 Authorization Header에 자동 추가
   ↓
Feign Client → 외부 API 호출 시 OAuth2 인증 적용
```

👉 핵심 포인트 → **Feign에 RequestInterceptor 등록** + **OAuth2AuthorizedClientManager 연계**.

2️⃣ 의존성 추가

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>

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

3️⃣ application.yml 설정

OAuth2 ClientRegistration 설정 → Client Credentials Flow 예시.

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          example-client:
            client-id: your-client-id
            client-secret: your-client-secret
            authorization-grant-type: client_credentials
            token-uri: https://auth.example.com/oauth/token
        provider:
          example-provider:
            token-uri: https://auth.example.com/oauth/token
```

주요 포인트

항목	설명
registration.client-id / client-secret	OAuth2 Client Credentials
authorization-grant-type	Client Credentials Flow 사용
token-uri	Token 발급 Endpoint

4️⃣ OAuth2AuthorizedClientManager Bean 구성

```java
@Configuration
public class OAuth2ClientConfig {

    @Bean
    public OAuth2AuthorizedClientManager authorizedClientManager(
            ClientRegistrationRepository clientRegistrationRepository,
            OAuth2AuthorizedClientService authorizedClientService) {

        OAuth2AuthorizedClientProvider authorizedClientProvider =
                OAuth2AuthorizedClientProviderBuilder.builder()
                    .clientCredentials()
                    .build();

        DefaultOAuth2AuthorizedClientManager authorizedClientManager =
                new DefaultOAuth2AuthorizedClientManager(clientRegistrationRepository,
authorizedClientService);

        authorizedClientManager.setAuthorizedClientProvider(authorizedClientProvider);

        return authorizedClientManager;
    }
}
```

✅ 핵심 → **clientCredentials()** 전략을 명시 → Feign 요청 시 OAuth2 토큰 발급 지원.

5️⃣ Feign RequestInterceptor 구성 🚀

OpenFeign 요청 시 Access Token을 자동으로 Authorization Header에 추가.

```
@Configuration
public class FeignClientOAuth2InterceptorConfig {

    private final OAuth2AuthorizedClientManager authorizedClientManager;

    public FeignClientOAuth2InterceptorConfig(OAuth2AuthorizedClientManager authorizedClientManager) {
        this.authorizedClientManager = authorizedClientManager;
    }

    @Bean
    public RequestInterceptor oauth2FeignRequestInterceptor() {
        return requestTemplate -> {
            OAuth2AuthorizeRequest authorizeRequest = OAuth2AuthorizeRequest.withClientRegistrationId("example-client")
                .principal("feign-client")
                .build();

            OAuth2AuthorizedClient authorizedClient = authorizedClientManager.authorize(authorizeRequest);

            if (authorizedClient != null) {
                String tokenValue = authorizedClient.getAccessToken().getTokenValue();
                requestTemplate.header("Authorization", "Bearer " + tokenValue);
            }
        };
    }
}
```

👉 **RequestInterceptor**를 등록하면 → Feign Client 요청 시 자동으로 **Bearer Token** 추가 ✨.

6️⃣ FeignClient 선언

```
@FeignClient(name = "exampleClient", url = "https://api.example.com")
public interface ExampleClient {

    @GetMapping("/protected-data")
    String getProtectedData();
}
```

Service에서 사용:

```
@Service
@RequiredArgsConstructor
public class ExampleService {

    private final ExampleClient exampleClient;

    public String callProtectedApi() {
        return exampleClient.getProtectedData();
    }
}
```

7️⃣ 종합 구성 흐름 🗂️

```
Application Startup
   ↓
@EnableFeignClients → FeignClient 등록
   ↓
OAuth2AuthorizedClientManager 구성
   ↓
RequestInterceptor 등록 → Feign 요청 시 Authorization Header 추가
   ↓
Feign Client 호출 시 Access Token 자동 적용
```

8️⃣ 주의사항 ⚠️

- **Client Credentials Flow**는 일반적으로 **머신-투-머신 호출(M2M)** 에 적합.
- OAuth2AuthorizedClientManager는 Access Token의 **캐싱**과 **자동 재발급**을 지원.
- 다른 Flow(Authorization Code 등)는 필요 시 해당 Provider 추가 구성 필요.
- Token 발급 빈도와 유효 기간 관리 주의 → Rate Limit 대응 필요.

결론 🗂️

✅ OpenFeign + OAuth2 Client 연동 시 **RequestInterceptor 구성**으로 OAuth2 Token 자동 처리 가능.
✅ OAuth2AuthorizedClientManager 구성 → Token 발급/재발급 관리 자동화.
✅ Client Credentials Flow 외 다른 Flow(Authorization Code, Password, Refresh Token 등)도 지원 가능.
✅ MSA 환경 또는 외부 API 연동 시 **중앙 집중적 Token 관리 구조 설계 가능** → 운영 효율성 향상.

타임아웃, 재시도, 서킷 브레이커

마이크로서비스 아키텍처 또는 외부 API 연동 시 **네트워크 지연/실패**는 불가피하게 발생한다.
OpenFeign 클라이언트에 다음과 같은 **회복 탄력성 기능**을 구성하면 운영 환경에서 서비스의 안정성을 높일 수 있다:

- 타임아웃(Timeouts) → **응답 지연에 대한 빠른 회복**
- 재시도(Retry) → **일시적 장애에 대한 자동 재시도**

- 서킷 브레이커(Circuit Breaker) → **장기적 장애 시 빠른 차단 및 복구 전략**

1️⃣ 타임아웃 설정 🕑

기본 원리

- **연결 타임아웃**: 서버 연결 시도 최대 대기 시간
- **읽기 타임아웃**: 응답 읽기 최대 대기 시간

설정 방법

```
feign:
  client:
    config:
      default:   # 또는 특정 client 이름 사용 가능
        connectTimeout: 5000     # ms
        readTimeout: 10000       # ms
```

동작 흐름

```
요청 → 서버 응답 지연 시 타임아웃 발생
→ TimeoutException throw
→ 글로벌 예외 핸들링 가능
```

주의사항 ⚠️

- 타임아웃은 **Fail Fast** 원칙을 지키는 데 매우 중요하다.
- 과도하게 긴 타임아웃 설정은 **Thread 자원 고갈 위험** 발생 가능.

2️⃣ 재시도(Retry) 설정 🔁

기본 전략

Feign에는 기본 Retryer가 내장되어 있으며, Spring Cloud에서는 `feign.client.config.retryer` 설정으로 제어 가능하다.

기본 설정

```
feign:
  client:
    config:
      default:
        retryer:
          period: 100          # 첫 재시도까지 delay(ms)
          maxPeriod: 1000      # 최대 delay(ms)
          maxAttempts: 3       # 최대 시도 횟수
```

Retryer 커스터마이징 (Java Config)

```
@Bean
public Retryer feignRetryer() {
    return new Retryer.Default(100, 1000, 3);
}
```

동작 흐름

```
1차 요청 → 실패 시 delay → 재시도 → 최대 횟수 초과 시 Exception throw
```

주의사항 ⚠️

- 비멱등성(POST 등) 요청에는 재시도 적용 주의 → 데이터 중복 처리 필요.
- Idempotent 요청(GET, PUT 등)에 Retry 권장.

3️⃣ 서킷 브레이커(Circuit Breaker) 구성 ⚡

개요

서킷 브레이커는 **반복되는 실패**가 발생할 경우 자동으로 호출 차단 후
일정 기간 후 다시 재시도하는 회복 전략이다.

Spring Cloud에서는 주로 **Resilience4j** 또는 **Hystrix(Deprecated)** 를 이용한다.
현재는 Resilience4j가 공식 권장됨.

의존성 추가

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot3</artifactId>
</dependency>

<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-cloud2</artifactId>
</dependency>
```

설정 방법 (application.yml)

```yaml
resilience4j:
  circuitbreaker:
    instances:
      exampleClientCircuitBreaker:
        registerHealthIndicator: true
        slidingWindowSize: 10
        failureRateThreshold: 50    # 실패율(%) → 50% 이상 시 Open 상태 전환
        waitDurationInOpenState: 10000   # Open 상태 유지(ms)
        permittedNumberOfCallsInHalfOpenState: 3
        slidingWindowType: COUNT_BASED
```

Feign Client와 연계 방법

1️⃣ Feign Client 호출 Service Layer에 **@CircuitBreaker** 적용

```java
@Service
@RequiredArgsConstructor
public class ExampleService {

    private final ExampleClient exampleClient;

    @CircuitBreaker(name = "exampleClientCircuitBreaker", fallbackMethod =
"fallbackExample")
    public String callExampleApi() {
        return exampleClient.getProtectedData();
    }

    public String fallbackExample(Throwable t) {
        return "Fallback Response";
    }
}
```

👉 CircuitBreaker 적용 시 **Open 상태**에서는 fallback 메서드가 즉시 호출됨 → 빠른 장애 대응 🚀.

4️⃣ 전체 구성 흐름 🗂️

```
Feign Client 요청 → 타임아웃 설정 → 일정 시간 내 응답 없으면 Timeout 발생
    ↓
Retryer 적용 시 → 자동 재시도(설정 횟수만큼)
    ↓
CircuitBreaker 적용 시 → 실패율 누적 분석 → Open 상태 시 요청 차단 및 fallback 처리
```

5️⃣ 정리 비교표 🗂️

기능	목적	적용 위치	효과
Timeout	느린 응답에 빠르게 실패	Feign config	Thread 낭비 방지
Retry	일시적 장애 자동 재시도	Retryer or Resilience4j Retry	장애 자동 복구 시도
Circuit Breaker	반복 실패 차단 + fallback 처리	Resilience4j CircuitBreaker	장애 격리, 서비스 보호

결론 🗂️

✅ Feign Client는 **Timeout + Retry + Circuit Breaker** 조합을 통해 **고가용성** 확보 가능.
✅ Resilience4j는 Feign Client 뿐 아니라 모든 서비스 호출 패턴에 유연하게 적용 가능.
✅ Circuit Breaker는 fallback 메서드 제공을 통해 **서비스 degrade** 시 안정성 보장.
✅ 적절한 Timeout 및 Retry 구성은 **시스템 전체 성능 보호**에 핵심 역할.

Retry, Resilience4j

Resilience4j는 모던한 Java 마이크로서비스 아키텍처에서 **회복 탄력성(Resilience)** 을 강화하기 위한 라이브러리다.
다음과 같은 기능을 모듈화된 구조로 제공한다:

- CircuitBreaker → 장애 발생 시 자동 차단
- Retry → 일시적 장애에 대한 **자동 재시도**
- RateLimiter → 요청 속도 제한
- Bulkhead → 병렬 처리량 제한 (격리)
- TimeLimiter → 시간 초과 관리

이번 절에서는 **Retry 기능 중심**으로 구성 방법과 활용 전략을 설명한다.

1️⃣ Retry 개요

Retry 사용 목적

- 네트워크 지연, 일시적 서비스 불가 등 **일시적 장애** 발생 시 자동으로 재시도하여 성공 확률을 높인다.
- 장애 대응 프로세스를 자동화하여 **서비스 안정성**을 향상시킨다.

적용 대상

- 외부 API 호출 (OpenFeign, RestTemplate, WebClient 등)
- DB 또는 Messaging 시스템
- 자체 서비스 내부 요청

---

2 기본 구성 흐름

```
Service 호출 시 → 실패 감지 → Retry 기능으로 자동 재시도
    ↓
성공 → 정상 결과 반환
    ↓
최대 재시도 초과 시 → 예외 발생 or fallback 처리
```

3 의존성 추가

```xml
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot3</artifactId>
</dependency>

<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-retry</artifactId>
</dependency>
```

4 application.yml 설정

```yaml
resilience4j:
  retry:
    instances:
      exampleRetry:
        max-attempts: 3          # 최대 시도 횟수
        wait-duration: 2s        # 재시도 간 간격
        retry-exceptions:
          - java.io.IOException
          - feign.FeignException$ServerError
        ignore-exceptions:
          - com.example.exception.CustomBadRequestException
```

주요 속성 설명

속성	설명
max-attempts	최대 재시도 횟수 (기본 3)
wait-duration	재시도 간 대기 시간 (기본 500ms)
retry-exceptions	재시도 대상 예외 목록
ignore-exceptions	재시도 제외 예외 목록

5️⃣ @Retry 어노테이션 사용 🚀

간단한 예제

```
@Service
@RequiredArgsConstructor
public class ExampleService {

    private final ExampleClient exampleClient;

    @Retry(name = "exampleRetry", fallbackMethod = "fallbackExample")
    public String callExampleApi() {
        return exampleClient.getProtectedData();
    }

    public String fallbackExample(Throwable t) {
        return "Fallback Response";
    }
}
```

동작 흐름

```
callExampleApi() 호출
    ↓
실패 발생 시 → exampleRetry 구성 적용 → 재시도
    ↓
최대 시도 초과 시 fallbackExample() 호출
```

주의 사항

- **fallbackMethod**의 시그니처는 원 메서드와 동일하거나 `Throwable` 추가 인자 허용.
- fallback 메서드가 없으면 **최종 예외 발생**.

6️⃣ Retry & OpenFeign 통합

Feign Client 호출에도 쉽게 적용 가능:

```
@Service
@RequiredArgsConstructor
public class ExampleService {

    private final ExampleClient exampleClient;

    @Retry(name = "exampleRetry", fallbackMethod = "fallbackExample")
    public String callExampleApi() {
        return exampleClient.getProtectedData();
    }

    public String fallbackExample(Throwable t) {
        return "Fallback Response";
```

```
    }
}
```

👉 FeignException, IOException 등을 대상으로 Retry 적용 가능.
👉 **Resilience4j Retry는 Feign 내부 Retryer보다 더 강력한 기능 제공** (Fallback 등 지원).

7️⃣ Retry + CircuitBreaker 결합 ⚡

Resilience4j는 **Retry + CircuitBreaker**를 쉽게 결합 구성 가능.

```yaml
resilience4j:
  retry:
    instances:
      exampleRetry:
        max-attempts: 3
        wait-duration: 2s

  circuitbreaker:
    instances:
      exampleCircuitBreaker:
        slidingWindowSize: 10
        failureRateThreshold: 50
        waitDurationInOpenState: 10s
```

```java
@Retry(name = "exampleRetry", fallbackMethod = "fallbackExample")
@CircuitBreaker(name = "exampleCircuitBreaker", fallbackMethod = "fallbackExample")
public String callExampleApi() {
    return exampleClient.getProtectedData();
}
```

- Retry → 일시적 장애 시 빠른 회복 시도
- CircuitBreaker → 지속적 장애 발생 시 차단 후 fallback 처리

👉 이 조합이 가장 많이 사용되는 패턴 🚀.

8️⃣ Retry Event 기반 모니터링

Resilience4j는 Retry Event 발행을 통해 **Metrics / 모니터링**도 지원.

```java
@Bean
public RetryEventConsumer<RetryOnRetryEvent> retryOnRetryEventConsumer() {
    return event -> System.out.println("Retry Event 발생: " + event.toString());
}
```

또는 Micrometer Metrics 연계 시 → Prometheus + Grafana 기반 모니터링 가능 📈.

9️⃣ 주요 활용 팁 🎁

- ✅ Retry 대상 예외는 **명확히 정의**할 것 → 모든 예외 재시도는 금물.
- ✅ Retry + CircuitBreaker 결합 시 **효과적 장애 대응** 가능.
- ✅ Retry 시 재시도 간 간격(Backoff)을 **적절히 조정**하여 과도한 요청 방지.
- ✅ fallbackMethod 구현 시 **서비스 degrade 처리 전략** 사전 설계 필요.

결론 🗂️

- ✅ Resilience4j Retry는 **네트워크/시스템 장애 대응 자동화** 기능 제공.
- ✅ 단일 구성으로 다양한 서비스 호출에 적용 가능.
- ✅ @Retry 어노테이션 사용 → 기존 코드에 쉽게 적용 가능.
- ✅ Retry + CircuitBreaker 조합 → MSA 서비스 회복 탄력성 향상.
- ✅ Prometheus 등과 연계 시 **실시간 모니터링** 가능.