1. Spring AOP와 트랜잭션 관리

가. 스프링의 중요한 특징으로 뽑히는 1순위는 '의존성 주입'과 관련된 내용이다.

나. 그 다음으로 뽑히는 특징은 역시 AOP(Aspect Oriented Programming)라는 기능이다. 스프링은 기존의 비즈니스 로직 외 작성해야 하는 코드를 별도로 분리함으로써 개발자가 좀 더 비즈니스 로직에만 집중해서 처리할 수 있는 방법을 제공한다.

다. 스프링이 가진 AOP를 사용해서 개발할 때 공통적이고,반복적인 그러나 비즈니스 로직의 핵심이 아닌 부분을 어떻게 처리하는지를 알아보려고 한다.

라. 또한 AOP와 더불어 데이터베이스 처리에 가장 중요한 트랜잭션 처리와 관련된 부분도 진행한다.

마. AOP의 'Aspect'라는 용어는 사전적 의미로는 측면 혹은 어떤 건물의 도형의 면을 의미하지만 실제 개발에서는 비즈니스 로직은 아니지만 반드시 해야 하는 작업 정도로 해석할 수 있다. 어플리케이션의 핵심 기능은 아니지만,어플리케이션을 구성하는 중요한 요소이고,부가적인 기능을 담당하는 요소 어플리케이션의 핵심적인 기능에서 부가적인 기능을 분리해서 애스펙트라는 모듈로 만들어서 설계하고 개발하는 방법을 AOP(Aspect Oriented Programming)이라고 한다

바. 스프링의 AOP 지원은 개발의 핵심적인 비즈니스 로직을 개발하는 데에만 집중하고, 나머지 부가적인 기능은 설정을 통해서 조정한다.

2. AOP의 중요 용어

가. Aspect : 공통 관심사에 대한 추상적인 명칭. 예를 들어 로깅이나 보안, 트랜잭션과 같은 기능 자체에 대한 용어

나. Advice : 실제로 기능을 구현한 객체

다. Join points : 공통 관심사를 적용할 수 있는 대상. 스프링 AOP에서는 각 객체의 메서드가 이에 해당

라. Pointcuts : 여러 메서드 중 실제 advice가 적용될 대상 메서드

마. target : 대상 메서드가 가지는 객체

3. Advice 종류

Advice는 실제 구현된 클래스로 생각할 수 있다.

타입	기능
Before Advice	target의 메서드를 호출전에 적용
After returning	target의 메서드를 호출후에 적용
After throwing	target의 예외 발생 후 적용
After	target의 메서드 호출 후 예외의 발생에 관계없이 적용
Around	target의 메서드 호출 이전과 이후 모두 적용(가장 광범위하게 사용됨)

여러 종류의 타입 중에서 가장 중요한 타입은 Around이다.

4. AOP를 적용할 수 있는 방식 역시 XML을 이용할 수 있고, 애노테이션을 이용할 수도 있다. 스프링 AOP를 활용할려면 관련 라이브러리가 필요하고, 트랜잭션을 처리 하기 위해서도 'spring-tx' 라이브러리를 사용해야 한다. pom.xml 관련 라이브러리 추가

```
<!-- 스프링 트랜잭션 적용 -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-tx</artifactId>
                <version>${org.springframework-version}</version>
        </dependency>

<!-- 스프링 aop 설정 라이브러리 시작-->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-aop</artifactId>
                <version>${org.springframework-version}</version>
        </dependency>
```

5. AOP 기능을 적용하기 위해서는 AspectJ 언어 문법을 이용하기 때문에 이와 관련된 라이브러리 역시 추가해야 한다. pom.xml 수정

```
<dependency>
        <groupId>aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
        <version>1.5.4</version>
</dependency>

<dependency>
        <groupId>aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.5.4</version>
</dependency>
```

6. root-context.xml에서 AOP의 설정을 통한 자동적인 프록시 객체 생성을 위해서 다음과 같은 설정을 미리 추가한다.

```
<aop:aspectj-autoproxy />
        <!-- aop 자동 프록시 설정.  -->
```

7. 샘플용 테이블을 설계

```
--tbl_user 테이블 생성
create table tbl_user(
 uid2 varchar2(50) primary key --회원 아이디
 ,upw varchar2(50) not null --비번
 ,uname varchar2(100) not null --회원이름
 ,upoint number(38) default 0 --메시지가 보내지면 포인터
 --점수 10점이 업
```

```
);
insert into tbl_user (uid2,upw,uname) values('user00',
'user00','홍길동');
insert into tbl_user (uid2,upw,uname) values('user01',
'user01','이순신');
select * from tbl_user;

--tbl_message 테이블 생성
create table tbl_message(
 mid number(38) primary key
 ,targetid varchar2(50) not null --외래키 제약조건으로 설
 --정. tbl_user 테이블의 uid2 컬럼 아이디값을 가져와 저장
 ,sender varchar2(50) not null--보낸사람
 ,message varchar2(1000) not null --보낸 메시지
 ,senddate date --보낸 날짜
);
--외래키 제약조건 추가
alter table tbl_message add constraint fk_usertarget
foreign key(targetid) references tbl_user(uid2);

--mid_no_seq 시퀀스 생성
create sequence mid_no_seq
start with 1
increment by 1
nocache;

select mid_no_seq.nextval from dual;

select * from tbl_message;
```

8. 데이터 저장빈 클래스

```
package org.zerock.domain;

public class MessageVO {

	private int mid;
	private String targetid;
	private String sender;
	private String message;
	private String senddate;
```

```
	public int getMid() {
		return mid;
	}
	public void setMid(int mid) {
		this.mid = mid;
	}
	public String getTargetid() {
		return targetid;
	}
	public void setTargetid(String targetid) {
		this.targetid = targetid;
	}
	public String getSender() {
		return sender;
	}
	public void setSender(String sender) {
		this.sender = sender;
	}
	public String getMessage() {
		return message;
	}
	public void setMessage(String message) {
		this.message = message;
	}
	public String getSenddate() {
		return senddate;
	}
	public void setSenddate(String senddate) {
		this.senddate = senddate;
	}
}

package org.zerock.domain;

public class UserVO {

	private String uid2;
	private String upw;
	private String uname;
	private int upoint;
```

```
	public String getUid2() {
		return uid2;
	}
	public void setUid2(String uid2) {
		this.uid2 = uid2;
	}
	public String getUpw() {
		return upw;
	}
	public void setUpw(String upw) {
		this.upw = upw;
	}
	public String getUname() {
		return uname;
	}
	public void setUname(String uname) {
		this.uname = uname;
	}
	public int getUpoint() {
		return upoint;
	}
	public void setUpoint(int upoint) {
		this.upoint = upoint;
	}
}
```

9. MessageDAO 인터페이스 설계

```
package org.zerock.persistence;

import org.zerock.domain.MessageVO;

public interface MessageDAO {

	void create(MessageVO vo);

}
```

10. MessageDAOImpl 작성

```
package org.zerock.persistence;

import org.apache.ibatis.session.SqlSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.zerock.domain.MessageVO;

@Repository
public class MessageDAOImpl implements MessageDAO {

        @Autowired
        private SqlSession sqlSession;

        @Override
        public void create(MessageVO vo) {
          this.sqlSession.insert("m_in2",vo);
        }
}
```

11. PointDAO 인터페이스

메시지 전송에 대한 포인트 점수를 처리하기 위해서 필요하다.

```
package org.zerock.persistence;

public interface PointDAO {

        void updatePoint(String sender, int i);

}
```

12.PointDAOImpl 구현

```
package org.zerock.persistence;

import java.util.HashMap;
import java.util.Map;

import org.apache.ibatis.session.SqlSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class PointDAOImpl implements PointDAO {
```

```
	@Autowired
	private SqlSession sqlSession;

	@Override
	public void updatePoint(String sender, int point) {
		Map<String,Object> pm=new HashMap<>();
		pm.put("sender",sender);//보낸사람
		pm.put("point",point);//포인터 점수
		this.sqlSession.update("pointUp",pm);
	}
}
```

13. message.xml 과 point.xml 매퍼 태그

message.xml

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

<!DOCTYPE mapper
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

 <mapper namespace="Message">

 <!-- 메시지 추가 -->
  <insert id="m_in2">
   insert into tbl_message (mid,targetid,sender,message,
   senddate)  values(mid_no_seq.nextval,#{targetid},
   #{sender},#{message},sysdate)
  </insert>
 </mapper>
```

point.xml

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

<!DOCTYPE mapper
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

 <mapper namespace="Point">

  <!-- 포인터 점수 10점 업 -->
  <update id="pointUp">
   update tbl_user set upoint=upoint+#{point}
```

```
    where uid2=#{sender}
  </update>
</mapper>
```

14. MessageService.java

```
package org.zerock.service;

import org.zerock.domain.MessageVO;

public interface MessageService {

        void addMessage(MessageVO vo);

}
```

15. MessageServiceImpl.java

기존의 서비스 객체와 다른 점은 두 개의 DAO를 이용해서 하나의 로직이 완성되는 형태라는 것이다. 즉 MessageDAO와 PointDAO를 함께 사용했다는 점이다. 이 부분은 고객의 추가 요구 사항 반영과 AOP를 통한 트랜잭션 적용을 통해서 사이트 신뢰도와 일관성을 유지하게 한다.

@Transactional 애노테이션은 인터페이스와 클래스 선언, 메서드 선언에서 사용할 수 있다. 적용 우선순위는 메서드에 설정한 것이 가장 높고, 그 다음이 클래스 선언, 인터페이스 선언 순번으로 우선순위가 적용된다.

```
package org.zerock.service;

import javax.inject.Inject;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.zerock.domain.MessageVO;
import org.zerock.persistence.MessageDAO;
import org.zerock.persistence.PointDAO;

@Service
public class MessageServiceImpl
implements MessageService {

        @Autowired
        private MessageDAO messageDAO;
```

```
        @Inject
        private PointDAO pointDAO;

        //aop를 통한 트랜잭션 적용
        @Transactional
        @Override
        public void addMessage(MessageVO vo) {
/* 왼쪽에 보이는 위아래 방향 화살표는 스프링 aop의 어드바이
 * 스의 적용범위를 나타내는 Around타입을 뜻한다.
 *  1.스프링 aop용어 정리
 *   가. Advice(어드바이스):실제 기능을 구현한 객체
 *   나. Around타입:target(타켓) 메서드 호출전 이후 모두
 *   적용(가장 광범위하게 사용됨)
 *   다. target(타켓): 대상메서드를 가지는 객체
 *
 */
                this.messageDAO.create(vo);//메시지 추가
                this.pointDAO.updatePoint(vo.getSender(),10);
                //메시지를 보낸사람에게 포인터 점수 10 추가
        }
}
```

16. root-context.xml 에 AOP 기능 설정
이 설정은 향후에 XML방식으로 AOP 기능을 설정할 때 사용한다.

```
 <aop:config></aop:config>
        <!-- aop 설정=>AOP를 애너테이션 말고
        XML방식으로 설정할 때 사용 -->
```

17.작업을 테스트 하기 위해서 MessageController를 작성한다.
이 컨트롤러는 @RestController로 작성되어 아작스를 통한 메시지를 전송할 수 있다.

```
package org.zerock.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.zerock.domain.MessageVO;
import org.zerock.service.MessageService;
```

```
@RestController
@RequestMapping("/message")
public class MessageController {

	@Autowired
	private MessageService messageService;

	//메시지 추가
	@RequestMapping(value="/",
			method=RequestMethod.POST)
	public ResponseEntity<String> addMessage(
			@RequestBody MessageVO vo){
		ResponseEntity<String> entity=null;

		try {
			this.messageService.addMessage(vo);
entity=new ResponseEntity<>("SUCCESS",HttpStatus.OK);
		}catch(Exception e) {
			e.printStackTrace();
			entity=new ResponseEntity<>(e.getMessage(),
					HttpStatus.BAD_REQUEST);
		}
		return entity;
	}
}
```

18. 스프링의 트랜잭션 처리

가. 데이터베이스의 트랜잭션 처리는 AOP의 설정을 응용하기 때문이다.

나. 스프링에서 트랜잭션 처리는 기본적으로 XML을 사용해서 하는 방식과 애노테이션을 활용하는 방식으로 나누어 진다.

다. 초급 개발자가 트랜잭션에 대한 처리를 빼먹는 경우가 많다.

라. 사이트에서 흔하게 처리하는 트랜잭션 적용 상황 예

회원이 특정 게시판에 게시글을 추가하면 회원의 포인트 점수가 올라가야 하는 경우

문의 게시판에 글을 등록하면 데이터베이스에도 글이 등록되지만, 담당자에게도 메일이 발송 돼야 하는 경우

마. 트랜잭션 기본 원칙

원자성: 되면 모두 되고, 안되면 모두 다 안된다.

일관성: 트랜잭션으로 처리된 데이터와 일반 데이터 사이에는 전혀 차이가 없어야만 한다.

바. 사용자가 다른 사용자에게 메시지를 남기면 메시지 등록과 함께 10포인트 증가 작업이 함께 이루어 져야 한다. 이런 경우는 트랜잭션 적용 대상이 된다.

사. 스프링에서 트랜잭션을 처리하기 위해서는 @Transactional 애너테이션을 이용하면 간단히 적용할 수 있다.

19. 트랜잭션 매니저의 설정
root-context.xml 의 일부

```
 <!-- 트랜잭션 설정 -->
    <bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                    <property name="dataSource" ref="dataSource"></property>
          </bean>
          <tx:annotation-driven />
          <!-- 트랜잭션 설정 애너테이션 ,
          @Transactional 애너테이션을 이용한 트랜잭션
          관리가 가능. -->
```