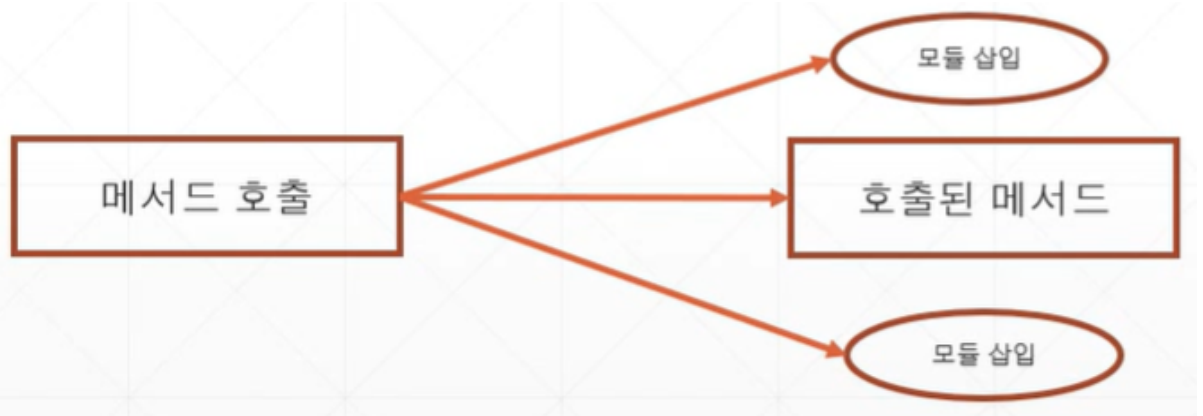
Part 1. 개인발표_권태구

참고. AOP(Aspect Oriented Programming)

하나의 프로그램을 관점(혹은 관심사)라는 논리적 단위로 분리하여 관리

- 로깅, 감사, 선언적 트랜잭션, 보안, 캐싱 등 다양한 곳에서 사용
- 관심사를 통해 Spring Framework가 어떤 메서드가 호출되는지 관심있게 지켜보다가 특정 메서드가 호출 되면 자동으로 메서드 전과 후에 다른 메서드가 호출될 수 있도록 유도
- 원본 메서드(여기선 '호출된 메서드')가 수정되는 것 없이 해당 과정을 수행 가능한 것



1. 어떤 메서드 호출을 함
2. 자동으로 해당 메서드가 실행되기전 과 후에 모듈을 삽입해 해당 메서드의 전과 후에 어떠한 동작을 취하고 메서드가 동작하도록 유도

Spring AOP 용어

Joint Point	모듈이 삽입되어 동작하게 되는 위치(메서드 호출 등) 관심사
Point Cut	다양한 Joint Point 중에 어떤 것을 사용할지 선택
Advice	Joint Point에 삽입되어 동작할 수 있는 코드 그림에서 '모듈 삽입'에 해당

Weaving	Advice를 핵심 로직 코드에 적용하는 것 메서드 호출이 되었을 때 '호출된 메서드'와 '모듈 삽입'을 수행해 하나의 동작으로 만드는 행위
Aspect	Point Cut + Advice 위의 수행 과정

Spring AOP Advice 종류

before	메서드 호출 전에 동작하는 Advice
after-returning	예외없이 호출된 메서드의 동작이 완료되면 동작하는 Advice
after-throwing	호출된 메서드 동작 중 예외가 발생했을 때 동작한느 Advice
after	예외 발생 여부에 관계없이 호출된 메서드의 동작이 완료되면 동작하는 Advice
around	메서드 호출 전과 후에 동작하는 Advice

Execution 명시자

> Pointcut을 지정할 때 사용하는 문법execution([접근제한자] [리턴 타입] [패키지 경로] [클래스명].메소드명(매개 변수))
> ex)
> execution(void *.method1(int))
> execution(* method1(..))
> execution(void kr.co.softcampus.beans.TestBean1.method1(int));

- 접근 제한자 = public 만 지원 - 생략 가능
- * : 한개짜리인 모든 것을 의미
- .. : 개수에 상관없이 모든 것을 의미
- 패키지가 모든 패키지를 대상일 경우 *도 생략 가능

@AspectJ

Advisor 역할을 할 Bean 설정 가능

지원 어노테이션

@Before	관심사 동작 이전에 호출
@After	관심사 동작 이후에 호출
@Around	관심사 동작 이전 이후를 의미
@AfterReturning	예외없이 정상적으로 종료되었을 때 호출
@AfterThrowing	예외가 발생하여 종료되었을 때 호출

<BeanConfigClass.java>

```
@Configuration
@ComponenetScan(basePackages = {"kr.co.softcampus.beans"}, {"kr.co.softcampus.advisor"})
@EnableAspectJAutoProxy
public class BeanConfigClass{

}
```

- @EnableAspectJAutoProxy
  - advisor 클래스에 설정되어 있는 Annotation을 분석하여 AOP세팅을 할 것을 알림

<TestBean1.java>

```
@Component
public class TestBean1{
        public void method1(){
        System.out.println("TestBean1의 메서드1");
    }
}
```

<AdvisorClass.java>

```
@Component
@AspectJ
public class AdvisorClass{

        @Before("execution(* method1())")
    public void beforeMethod(){
        System.out.println("before 메서드");
    }

    @After("execution(* method1())")
    public void afterMethod(){
        System.out.println("after 메서드");
    }

    @Around("execution(* method1())")
    public Object aroundMethod(ProceedingJoinPoint pjp) throws Throwable{
        System.out.println("around 메서드");
        Object result = pjp.proceed(); //실행하고자 하는 메서드의 반환값을 받기위해 설정
        System.out.println("around 메서드");

        return result;
    }

    //오류없이 정상적으로 끝낼 경우 호출될 메서드 등록
    @AfterReturning("execution(* method1())")
    public void afterReturningMethod(){
        System.out.println("afterReturning 메서드");
    }

    @AfterThrowing("execution(* method1())")
    public void afterThrowingMethod(){
        System.out.println("afterThrowing 메서드");
    }

}
```

<MainClass.java>

```
public class MainClass{
        public static void main(String[] args)
        AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(
                        BeanConfigClass.class)

        ctx.close;
}
```

1. Logging

> 배포환경에서의 동작 상태 확인을 위해서는 Test코드 작성, System.out.println() 으로는 불충분프로그램 동작시 발생하는 **모든 일(최소한의 목적)**을 기록하는 행위

로깅할 내용(모든일)

- 프로젝트의 성격에 맞게(팀에 맞게) 정의하면 됨
  - 서비스 동작 상태
    - 시스템이 동작되기위해 필요한 부분
      - 시스템 로딩
      - HTTP 통신
      - 트랜젝션
      - DB요청
      - 의도를 가진 Exception ...
  - 장애(exception. error)
    - 개발자가 의도하지 않은 일
      - I/O Exception
      - NullPointException
      - 의도하지 않은 Exception ...

로깅을 하는 시점

- 앞서 살펴본 최소한의 목적을 우선적으로 생각하고 요구사항에 맞게 작성하면 됨
- 즉, 로깅 시점은 프로젝트 별로 다름

어떻게 기록할 것인가?

- System.out.println("로깅");
- System.err.pritnln("에러로깅");
  - ↑로그를 남기는 가장 쉬운 방법
- **로깅 프레임워크**
  - SLF4J, Logback
  - 출력 형식을 지정할 수 있음
  - 로그 레벨에 따라 남기고 싶은 로그를 별도로 지정가능

- 콘솔 뿐만아니라 파일이나 네트워크등 로그를 별도의 위치에 남길 수 있음

로그를 어떻게 기록할까?

- **로그레벨**을 사용

레벨	설명
Fatal	매우 심각한 에러. 프로그램이 종료되는 경우가 많음
Error	의도하지 않은 에러가 발생한 경우. 프로그램이 종료되진 않음
Warn	에러가 될 수 있는 잠재적 가능성이 있는 경우
Info	명확한 의도가 있는 에러, 요구사항에 따라 시스템 동작을 보여줄 때
Debug	Info 레벨보다 더 자세한 정보가 필요한 경우. Dev 환경
Trace	Debug 레벨보다 더 자세함. Dev 환경에서 버그를 해결하기 위해 사용

예시)
회원가입 시 DB에 동일한 email을 가진 회원이 있을 때, DuplicationException(중복 에러)을 던진다면
이 이벤트의 로그는 어떤 레벨을 적용?

-> Info

- 개발자가 의도한 예외이기 때문

디버깅과의 차이

> 프로그래밍의 절반은 디버깅

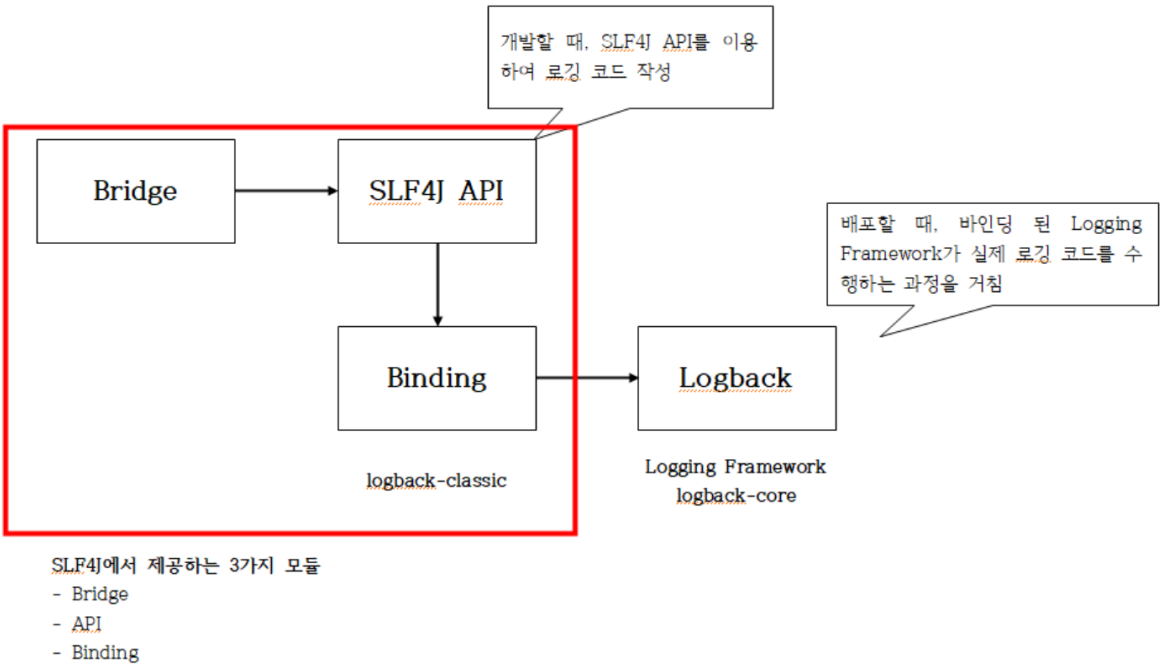
- 디버깅을 할 수 없는 상황에서는 로깅이 최선의 선택(e.g. 실 서버 구동 중)
- 디버깅을 쓸 수 있다면 디버깅을 최대한 활용하는것이 좋음

2. SLF4J(Simple Loggin Facade for Java)

logging 추상화 라이브러리다양한 로깅 프레임 워크에 대해 추상화(인터페이스) 역할

- 단독으로는 사용 불가능
- 최종 사용자가 배포시 원하는 구현체 선택

SLF4J 동작 과정



SLF4J에서 제공하는 3가지 모듈

Bridge

다른 로깅 API로의 Logger 호출을 SLF4J API로 연결SLF4J 이외의 다른 로깅 API로의 Logger 호출을 SLF4J API가 대신 처리할 수 있도록 하는 라이브러리

- 일종의 어댑터 역할
- 이전의 레거시 로깅 프레임워크를 위한 라이브러리
- 여러개 사용 가능
- **Binding 모듈에서 사용될 프레임워크와 달라야 함**

API

로깅에 대한 추상 레이어(인터페이스) 제공

- 하나의 API 모듈에 하나의 Binding 모듈

Binding

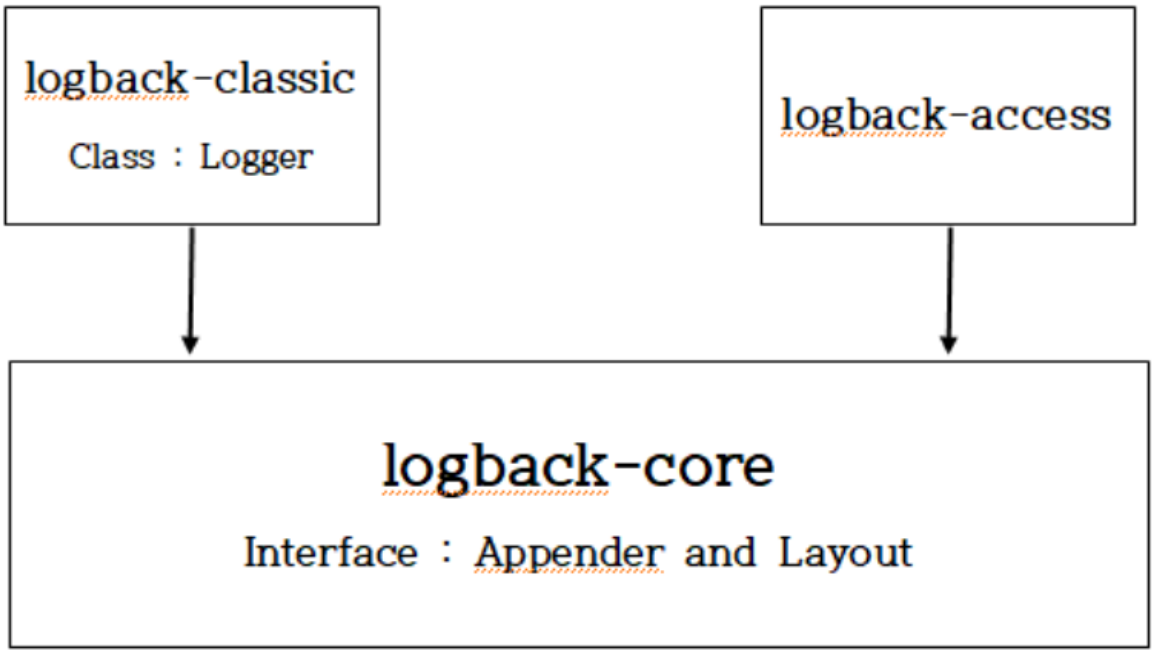
SJF4J API를 로깅 구현체(Logging Framework)와 연결

- 하나의 API 모듈에 하나의 Binding 모듈

---

3. Logback

SLF4J의 구현체인 로깅 프레임워크



logback-core

다른 두 모듈을 위한 기반역할을 하는 모듈

- Appender와 Layout 인터페이스가 이 모듈에 속함

logback-classic

logback-core를 가지며(logback-core에서 확장된 모듈) SLF4J를 구현

- Logger 클래스가 이 모듈에 속함
- 추가적으로 logback-classic에 포함된 라이브러리들은 해당 artifact의 올바른 버전 사용이 필요하고 명시적으로 사용하는 것이 좋음
  - 이걸 사용할 때는 exclude 해주는 것이 좋음

logback-access

Servlet Container와 통합되어 HTTP 액세스에 대한 로깅 기능을 제공

- 웹 애플리케이션 레벨이 아닌 컨테이너 레벨에서 설치되어야 함.

※추가정보

설정요소

Logback을 이용해 로깅을 수행하기 위해서 필요한 주요 설정요소로는 Logger, Appender, Layout(Encoder) 의 3가지

Logger

실제 로깅을 수행하는 구성요소

- 어떻게 기록할까?
- 출력 레벨 조정
  - TRACE < DEBUG < INFO < WARN < ERROR

```
package lab;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LabApplication {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger(LabApplication.class);

        for (int count = 1; count <= 10; count++) {
```

```
            logger.trace("trace 로깅이야!!! {}", count);
            logger.debug("debug 로깅이야!!! {}", count);
            logger.info("info 로깅이야!!! {}", count);
            logger.warn("warn 로깅이야!!! {}", count);
            logger.error("error 로깅이야!!! {}", count);
        }
    }
}
```

Appender

> 로그 메세지가 출력할 대상 결정

- 어디에다 기록할까?
- Logback은 로그이벤트를 쓰는작업을 Appender에게 위임

Appender 구현 클래스

- ch.qos.logback.core.UnsynchronizedAppenderBase
  - synchronized한 동작이 필요하지 않을 때 사용
- ch.qos.logback.core.OutputStreamAppender
  - java.io.OutputStream 로그 이벤트를 append
  - 추상 클래스이기 때문에 이를 직접 쓰지 않고 하위 클래스에 책임을 위임
- ch.qos.logback.core.ConsoleAppender
  - 콘솔에 System.out 또는 System.err를 이용하여 로그이벤트를 append
  - 사용자가 지정한 endoer를 통해 이벤트 포맷의 형식 지정 가능'

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%-4relative [%thread] %-5level %logger{35} - %msg %n</pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

- ch.qos.logback.core.FileAppender

    - 파일에 로그이벤트를 append
    - 매 실행 마다 Unique한 이름의 새로운 로그파일을 만드는 것이 좋기에 아래와 같이 timestamp를 이용하여 타깃 파일을 동적으로 설정 가능

```
<configuration>
  <timestamp key="bySecond" datePattern="yyyyMMdd'T'HHmmss"/>

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <file>log-${bySecond}.txt</file>
    <encoder>
      <pattern>%logger{35} - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

- ch.qos.logback.core.rolling.RollingFileAppender
    - FileAppender를 상속하여 로그파일을 rollover(타켓 파일을 바꾸는 것)
        - log.txt를 타깃파일로 로그 메시지를 append하다가 어느 조건(시간, 용량)에 다다르면, 이전 파일을 저장하고 타겟파일을 바꿈

Rolling Policies

TimeBasedRollingPolicy

- 가장 많이 알려진 RollingPolicy 종류 중 하나
- 시간에 기반하여 rollover 정책을 정의, 주로 일 or 월 단위로 rollover
- rollover 뿐만 아닌 Trigger에 대한 책임도 지기 때문에 RollingPolicy와 TriggeringPolicy 인터페이스를 모두 implements
- TimeBasedRollingPolicy는 필수적으로 fileNamePattern 속성을 가짐

```
<configuration>
  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>logFile.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <!-- 하루 동안의 log를 남김 -->
      <fileNamePattern>logFile.%d{yyyy-MM-dd}.log</fileNamePattern>

      <!-- 30일동안, 총 최대 3GB의 log를 저장함-->
      <maxHistory>30</maxHistory>
      <totalSizeCap>3GB</totalSizeCap>
```

```
    </rollingPolicy>

    <encoder>
      <pattern>%-4relative [%thread] %-5level %logger{35} - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

- <fileNamePattern> : 파일 쓰기가 종료된 log 파일명의 패턴을 지정

fileNamePattern에 명시된 dateTime 패턴의 최소 단위에 따라 rollover 단위가 달라짐
아래의 예시를 통해 rollover가 trigger 되는 시점을 확인 가능

> fileName의 dateTime 패턴

.%d: default %d는 yyyy-MM-dd 매일 자정에 새로운 로그 파일로 rollover.%d{yyyy-MM-dd_HH-mm}:매 분 새로운 로그 파일로 rollover/%d{yyyy/MM}/foo.txt: 매월 새로운 디렉토리를 만들어 하위에 foo.txt 파일로 rollover.%d.gz: 매일 새로운 로그 파일로 rollover하고, 이전 로그파일은 GZIP으로 압축

- <maxHistory> : 최대 파일 생성 갯수를 정의

maxHistory가 30이고, Rolling 정책을 일 단위로 하면 30일 동안만 저장되고, 월 단위로 하면 30개월간 저장
예를들어 30일동안 30개의 파일이 유지됐다면 오래된 파일부터 삭제

- <totalSizeCap>: 저장소의 최대 크기를 지정

maxHistory와 함께 쓰일 경우 1순위로 maxHistory에 대하여 처리된 후
totalSizeCap 이 적용

SizeAndTimeBasedRollingPolicy

- TimeBasedRollingPolicy에서 각각의 로그 파일에 대한 크기를 제한을 하는 부분이 추가됨
- fileNamePattern에서 %i와 %d가 필수적으로 포함되어야 함

```
<configuration>
  <appender name="ROLLING" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>mylog.txt</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
      <!-- 하루 동안의 log를 남김 -->
      <fileNamePattern>mylog-%d{yyyy-MM-dd}.%i.zip</fileNamePattern>
      <!-- 각각의 파일은 100MB로 저장되고, 30일동안, 최대 3GB의 log를 저장함-->
       <maxFileSize>100MB</maxFileSize>
       <maxHistory>30</maxHistory>
       <totalSizeCap>3GB</totalSizeCap>
    </rollingPolicy>
    <encoder>
      <pattern>%msg%n</pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="ROLLING" />
  </root>

</configuration>
```

- <maxFileSize>: 한 파일당 최대 파일 용량을 저장

log 내용의 크기도 IO성능에 영향을 미치기 때문에 되도록 너무 크지 않는 사이즈로 지정하는게 좋음(10MB 권장)용량의 단위는 KB, MB, GB 3가지를 지정 가능

- %i : 롤링 순번을 자동적으로 지정함 (ex) 0, 1, 2 , 3 ...)

Encoder(Layout)

> 사용자가 지정한 형식으로 표현될 로그메시지를 변환하는 역할로그 이벤트를 바이트 배열로 변환하고, 해당 바이트 배열을 OutputStream에 쓰는 작업을 담당

- 어떻게 출력할까?
- Encoder는 로그이벤트를 바이트 배열로 변환하고 해당 바이트 배열을 OutputStream에 쓰는 작업 담당
  - 즉, Appender에 포함되어 사용자가 지정한 형식으로 표현될 로그 메시지를 변환하는 역할을 담당하는 요소

- FileAppender와 하위 클래스는 Encoder를 필요로 하고 현재 layout을 사용하지 않음
  - encoder를 사용

PatternLayoutEncoder

```
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
  <file>testFile.log</file>
  ...
  <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <pattern>%msg%n</pattern>
  </encoder>
</appender>
```

- 헤더에 패턴 출력방법

```
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
  <file>foo.log</file>
  <encoder>
    <pattern>%d %-5level [%thread] %logger{0}: %msg%n</pattern>
    <outputPatternAsHeader>true</outputPatternAsHeader> <!-- 헤더에 패턴 출력 -->
  </encoder>
</appender>
```

Log Pattern

%logger : 패키지 포함 클래스 정보

%logger{0} : 패키지를 제외한 클래스 이름만 출력

%logger{length} : Logger name을 축약할 수 있음. {length}는 최대 자리 수, ex)logger{35}

%-5level : 로그 레벨, -5는 출력의 고정폭 값(5글자), 로깅레벨이i nfo일 경우 빈칸 하나 추가

${PID:-} : 프로세스 아이디

%d : 로그 기록시간 출력

%p : 로깅 레벨 출력

%F : 로깅이 발생한 프로그램 파일명 출력

%M : 로깅일 발생한 메소드의 명 출력

%line : 로깅이 발생한 호출지의 라인

%L : 로깅이 발생한 호출지의 라인

%thread : 현재 Thread 명

%t : 로깅이 발생한 Thread 명

%c : 로깅이 발생한 카테고리

%C : 로깅이 발생한 클래스 명 (%C{2}는 somePackage.SomeClass 가 출력됨)

%m : 로그 메시지

%msg : - 로그 메시지 (=%message)

%n : 줄바꿈(new line)

%% : %를 출력

%r : 애플리케이션 시작 이후부터 로깅이 발생한 시점까지의 시간(ms)

%d{yyyy-MM-dd-HH:mm:ss:sss} : %d는 date를 의미하며 중괄호에 들어간 문자열은 dateformat을 의미. 따라서 [2021-07-12 12:42:78]과 같은 날짜가 로그에 출력됨.

%-4relative : %relative는 초 아래 단위 시간(밀리초)을 나타냄. -4를하면 4칸의 출력 폼을 고정으로 가지고 출력. 따라서 숫자에 따라 [2021-07-12 12:42:78:232] 혹은 [2021-07-12 12:42:78:2332]와 같이 표현됨

※Extra
<springProfile> : logback 설정 파일에서 복수 개의 프로파일 설정 가능

```
<springProfile name="!prod">
```

```
        <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
            <encoder>
                <pattern>${LOG_PATTERN}</pattern>
            </encoder>
        </appender>

        <root level="INFO">
            <appender-ref ref="CONSOLE"/>
        </root>

</springProfile>

<springProfile name="local">
        <property resource="logback-local.properties"/>
</springProfile>
```

<Filter> : 해당 패키지에서 무조건 로그를 찍는 것 말고도 필터링이 필요한 경우 사용하는 기능

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%-4relative [%thread] %-5level %logger{35} - %msg %n</pattern>
    </encoder>
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
        <level>ERROR</level>
        <onMatch>ACCEPT</onMatch>
        <onMismatch>DENY</onMismatch>
    </filter>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

---

4. 프로젝트에 적용해보기

스프링 부트 에서는 logback-spring.xml을 사용해서 Spring이 logback을 구동할 수 있도록 지원해 주며 이를 이용하여 profile, 즉 배포 환경에 따른 (spring.profiles.active을 활용하여) application.xml에 설정된 properties를 읽어올 수 있음.

Log 설정 참조 순서

1. classpath(resources디렉토리 밑)에 **logback-spring.xml파일**이 있으면 설정파일을 읽음.
2. logback-spring.xml파일이 없다면 **.yml(.properties)파일**의 설정을 읽음.
3. logback-spring.xml파일과 .yml(.properties)파일이 동시에 있으면 .yml(.properties) 설정 파일을 적용 후 **xml파일이 적용됨**.

콘솔에 출력하기

```
private Team findTeam(String teamName) {
        Optional<Team> team = teamRepository.findByName(teamName);
        if (team.isEmpty()) {
            String detailMessage = String.format("존재하지 않는 팀입니다. 입력값: %s",
teamName);
            logger.info(detailMessage);
            throw new IllegalArgumentException(detailMessage);
        }
        return team.get();
    }
```

<logback-spring.xml>

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>[%d{yyyy-MM-dd HH:mm:ss}:%-4relative] %green([%thread])
                        %highlight(%-5level) %boldWhite([%C.%M:%yellow(%L)]) -
%msg%n</pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
    </root>
</configuration>
```

<콘솔창 출력 결과>

```
[2021-07-12 21:11:18:9095] [http-nio-8080-exec-2] INFO
[com.livenow.slf4jlogbacklab.service.TeamService.findTeam:54] - 존재하지 않는 팀입니다. 입
력값: 좋은 팀
[2021-07-12 21:11:18:9099] [http-nio-8080-exec-2] ERROR
[com.livenow.slf4jlogbacklab.Slf4jRestControllerAdvice.illegalArgumentException:25] -
IllegalArgumentException: 존재하지 않는 팀입니다. 입력값: 좋은 팀
```

파일에 출력하기

- 테스트 환경이 아닌, 실제 프로덕션 환경에서는 로그를 파일로 주로 저장
- <springProfile>을 통해 수정

<logback-spring.xml>

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
    <timestamp key="BY_DATE" datePattern="yyyy-MM-dd"/>
    <property name="LOG_PATTERN"
              value="[%d{yyyy-MM-dd HH:mm:ss}:%-4relative] %green([%thread])
                        %highlight(%-5level) %boldWhite([%C.%M:%yellow(%L)]) - %msg%n"/>

    <springProfile name="!prod">
        <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
            <encoder>
                <pattern>${LOG_PATTERN}</pattern>
            </encoder>
        </appender>

        <root level="INFO">
            <appender-ref ref="CONSOLE"/>
        </root>
    </springProfile>

    <springProfile name="prod">
        <appender name="FILE-INFO"
class="ch.qos.logback.core.rolling.RollingFileAppender">
            <file>./log/info/info-${BY_DATE}.log</file>
            <filter class = "ch.qos.logback.classic.filter.LevelFilter">
                <level>INFO</level>
                <onMatch>ACCEPT</onMatch>
                <onMismatch>DENY</onMismatch>
            </filter>
            <encoder>
                <pattern>${LOG_PATTERN}</pattern>
            </encoder>
            <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
                <fileNamePattern> ./backup/info/info-%d{yyyy-MM-
dd}.%i.log</fileNamePattern>
                <maxFileSize>100MB</maxFileSize>
                <maxHistory>30</maxHistory>
                <totalSizeCap>3GB</totalSizeCap>
```

```
            </rollingPolicy>
        </appender>

        <appender name="FILE-WARN"
class="ch.qos.logback.core.rolling.RollingFileAppender">
            <file>./log/warn/warn-${BY_DATE}.log</file>
            <filter class = "ch.qos.logback.classic.filter.LevelFilter">
                <level>WARN</level>
                <onMatch>ACCEPT</onMatch>
                <onMismatch>DENY</onMismatch>
            </filter>
            <encoder>
                <pattern>${LOG_PATTERN}</pattern>
            </encoder>
            <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
                <fileNamePattern> ./backup/warn/warn-%d{yyyy-MM-
dd}.%i.log</fileNamePattern>
                <maxFileSize>100MB</maxFileSize>
                <maxHistory>30</maxHistory>
                <totalSizeCap>3GB</totalSizeCap>
            </rollingPolicy>
        </appender>

        <appender name="FILE-ERROR"
class="ch.qos.logback.core.rolling.RollingFileAppender">
            <file>./log/error/error-${BY_DATE}.log</file>
            <filter class = "ch.qos.logback.classic.filter.LevelFilter">
                <level>ERROR</level>
                <onMatch>ACCEPT</onMatch>
                <onMismatch>DENY</onMismatch>
            </filter>
            <encoder>
                <pattern>${LOG_PATTERN}</pattern>
            </encoder>
            <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
                <fileNamePattern> ./backup/error/error-%d{yyyy-MM-
dd}.%i.log</fileNamePattern>
                <maxFileSize>100MB</maxFileSize>
                <maxHistory>30</maxHistory>
                <totalSizeCap>3GB</totalSizeCap>
            </rollingPolicy>
        </appender>

        <root level="INFO">
            <appender-ref ref="FILE-INFO"/>
            <appender-ref ref="FILE-WARN"/>
            <appender-ref ref="FILE-ERROR"/>
        </root>
    </springProfile>
```

```
</configuration>
```