Spring Boot와 MSA(Micro Service Architecture)

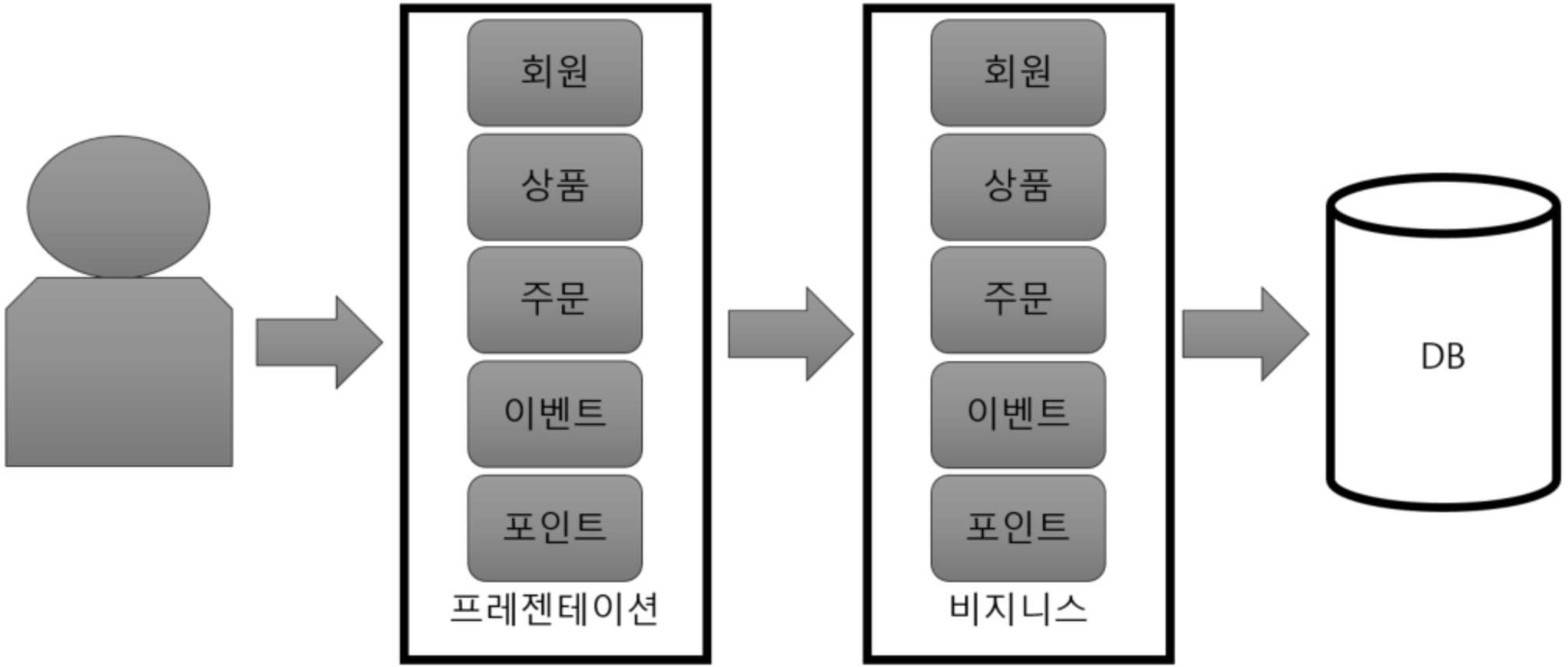
- 강경미 (carami@nate.com)

Monolithic Architecture

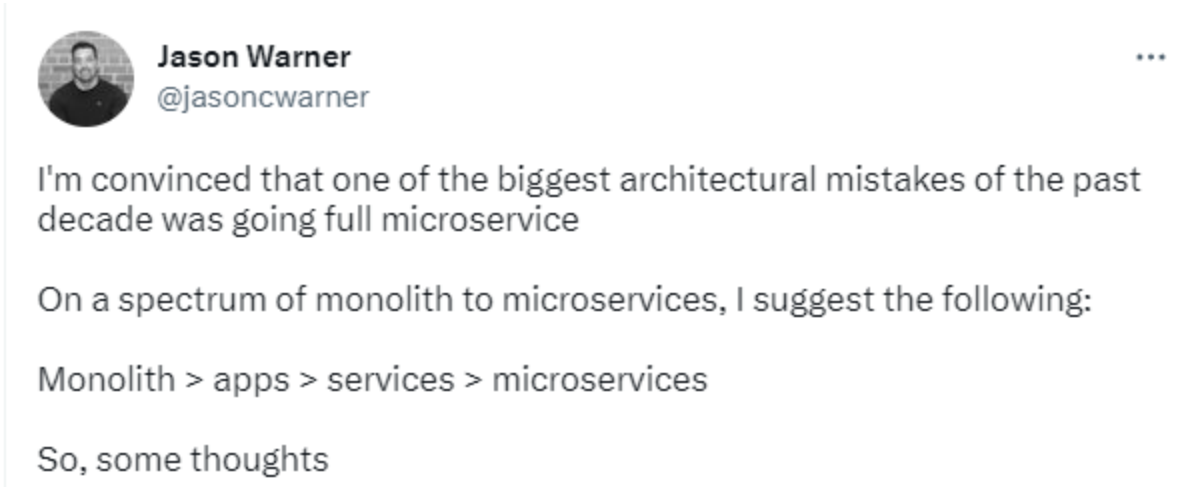
- Monolithic Architecture는 전통적인 애플리케이션 아키텍처로, 하나의 큰 애플리케이션으로 구성됩니다. 모든 기능이 하나의 애플리케이션 내부에 구현되어 있고, 데이터베이스, 사용자 인터페이스, 비즈니스 로직 등이 함께 구현되어 있습니다.
- Monolithic Architecture의 가장 큰 장점은 구현과 배포가 상대적으로 쉽다는 것입니다. 모든 코드가 함께 있기 때문에 개발자가 애플리케이션 전체를 이해하기 쉽습니다. 또한, 배포할 때도 단일 애플리케이션 파일을 배포하기 때문에, 배포 프로세스도 단순합니다.

- 하지만 Monolithic Architecture는 큰 단점이 있습니다. 애플리케이션이 커질수록 유지보수가 어려워지고, 확장성이 제한됩니다. 큰 애플리케이션에서는 다양한 기능이 함께 동작하기 때문에, 하나의 기능을 수정하거나 추가하면 전체 애플리케이션을 다시 배포해야 합니다. 이러한 문제는 애플리케이션의 규모가 커질수록 더욱 심각해집니다. 또한, 애플리케이션의 확장성도 제한됩니다. 모든 기능이 함께 동작하기 때문에, 특정 기능만 확장하거나 분리하는 것이 어렵습니다.
- 따라서, Monolithic Architecture는 작은 규모의 애플리케이션에서는 효율적일 수 있지만, 애플리케이션의 규모가 커질수록 유지보수와 확장성에 어려움을 겪는 구조입니다. 이러한 문제를 해결하기 위해 최근에는 마이크로서비스 아키텍처가 주목받고 있습니다.

- Monolithic Architecture의 예



- Github CTO : 10년간 가장 큰 아키텍처 실수 중 하나가 마이크로서비스로 전환한 것이라고 확신합니다.



https://twitter.com/jasoncwarner/status/1592227285024636928

https://news.hada.io/topic?id=7839

- 작은 회사는 Monolithic Architecture를 사용한다. 최대한 늦게 MSA로 전환한다.
- 규모가 큰 회사들은 이미 많은 서버들로 구성되어 있다.
- 그렇다면 어떻게 이 많은 서버들이 구성되었을까? 종로 vs 판교
- 회사들이 처음부터 계획하고 발전했을까? 어떻게 보면 살기위해 계속 추가.추가.추가 되었을 것이다.
- 보통 MSA를 한다는 곳은 운영되는 서비스를 잘 정리하는 것이다.
- 이말은 서버들, 컴포넌트간의 의존성을 관리하는 것이 굉장히 중요하다. 의존성을 잘 분리하는 작업. 기술보다 이게 더 중요하다.

MSA(Micro Service Architecture)란?

- 마이크로서비스 아키텍처란, 전통적인 단일 모놀리식 애플리케이션 대신, 여러 개의 작은 서비스들이 분산되어 각자의 역할을 수행하면서 함께 동작하는 애플리케이션 아키텍처를 말합니다. 각각의 마이크로서비스는 독립적으로 배포되고 실행될 수 있으며, 다른 마이크로서비스와 통신할 수 있어야 합니다.
- 이렇게 구성된 마이크로서비스 아키텍처는 기존의 단일 모놀리식 애플리케이션과 비교하여 유연성과 확장성이 뛰어나다는 장점이 있습니다. 하나의 문제가 발생했을 때 전체 애플리케이션을 다시 배포할 필요 없이, 해당 문제가 발생한 서비스만 수정하고 배포할 수 있습니다. 또한, 각각의 마이크로서비스는 자체적으로 확장 가능하므로, 필요에 따라 특정 서비스만 확장하고, 다른 서비스는 그대로 둘 수 있습니다. 이를 통해 자원의 효율적인 사용이 가능해집니다.

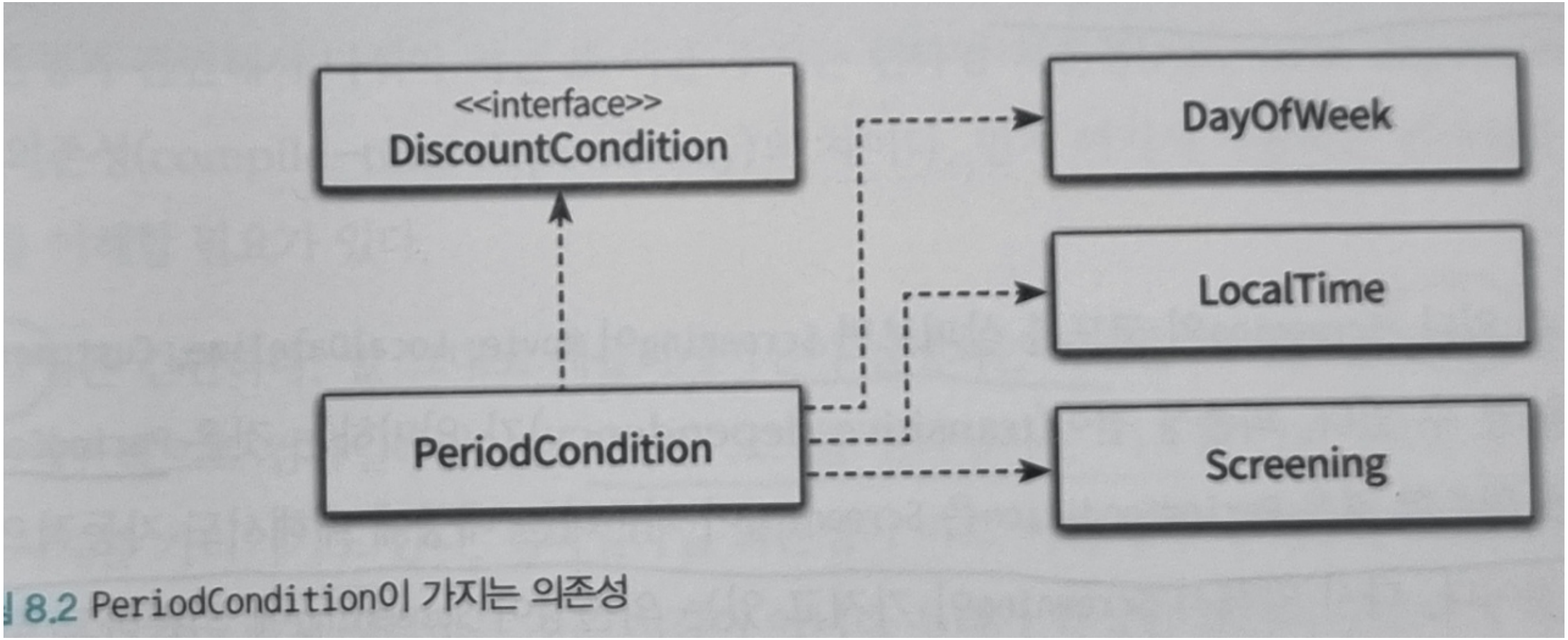
- 마이크로서비스 아키텍처는 여러 개의 작은 서비스들이 분산되어 동작하기 때문에, 각각의 서비스가 어떻게 통신하는지, 어떤 프로토콜을 사용하는지, 서비스의 위치는 어디인지 등을 관리하는 것이 중요합니다. 이를 위해 마이크로서비스 아키텍처에서는 API 게이트웨이, 로드 밸런싱, 서킷 브레이커, 서비스 디스커버리 등의 패턴과 기술이 사용됩니다.

의존성 관리가 중요!

의존성이란?

- 두 요소 사이의 의존성은 의존되는 요소가 변경될 때 의존하는 요소도 함께 변경될 수 있다는 것을 의미한다.
- 어떤 객체가 협력하기 위해 다른 객체를 필요로 할 때 두 객체 사이에 의존성이 존재
- 의존성은 방향성을 가지며 항상 단방향이다.
- 의존성은 변경에 의한 영향의 전파 가능성을 암시한다.
- 즉, 의존성은 함께 변경될 수 있는 가능성을 의미한다.
- 조영호 (오브젝트) 책 참조

예



8.2 PeriodCondition이 가지는 의존성

- UML에서는 세모에 점선이 구현을 나타내는데, 이 그림은 방향성을 중요하게 표현하였다.

Monolithic Architecture로 만들더라도, 분리가 쉽게 만들 필요가 있다.

그래야, MSA전환이 쉽다.

MSA의 이벤트 프로그래밍

MSA에서 이벤트 프로그래밍을 하는 이유

MSA(Microservices Architecture)는 하나의 대규모 애플리케이션을 작은 서비스 단위로 나누어 개발하고, 각각 독립적으로 실행 및 확장할 수 있게 하는 아키텍처입니다. 이러한 MSA에서 이벤트 프로그래밍은 여러 가지 이유로 중요합니다.

1. 비동기 통신: MSA에서 각각의 서비스는 독립적으로 실행되기 때문에, 이벤트 프로그래밍을 통해 비동기적으로 서비스간의 통신이 가능해집니다. 이로 인해, 서비스가 서로 블로킹(blocking)하지 않고, 효율적으로 처리될 수 있습니다.
2. 탄력성: 이벤트 프로그래밍을 활용하면, 서비스 장애 발생 시 이벤트를 통해 다른 서비스가 영향을 받지 않도록 처리할 수 있습니다. 이는 전체 시스템의 탄력성을 높이는 데 도움이 됩니다.
3. 확장성: 이벤트 프로그래밍은 서비스간의 통신을 이벤트 기반으로 처리함으로써, 서비스의 확장성을 향상시킵니다. 특정 서비스가 부하가 많을 경우, 해당 서비스만 증설하면 됩니다.

4. 결합도 감소: 이벤트 프로그래밍을 사용하면 서비스간의 직접적인 의존성을 줄일 수 있습니다. 이는 각 서비스가 독립적으로 변경되고 개선될 수 있게 하여, 전체적인 시스템의 유연성과 관리 용이성을 향상시킵니다.
5. 이벤트 추적 및 모니터링: 이벤트 프로그래밍을 통해 발생하는 이벤트들은 로깅 및 모니터링을 통해 추적할 수 있습니다. 이를 통해 서비스 간의 데이터 흐름 및 에러 처리를 보다 효과적으로 관리할 수 있습니다.

결론적으로, MSA에서 이벤트 프로그래밍은 서비스간의 통신, 탄력성, 확장성, 결합도 감소, 이벤트 추적 및 모니터링 등 여러 가지 측면에서 중요한 역할을 합니다. 이로 인해 MSA 기반의 프로젝트에서는 이벤트 프로그래밍을 적극 활용하는 것이 좋습니다.

서버간 이벤트 프로그래밍을 하려면?

- Spring Boot로 같은 서버 안에서 이벤트 처리를 하려면 ApplicationEvent로 처리 가능
- 서로 간에 이벤트 처리를 하려면 메시지 브로커(Message Broker)가 필요하다.
  - RabbitMQ: AMQP(Advanced Message Queuing Protocol)을 구현한 오픈소스 메시지 브로커로, 높은 확장성과 안정성을 제공합니다.
  - Apache Kafka: 분산 이벤트 스트리밍 플랫폼으로, 대용량 메시지 처리에 적합하며, 메시지 전달의 지연 시간을 최소화합니다.
  - ActiveMQ: Java Message Service(JMS)를 구현한 오픈소스 메시지 브로커로, Java 어플리케이션에 쉽게 통합할 수 있습니다.
  - Redis: 고성능 인메모리 데이터 스토어로써, 메시지 브로커로도 사용할 수 있습니다. Pub/Sub 모델을 지원하며, 빠른 응답 속도를 제공합니다.

이벤트를 사용하지 않는 예제

- 순차적으로 로직이 수행.

```
@Service
public class RegisterService {
    public void register(String name) {
        // 회원가입 처리 로직
        System.out.println("회원 추가 완료");

        // 가입 축하 메세지 전송
        System.out.println(name + "님에게 가입 축하 메세지를 전송했습니다.");

        // 가입 축하 쿠폰 발급
        System.out.println(name + "님에게 쿠폰을 전송했습니다.");
    }
}
```

이벤트를 사용하는 예제

```
@Service
public class RegisterEventService {

    @Autowired
    ApplicationEventPublisher publisher;

    public void register(String name) {
        // 회원가입 처리 로직
        System.out.println("회원 추가 완료");

        publisher.publishEvent(new RegisteredEvent(name)); // 2
    }
}
```

이벤트를 사용하는 예제 - Event 객체

```
public class RegisteredEvent {

    private String name;

    public RegisteredEvent(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

이벤트를 사용하는 예제 - 이벤트 핸들러

```
@Component
public class SmsEventHandler {

    @EventListener // 1
    public void sendSms(RegisteredEvent event) {
        System.out.println(event.getName() + "님에게 가입 축하 메세지를 전송했습니다.");
    }

    @EventListener
    public void makeCoupon(RegisteredEvent event) {
        System.out.println(event.getName() + "님에게 쿠폰을 전송했습니다.");
    }
}
```

이벤트를 사용하는 예제 - 테스트 컨트롤러

```
@RestController
@RequiredArgsConstructor
public class TestController {

    private final RegisterEventService service;

    @GetMapping("/register/{name}")
    public void register(@PathVariable String name) {
        service.register(name);
        System.out.println("회원가입을 완료했어요");
    }

}
```

이 예제의 문제점!

ApplicationEvent를 처리하는 메소드들은 같은 Thread를 이용한다.

```java
@EventListener // 1
public void sendSms(RegisteredEvent event) {
    System.out.println("sendSms thread name : " + Thread.currentThread().getName());
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    System.out.println(event.getName() + "님에게 가입 축하 메세지를 전송했습니다.");
}

@EventListener
public void makeCoupon(RegisteredEvent event) {

    System.out.println("makeCoupon thread name : " + Thread.currentThread().getName());
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    System.out.println(event.getName() + "님에게 쿠폰을 전송했습니다.");
}
```

실행결과

```
회원 추가 완료
sendSms thread name : http-nio-8080-exec-1 -- 출력후 2초뒤에 다음 실행
kim님에게 가입 축하 메세지를 전송했습니다.
makeCoupon thread name : http-nio-8080-exec-1 -- 출력후 2초뒤에 다음 실행
kim님에게 쿠폰을 전송했습니다.
회원가입을 완료했어요
```

Spring Boot의 서버 쓰레드 설정하기

application.yml

- 원래 서버 Thread는 Max는 200개가 기본. 기본적으로 10개로 시작.
- 요청부터 응답까지 서버스 Thread가 1개 사용된다.

```
server:
  tomcat:
    threads:
      max: 1
      min-spare: 1
```

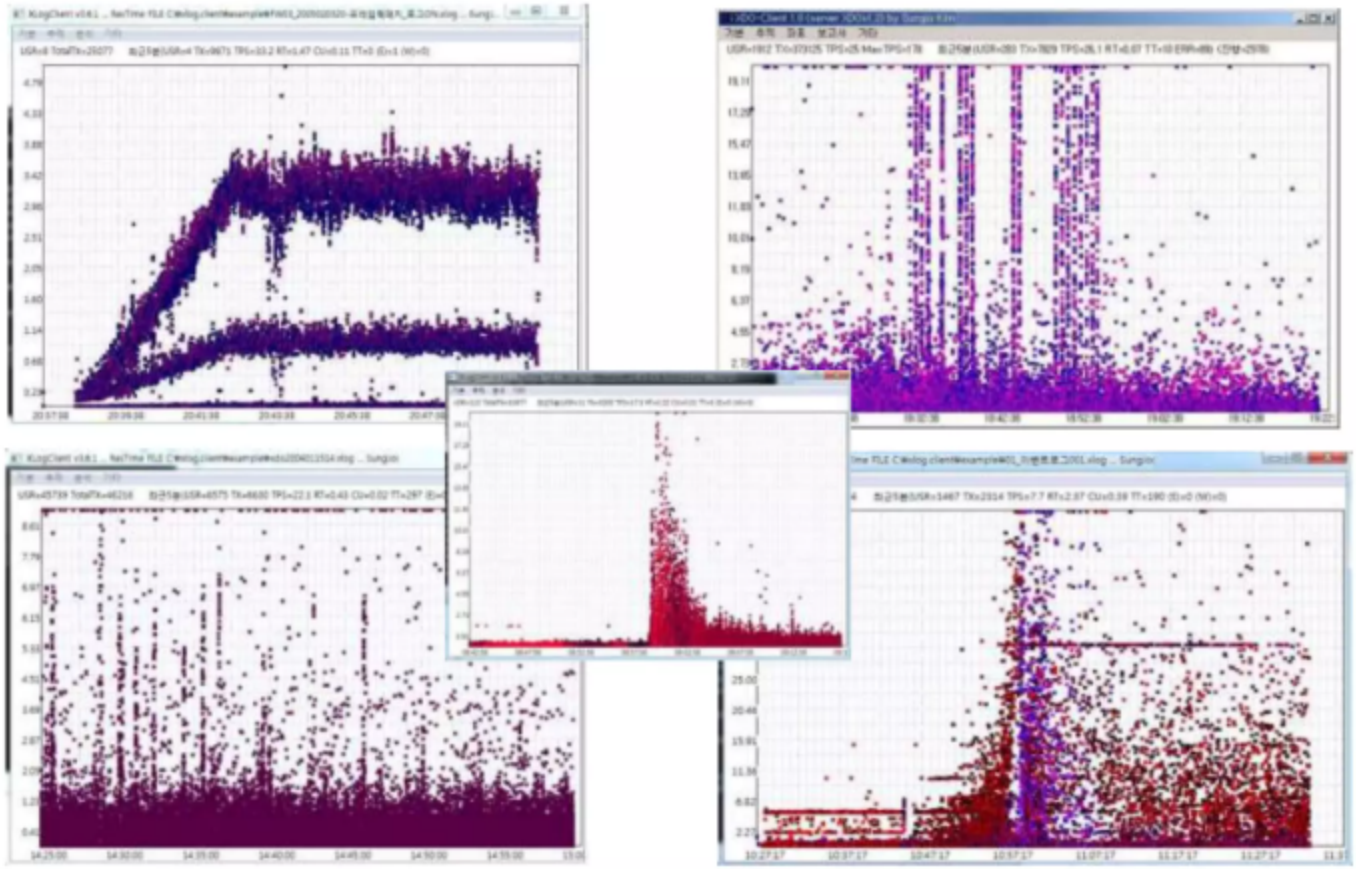
```java
@GetMapping("/hello")
public String hello(){
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    return "hello : " + Thread.currentThread().getName();
}
```

http://localhost:8080/hello 를 요청한다.

다음을 생각하자.

- 200개가 서버스레드 기본이다.
- 초당 500개의 요청이오고, 각 요청은 응답까지 평균 1초가 걸린다면 어떤일이 벌어질까?

- 스카우터(scouter) 같은 도구로 모니터링하면 다음과 같은 장애를 보게 된다.



해결방법

- 스레드수를 늘리던지 (메모리 등이 추가로 필요 - 스케일업)
- 서버수를 늘리던지 (스케일아웃)
- DBMS가 처리할 수 있는 양이 있다. DBMS는 커넥션의 제한이 있다. 굉장히 많은 요청이 올 경우 하나의 DBMS가 처리가 힘들게 된다. 그렇기 때문에 DBMS도 분리가 된다. MSA에서는 각각의 MS(Micro Service)별로 DBMS가 분리된다. 이말은 Table과 Table간의 의존성도 분리되도록 미리 설계되야 한다.

ApplicationEvent처리까지 서버 스레드로 실행된다.

- 이벤트는 비동기적으로 실행할 수 없을까?
- 같은 서버 스레드라 Controller도 요청이 끝나지 않고 있다.
- 이벤트는 던지고, 응답만 하면 좋을텐데!

Spring Boot와 Async 프로그래밍

Spring Boot와 Async 프로그래밍의 개요

- Spring Boot는 스프링 기반 애플리케이션을 빠르게 개발하고 실행할 수 있도록 도와주는 프레임워크입니다. 스프링 부트의 목표는 개발자들이 빠르게 웹 애플리케이션을 개발하고 배포할 수 있게 하여 생산성을 높이는 것입니다. 스프링 부트는 관례 위주의 설정(convention over configuration)을 통해 초기 설정을 최소화하고, 의존성 관리, 자동 설정, 애플리케이션 실행 및 배포를 용이하게 합니다.
- Async 프로그래밍은 비동기 프로그래밍이라고도 불리며, 동시에 여러 작업을 처리할 수 있도록 하는 프로그래밍 기법입니다. 이 방법을 사용하면, 일부 작업이 완료될 때까지 기다릴 필요 없이 다른 작업을 진행할 수 있습니다. 이를 통해 애플리케이션의 전체적인 처리 성능과 응답 속도를 향상시킬 수 있습니다.

- Spring Boot에서 Async 프로그래밍은 주로 @Async 애너테이션과 함께 사용됩니다. @Async 애너테이션을 메서드에 붙이면, 그 메서드는 비동기적으로 실행되게 됩니다. 비동기 실행은 별도의 스레드에서 이루어지기 때문에 메인 스레드가 해당 작업을 기다리지 않고 다른 작업을 계속 진행할 수 있습니다.
- 또한 Spring Boot에서는 Async 프로그래밍을 설정하기 위해 @EnableAsync 애너테이션을 사용하여 비동기 기능을 활성화할 수 있습니다. 이렇게 하면, 스프링 애플리케이션의 여러 부분에서 비동기 작업을 수행할 수 있게 됩니다. 이를 위해선 설정 클래스에서 AsyncConfigurer 인터페이스를 구현하여 별도의 스레드풀 설정이 가능하며, 이를 통해 비동기 작업의 성능을 더욱 최적화할 수 있습니다.
- 요약하면, Spring Boot는 스프링 기반 애플리케이션의 개발과 실행을 간소화하고 빠르게 만드는 프레임워크이며, Async 프로그래밍은 동시에 여러 작업을 처리하는 데 도움이 되는 기법입니다. Spring Boot에서 Async 프로그래밍을 사용하면 애플리케이션의 성능과 응답 속도를 개선할 수 있습니다

Async 프로그래밍 설정

```
@EnableAsync
@Configuration
public class AsyncConfig implements AsyncConfigurer {
    @Bean(name = "taskExecutor")
    public Executor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(50);
        executor.setMaxPoolSize(50);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("AsyncThread-");
        executor.initialize();
        return executor;
    }

    @Override
    public Executor getAsyncExecutor() {
        return taskExecutor();
    }

    @Override
    public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
        return new SimpleAsyncUncaughtExceptionHandler();
    }
}
```

```
@EnableAsync
```

이 애너테이션은 스프링 프레임워크의 비동기 기능을 활성화합니다. 즉, @Async 애너테이션이 붙은 메서드를 비동기로 실행하게 해줍니다.

```
@Configuration
```

이 애너테이션은 해당 클래스를 스프링 설정 클래스로 지정합니다. 이 클래스에서는 스프링 빈을 정의하고, 설정 값을 지정할 수 있습니다.

```
public class AsyncConfig implements AsyncConfigurer
```

AsyncConfig 클래스는 AsyncConfigurer 인터페이스를 구현합니다. 이를 통해 비동기 처리에 필요한 설정을 제공할 수 있습니다.

```
@Bean(name = "taskExecutor")
```

이 애너테이션은 taskExecutor() 메서드가 반환하는 객체를 스프링 빈으로 등록하며, 이름을 "taskExecutor"로 지정합니다.

```
public Executor taskExecutor()
```

ThreadPoolTaskExecutor 객체를 생성하고, 스레드풀 관련 설정을 지정한 뒤 초기화하여 Executor 타입으로 반환합니다.

```
executor.setCorePoolSize(2);
```

스레드풀의 기본 크기를 2로 설정합니다. 즉, 최소한 두 개의 스레드가 유지됩니다.

```
executor.setMaxPoolSize(10);
```

스레드풀의 최대 크기를 10으로 설정합니다. 스레드 수는 10개를 넘지 않게 됩니다.

```
executor.setQueueCapacity(100);
```

작업 대기열의 크기를 100으로 설정합니다. 최대 100개의 작업을 대기열에 넣을 수 있습니다.

```
executor.setThreadNamePrefix("AsyncThread-");
```

스레드풀에서 생성되는 스레드의 이름 접두사를 "AsyncThread-"로 설정합니다. 예를 들어, "AsyncThread-1", "AsyncThread-2" 등이 됩니다.

```
executor.initialize();
```

ThreadPoolTaskExecutor 객체를 초기화합니다. 초기화 후 스레드풀이 사용 가능해집니다.

```
@Override
public Executor getAsyncExecutor()
```

이 메서드는 비동기 작업에 사용될 Executor를 반환합니다. 여기서는 taskExecutor() 메서드를 호출하여 반환하므로, 우리가 정의한 스레드풀이 사용됩니다.

```
@Override
public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler()
```

이 메서드는 비동기 작업 중 발생하는 예외를 처리할 AsyncUncaughtExceptionHandler를 반환합니다. 여기서는 SimpleAsyncUncaughtExceptionHandler를 사용하므로, 비동기 작업 중 발생하는 예외는 단순히 로그로 출력됩니다.

기존 핸들러 코드의 수정

```
@Component
public class SmsEventHandler {

    @Async
    @EventListener // 1
    public void sendSms(RegisteredEvent event) throws InterruptedException{
        Thread.sleep(3000);
        System.out.println("sendSms : " + Thread.currentThread().getName());
        System.out.println(event.getName() + "님에게 가입 축하 메세지를 전송했습니다.");
    }

    @Async
    @EventListener
    public void makeCoupon(RegisteredEvent event) throws InterruptedException {
        Thread.sleep(5000);
        System.out.println("makeCoupon : " + Thread.currentThread().getName());
        System.out.println(event.getName() + "님에게 쿠폰을 전송했습니다.");
    }
}
```

Async 예제 1

```java
@Service
public class GetOneService {
    public String getOne(){
        try{
            Thread.sleep(2000);
        }catch (Exception ex){
            ex.printStackTrace();
        }
        System.out.println("getOne : " + Thread.currentThread().getName());
        return "one";
    }
}
```

```java
@Service
public class GetTwoService {
    public String getTwo(){
        try{
            Thread.sleep(2000);
        }catch (Exception ex){
            ex.printStackTrace();
        }
        System.out.println("getTwo : " + Thread.currentThread().getName());
        return "two";
    }
}
```

```java
@Service
@RequiredArgsConstructor
public class AsyncTestService {
    private final GetOneService getOneService;
    private final GetTwoService getTwoService;
    public String executeWithoutAsync(){
        long start = System.nanoTime();
        String value1 = getOneService.getOne();
        String value2 = getTwoService.getTwo();
        long end = System.nanoTime();

        System.out.println("value1 : " + value1);
        System.out.println("value2 : " + value2);

        String str = "executeWithoutAsync 걸린 시간 : " + formatNanoseconds(end - start);
        System.out.println(str);

        return str;

    }
}
```

```java
@RestController
@RequiredArgsConstructor
public class TestController {

    private final AsyncTestService asyncTestService;

    @GetMapping("/test1")
    public String test1(){
        return asyncTestService.executeWithoutAsync();
    }

}
```

Async 예제 2

```java
@Service
@RequiredArgsConstructor
public class AsyncTestService {
    private final GetOneService getOneService;
    private final GetTwoService getTwoService;

    public String executeWithCompletableFuture() {
        long start = System.nanoTime();

        CompletableFuture<String> value1Future = CompletableFuture.supplyAsync(() -> getOneService.getOne());
        CompletableFuture<String> value2Future = CompletableFuture.supplyAsync(() -> getTwoService.getTwo());

        CompletableFuture.allOf(value1Future, value2Future).join();

        String value1 = value1Future.join();
        String value2 = value2Future.join();

        System.out.println("value1 : " + value1);
        System.out.println("value2 : " + value2);

        long end = System.nanoTime();
        String str = "executeWithAsync 걸린 시간 : " + formatNanoseconds(end - start);
        System.out.println(str);

        return str;
    }
```

```java
@RestController
@RequiredArgsConstructor
public class TestController {

    private final AsyncTestService asyncTestService;


    @GetMapping("/test2")
    public String test3(){
        return asyncTestService.executeWithCompletableFuture();
    }
}
```

미리 설정해둔 커넥션 풀 사용하도록 코드 수정

```
@Service
@RequiredArgsConstructor
public class AsyncTestService {
    private final GetOneService getOneService;
    private final GetTwoService getTwoService;

    @Autowired
    @Qualifier("taskExecutor")
    private Executor taskExecutor;

    public String executeWithCompletableFuture() {
        long start = System.nanoTime();

        CompletableFuture<String> value1Future = CompletableFuture.supplyAsync(() -> getOneService.getOne(), taskExecutor);
        CompletableFuture<String> value2Future = CompletableFuture.supplyAsync(() -> getTwoService.getTwo(), taskExecutor);

        CompletableFuture.allOf(value1Future, value2Future).join();

        String value1 = value1Future.join();
        String value2 = value2Future.join();

        System.out.println("value1 : " + value1);
        System.out.println("value2 : " + value2);

        long end = System.nanoTime();
        String str = "executeWithAsync 걸린 시간 : " + formatNanoseconds(end - start);
        System.out.println(str);

        return str;
    }

}
```

Async 예제 3

```java
@Service
public class GetOneService {
    @Async
    public Future<String> getAsyncOne(){
        try{
            Thread.sleep(2000);
        }catch (Exception ex){
            ex.printStackTrace();
        }
        System.out.println("getOne : " + Thread.currentThread().getName());
        return new AsyncResult<>("one");
    }
}
```

```java
@Service
public class GetTwoService {
    @Async
    public Future<String> getAsyncTwo(){
        try{
            Thread.sleep(2000);
        }catch (Exception ex){
            ex.printStackTrace();
        }
        System.out.println("getTwo : " + Thread.currentThread().getName());
        return new AsyncResult<>("two");
    }
}
```

```java
@Service
@RequiredArgsConstructor
public class AsyncTestService {
    private final GetOneService getOneService;
    private final GetTwoService getTwoService;

    public String executeWithAsync() {
        long start = System.nanoTime();

        Future<String> value1Future = getOneService.getAsyncOne();
        Future<String> value2Future = getTwoService.getAsyncTwo();

        while (!value1Future.isDone() || !value2Future.isDone()) {
            // value1Future와 value2Future가 모두 완료될 때까지 기다립니다.
            //System.out.print(".");
        }

        try {
            String value1 = value1Future.get();
            String value2 = value2Future.get();

            System.out.println("value1 : " + value1);
            System.out.println("value2 : " + value2);
        }catch(Exception ex){
            ex.printStackTrace();
        }

        long end = System.nanoTime();
        String str = "executeWithAsync 걸린 시간 : " + formatNanoseconds(end - start);
        System.out.println(str);

        return str;
    }

}
```

```
@RestController
@RequiredArgsConstructor
public class TestController {

    private final AsyncTestService asyncTestService;

    @GetMapping("/test3")
    public String test2(){
        return asyncTestService.executeWithAsync();
    }

}
```

외부 API 호출하기

Spring Boot에서 RestTemplate을 이용한 외부 API 호출하기

Spring Boot를 사용하여 개발하다 보면, 외부 서비스의 API를 호출해야 할 경우가 종종 발생합니다. 이때, Spring Boot에서는 RestTemplate이라는 라이브러리를 사용하여 편리하게 외부 API와 통신할 수 있습니다. 본 문서에서는 Spring Boot에서 RestTemplate을 이용한 외부 API 호출의 개요에 대해 설명합니다.

1. RestTemplate 소개
   RestTemplate은 Spring Framework에서 제공하는 HTTP 클라이언트로, RESTful 웹 서비스와 손쉽게 통신할 수 있는 기능을 제공합니다. RestTemplate은 다양한 HTTP 메서드를 사용하여 API 호출을 처리할 수 있으며, 자동으로 JSON 형식의 데이터를 Java 객체로 변환해줍니다.

- HttpClient를 이용할 수도 있습니다.

2. RestTemplate 빈 생성

Spring Boot 2.3 이후부터는 RestTemplate 빈이 기본적으로 생성되지 않기 때문에, 직접 빈을 생성해야 합니다. 아래 코드를 Java 설정 파일에 추가하여 RestTemplate 빈을 생성할 수 있습니다.

```
@Configuration
public class RestTemplateConfig {

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

3. 외부 API 호출 예제

RestTemplate을 사용하여 외부 API를 호출하는 방법은 다양한 HTTP 메서드(GET, POST, PUT, DELETE 등)에 따라 다릅니다. 아래는 GET 메서드를 사용한 예제입니다.

```
@Service
public class ApiService {
    private final RestTemplate restTemplate;

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

    public String getExternalData() {
        String apiUrl = "https://api.example.com/data";
        ResponseEntity<String> response = restTemplate.getForEntity(apiUrl, String.class);
        return response.getBody();
    }
}
```

위의 예제에서는 외부 API의 주소를 apiUrl에 지정하고, restTemplate.getForEntity() 메서드를 사용하여 GET 요청을 보냅니다. 그리고 받은 응답을 ResponseEntity 객체로 받아, 필요한 데이터를 처리합니다.

Spring Boot에서 RestTemplate을 사용하면, 외부 API 호출을 손쉽게 처리할 수 있습니다. RestTemplate은 다양한 HTTP 메서드를 지원하며, JSON 형식의 데이터를 자동으로 변환해줍니다. 이를 통해 개발자는 외부 서비스와의 통신 과정을 쉽게 구현할 수 있습니다.

예

Todos Controller

```
@RequestMapping("/todos")
@RestController
public class TodosController {
    private List<Todo> todos;
    private static int ID = 1;

    public TodosController() {
        todos = new ArrayList<>();
        todos.add(new Todo(ID++, "낮잠자기", false));
        todos.add(new Todo(ID++, "비타민먹기", false));
        todos.add(new Todo(ID++, "사과먹기", false));
        todos.add(new Todo(ID++, "책읽기", false));
        todos.add(new Todo(ID++, "청소하기", true));
    }
```

```java
@GetMapping
public List<Todo> getTodos(){
    return todos;
}

@GetMapping("/{id}")
public Todo getTodo(@PathVariable("id") int id){
    return todos.stream()
            .filter(todo -> todo.getId() == id)
            .findFirst()
            .orElse(null);
}
```

```java
@DeleteMapping("/{id}")
public Todo delete(@PathVariable("id") int id){
    Todo deletedTodo = todos.stream()
            .filter(todo -> todo.getId() == id)
            .findFirst()
            .orElse(null);

    if (deletedTodo != null) {
        todos.remove(deletedTodo);
    }

    return deletedTodo;
}

@PostMapping
public Todo addTodo(@RequestBody Todo todo){
    todo.setId(ID++);
    todos.add(todo);
    return todo;
}
```

```
    @PatchMapping
    public Todo updateTodo(@RequestBody Todo todo) {
        Todo updatedTodo = todos.stream()
                .filter(t -> t.getId() == todo.getId())
                .findFirst()
                .orElse(null);

        if (updatedTodo != null) {
            updatedTodo.setTitle(todo.getTitle());
            updatedTodo.setDone(todo.isDone());
        }

        return updatedTodo;
    }
}
```

TodosProxyController

```
@RestController
@RequestMapping("/todos-proxy")
public class TodosProxyController {

    private final String todosUrl = "http://localhost:8080/todos";
    private final RestTemplate restTemplate;

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

    @GetMapping
    public List<Todo> getTodos() {
        ResponseEntity<Todo[]> response = restTemplate.getForEntity(todosUrl, Todo[].class);
        return Arrays.asList(response.getBody());
    }
```

```
@GetMapping("/{id}")
public Todo getTodo(@PathVariable("id") int id) {
    ResponseEntity<Todo> response = restTemplate.getForEntity(todosUrl + "/" + id, Todo.class);
    return response.getBody();
}

@DeleteMapping("/{id}")
public Todo delete(@PathVariable("id") int id) {
    ResponseEntity<Todo> response = restTemplate.exchange(todosUrl + "/" + id, HttpMethod.DELETE, null, Todo.class);
    return response.getBody();
}
```

```
    @PostMapping
    public Todo addTodo(@RequestBody Todo todo) {
        ResponseEntity<Todo> response = restTemplate.postForEntity(todosUrl, todo, Todo.class);
        return response.getBody();
    }

    @PatchMapping
    public Todo updateTodo(@RequestBody Todo todo) {
        HttpEntity<Todo> requestEntity = new HttpEntity<>(todo);
        ResponseEntity<Todo> response = restTemplate.exchange(todosUrl, HttpMethod.PATCH, requestEntity, Todo.class);
        return response.getBody();
    }
}
```

예

```
@RequestMapping("/posts")
@RestController
public class PostController {
    private List<Post> posts;
    private static int ID = 1;

    public PostController() {
        posts = new ArrayList<>();
        posts.add(new Post(ID++, "Title1", "Content1"));
        posts.add(new Post(ID++, "Title2", "Content2"));
        posts.add(new Post(ID++, "Title3", "Content3"));
        posts.add(new Post(ID++, "Title4", "Content4"));
    }

    @GetMapping("/{id}")
    public Post getPost(@PathVariable("id") int id){
        return posts.stream()
                .filter(post -> post.getId() == id)
                .findFirst()
                .orElse(null);
    }
}
```

```
public class PostDetail {
    private Post post;
    private List<Comment> comments;

    public PostDetail(Post post, List<Comment> comments) {
        this.post = post;
        this.comments = comments;
    }

    // Getters and Setters
}
```

```java
@RestController
@RequestMapping("/post-details")
public class PostDetailsProxyController {

    private final String postsUrl = "http://localhost:8080/posts";
    private final String commentsUrl = "http://external-api.example.com/comments";
    private final RestTemplate restTemplate;

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

    @GetMapping("/{id}")
    public PostDetail getPostDetail(@PathVariable("id") int id) {
        ResponseEntity<Post> postResponse = restTemplate.getForEntity(postsUrl + "/" + id, Post.class);
        Post post = postResponse.getBody();

        ResponseEntity<Comment[]> commentsResponse = restTemplate.getForEntity(commentsUrl + "/by-post/" + id, Comment[].class);
        List<Comment> comments = Arrays.asList(commentsResponse.getBody());

        return new PostDetail(post, comments);
    }
}
```

예

- 게시물 등록,수정,삭제, 1건조회, 페이징처리 조회
- 댓글 등록, 댓글 삭제, 댓글 목록 조회
- 위의 2가지 API를 호출해서 결과를 만들어내는 API

```java
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String content;

    // 생성자, 게터, 세터
}
```

```
public interface PostRepository extends JpaRepository<Post, Long> {
}
```

```java
@RestController
@RequestMapping("/posts")
public class PostController {
    private final PostRepository postRepository;

    public PostController(PostRepository postRepository) {
        this.postRepository = postRepository;
    }

    @GetMapping
    public Page<Post> getAllPosts(@RequestParam(defaultValue = "0") int page,
                                  @RequestParam(defaultValue = "10") int size) {
        return postRepository.findAll(PageRequest.of(page, size));
    }
```

```
    @GetMapping("/{id}")
    public Post getPostById(@PathVariable Long id) {
        return postRepository.findById(id).orElse(null);
    }

    @PostMapping
    public Post createPost(@RequestBody Post post) {
        return postRepository.save(post);
    }
```

```
    @PutMapping("/{id}")
    public Post updatePost(@PathVariable Long id, @RequestBody Post updatedPost) {
        return postRepository.findById(id).map(post -> {
            post.setTitle(updatedPost.getTitle());
            post.setContent(updatedPost.getContent());
            return postRepository.save(post);
        }).orElse(null);
    }

    @DeleteMapping("/{id}")
    public void deletePost(@PathVariable Long id) {
        postRepository.deleteById(id);
    }
}
```

```java
@Entity
public class Comment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long postId;
    private String author;
    private String content;

    // 생성자, 게터, 세터
}
```

```
public interface CommentRepository extends JpaRepository<Comment, Long> {
    List<Comment> findByPostId(Long postId);
}
```

```java
@RestController
@RequestMapping("/comments")
public class CommentController {
    private final CommentRepository commentRepository;

    public CommentController(CommentRepository commentRepository) {
        this.commentRepository = commentRepository;
    }

    @GetMapping("/by-post/{postId}")
    public List<Comment> getCommentsByPostId(@PathVariable Long postId) {
        return commentRepository.findByPostId(postId);
    }
```

```
    @PostMapping
    public Comment createComment(@RequestBody Comment comment) {
        return commentRepository.save(comment);
    }

    @DeleteMapping("/{id}")
    public void deleteComment(@PathVariable Long id) {
        commentRepository.deleteById(id);
    }
}
```

```
// 필드, 생성자, 게터, 세터
public class Post {
    private Long id;
    private String title;
    private String content;
}

package com.example.postcomment.dto;

// 필드, 생성자, 게터, 세터
public class Comment {
    private Long id;
    private Long postId;
    private String author;
    private String content;
}
```

```
public class PostDetail {
    private Post post;
    private List<Comment> comments;

    // 생성자, 게터, 세터
}
```

```java
@RestController
public class PostCommentController {
    private final RestTemplate restTemplate = new RestTemplate();
    private final String postServiceUrl = "http://localhost:8081/posts";
    private final String commentServiceUrl = "http://localhost:8082/comments/by-post";

    @GetMapping("/post-details/{postId}")
    public PostDetail getPostDetail(@PathVariable Long postId) {
        ResponseEntity<Post> postResponse = restTemplate.getForEntity(postServiceUrl + "/" + postId, Post.class);
        Post post = postResponse.getBody();

        ResponseEntity<Comment[]> commentResponse = restTemplate.getForEntity(commentServiceUrl + "/" + postId, Comment[].class);
        List<Comment> comments = Arrays.asList(commentResponse.getBody());

        return new PostDetail(post, comments);
    }
}
```

```java
@RestController
public class PostCommentController {
    private final RestTemplate restTemplate = new RestTemplate();
    private final String postServiceUrl = "http://localhost:8081/posts";
    private final String commentServiceUrl = "http://localhost:8082/comments/by-post";

    @GetMapping("/post-details/{postId}")
    public PostDetail getPostDetail(@PathVariable Long postId) throws ExecutionException, InterruptedException {
        CompletableFuture<Post> postFuture = CompletableFuture.supplyAsync(() -> {
            ResponseEntity<Post> postResponse = restTemplate.getForEntity(postServiceUrl + "/" + postId, Post.class);
            return postResponse.getBody();
        });

        CompletableFuture<List<Comment>> commentsFuture = CompletableFuture.supplyAsync(() -> {
            ResponseEntity<Comment[]> commentResponse = restTemplate.getForEntity(commentServiceUrl + "/" + postId, Comment[].class);
            return Arrays.asList(commentResponse.getBody());
        });

        CompletableFuture.allOf(postFuture, commentsFuture).join();

        Post post = postFuture.get();
        List<Comment> comments = commentsFuture.get();

        return new PostDetail(post, comments);
    }
}
```

API 재시도 및 리트라이

Spring Boot에서 API 호출 시 실패한 경우 재시도를 수행하거나 대안 흐름을 실행하기 위한 방법들을 다음과 같이 소개합니다.

재시도를 위한 Spring Retry 라이브러리

Spring Retry는 메서드 호출 실패 시 자동으로 재시도할 수 있는 기능을 제공하는 라이브러리입니다. Spring Boot에서 사용하려면 먼저 의존성을 추가해야 합니다.

```
implementation 'org.springframework.retry:spring-retry:1.3.1'
```

재시도를 적용할 메서드에 @Retryable 애너테이션을 추가하고, 재시도 관련 설정을 지정합니다.

```
@Service
public class ApiService {
    private final RestTemplate restTemplate;

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

    @Retryable(
        value = {HttpStatusCodeException.class},
        maxAttempts = 3,
        backoff = @Backoff(delay = 1000)
    )
```

```
    public String getExternalData() {
        String apiUrl = "https://api.example.com/data";
        ResponseEntity<String> response = restTemplate.getForEntity(apiUrl, String.class);
        return response.getBody();
    }
}
```

위 코드에서는 HttpStatusCodeException 발생 시 최대 3회 재시도를 하며, 각 재시도 사이에 1초의 딜레이를 설정했습니다.

재시도가 모두 실패한 경우 처리를 원한다면, @Recover 애너테이션을 사용하여 대안 메서드를 구현할 수 있습니다.

```
@Recover
public String recover(HttpStatusCodeException e) {
    // 대안 흐름을 실행
    return "Alternative data";
}
```

Resilience4j 라이브러리 사용

Resilience4j는 네트워크 호출에 대한 회복 전략을 구현하는 데 도움이 되는 라이브러리입니다. 재시도, 서킷 브레이커, 타임아웃 등 다양한 기능을 제공합니다. 의존성을 추가하고 설정을 적용한 뒤 사용하면 됩니다.

```
implementation 'io.github.resilience4j:resilience4j-spring-boot2:1.7.0`
```

application.yml에 Resilience4j 설정을 추가합니다.

```
resilience4j.retry:
  configs:
    default:
      maxAttempts: 3
      waitDuration: 1000ms
      retryExceptions:
        - org.springframework.web.client.HttpStatusCodeException
  instances:
    retryAPI:
      baseConfig: default
```

서비스에서 Retry 객체를 사용하여 API 호출을 처리합니다.

```
@Service
public class ApiService {
    private final RestTemplate restTemplate;
    private final RetryRegistry retryRegistry;

    public ApiService(RestTemplate restTemplate, RetryRegistry retryRegistry) {
        this.restTemplate = restTemplate;
        this.retryRegistry = retryRegistry;
    }

    public String getExternalData() {
        Retry retry = retryRegistry.retry("retryAPI");
        String apiUrl = "https://api.example.com/data";

        CheckedFunction0<ResponseEntity<String>> retryRequest = Retry.decorateCheckedSupplier(
                retry,
                () -> restTemplate.getForEntity(apiUrl, String.class)
        );

        try {
            ResponseEntity<String> response = retryRequest.apply();
            return response.getBody();
        } catch (Throwable e) {
            // 대안 흐름을 실행
            return "Alternative data";
        }
    }
}
```

위 코드에서는 `retryRegistry.retry("retryAPI")`를 통해 설정한 Retry 객체를 가져옵니다. 그런 다음, `Retry.decorateCheckedSupplier()` 메서드를 사용하여 API 호출을 재시도 로직과 결합합니다. 이렇게 설정하면, 설정한대로 재시도가 적용됩니다. 모든 재시도가 실패한 경우에 대안 흐름을 실행하도록 예외 처리를 구현했습니다.

요약하면, Spring Boot에서 외부 API 호출 시 실패한 경우 재시도를 수행하거나 대안 흐름을 실행하기 위해 Spring Retry 라이브러리나 Resilience4j 라이브러리를 사용할 수 있습니다. 이 두 라이브러리를 활용하면, 다양한 재시도 전략을 적용하고 대안 흐름을 구현할 수 있어 서비스의 안정성을 높일 수 있습니다.

MSA를 위한 Spring Cloud

MSA를 위한 Spring Cloud

- 서비스 디스커버리 (Service Discovery) : Spring Cloud에서는 Eureka, Consul, Zookeeper 등의 서비스 디스커버리 서버를 지원합니다. 서비스 디스커버리를 사용하면, 마이크로서비스 간 통신을 위한 네트워크 위치 정보를 찾고, 관리할 수 있습니다.
- API 게이트웨이 (API Gateway) : Spring Cloud Gateway를 사용하여 API 게이트웨이를 구현할 수 있습니다. 게이트웨이를 통해 요청을 특정 서비스로 라우팅하고, 로드 밸런싱, 인증 및 권한 부여 등의 기능을 수행할 수 있습니다.
- 클라이언트 사이드 로드 밸런싱 (Client-Side Load Balancing) : Ribbon을 사용하여 클라이언트 사이드 로드 밸런싱을 구현할 수 있습니다. 여러 서비스 인스턴스 간에 부하를 분산시키고, 장애를 격리할 수 있습니다.

- 서킷 브레이커 (Circuit Breaker) : Hystrix 또는 Resilience4j를 사용하여 서킷 브레이커 패턴을 구현할 수 있습니다. 서킷 브레이커는 외부 서비스에 대한 과도한 요청을 차단하고, 대안 흐름을 제공함으로써 전체 시스템의 안정성을 높입니다.
- 분산 추적 (Distributed Tracing) : Spring Cloud Sleuth와 Zipkin을 사용하여 마이크로서비스 환경에서의 분산 추적을 수행할 수 있습니다. 각 서비스 간의 통신 과정을 모니터링하고, 문제를 신속하게 파악할 수 있습니다.
- 외부 설정 중앙화 (Externalized Configuration) : Spring Cloud Config를 사용하여 외부 설정을 중앙화할 수 있습니다. 각 마이크로서비스가 별도의 설정 파일을 갖는 대신, 중앙 서버에서 설정을 관리하고 배포할 수 있습니다.
- 메시지 기반 마이크로서비스 (Message-Driven Microservices) : Spring Cloud Stream과 Spring Cloud Bus를 사용하여 메시지 기반의 비동기 마이크로서비스 아키텍처를 구축

Spring Cloud를 대체하는 기술들

1. API 게이트웨이:
    - Kong: 오픈소스 API 게이트웨이로서, Lua를 기반으로 한 강력한 플러그인 아키텍처를 제공합니다. 여러 기능을 확장할 수 있고, 손쉽게 관리할 수 있습니다.
    - Apigee: 구글에서 제공하는 상용 API 관리 플랫폼입니다. 기업 규모의 API 게이트웨이로서, 고성능과 다양한 기능을 제공합니다.
2. 서비스 디스커버리:
    - Consul: HashiCorp에서 개발한 오픈소스 서비스 디스커버리 솔루션입니다. 서비스 메시 구현과 키-값 스토어 기능도 제공합니다.
    - etcd: CNCF에서 관리하는 오픈소스 분산 키-값 스토어로서, 서비스 디스커버리 기능도 제공합니다.

3. 로드 밸런서:

   - Envoy: Lyft에서 개발한 고성능 오픈소스 로드 밸런서 및 프록시입니다. 클라이언트 및 서버 사이드 로드 밸런싱을 제공하며, gRPC, HTTP/2 등의 최신 프로토콜을 지원합니다.
   - HAProxy: 널리 사용되는 오픈소스 로드 밸런서로서, 고성능과 안정성을 제공합니다. 여러 프로토콜과 플랫폼을 지원하며, 다양한 환경에서 구성할 수 있습니다.

4. 서킷 브레이커:

   - Istio: 서비스 메시 구현을 위한 오픈소스 플랫폼으로, 내장 서킷 브레이커 기능을 제공합니다. 마이크로서비스의 통신 제어, 관찰 및 안전 기능을 지원합니다.
   - Polly: .NET 기반의 오픈소스 라이브러리로서, 서킷 브레이커, 블키헤드, 타임아웃 등의 장애 복구 기능을 제공합니다.

5. 분산 추적:
    - Jaeger: Uber에서 개발한 오픈소스 분산 추적 시스템입니다. 성능 최적화와 문제 진단을 위한 강력한 기능을 제공합니다.
    - Elastic APM: Elasticsearch에서 제공하는 응용 프로그램 성능 모니터링 솔루션입니다. 분산 추적 기능 외에도 메트릭 및 로깅을 통합하여 애플리케이션 성능 문제를 신속하게 파악할 수 있습니다.
6. 외부 설정 중앙화:
    - Consul: 앞서 언급한 바와 같이, Consul은 분산 키-값 스토어를 제공하므로 외부 설정 중앙화에 사용할 수 있습니다.
    - etcd: 이 역시 앞서 언급한 바와 같이, etcd는 분산 키-값 스토어를 제공하므로 외부 설정 중앙화에 사용할 수 있습니다.

7. 메시지 기반 마이크로서비스:
    - Apache Kafka: 고성능의 분산 스트리밍 플랫폼으로서, 메시지 기반의 마이크로서비스 아키텍처 구축에 적합합니다. 대규모 데이터 처리와 실시간 스트리밍 처리를 지원합니다.
    - RabbitMQ: 널리 사용되는 오픈소스 메시지 브로커로서, 메시지 기반의 마이크로서비스 아키텍처 구축에 사용할 수 있습니다. 다양한 메시지 전달 패턴과 메시지 라우팅 기능을 제공합니다.

AWS의 MSA를 위한 서비스

1. API 게이트웨이:

- Amazon API Gateway: AWS에서 제공하는 완전관리형 API 게이트웨이 서비스입니다. 다양한 백엔드 서비스와 연결하고, 요청에 대한 인증 및 권한 부여, 캐싱, 로깅 및 모니터링 등의 기능을 지원합니다.

2. 서비스 디스커버리:

- AWS Cloud Map: AWS에서 제공하는 서비스 디스커버리 솔루션으로, 애플리케이션 리소스를 등록하고 검색할 수 있습니다. 로드 밸런서, 람다, RDS 등 다양한 리소스를 관리할 수 있습니다.

3. 로드 밸런서:

- Application Load Balancer: HTTP 및 HTTPS 트래픽을 처리하는 레이어 7 로드 밸런서입니다. 복잡한 라우팅 기능을 제공하며, 웹소켓 및 gRPC 프로토콜을 지원합니다.
- Network Load Balancer: 레이어 4 로드 밸런서로, TCP 및 UDP 트래픽을 처리합니다. 고성능과 초저지연을 제공하며, 전체 지역에서 작동합니다.

4. 서킷 브레이커:

- AWS App Mesh: AWS에서 제공하는 서비스 메시 구현으로, Envoy 프록시를 사용하여 애플리케이션 네트워크 트래픽을 관리합니다. 서킷 브레이커 기능 외에도, 리트라이 정책, 요청 라우팅, 로드 밸런싱 등을 제공합니다.

5. 분산 추적:

- AWS X-Ray: AWS에서 제공하는 분산 추적 시스템으로, 애플리케이션 내 서비스 간의 통신 및 성능 분석을 수행할 수 있습니다. 다양한 AWS 서비스와 통합되어 있어, 상세한 분석 및 시각화 기능을 제공합니다.

6. 외부 설정 중앙화:

- AWS Systems Manager Parameter Store: AWS에서 제공하는 중앙화된 설정 저장소로, 애플리케이션 설정 및 인프라 정보를 안전하게 저장하고 관리할 수 있습니다.

7. 메시지 기반 마이크로서비스:

- Amazon SQS (Simple Queue Service): 완전관리형 메시지 대기열 서비스로, 메시지 기반 마이크로서서비스 아키텍처 구축에 사용할 수 있습니다. 스케일링, 분산 처리 및 서비스 간 메시지 전달을 지원하며, 손쉬운 통합과 확장성을 제공합니다.
- Amazon SNS (Simple Notification Service): 완전관리형 퍼블리시/구독 메시징 서비스로, 서로 다른 서비스들 사이의 메시지 전달을 손쉽게 구현할 수 있습니다. 애플리케이션, 모바일 기기, 이메일 및 SMS와 같은 다양한 엔드포인트에 메시지를 전달할 수 있습니다.

8. 컨테이너 서비스:

- Amazon ECS (Elastic Container Service): 완전관리형 컨테이너 오케스트레이션 서비스로, 도커 컨테이너를 손쉽게 배포, 관리 및 확장할 수 있습니다. 클러스터 관리와 작업 스케줄링 기능을 제공합니다.
- Amazon EKS (Elastic Kubernetes Service): 완전관리형 쿠버네티스 서비스로, 쿠버네티스 클러스터를 손쉽게 설정하고 운영할 수 있습니다. 다양한 AWS 서비스와 통합되어 있어 효율적인 리소스 관리가 가능합니다.

9. 서버리스 아키텍처:

- AWS Lambda: 이벤트 기반의 서버리스 컴퓨팅 서비스로, 코드를 작성하고 이벤트에 따라 자동으로 실행할 수 있습니다. 사용한 컴퓨팅 자원에 대해서만 비용을 지불하며, 자동 스케일링 기능을 제공합니다.

회사에서는 MSA를 구성하기 위해 굉장히 다양한 기술들이 조합된다. 꼭 Spring 관련 기술만 사용되는 것이 아니다. 이런것을 어떻게 잘 조합하는지 결정하는 것은 해당 회사의 아키텍트의 몫이다.

- 우리가 지금까지 배운 Spring Boot프로그래밍은 MSA를 가장 기본적인 기술이다.
- 어떤 아키텍처가 결정되었느냐에 따라서, 관련된 기술을 Spring에서 사용해야 한다.

끝!

강경미(carami@nate.com)