



[한국ICT인재개발원] 스프링 프레임워크

## 10. 스프링의 AOP와 트랜잭션

前) 광고데이터 분석 1년

前) IT강의 경력 2년 6개월

前) 머신러닝을 활용한 데이터 분석 프로젝트반 운영 1년

前) 리그오브 레전드 데이터 분석 등...

現) 국비반 강의 진행중

AOP는 핵심 로직과 보조 로직을 분리하자는 개념에서 출발합니다.

여태까지는 모든 메서드의 실행소요시간이 몇 초인지 구하고 싶으면

메서드마다 기능을 추가해야 했지만, 이제는 실행소요시간 구하는 기능을

하나만 만들어 두면, 모든 메서드에 기능을 전역적으로 추가하거나

혹은 기능을 추가할 메서드 범위를 지정할 수도 있습니다.

이런 개념을 AOP라고 부르며, 주로 로그, 보안등 주요 기능은 아니지만

주요 기능을 보조하는 기능을 구현할 때 많이 사용합니다.

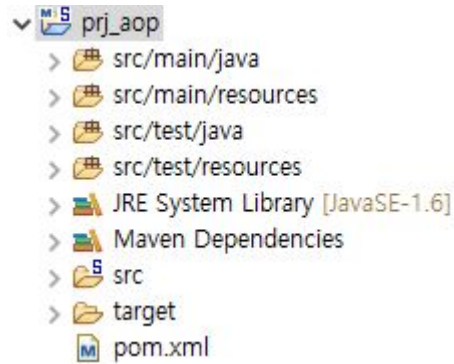
용어	의미
Aspect	관심사. 공통적으로 사용할 기능을 통틀어 부르는 명칭.
Advice	Aspect를 적용받은 실제 객체.
Join point	Aspect를 적용할 수 있는 후보 객체.
Pointcut	Join point 중 실제로 적용할 예정인 객체.
Target	Join point가 가지고 있는 메서드들.
Proxy	Advice가 적용된 객체를 생성한 것.
Introduction	target에 없는 새로운 메소드, 변수등을 추가하는 경우
Weaving	Advice를 Target에 적용하는 행위.

먼저 AOP에 대한 용어 정리입니다.

위 용어를 알아야 공부를 진행할 수 있으니 유심히 봐 주세요.

타입	기능
Before	주요 메서드 실행 전에 먼저 실행하는 aop
AfterReturning	메서드 정상 호출 후에 return구문이 끝나고 실행하는 aop
AfterThrowing	에러처리를 한 뒤에 실행되는 aop
After	예외여부 상관없이 메서드호출이 끝난 뒤 호출할때
Around	호출 이전부터 호출 이후까지 꼭실행할때

**Advice**를 적용할 시점이 다르기 때문에 시점에 대해서도 이해해야합니다.



```
<!-- Test -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

```
10 <properties>
11   <java-version>1.8</java-version>
12   <org.springframework-version>5.0.7.RELEASE</
13   <org.aspectj-version>1.9.0</org.aspectj-vers
14   <org.slf4j-version>1.7.25</org.slf4j-version
15 </properties>
16
17 <!-- Servlet -->
18 <dependency>
19   <groupId>javax.servlet</groupId>
20   <artifactId>javax.servlet-api</art
21   <version>3.1.0</version>
22   <scope>provided</scope>
23 </dependency>
24
25 <plugin>
26   <groupId>org.apache.maven.plugins</
27   <artifactId>maven-compiler-plugin</
28   <version>2.5.1</version>
29   <configuration>
30     <source>1.8</source>
31     <target>1.8</target>
```

샘플 프로젝트를 생성하고 pom.xml 최 상단의 값을 위와 같이 변경합니다.  
추가로 테스트를 위해 junit도 4.12로 변경합니다.  
servlet을 3.1.0으로 변경하고 maven-compiler-plugin도 1.8로 바꿉니다.



## 1. Spring TestContext Framework

[org.springframework.org/spring-test](https://org.springframework.org/spring-test)

Spring TestContext Framework

Last Release on Mar 16, 2021

5.07버전



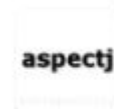
## 1. Project Lombok

[org.projectlombok.org/lombok](https://org.projectlombok.org/lombok)

Spice up your java: Automatic F

Last Release on Jan 28, 2021

1.18.20버전



## 1. AspectJ Weaver

[org.aspectj.org/aspectjweaver](https://org.aspectj.org/aspectjweaver)

The AspectJ weaver introduc

Last Release on Jul 22, 2020

1.9.0버전



## 1. AspectJ Runtime

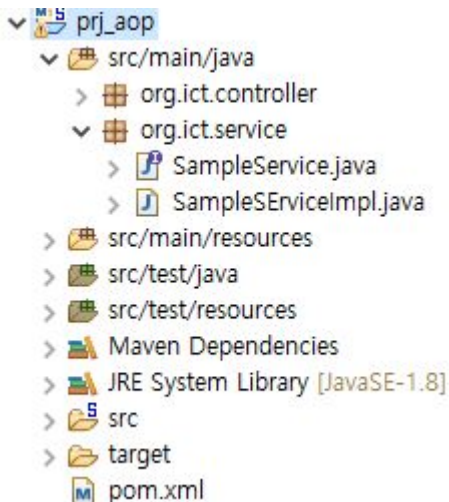
[org.aspectj.org/aspectjrt](https://org.aspectj.org/aspectjrt)

The runtime needed to execu

Last Release on Jul 22, 2020

1.9.0버전

테스트 프로젝트에서는 위 4개를 먼저 설치합니다.  
이후 프로젝트 업데이트를 실행해주세요.



```
package org.ict.service;

public interface SampleService {

    public Integer doAdd(String str1, String str2) throws Exception;
}

import org.springframework.stereotype.Service;

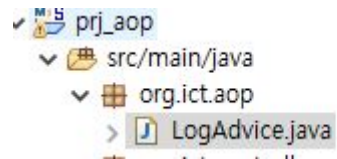
@Service
public class SampleServiceImpl implements SampleService {

    @Override
    public Integer doAdd(String str1, String str2) throws Exception {

        return Integer.parseInt(str1) + Integer.parseInt(str2);
    }
}
```

org.ict.service 에는 위와 같이 SampleService 인터페이스와 SampleServiceImpl 구현클래스를 만들어줍니다.





```
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

import lombok.extern.log4j.Log4j;

@Aspect
@Log4j
@Component
public class LogAdvice {

}
```

Service의 코드는 현재 로그를 기록하는 기능이 없습니다.  
보조 기능과 핵심 기능을 따로 개발하는 AOP 개념을 적용해  
로깅기능을 추가할 것이기 때문입니다.  
로그를 자동기록해주는 Advice를 지금부터 작성해보겠습니다.

먼저 org.ict.aop 패키지 내에 LogAdvice 클래스를 생성합니다.

```
<groupId>log4j</groupId>
<artifactId>log4j</artifactId>
<version>1.2.15</version>
<exclusions>
  <exclusion>
    <groupId>javax.mail</groupId>
    <artifactId>mail</artifactId>
  </exclusion>
  <exclusion>
    <groupId>javax.jms</groupId>
    <artifactId>jms</artifactId>
  </exclusion>
  <exclusion>
    <groupId>com.sun.jdmk</groupId>
    <artifactId>jmxtools</artifactId>
  </exclusion>
  <exclusion>
    <groupId>com.sun.jmx</groupId>
    <artifactId>jmxri</artifactId>
  </exclusion>
</exclusions>
<!-- <scope>runtime</scope> -->
```

```
8 @Aspect
9 @Log4j
10 @Component
11 public class LogAdvice {
12
13 }
14
```

먼저 @Log4j가 에러가 나는 경우가 있습니다.

이 경우, pom.xml의 log4j내부 <scope>를 주석처리하면 풀립니다.

LogAdvice.java

```
11 import org.aspectj.lang.annotation.  
12 import org.aspectj.lang.annotation.  
13 import org.springframework.  
14 import lombok.extern.log4j.  
15  
16 @Aspect  
17 @Log4j  
18 @Component  
19 public class LogAdvice {  
20
```

prj\_aop/pom.xml

```
<!-- https://mvnrepository.com/artifact/org  
<dependency>  
    <groupId>org.aspectj</groupId>  
    <artifactId>aspectjrt</artifactId>  
    <version>1.9.0</version>  
    <!-- <scope>runtime</scope> -->  
</dependency>
```

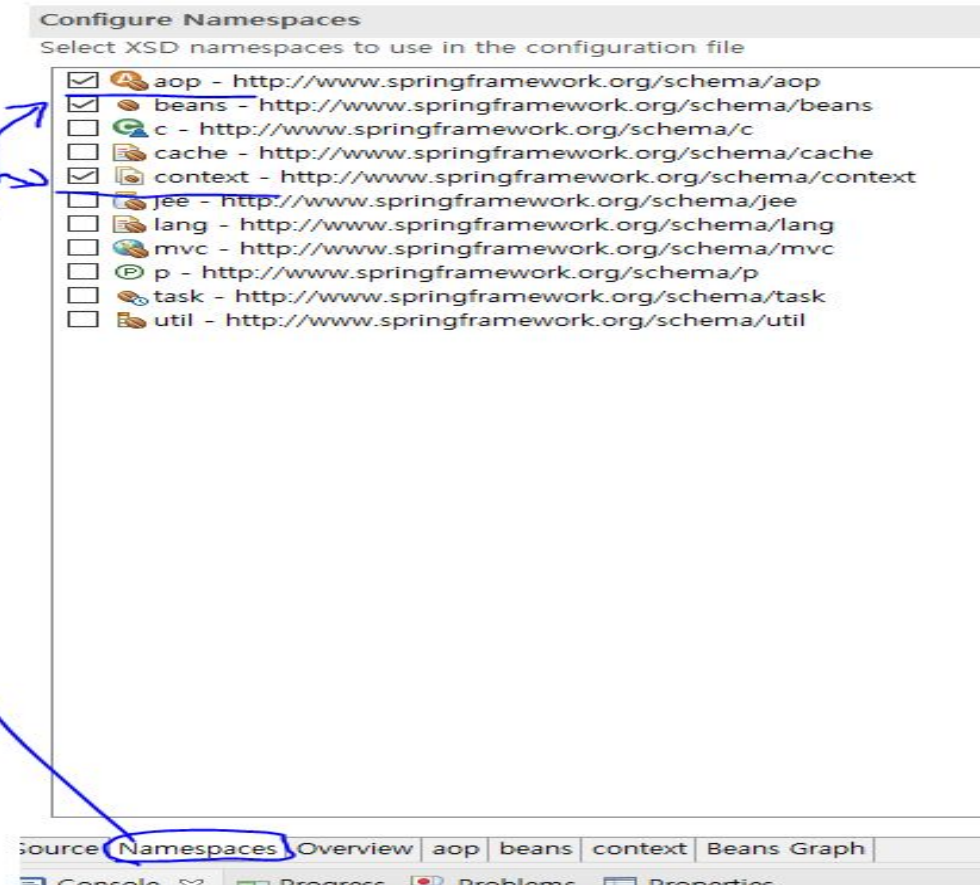
LogAdvice 파일에서 @Aspect를 못 잡는 경우가 있는데 이런 경우  
pom.xml의 aspectjrt의 scope도 주석처리합니다.

```
public class LogAdvice {  
    @Before("execution(* org.ict.service.SampleService*.*(..))")  
    public void logBefore() {  
        Log.info("=====");  
    }  
}
```

이제 LogAdvice 내부에 위와 같이 작성합니다.

@Before는 핵심로직 실행 전에 logBefore메서드가 실행됨을 나타내며, 내부 문자열은 규칙에 맞는 메서드 전부를 실행하기 전에 logBefore메서드가 실행됨을 나타냅니다.

위의 경우 맨 앞의 \*은 접근제한자 상관 없음, 가운데는 어느 패키지 어느 클래스까지, 뒤의 \*은 클래스명, 메서드명, 파라미터 종류입니다.



다음으로, 컴포넌트 스캔과 AOP설정을 모두 다 해보겠습니다.

root-context.xml의 하단 namespaces 탭에서 aop, context를 체크합니다.



```
<!-- Root Context: defines shared resources visible to all other web components -->  
<context:annotation-config></context:annotation-config>
```

```
<context:component-scan base-package="org.ict.service"></context:component-scan>  
<context:component-scan base-package="org.ict.aop"></context:component-scan>
```

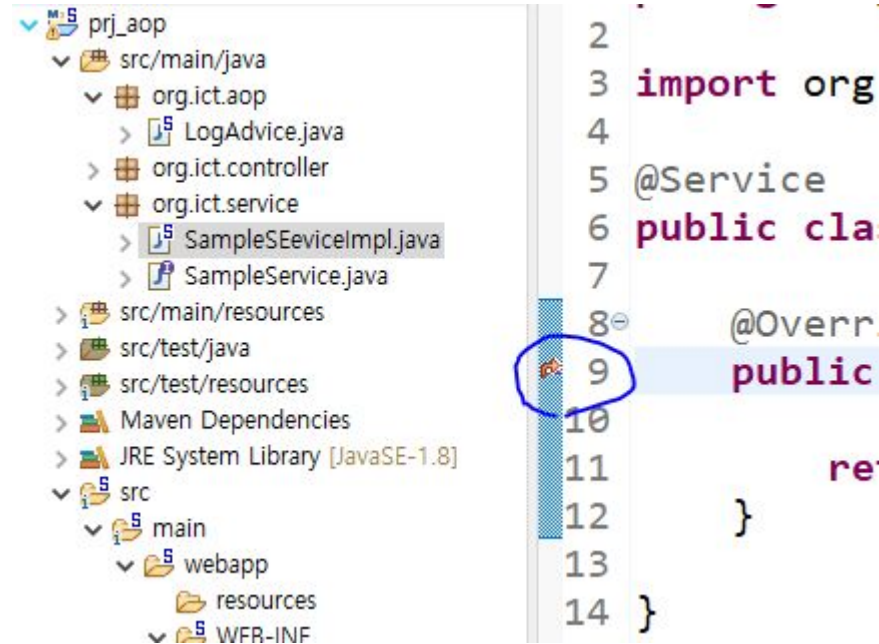
```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>  
</beans>
```

```
9 @Aspect  
0 @Log4j  
1 @Component  
2 public class LogAdvice {  
3  
4     // ...  
5 }
```

내부에 위와 같이 작성합니다.

특히 하단의 **aspectj-autoproxy**까지 작성해주셔야 합니다.

하단 태그의 역할은 **@Aspect**를 받은 클래스는 AOP의 타겟임을 알리는 것입니다.



이제 컴포넌트 스캔이 되면서 위와 같이 메서드 왼쪽에 화살표 마크가 추가됩니다.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("file:src/main/webapp/WEB-INF/spring/root-context.xml")
@Log4j
public class SampleServiceTests {

    @Autowired
    private SampleService service;

    @Test
    public void testClass() {
        Log.info(service);
        Log.info(service.getClass().getName());
    }
}
```

다음으로 테스트 코드를 작성해서 실제로 AOP가 적용되었는지 확인합니다.

src/test/java 하위에 org.ict.service 패키지를 추가하고  
SampleServiceTests 클래스를 추가합니다.  
그리고 위와 같이 작성합니다.



```
...  
- org.ict.service.SampleServiceImpl@2bc12da  
- com.sun.proxy.$Proxy23  
t.GenericApplicationContext - Closing org.spring
```

---

위와 같이 Proxy 객체가 확인되어야 AOP 적용이 완벽하게 된 것입니다.

```
@Test
public void testAdd() throws Exception {
    Log.info(service.doAdd("123", "456"));
}
```

```
IG: All illegal access operations will be denied in
org.ict.aop.LogAdvice - =====
org.ict.service.SampleServiceTests - 579
org.springframework.context.support.GenericApplica
```

이제 위와 같이 Advice를 적용받은 doAdd를 작성해 실행해봅시다.

그러면 우측처럼 123 + 456을 계산하기 전, 먼저 ===== 라고 로깅부터 하는것을 볼 수 있습니다.

```
@Before("execution(* org.ict.service.SampleService*.doAdd(String, String)) && args(str1, str2)")
public void logBeforeWithParam(String str1, String str2) {
    |
    log.info("str1: " + str1);
    log.info("str2: " + str2);
}
```

이번에는 `execute` 구문에 `args`를 이용해 파라미터를 추적해보겠습니다.  
분명히 로깅 자체는 마음에 들지만, 파라미터별로 무슨 값을 받았는지 알 수  
있다면 더 좋을 때문에 `LogAdvice`에 추가로 위와같이 메서드를 추가합니다.

접근제한자 모두 허용  
`orc.ict.service.SampleService`로 시작하는 모든 클래스  
`doAdd(String, String)`인 메서드 타겟이며  
파라미터는 각각 `str1`, `str2`로 명명

```
INFO : org.ict.aop.LogAdvice - =====  
INFO : org.ict.aop.LogAdvice - str1: 123  
INFO : org.ict.aop.LogAdvice - str2: 456  
INFO : org.ict.service.SampleServiceTests - 579  
INFO : org.springframework.context.support.Generic
```

Advice는 한 메서드에 2개 이상도 동시에 적용받을 수 있기 때문에 테스트 코드를 실행하면 이번에는 2개가 모두 실행됩니다.

둘 다 @Before이기 때문에 원본 메서드 이전에 작성된 순서대로 실행 후 핵심로직이 실행됩니다.

```
@AfterThrowing(pointcut = "execution(* org.ict.service.SampleService*.*(..))", throwing="exception")
public void logException(Exception exception) {

    log.info("Exception....!!!!");
    log.info("exception: " + exception);
}
```

다음은 예외가 발생했을때만 작동하는 @AfterThrowing을 추가해보겠습니다.

LogAdvice 내부에 위와같이 작성합니다.

```
@Test
public void testAdd() throws Exception {

    Log.info(service.doAdd("123", "ABC"));
}
```

```
INFO : org.ict.aop.LogAdvice - =====
INFO : org.ict.aop.LogAdvice - str1: 123
INFO : org.ict.aop.LogAdvice - str2: ABC
INFO : org.ict.aop.LogAdvice - Exception....!!!!
INFO : org.ict.aop.LogAdvice - exception: java.lang.NumberFormatException: For input string: "ABC"
INFO : org.springframework.context.support.GenericApplicationContext - Closing org.springframework.
```

SampleServiceTests에서는 일부러 에러를 발생시켜봅니다.

결과는 위와 같이 에러가 발생하면서 에러가 뭘였는지 로깅을 해 줍니다.

```
@Around("execution(* org.ict.service.SampleService*.*(..))")
public Object logTime(ProceedingJoinPoint pjp) {

    long start = System.currentTimeMillis();

    Log.info("Target: " + pjp.getTarget());
    Log.info("Param: " + Arrays.toString(pjp.getArgs()));

    Object result = null;

    try {
        result = pjp.proceed();
    } catch (Throwable e) {
        e.printStackTrace();
    }

    long end = System.currentTimeMillis();

    Log.info("TIME: " + (end - start));

    return result;
}
```

마지막으로 @Around는 메서드 실행 내내 실행되는 특이한 경우입니다. 특히, @Around와 함께 사용하는 ProceedingJoinPoint객체를 활용하면 파라미터와 예외 등등을 한 번에 처리할 수 있습니다. 위 코드는 메서드의 실행소요시간을 구하는 Advice입니다.



```
@Test
public void testAdd() throws Exception {
    Log.info(service.doAdd("123", "456"));
}
```

```
org.ict.aop.LogAdvice - Target: org.ict.service.SampleServiceImpl@15723761
org.ict.aop.LogAdvice - Param: [123, 456]
org.ict.aop.LogAdvice - =====
org.ict.aop.LogAdvice - str1: 123
org.ict.aop.LogAdvice - str2: 456
org.ict.aop.LogAdvice - TIME: 2
org.ict.service.SampleServiceTests - 579
org.springframework.context.support.GenericApplicationContext - Closing org.
```

테스트코드 실행시 위와 같이 시간을 구해서 출력해줍니다.



원자성 (Atomicity)	하나의 트랜잭션은 하나의 단위로 처리되어야 함. 예) 게시판의 글과 댓글을 함께 삭제하는 로직이 있다면 글과 댓글은 무조건 함께 삭제되거나 함께 삭제되지 말아야 함. 만약 글만 삭제되고 댓글은 삭제가 되지 않거나 댓글만 삭제되고 글은 삭제되지 않는다면, 비정상적 상황이므로 원래대로 돌려놔야함.
일관성 (Consistency)	트랜잭션으로 처리된 데이터는 트랜잭션이 아닌 일반로직으로 처리된 데이터와 차이가 없어야 함.
격리 (Isolation)	트랜잭션으로 처리되는 도중 외부 간섭을 받아서는 안 됨.
영속성 (Durability)	트랜잭션이 일단 처리되면, 처리 결과는 돌이킬 수 없어야함.

이제 AOP를 응용해 트랜잭션을 이해하고 적용해보겠습니다.  
먼저 트랜잭션은 2개 이상의 쿼리문이 실행된다면 전부 실행되고  
실패한다면 전부 실패하게 만드는 것을 의미합니다.  
위 4가지 원칙에 따라 기능을 묶어주는것을 트랜잭션이라고 합니다.



1. **Spring JDBC**  
[org.springframework](#) » [spring-jdbc](#)  
Spring JDBC  
Last Release on Mar 16, 2021

5.0.7버전



1. **Spring Transaction**  
[org.springframework](#) » [spring-tx](#)  
Spring Transaction  
Last Release on Mar 16, 2021

5.0.7버전



1. **HikariCP**  
[com.zaxxer](#) » [HikariCP](#)  
Ultimate JDBC Connection Pool  
Last Release on Mar 3, 2021

4.0.3버전



1. **Log4Jdbc Log4j2 JDBC 4**  
[org.bgee.log4jdbc-log4j2](#) » [log4jdbc-log4j2-jdbc4](#)  
Log4Jdbc Log4j2 JDBC 4  
Last Release on Dec 12, 2013

1.16버전



1. **MyBatis**  
[org.mybatis](#) » [mybatis](#)  
The MyBatis SQL mapper fram  
object relational mapping tool:  
Last Release on Oct 6, 2020

3.5.6버전



2. **MyBatis Spring**  
[org.mybatis](#) » [mybatis-spring](#)  
An easy-to-use Spring bridge f  
Last Release on Nov 14, 2020

2.0.6버전

Home » [com.oracle.database.jdbc](#) » [ojdbc8](#) » 18.15.0.0

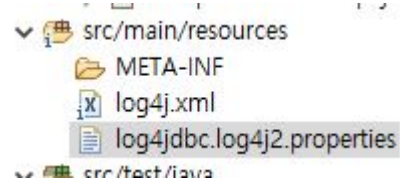
**Ojdbc8 » 18.15.0.0**  
Oracle JDBC Driver compatible with JDK8, JDK11, JI

HomePage	<a href="https://www.oracle.com/database/techn">https://www.oracle.com/database/techn</a>
Date	(Aug 24, 2021)
Files	<a href="#">pom (1 KB)</a>   <a href="#">jar (4.0 MB)</a>   <a href="#">View All</a>

18.15.0.0 버전

트랜잭션 실습을 위해 prj\_aop 프로젝트의 pom.xml에 라이브러리를 추가합니다.

mvnrepository에서 추가하면 됩니다.



```
1 log4jdbc.spylogdelegator.name=net.sf.log4jdbc.log.slf4j.Slf4jSpyLogDelegator
```

src/main/resources 내부에 log4jdbc.log4j2.properties 파일 역시 넣어줍니다.

## Namespaces 3 errors

### Configure Namespaces

Select XSD namespaces to use in the configuration file

- ☒ aop - <http://www.springframework.org/schema/aop>
- ☒ beans - <http://www.springframework.org/schema/beans>
- ☐ c - <http://www.springframework.org/schema/c>
- ☐ cache - <http://www.springframework.org/schema/cache>
- ☒ context - <http://www.springframework.org/schema/context>
- ☐ jdbc - <http://www.springframework.org/schema/jdbc>
- ☐ jee - <http://www.springframework.org/schema/jee>
- ☐ lang - <http://www.springframework.org/schema/lang>
- ☐ mvc - <http://www.springframework.org/schema/mvc>
- ☒ mybatis-spring - <http://mybatis.org/schema/mybatis-spring>
- ☐ p - <http://www.springframework.org/schema/p>
- ☐ task - <http://www.springframework.org/schema/task>
- ☒ tx - <http://www.springframework.org/schema/tx>
- ☐ util - <http://www.springframework.org/schema/util>

```
<bean id="hikariConfig" class="com.zaxxer.hikari.HikariConfig">
  <property name="driverClassName"
    value="net.sf.log4jdbc.sql.jdbcapi.DriverSpy" />
  <property name="jdbcUrl"
    value="jdbc:log4jdbc:oracle:thin:@localhost:1521/XEPDB1" />
  <property name="username" value="mytest" />
  <property name="password" value="mytest" />
</bean>

<bean id="dataSource"
  class="com.zaxxer.hikari.HikariDataSource" destroy-method="close"
  <constructor-arg ref="hikariConfig" />
</bean>

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactory"
  <property name="dataSource" ref="dataSource" />
</bean>
```

먼저 root-context.xml을 열고 Namespaces 탭에 tx를 체크합니다.  
다음 기본 DB연결설정을 해 줍니다.

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"></property>
</bean>

<tx:annotation-driven/>

<mybatis-spring:scan base-package="org.ict.mapper"/>
```

계속해서 root-context.xml 내부에 트랜잭션 실행을 위한 transactionManager 객체 생성과 annotation-driven 태그 작성, 쿼리문을 입력할 마이바티스 매퍼를 스캔합니다.

```
CREATE TABLE tbl_test1( col1 varchar2(50));  
CREATE TABLE tbl_test2( col2 varchar2(5));
```

이제 같은 명령을 받으면 둘 중 하나만 돌아갈 가능성이 존재하는 테스트 테이블을 만들어보겠습니다.

sqlDeveloper에서 위와 같이 test 테이블2개를 생성합니다.

위 테이블들은 각각 50글자와 5글자를 저장할 수 있는 컬럼을 가집니다.



```
package org.ict.mapper;

import org.apache.ibatis.annotations.Insert;

public interface Sample1Mapper {
    @Insert("insert into tbl_test1 (col1) values (#{data})")
    public int insertCol1(String data);
}

import org.apache.ibatis.annotations.Insert;

public interface Sample2Mapper {
    @Insert("insert into tbl_test2 (col2) values (#{data})")
    public int insertCol2(String data);
}
```

Sample1Mapper 인터페이스와 Sample2Mapper 인터페이스는 mapper xml은 따로 작성하지 않고, 어노테이션으로 구문을 작성합니다.

```
package org.ict.service;

public interface SampleTxService {

    public void addData(String value);
}
```

```
@Service
@Log4j
public class SampleTxServiceImpl implements SampleTxService {

    @Autowired
    private Sample1Mapper mapper1;

    @Autowired
    private Sample2Mapper mapper2;

    @Override
    public void addData(String value) {
        Log.info("mapper1.....");
        mapper1.insertCol1(value);
        Log.info("mapper2.....");
        mapper2.insertCol2(value);
    }
}
```

다음, 방금 생성한 두 테이블에 같은 길이의 문자열을 집어넣는 서비스를 생성하기 위해 **SampleTxService** 인터페이스, **SampleTxServiceImpl** 구현클래스를 만들어줍니다.



```
@RunWith(SpringJUnit4ClassRunner.class)
@Configuration("file:src/main/webapp/WEB-INF
@Log4j
public class SampleTxServiceTests {

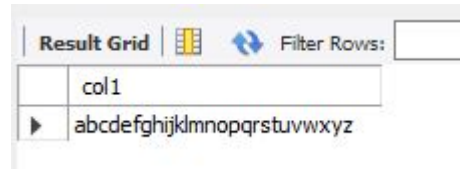
    @Autowired
    private SampleTxService service;

    @Test
    public void testInsert() {
        String str = "abcdefghijklmnopqrstuvwxyz";

        service.addData(str);
    }
}
```

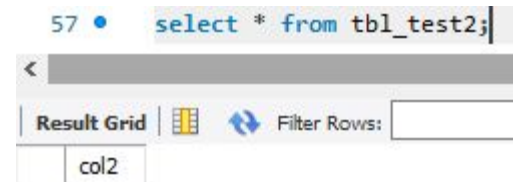
이제 src/test/java 내부의 org.ict.service 패키지 내부에다 SampleTxServiceTests 클래스를 만들어서 테스트를 해 봅니다. 26글자를 테이블 두 개에 넣어주는 로직입니다.

```
teTestRunner.main(RemoteTestRunner.java:210)  
) FAILED! insert into tbl_test2 (col2) values ('abcdefghijklmnopqrstuvwxyz')  
  
a truncation: Data too long for column 'col2' at row 1  
Mapping.translateException(SQLExceptionsMapping.java:104)  
executeInternal(ClientPreparedStatement.java:953)
```



A screenshot of a database client's 'Result Grid' window. It shows a table with two columns: 'col1' and 'col2'. The 'col1' column contains the value 'abcdefghijklmnopqrstuvwxyz'. The 'col2' column is empty.

col1	col2
abcdefghijklmnopqrstuvwxyz	



A screenshot of a database client's 'Result Grid' window. It shows a table with two columns: 'col1' and 'col2'. The 'col1' column is empty. The 'col2' column contains the value 'abcdefghijklmnopqrstuvwxyz'. Above the grid, a SQL query 'select \* from tbl\_test2;' is visible.

col1	col2
	abcdefghijklmnopqrstuvwxyz

실행시 varchar(5)짜리 테이블에 26글자를 넣으려해서 에러가 납니다.  
단, 순서는 varchar(50)인 tbl\_test1에 먼저 넣었고  
다음 varchar(5)인 tbl\_test2에 넣었기 때문에 tbl\_test1은 데이터가 있습니다.



```
21 @Transactional
22 @Override
23 public void addData(String value) {
24     Log.info("mapper1.....");
25     mapper1.insertCol1(value);
26     Log.info("mapper2.....");
27     mapper2.insertCol2(value);
28 }
```

```
drop table tbl_test1;
```

이제, 두 개의 mapper1, mapper2를 동시에 호출하는 addData 메서드에 @Transactional 어노테이션을 붙여줍니다.  
그리고, tbl\_test1 테이블을 drop했다가 다시 생성합니다.

```
56 • select * from tbl_test1;
```

<

Result Grid   Filter Rows:

col1
------

36

```
alter table ictboard add column replycount int default 0;
```

```
SET SQL_SAFE_UPDATES = 0;
```

```
update ictboard set replycount = (select count(rno) from ictreply where bno = ictboard.bno) where bno > 0;
```

이제 게시글마다 추가로 댓글이 몇 개가 달렸는지 표출해보겠습니다.  
게시판 프로젝트로 다시 돌아와주세요!

먼저, ictboard에 replycount 컬럼을 추가하고, 실제로 갯수를 표현하도록 하겠습니다.

workbench에서 update구문을 대량으로 처리하지 못하도록 safe update가 걸려있으므로 해제해주고 위의 구문을 실행합니다.



# 실 게시판 글 삭제시 트랜잭션 적용하기.

result Grid

Filter Rows:

Edit:

Export/Import:

Wrap Cell Content:

Fetch rows:

btno	content	title	writer	regdate	updatedate	replycount
30729	ㅎㅋㄷㅎㅋ	ㅋㄴㄷㄷ	ㅎㅋㄴㄷㅎ	2021-03-21 23:27:25	2021-03-21 23:27:25	1
30728	dsafa	수정했다222	efafew	2021-03-21 23:27:25	2021-03-21 23:27:25	4
30727	4124	훈련교강사 점수 판정에 대한 기준	15125	2021-03-21 23:27:25	2021-03-21 23:27:25	5
30726	42141	412	4124	2021-03-21 23:27:25	2021-03-21 23:27:25	2
30725	Mock본문	Mock제목	Mock글쓴이	2021-03-21 23:27:25	2021-03-21 23:27:25	1
30724	테스트 본문	테스트 제목	테스트 글쓴이	2021-03-21 23:27:25	2021-03-21 23:27:25	0

```
@Data
public class BoardVO {

    private Long bno;
    private String title;
    private String content;
    private String writer;
    private Date regDate;
    private Date updateDate;
    private int replyCount;
}
```

위와 같이 replycount 부분에 리플개수가 표시되면 성공입니다.

더불어 BoardVO 역시 replycount를 받을 수 있도록 멤버변수를 추가합니다.

```
<select id="listPage" resultType="org.ict.domain.BoardVO">
  <!-- 아래 <![CDATA[ 는 닫는부분인 ]]> 사이의 모든 문자를
  쿼리문으로만 인식하고 태그요소로 인식하지 않게 함. -->
  <![CDATA[
    SELECT bno, title, content, writer, regdate, updatedate, replycount
    FROM ictboard
    WHERE bno > 0
  ]]>
  <include refid="search"></include>

  <![CDATA[
    ORDER BY bno DESC
    limit #{pageStart}, #{number}
  ]]>
</select>
```

boardMapper.xml에서 SQL문도 다음과 같이 고쳐줍니다.

```
public void updateReplyCount(@Param("bno") Long bno,  
                             @Param("amount") int amount);
```

다음 이제 댓글이 늘어날때마다 ReplyCount를 증가시켜주는 메서드를 BoardMapper 인터페이스에 먼저 추가합니다.

bno는 해당 게시물의 번호, amount는 변동값을 나타내며 마이바티스는 기본적으로 하나의 파라미터를 쓰는것을 전제로 하기 때문에 위와같이 2개 이상을 정의할때는 @Param 어노테이션을 붙여줍니다.



```
<update id="updateReplyCount">  
  update ictboard  
  set  
    replycount = replycount + #{amount} where bno = #{bno}  
</update>
```

위와 같이 **amount**로 댓글의 증감을 나타내고, **bno**로 해당 글이 몇 번이었는지를 카운팅해 댓글숫자를 가늠합니다.

```
@Service
public class ReplyServiceImpl implements ReplyService {

    @Autowired
    private ReplyMapper mapper;

    @Autowired
    private BoardMapper boardMapper;
}
```

이제 댓글 추가와, replycount 컬럼 업데이트라는 작업이 댓글 쓰기라는 하나의 서비스에 동반실행됩니다.

즉, 이 역시 트랜잭션의 대상이 됩니다.

먼저 ReplyServiceImpl 내부에서 BoardMapper를 쓸 수 있도록 처리합니다.

```
public interface ReplyMapper {  
    public List<ReplyVO> getList(int l  
  
    public void create(ReplyVO vo);  
    public void update(ReplyVO vo);  
    public void delete(int rno);  
  
    public int getBno(int rno);  
}
```

```
<select id="getBno" resultType="Long">  
    SELECT bno FROM ictreply  
        WHERE rno = #{rno}  
</select>
```

그리고, 삭제시 댓글갯수를 1 줄이는 작업부터 하겠습니다.  
먼저 rno만으로 bno를 유추해야 하기 때문에 ReplyMapper쪽에 위와 같이 getBno 메서드를 정의 및 구현합니다.

```
@Transactional  
@Override  
public void removeReply(int rno) {  
    mapper.delete(rno);  
    Long bno = mapper.getBno(rno);  
    boardMapper.updateReplyCount(bno, -1);  
}
```

이제 이 메서드는 리플 삭제라는 하나의 서비스 기능 구현에 실행시에는 정작 3개의 쿼리문을 돌려야 하기 때문에 반드시 트랜잭션이 적용되어야 합니다.

**register**쪽은 직접 구현해보세요!

## 실 게시판 글 삭제시 트랜잭션 적용하기.

30729	<a href="#">크르르르 [1]</a>
30728	<a href="#">수정했다222 [4]</a>
30727	<a href="#">훈련교강사 점수 판정에 대한 기준 [5]</a>
30726	<a href="#">412 [2]</a>
30725	<a href="#">Mock제목 [1]</a>
30724	<a href="#">테스트 제목 [0]</a>
30723	<a href="#">테스트 제목 [1]</a>
30722	<a href="#">크르르르 [0]</a>
30721	<a href="#">수정했다222 [0]</a>
30720	<a href="#">훈련교강사 점수 판정에 대한 기준 [0]</a>

마지막으로 화면 출력을 위해 `list.jsp` 역시 표현부를 조금 수정해주세요.

위와 같이 댓글갯수를 모두 표출해주시면 됩니다  
(hint: 제목 표출부에 `replycount`를 사용하세요)