Open Session In View Pattern

Eternity(http://aeternum.egloos.com/)

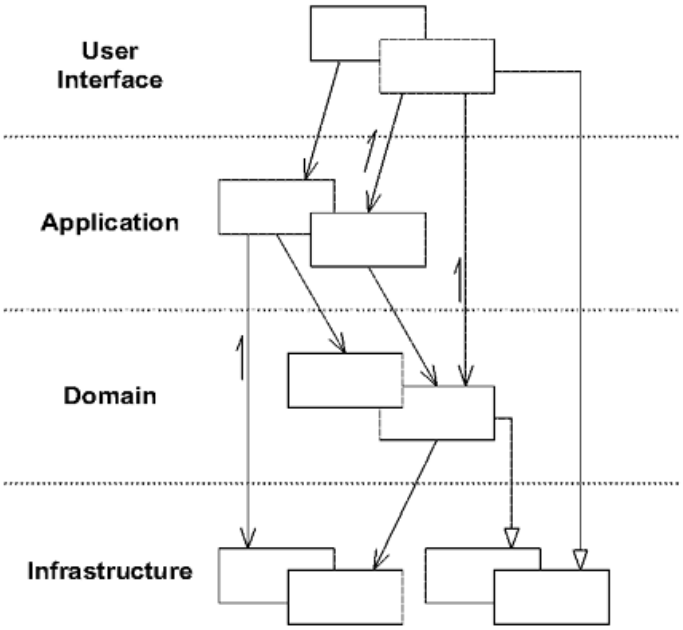
레이어 아키텍처(Layered Architecture)

사용자 인터페이스와 영속성 메커니즘을 포함하는 대부분의 엔터프라이즈 애플리케이션은 "레이어 아키텍처(Layered Architecture)"[Buschman POSA1] 패턴을 기반으로 하고 있다. 레이어 아키텍처 패턴의 기본 아이디어는 시스템을 유사한 책임을 지닌 레이어로 분해한 후 각각의 레이어가 하위의 레이어에만 의존하도록 시스템을 구성하는 것이다. 현대적인 엔터프라이즈 애플리케이션은 각 레이어가 바로 하위 레이어에만 의존해야 한다는 제약을 완화시킨 '완화된 레이어 시스템(relaxed layered system)'인 동시에 프레임워크의 인터페이스나 클래스를 상속 받아 애플리케이션의 핵심 클래스를 구현하는 '상속을 통한 레이어 구성(layering through inheritance)' 방식을 적용하는 것이 일반적이다.

다른 아키텍처 패턴처럼 레이어 아키텍처의 목적 역시 시스템의 '관심사를 분리(separation of concerns)'하는 것이다. 서로 다른 관심사를 서로 다른 레이어로 분리하여 전체적인 시스템의 결합도를 낮추고 인지 과부하를 방지함으로써 재사용성을 높이고 유지보수성을 향상시키는 것이 레이어 아키텍처의 근본 목적이다.

> 서로 다르고 관련이 없는 책임(responsibility)들은 소프트웨어 내에서 서로 분리해야 한다. 그 예로 책임들을 서로 다른 컴포넌트에 부여하는 방법을 들 수 있겠다. 특정 태스크의 해결을 위해 매진하는 협력(collaboration) 컴포넌트들은 다른 태스크의 계산에서 사용되는 컴포넌트들로부터 분리되어야 한다. 만약 어떤 컴포넌트가 각기 다른 정황에서 각기 다른 역할들을 담당한다면, 이 역할들은 그 컴포넌트 내에 각각 독립적으로 분리되어야 한다. 이 책에서 제시한 패턴 시스템의 거의 모든 패턴이 나름의 방법으로 이 기본 원칙을 처리한다. 예를 들어 Model-View-Controller 패턴은 역할을 내부적으로 모델, 사용자 프레젠테이션, 입력 처리로 분리한다.
>
> \- Frank Buschman, Pattern-Oriented Software Architecture Vol. 1

레이어의 명칭이나 개수는 문헌에 따라 약간의 차이가 있지만 대부분의 애플리케이션은 사용자 화면을 구성하는 "사용자 인터페이스 레이어(User Interface Layer)", 애플리케이션의 제어 흐름을 관리하는 "애플리케이션 레이어(Application Layer)", 도메인의 핵심 로직을 포함하는 "도메인 레이어(Domain Layer)", 상위 계층을 지원하기 위한 "인프라스트럭처 레이어(Infrastructure Layer)"로 구성된다.



<그림 1> 일반적인 엔터프라이즈 애플리케이션의 레이어 구성

[출처: Domain-Driven Design]

일반적인 엔터프라이즈 애플리케이션은 다음과 같은 원칙에 따라 레이어를 분리한다.

- **모델-뷰 분리(Model-View Separation)**[Fowler PEAA, Larman AUP] - 도메인 로직과 사용자 인터페이스는 서로 다른 관심사를 다룬다. 따라서 모델(도메인 로직과 관련된 관심사)과 뷰(사용자 인터페이스와 관련된 관심사)는 별도의 레이어로 분리해야 한다. 모델은 화면 출력과 관련된 로직을 포함해서는 안되며 뷰는 비즈니스 로직을 포함해서는 안된다. 모델-뷰 분리 원칙에서 가장 중요한 것은 의존성의 방향이다. 뷰는 모델에 의존할 수 있지만 모델은 뷰에 의존할 수 없다. 따라서 사용자 인터페이스 레이어에서 도메인 레이어로의 접근은 허용되지만 도메인 레이어에서 사용자 인터페이스로의 접근은 허용되지 않는다.
- **깔끔하고 얇은 뷰(Clean and Thin View)**[Johnson J2EEDD] - 모델-뷰 분리 원칙에 따라 모델과 뷰를 분리했다면 뷰는 오직 마크업과 화면 출력 로직만을 포함해야 하며(Clean View), 시스템의 상태를 변경시킬 수 있는 비즈니스 로직을 포함해서는 안 된다(Thin View). OPEN SESSION IN VIEW 패턴에 대한 몇 가지 논쟁은 "얇은 뷰(Thin View)"의 정의에 대한 의견 차이로부터 기인하는 것으로 보인다.
- **영속성 분리(PI, Persistence Ignorance)**[Nilsson ADDD] - 모델(도메인 로직)과 뷰(사용자 인터페이스 로직)가 서로 다른 관심사를 다루는 것과 마찬가지로, 도메인 로직과 영속성 로직 역시 서로 다른 관심사를 다룬다. 따라서 도메인 로직과 영속성 로직은 서로 다른 레이어로 분리해야 한다. 모델-뷰 분리 원칙과 마찬가지로 여기에서도 의존성의 방향이 중요하다. 영속성 메커니즘은 도메인 객체에 의존하지만 도메인 객체는 영속성 메커니즘으로부터 독립적이어야 한다. 영속성 메커니즘과 도메인을 분리하기 위한 가장 일반적인 방법은 도메인 객체를 데이터베이스 관련

인프라스트럭처에 독립적인 POJO(Plain Old Java Object)로 개발하고 DAO(Data Access Object) 패턴[Alur CORE]을 이용해서 인터페이스 하부로 영속성 메커니즘을 캡슐화하는 것이다.

- **도메인 레이어 고립(Domain Layer Isolation)**[Evans DDD] – 도메인 레이어는 비즈니스를 구성하는 핵심 개념과 중요한 정보, 준수해야 하는 비즈니스 규칙을 표현하는 곳이다. 시스템을 단순하고 유연한 상태로 유지하기 위해서는 도메인 레이어를 기술적인 이슈로부터 고립시켜야 한다. 이 원칙은 "모델-뷰 분리 원칙"과 "영속성 분리 원칙"을 통해 달성하고자 하는 최종 목표를 나타내고 있다.

엔터프라이즈 애플리케이션에 있어서 레이어 아키텍처의 가장 큰 가치는 도메인의 핵심 지식을 포함하는 동시에 시스템의 실제적인 가치를 제공하는 도메인 레이어를 인프라스트럭처나 기술과 관련된 부가적인 관심사로부터 격리시킬 수 있다는 점이다. 이것은 근본적으로 Alistair Cockburn 의 HEXAGONAL ARCHITECTURE 패턴(또는 PORT & ADAPTER 패턴)[Cockburn HEXAGONAL]의 목적과 동일하다.

> [사용자 인터페이스로 비즈니스 로직이 누수되는] 사용자측 문제와 [애플리케이션 로직이 외부 데이터베이스나 타 서비스와 강하게 결합되는] 서버측 문제 모두 실제로는 설계와 프로그래밍 상에서의 동일한 오류로 인한 것이다 – 외부 엔티티와의 상호작용과 비즈니스 로직 간의 결합. 이용해야 하는 비대칭성은 애플리케이션의 "왼쪽" 측면 [사용자 인터페이스]과 "오른쪽" 측면 [데이터베이스] 간이 아니라 애플리케이션의 "내부" 측면 [도메인 레이어]과 "외부" 측면 [사용자 인터페이스와 데이터베이스] 간이다. 준수해야하는 규칙은 "내부"에 존재하는 코드는 "외부" 파트로 누수되어서는 안된다는 것이다.
>
> - Alistair Cockburn, The Hexagonal Architecture (Ports & Adapters) Pattern

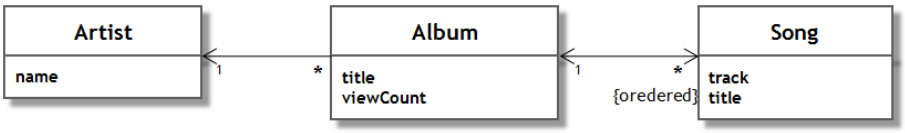
도메인 레이어는 도메인 모델이 살아가는 곳이다. 도메인 레이어에 도메인과 무관한 기술적인 이슈가 침투할수록 시스템을 변경하고 이해하기 어려워 진다.

> 복잡한 프로그램을 여러 개의 계층으로 분할하라. 응집력 있고 오직 하위의 계층에만 의존하는 각 계층에서 설계를 발전시켜라. 표준 아키텍처 패턴에 따라 상위 계층과의 결합은 느슨하게 유지하라. 도메인 모델에 관계된 코드는 모두 한 계층에 모으고 사용자 인터페이스 코드 및 애플리케이션 코드, 인프라스트럭처 코드로부터 격리하라. 화에 표시 및 저장, 애플리케이션 흐름 관리 등의 책임에서 자유로운 도메인 객체는 도메인 모델의 표현에만 집중할 수 있다. 이를 통해 모델은 진화를 거듭하여 업무의 본질적인 지식을 포착하고 해당 업무 지식을 적용시킬 수 있을 정도로 풍부하고 명확해질 것이다.
>
> - Eric Evans, Domain-Driven Design

예제 애플리케이션

어떤 개념이나 패턴의 요지를 가장 쉽게 전달하는 방법은 실제로 동작하는 예제를 살펴 보는 것이다. 여기에서는 앨범과 앨범에 수록된 음악의 목록을 관리하는 웹 사이트를 개발한다고 가정한다.

먼저 도메인에 관한 핵심적인 개념과 업무 규칙을 표현하는 도메인 레이어로부터 시작하자. 도메인 레이어는 도메인 모델에 표현된 개념과 개념간의 관계를 반영하여 구현된 클래스들을 포함하는 것이 이상적이다. 앨범 관리 사이트의 도메인 모델은 발매된 앨범을 표현하는 Album 과 앨범에 수록된 노래를 나타내는 Song, Album 을 발표한 가수나 음악가를 나타내는 Artist 로 구성되어 있다.



<그림 2> 앨범 관리 사이트를 위한 도메인 모델

영속성을 다루기 위한 프레임워크로 "하이버네이트"[Bauer JPAH]를 사용하기로 한다. 하이버네이트는 하부의 영속성 메커니즘으로부터 독립적으로 도메인 모델을 구현할 수 있도록 "투명한 영속성(transparent persistence)" 메커니즘을 제공해 주는 오픈소스 객체-관계 매핑(ORM, Object-Relational Mapping) 프레임워크다. 클래스와 객체 간의 매핑은 JPA(Java Persistence API)에 정의된 표준 어노테이션을 사용하여 클래스에 선언하는 것으로 한다.

앨범을 발매한 가수나 음악가를 나타내는 ENTITY[Evans DDD]인 Artist 는 음악가의 이름을 속성으로 가지는 간단한 클래스다.

Artist.java

```
@Entity
public class Artist {
  @Id
  @GeneratedValue
  private Long id;

  private String name;

  Artist() {
  }

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

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

```
  }

  public Long getId() {
    return id;
  }
}
```

Album 은 앨범을 발매한 가수나 음악가를 나타내는 Artist 와 앨범에 수록된 음악인 Song 의 목록을 포함하는 ENTITY 다. title 은 Album 의 타이틀을, viewCount 는 웹 상에서 해당 앨범이 조회된 횟수를 저장한다. Song 을 Album 에 추가하는 addSong() 메소드는 Album 과 Song 과의 양방향 연관관계를 설정하기 위해 Song 클래스의 setAlbum() 메소드를 호출하고 있다.

Album.java

```
@Entity
public class Album {
  @Id
  @GeneratedValue
  private Long id;

  private String title;
  private int viewCount;

  @ManyToOne(fetch=FetchType.LAZY, optional=false)
  @JoinColumn(name="ARTIST_ID")
  private Artist artist;

  @OneToMany(cascade=CascadeType.ALL, mappedBy="album")
  private List<Song> songs = new ArrayList<Song>();

  Album() {
  }

  public Album(String title, Artist artist) {
    this.title = title;
    this.artist = artist;
  }

  public Long getId() {
    return id;
  }

  public String getTitle() {
    return title;
  }

  public Artist getArtist() {
    return artist;
  }

  public void addSong(Song song) {
    songs.add(song);
    song.setAlbum(this);
  }
```

```
  public List<Song> getSongs() {
    return songs;
  }

  public void increaseViewCount() {
    viewCount++;
  }
}
```

Song 은 음악 제목을 나타내는 title 과 트랙번호를 나타내는 track, 자신이 수록된 Album 을 참조하는 album 속성을 포함한다.

Song.java

```
@Entity
public class Song {
  @Id
  @GeneratedValue
  private Long id;

  private Integer track;
  private String title;

  @ManyToOne
  @JoinColumn(name="ALBUM_ID")
  private Album album;

  Song() {
  }

  public Song(Integer track, String title) {
    this.track = track;
    this.title = title;
  }

  public Long getId() {
    return id;
  }

  public Integer getTrack() {
    return track;
  }

  public String getTitle() {
    return title;
  }

  void setAlbum(Album album) {
    this.album = album;
  }
}
```

데이터베이스로부터 Album 을 조회하는 영속성 메커니즘을 추상화하기 위해 REPOSITORY[Evans DDD] 인터페이스를 추가한다. REPOSITORY 내부에 영속성 메커니즘과

관련된 기술적인 세부사항을 캡슐화시킴으로써 "영속성 분리(Persistence Ignorance)" 원칙을 준수할 수 있다.

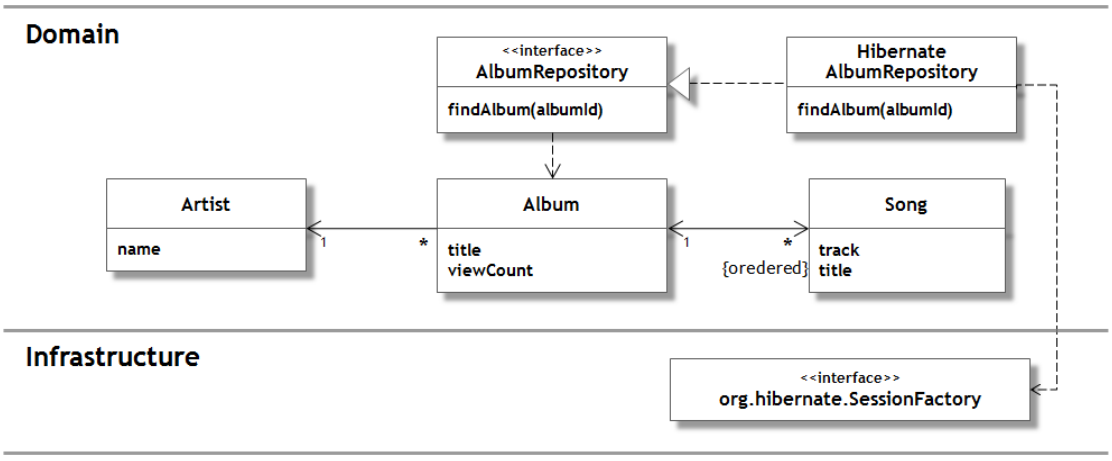
AlbumRepository.java

```
public interface AlbumRepository {
  Album findAlbum(Long albumId);
}
```

AlbumRepository 인터페이스의 구현체인 HibernateAlbumRepository 는 하이버네이트 Session 을 통해 데이터베이스로부터 Album 을 조회한다.

HibernateAlbumRepository.java

```
@Repository
public class HibernateAlbumRepository implements AlbumRepository {
  @Autowired
  private SessionFactory sessionFactory;

  @Override
  @Transactional(readOnly=true)
  public Album findAlbum(Long albumId) {
    return (Album)sessionFactory.getCurrentSession()
            .get(Album.class, albumId);
  }
}
```



<그림 3> REPOSITORY를 통한 영속성 분리

애플리케이션 레이어(Application Layer)는 도메인 레이어에 요청을 위임하고 애플리케이션의 흐름을 제어하는 얇은 계층으로 서비스 계층(Service Layer)[Fowler PEAA]이라고도 한다. AlbumService 는 앨범 정보를 조회한 후 앨범의 조회 카운트를 증가시키는 역할을 수행하는 애플리케이션 레이어 SERVICE[Evans DDD]이다.

먼저 AlbumService 인터페이스를 추가하자.

AlbumService.java

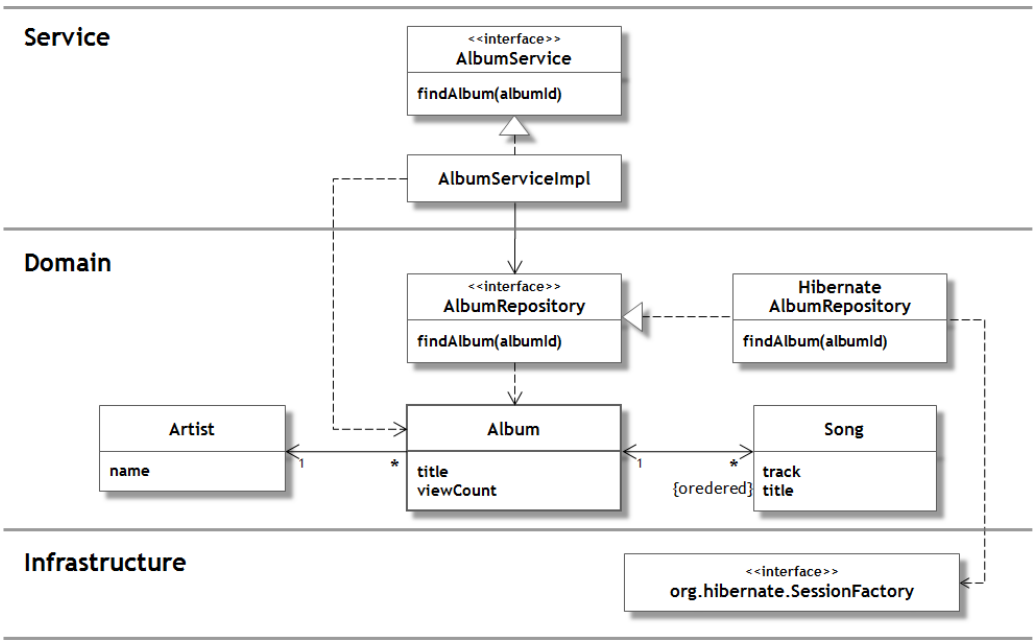
```
public interface AlbumService {
  Album findAlbum(Long albumId);
}
```

AlbumServiceImpl 구현체는 AlbumRepository 를 통해 원하는 Album ENTITY 를 조회하고, 로드된 Album 의 조회 카운트를 증가시킨 후 반환한다.

AlbumServiceImpl.java

```
@Service
public class AlbumServiceImpl implements AlbumService {
  @Autowired
  private AlbumRepository albumRepository;

  @Override
  @Transactional
  public Album findAlbum(Long albumId) {
    Album result = albumRepository.findAlbum(albumId);
    result.increaseViewCount();
    return result;
  }
}
```

애플리케이션 레이어는 애플리케이션의 트랜잭션 경계를 정의한다. 따라서 조회 카운트가 증가된 Album ENTITY 는 "작업 단위(Unit of Work)"[Fowler PEAA]인 findAlbum() 메소드가 종료될 때 "트랜잭션 단위의 쓰기 지연(transactional write-behind)" 메커니즘[Bauer JPAH]에 의해 데이터베이스에 갱신된 내용이 자동으로 커밋된다.



<그림 4> 애플리케이션 레이어 SERVICE 추가

사용자 인터페이스 레이어에 사용자 요청을 처리하기 위한 컨트롤러를 추가한다. 예제에서는 컨트롤러의 구현을 위해 SpringMVC 프레임워크를 사용하고 있지만 Struts 등의 다른 MVC 프레임워크를 사용해도 무방하다.

AlbumController.java

```
@Controller
public class AlbumController {
  @Autowired
  private AlbumService albumService;

  @RequestMapping("/album")
  public String execute(@RequestParam("albumId") long albumId,
      Model model) {
    Album album = albumService.findAlbum(albumId);

    model.addAttribute("album", album);

    return "album";
  }
}
```

애플리케이션 레이어 SERVICE 는 하부의 도메인 레이어를 캡슐화하고 외부에서 호출 가능한 인터페이스를 정의한다. 사용자 인터페이스 레이어와 도메인 레이어는 SERVICE 인터페이스를 경계로 상호 분리되며, 의존성의 방향은 사용자 인터페이스 레이어에서 애플리케이션 레이어 또는 도메인 레이어로 향하게 된다. 따라서 잘 정의된 애플리케이션 계층을 제공함으로써 "모델-뷰 분리(Model-View Separation)" 원칙을 준수할 수 있다.

도메인 객체와 영속성 메커니즘은 REPOSITORY 인터페이스를 경계로 상호 분리되며, REPOSITORY 구현 클래스는 비침투적인 하이버네이트 프레임워크를 사용하여 도메인 객체와 영속성 메커니즘을 분리하고 있다. 따라서 잘 정의된 REPOSITORY 인터페이스와 비침투적인 프레임워크의 지원을 통해 "영속성 도메인 분리(Persistence-Domain Separation)" 원칙을 준수할 수 있다.

이와 같이 "모델-뷰 분리(Model-View Separation)" 원칙과 "영속성 도메인 분리(Persistence-Domain Separation)" 원칙에 따라 아키텍처를 구성함으로써 "도메인 레이어 고립(Domain Layer Isolation)" 원칙을 준수할 수 있다.

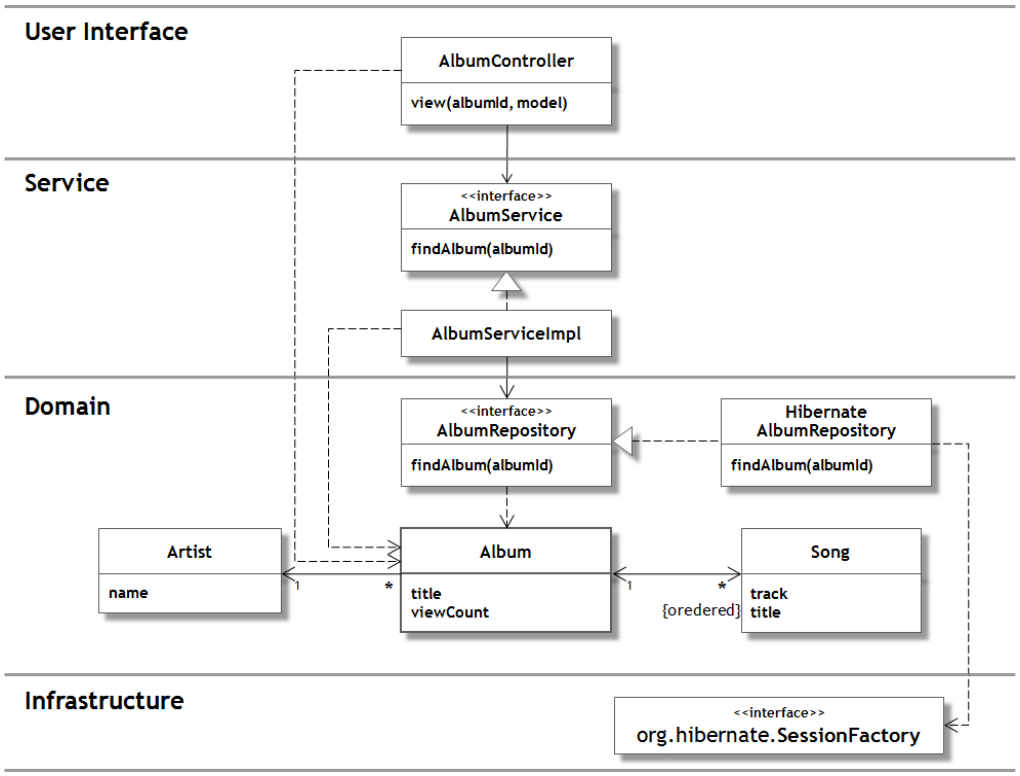
사용자에게 표시될 최종 결과를 담고 있는 뷰는 JSP 를 사용하여 렌더링 된다. JSP 페이지에서는 조회한 Album ENTITY 로부터의 연관 관계 탐색을 통해 Artist 와 Song 정보를 출력한다.

album.jsp

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
<title>앨범 정보</title>
</head>
<body>
  <h1>앨범 정보</h1>

  <h2>앨범명</h2>
  ${album.title}<br />
  <h2>가수명</h2>
  ${album.artist.name}<br/>
  <h2>수록곡</h2>
  <c:forEach var="song" items="${album.songs}">
    ${song.track} - ${song.name}<br />
  </c:forEach>
</body>
</html>
```

JSP 페이지는 화면을 구성하기 위해 마크업과 간단한 EL(Expression Language) 표현식만을 사용하고 있다. 따라서 이 뷰는 깔끔하다(Clean). 또한 반환된 Album ENTITY 를 사용해서 화면에 출력하는 작업만 수행하고 있다. 따라서 이 뷰는 얇다(Thin).

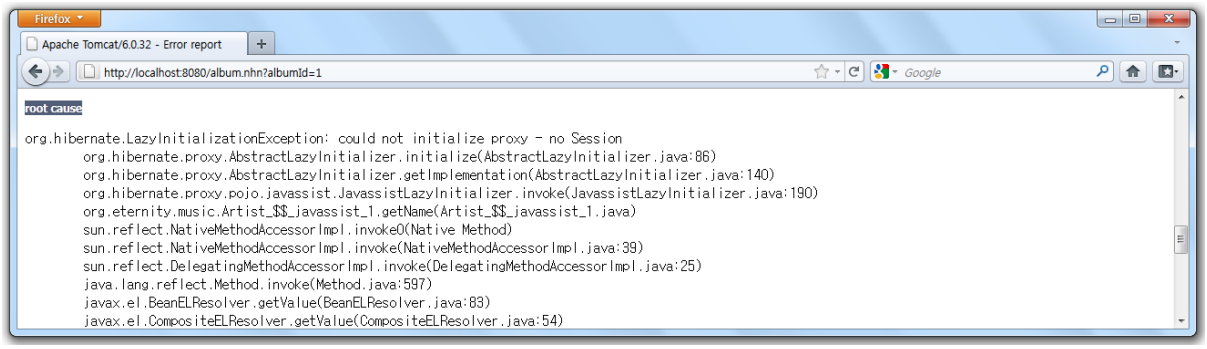


<그림 5> 모델-뷰 분리와 영속성 분리를 통한 도메인 레이어 고립

지금까지 살펴 본 바와 같이 앨범 관리 사이트의 아키텍처는 앞에서 살펴 본 엔터프라이즈 애플리케이션에서 레이어 아키텍처를 적용할 때 따라야 할 4 가지 핵심 원칙인 "모델 뷰

분리(Model-View Separation)" 원칙, "영속성 분리(Persistence Ignorance)" 원칙, "얇고 깔끔한 뷰(Clean and Thin View)" 원칙, "도메인 레이어 고립(Domain Layer Isolation)" 원칙 모두를 준수하고 있다.

문제는 Detached 상태의 Album 인스턴스를 사용해서 JSP 를 렌더링할 때 LazyInitializationException 이 발생한다는 것이다.



<그림 6> JSP 렌더링 도중 발생한 LazyInitializationException

앨범에 수록된 가수의 이름을 출력하기 위해 Album 객체의 artist 프로퍼티에 접근하는 순간 예상하지 못한 예외가 발생했음을 알 수 있다. Album 에 포함된 Artist 인스턴스에 접근하려고 할 때 예외가 발생하는 이유는 무엇일까? 하이버네이트를 사용할 경우 연관관계를 통해 객체 간의 관계를 탐색하는 일반적인 객체지향 접근 방법이 허용되지 않는 것일까?

문제의 원인을 이해하기 위해서는 무대 뒤편에서 은밀하게 움직이는 하이버네이트의 메커니즘을 이해할 필요가 있다.

무대 뒤에서

하이버네이트는 투명한 영속성(transparent persistence) 메커니즘을 제공하는 비침투적인(nonintrusive) 프레임워크다. 투명한 영속성이라는 의미는 도메인 레이어 객체들이 하부의 데이터 저장소와 영속성 메커니즘에 대해 알지 못한다는 것을 의미한다. 도메인 객체는 영속 프레임워크의 특정 인터페이스를 구현하거나 클래스를 상속받지 않더라도 데이터 저장소에 저장되거나 조회될 수 있다. 영속성과 관련된 모든 관심사는 도메인 객체로부터 격리된 채 하부의 프레임워크에서 제공하는 관리자 객체에 의해 투명하게 처리된다. 하이버네이트의 경우 Session 이 관리자 객체의 기능을 제공한다.

Session 은 "영속성 컨텍스트(persistence context)"를 포함한다. 영속성 컨텍스트의 개념을 이해하기 위해서는 퍼즐의 또 다른 조각인 "작업 단위(unit of work)"를 이해해야 한다. 작업 단위란 원자적으로 처리되어야 하는 오퍼레이션의 집합을 말한다. 일반적으로 하나의 작업 단위는 하나의 영속성 컨텍스트와 연결되며 작업 단위 동안 수정된 객체의 모든 상태는 영속성 컨텍스트 내에 저장된 후 작업 단위가 종료될 때 데이터 저장소에 동기화 된다.

따라서 하이버네이트 Session 을 하나의 작업 단위 동안 생성 및 조회, 수정, 삭제되는 객체의 상태를 보관하는 일종의 캐시로 간주해도 무방하다(실제로 하이버네이트 Session 을 "1 차 캐시"라고 부르기도 한다).

앞에서 살펴 본 AlbumService 의 경우 findAlbum() 메소드가 실행될 때 Spring @Transactional 어노테이션에 의해 자동으로 Session 이 열리고(따라서 영속성 컨텍스트가 생성되고) 새로운 작업 단위가 시작된다.

AlbumServiceImpl.java

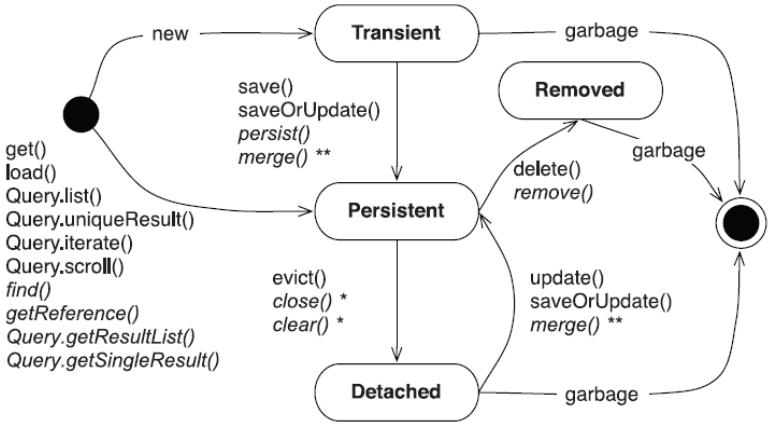
```
@Override
@Transactional
public Album findAlbum(Long albumId) {
  Album result = albumRepository.findAlbum(albumId);
  result.increaseViewCount();
  return result;
}
```

AlbumRepository 를 통해 조회된 Album ENTITY 는 영속성 컨텍스트에 캐시되고 작업 단위가 종료될 때까지 해당 ENTITY 에 대한 변경 사항이 추적된다. Album 의 increaseViewCount() 메소드를 호출해서 조회 카운트를 증가시키면 영속성 컨텍스트는 Album ENTITY 가 로드된 이후 상태가 변경되었음을 자동으로 감지한다. 이처럼 작업 단위 내에서 영속성 컨텍스트에 의해 관리되는 Persistent 객체의 상태 변경을 자동으로 감지하는 것을 "변경 감지(dirty checking)"라고 한다.

findAlbum() 메소드의 실행이 종료될 때 트랜잭션이 commit 되며 하이버네이트는 영속성 컨텍스트에 캐시되어 있던 Album ENTITY 의 변경사항을 데이터베이스에 반영하기 위해 자동으로 SQL UPDATE 문을 실행한다. 이처럼 영속성 컨텍스트에 캐시된 객체의 변경 사항을 데이터베이스로 반영하는 것을 '플러시한다(flushing)'고 말한다.

모든 데이터베이스와의 상호작용이 투명하게 이루어짐에 주목하라. Album ENTITY 는 데이터 저장소의 존재를 모를 뿐만 아니라 자신이 영속성 컨텍스트에 캐시되었다는 사실조차 알지 못한다. 하이버네이트를 사용하면 비침투적이고 투명한 방식으로 영속성 메커니즘으로부터 도메인 레이어를 고립시킬 수 있다.

하이버네이트에서 관리되는 객체는 Transient, Persistence, Detached, Removed 의 4 가지 상태를 가진다. 데이터베이스 식별자(주키)를 가지며 작업 단위 내에서 영속성 의해 관리되는 ENTITY 의 상태를 Persistence 상태라고 한다. Session.close() 메소드 호출로 인해 작업 단위가 끝나고 영속성 컨텍스트가 닫히면 Persistence 상태에 있던 객체들은 데이터베이스와 동기화된 후 영속성 컨텍스트로부터 분리된다. 이처럼 데이터베이스 식별자를 가지지만 영속성 컨텍스트로부터 분리되어 더 이상 데이터베이스와의 동기화가 보장되지 않는 ENTITY 의 상태를 Detached 상태라고 부른다.



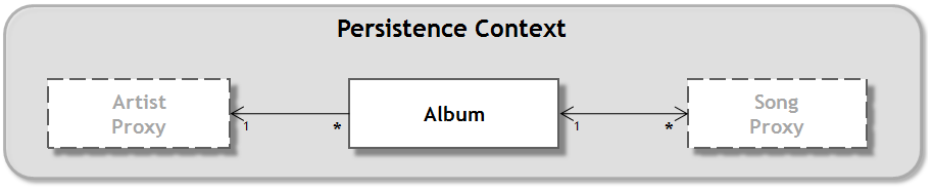
<그림 7> 영속성 관리자의 오퍼레이션에 의한 ENTITY의 상태 전이
[출처: 하이버네이트 완벽 가이드]

Session.close() 메소드는 Session 을 닫고 영속성 컨텍스트를 포함한 모든 자원을 반환하며, 관리하에 있는 모든 영속 인스턴스를 Detached 상태로 변경시킨다. 하이버네이트는 Detached 상태의 객체에 대해서는 변경사항을 추적하지 않으며, 따라서 데이터베이스와의 동기화 또한 수행하지 않는다.

findAlbum() 메소드에서 AlbumRepository 를 통해 조회된 Album 인스턴스는 Persistence 상태를 가진다. Album 이 Persistence 상태에 있기 때문에 Album 의 조회 카운트 증가와 같은 모든 상태 변경은 하이버네이트에 의해 모니터링되고 추적된다. 마지막으로 findAlbum()이 종료될 때 하이버네이트는 영속성 컨텍스트 내에서 관리하고 있는 Album 의 상태 변경 여부를 확인한 후 적절한 UPDATE 문을 실행하여 ENTITY 와 데이터베이스의 상태를 동기화한다. 작업 단위가 종료되고 영속성 컨텍스트가 닫힌 후에 Album 인스턴스의 상태는 Detached 상태가 되며 더 이상 데이터베이스와의 동기화가 보장되지 않는다.

기본적으로 하이버네이트는 모든 ENTITY 와 컬렉션에 대해 "지연 페치(Lazy Fetch)" 전략을 적용한다. 지연 페치 전략이란 기본적으로 쿼리 수행과 관련된 객체만 로드하고 해당 객체와 연관 관계를 맺고 있는 객체나 컬렉션은 로드하지 않는다는 것을 의미한다. 하이버네이트는 모든 연관 관계와 컬렉션에 대해 프록시(Proxy)를 생성하며 가능하면 실제 객체 대신 프록시를 반환하여 불필요한 데이터베이스 접근을 최소화하려고 노력한다. 따라서 연관 관계에 대한 설정을 변경하지 않는 이상 속성으로 포함된 모든 객체와 컬렉션은 실제 객체처럼 위장한 프록시다.

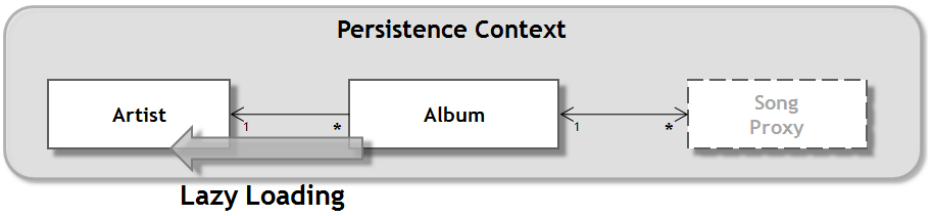
앞에서 살펴본 예제 애플리케이션의 경우 하이버네이트는 Album 인스턴스만을 직접 데이터베이스로부터 로드하고 Album 에 속한 artist 와 songs 속성에 대해서는 프록시 객체를 생성한다.



<그림 8> 데이터베이스로부터 로드된 Album 인스턴스

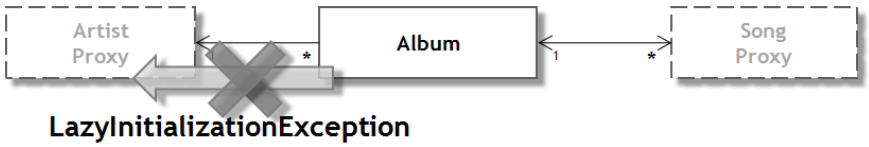
프록시는 `getId()`와 같은 식별자 접근 메소드를 제외한 메소드를 호출할 경우에 초기화된다(직접 필드 접근으로 식별자 프로퍼티를 매핑한 경우에는 식별자 접근 메소드 호출 시에도 프록시가 초기화된다). 이처럼 연관된 객체를 한꺼번에 로딩하지 않고 실제로 사용되는 시점에 로딩하는 것을 "지연 로딩(Lazy Loading)"이라고 한다. 따라서 하이버네이트는 `album.getArtist().getName()`나 `album.getSongs().size()`가 호출될 때 데이터베이스에 접근하여 해당 프록시 객체를 로딩하게 된다.

문제는 하이버네이트가 프록시 객체를 초기화하기 위해서는 해당 객체가 영속성 컨텍스트에서 관리되는 Persistence 상태여야 한다는 점이다. 따라서 Album 의 artist 와 songs 프록시를 정상적으로 초기화하기 위해서는 Session 이 열려 있는 상태에서 해당 객체의 메소드를 호출해야 한다.



<그림 9> Persistent 상태의 프록시 접근을 통한 지연 로딩

그러나 앞의 예제 애플리케이션에서 뷰 렌더링을 위해 artist 와 songs 프록시 객체에 접근하는 시점은 이미 Session 이 닫힌 후이며 따라서 Album 의 상태는 Persistent 상태에서 Detached 상태로 전이된 후이다. 이처럼 Detached 상태의 객체에 포함된 프록시 객체에 접근할 경우 하이버네이트는 프록시 객체를 지연 로딩할 수 없기 때문에 LazyInitializationException 예외를 던진다.



<그림 10> Detached 상태의 프록시 접근 시 LazyInitializationException 발생

Detached 상태의 프록시 초기화 문제를 해결하는 방법에는 크게 두 가지가 있다. 첫 번째 방법은 필요한 모든 연관 관계와 컬렉션을 뷰로 보내기 전에 애플리케이션 레이어에서 완전히

초기화시키는 것이다. 두 번째 방법은 영속성 컨텍스트를 뷰 렌더링이 끝나는 시점까지 개방한 상태로 유지하는 것이다. 두 번째 방법을 흔히 OPEN SESSION IN VIEW 패턴이라고 부른다.

지연 로딩 문제를 해결하기 위한 메커니즘을 이해하기 위해 먼저 영속성 컨텍스트와 트랜잭션, 그리고 JDBC 커넥션 간의 관계에 관해 살펴 보도록 하자.

영속성 컨텍스트, 트랜잭션, 그리고 커넥션

하이버네이트 Session 은 "트랜잭션 단위의 쓰기 지연(transactional write-behind)" 메커니즘을 제공한다. 하이버네이트는 영속성 컨텍스트에 의해 관리되는 객체의 변경 사항을 바로 데이터베이스로 동기화하지 않는다. 대신 트랜잭션을 커밋할 때 그 때까지의 모든 변경 사항을 합쳐서 데이터베이스로 플러시한다. 이를 통해 데이터베이스로 전송되는 쿼리의 수를 줄일 수 있으며 트랜잭션에 의해 데이터베이스에 락이 걸리는 시간을 최소화할 수 있다.

하이버네이트는 데이터베이스와 상호작용하는 모든 연산이 트랜잭션 내에서 실행될 것을 요구한다. 즉, 데이터베이스를 조회하거나 삽입, 수정, 삭제하는 모든 작업 전에 반드시 트랜잭션이 시작되어야 한다.

EJB 컨테이너를 사용하지 않는 비관리(non-managed) 환경에서 하이버네이트를 사용하는 경우 프로그래밍적인 방식으로 트랜잭션 경계를 선언해야 한다. 다음은 하이버네이트 레퍼런스 문서[HIBERNATE]에서 발췌한 것으로 비관리 환경에서 프로그래밍적으로 트랜잭션 경계를 설정하는 일반적인 방법을 보여준다.

비관리 환경에서의 트랜잭션 경계 설정

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();
    // do some work
    ...
    tx.commit();
} catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
} finally {
    sess.close();
}
```

프로그래밍적으로 트랜잭션 경계를 정의할 경우 도메인과 무관한 기술적인 관심사가 애플리케이션 레이어나 도메인 레이어에 침투하게 되며 트랜잭션 경계를 정의하는 코드가 중복되어 나타나는 문제점이 있다. 최근의 프레임워크는 관점-지향 프로그래밍(AOP, Aspect-Oriented Programming) 방식을 통해 트랜잭션 경계를 선언적으로 정의할 수 있는 다양한 방법을 제공한다.

Spring 프레임워크 역시 프록시 기반의 AOP 메커니즘을 통해 선언적으로 트랜잭션 경계를 정의할 수 있는 방법을 제공한다. Spring 프레임워크를 하이버네이트와 함께 사용하는 경우 선언적으로 트랜잭션 경계를 정의하는 가장 간단한 방법은 @Transactional 어노테이션을 사용하는 것이다. @Transactional 어노테이션이 명시된 메소드를 시작할 때 Spring 프레임워크는 자동으로 Session 을 오픈하고 트랜잭션을 시작하며, 메소드 종료 시에 자동으로 트랜잭션을 커밋(예외 발생 시에는 롤백)하고 Session 을 닫는다. @Transactional 어노테이션이 명시된 메소드가 종료될 때 트랜잭션이 커밋되기 때문에 메소드가 실행될 동안 영속성 컨텍스트에 캐시되어 있던 객체들의 모든 변경 사항이 데이터베이스로 플러시되며, Session 이 닫히기 때문에 플러시된 모든 객체들의 상태가 Detached 상태로 변경된다.

다시 한 번 AlbumService 의 구현 코드를 살펴 보자.

AlbumServiceImpl.java

```
@Override
@Transactional
public Album findAlbum(Long albumId) {
  Album result = albumRepository.findAlbum(albumId);
  result.increaseViewCount();
  return result;
}
```

이번에는 findAlbum() 메소드에 선언된 @Transaction 어노테이션에 주목하자. Spring 프레임워크는 AOP 프록시를 이용해 findAlbum() 메소드의 실행을 가로챈 후 메소드 실행 직전에 Session 을 열고 트랜잭션을 시작한다. findAlbum() 메소드 내에서 로드 된 result 객체는 Persistent 상태가 되어 영속성 컨텍스트에 캐시되며 '변경 감지(dirty checking)' 메커니즘에 의해 자동으로 상태 변경이 추적된다. result 의 조회 수를 증가시키고 메소드가 종료되면 Spring 프레임워크는 트랜잭션을 커밋하고 영속성 컨텍스트의 변경 내용을 데이터베이스로 플러시한 후 Session 을 닫는다. Session 이 닫힌 후 result 객체는 프록시가 초기화되지 않은 상에서 Detached 상태로 전이되기 때문에 뷰에서 렌더링될 때 LazyInitializationException 예외가 발생하게 된다.

이제 @Transactional 어노테이션과 REPOSITORY 와의 관계에 관해 살펴 보도록 하자. @Transactional 어노테이션을 사용하면 SERVICE 안에서 직접 Session 을 오픈하고 트랜잭션을 시작하거나 종료하는 코드를 추가하지 않더라도 자동으로 Session 과 트랜잭션을 관리할 수 있다. 문제는 @Transactional 어노테이션을 사용해서 오픈한 Session 을 REPOSITORY 에서도 접근할 수 있어야 한다는 것이다. 예제 프로그램의 경우 AlbumService 실행 전에 Spring 프레임워크가 오픈한 Session 을 AlbumRepository 에서도 사용할 수 있어야 한다. 어떻게 AlbumRepository 는 명시적으로 전달되지 않은 Session 을 사용해서 Album 을 로드할 수 있을까?

해답은 하이버네이트의 Sessoin 자동 전파(automatic propagation) 기능에 있다. 하이베네이트는 작업 단위에 참가하는 여러 클래스 간에 명시적으로 Session 을 전달하지 않아도 Session 을 공유할 수 있는 방법을 제공한다. 기본적인 아이디어는 모든 클래스가 참조할 수 있는 "현재 세션(current session)"을 제공하는 것이다. 트랜잭션에 참여하는 클래스는 SessionFactory.getCurrentSession() 메소드를 통해 현재 세션에 접근할 수 있다.

현재 세션 공유 방식은 hibernate.current_session_context_class 프로퍼티의 값을 설정해서 변경할 수 있다. 기본값은 thread 로 하이버네이트는 현재 세션을 실행 중인 스레드와 연결하며, 동일한 스레드 내에서 실행되는 모든 객체들은 SessionFactory.getCurrentSession() 메소드를 통해 동일한 Session 을 획득할 수 있다. 이처럼 현재 Session 을 스레드에 연결하는 방식을 "ThreadLocal Session" 패턴이라고 하며 하나의 HTTP 요청을 하나의 Session 을 통해 처리하는 일반적인 웹 애플리케이션 환경에 유용하다. 이처럼 하나의 Session 을 통해 개별 요청을 처리하는 방식을 "요청 당 세션(session per request)" 패턴이라고 한다.

AlbumRepository 코드를 다시 살펴 보면 Album 을 로드하기 위해 SessionFactory.getCurrentSession() 메소드를 호출하여 스레드에 연결된 현재 Session 을 얻어 오고 있음을 알 수 있다.

HibernateAlbumRepository.java

```
@Repository
public class HibernateAlbumRepository implements AlbumRepository {
  @Autowired
  private SessionFactory sessionFactory;

  @Override
  @Transactional(readOnly=true)
  public Album findAlbum(Long albumId) {
    return (Album)sessionFactory.getCurrentSession()
            .get(Album.class, albumId);
  }
}
```

앞에서 트랜잭션이 커밋될 때 Session 의 모든 변경사항을 데이터베이스로 플러시한다고 했지만 영속성 컨텍스트의 플러시 시점 역시 커스터마이징이 가능하다.

기본적으로 하이버네이트는 다음과 같은 경우에 Session 을 데이터베이스로 플러시한다.

- 하이버네이트 트랜잭션이 커밋되는 경우
- 쿼리를 실행 전 영속성 컨텍스트의 상태가 쿼리 결과에 영향을 미친다고 판단되는 경우
- session.flush()를 명시적으로 호출하는 경우

Session.setFlushMode() 메소드를 사용하면 하이버네이트의 기본 플러시 규칙을 변경할 수 있다. 하이버네이트는 Session 에 설정할 수 있는 플러시 모드로 다음의 4 가지를 제공한다. 기본값은 FlushMode.AUTO 로 위에서 설명한 3 가지 경우에 영속성 컨텍스트를 플러시한다.

- FlushMode.ALWAYS – 모든 쿼리 실행 전에 영속성 컨텍스트를 플러시한다.
- FlushMode.AUTO - 기본 플러시 모드로 위에서 설명한 3가지 경우에 영속성 컨텍스트를 플러시한다.
- FlushMode.COMMIT – 쿼리 실행 전에는 플러시하지 않으며 하이버네이트 트랜잭션이 커밋되거나 Session.flush()를 직접 호출할 경우에만 영속성 컨텍스트를 플러시한다.
- FlushMode.MANUAL - 명시적으로 flush()를 호출할 때만 영속성 컨텍스트를 플러시한다. 쿼리 실행 전과 하이버네이트 트랜잭션이 커밋되더라도 영속성 컨텍스트는 플러시되지 않는다.

예제 프로그램의 경우 기본값인 FlushMode.AUTO 를 사용하고 있으므로 AlbumService 의 findAlbum() 메소드가 종료되고 Spring 에 의해 트랜잭션이 커밋되는 순간 자동으로 영속성 컨텍스트의 모든 내용이 데이터베이스로 플러시된다.

마지막 이슈는 데이터베이스 커넥션과 관련된 것이다. 하이버네이트 Session 은 실제로 필요한 시점이 되기 전까지는 커넥션 풀로부터 JDBC 커넥션을 얻어오지 않는다. Session 이 커넥션을 얻어오는 시점은 Transaction.beginTransaction() 메소드를 실행하여 트랜잭션을 시작하는 시점이다. 트랜잭션이 시작되면 비로서 커넥션 풀로부터 JDBC 커넥션을 얻어오며 이 시점에서야 Session 이 데이터베이스와 연결된다.

앞에서 설명한 바와 같이 기본적으로 트랜잭션이 커밋될 때 영속성 컨텍스트가 플러시된다. 트랜잭션이 커밋되면 플러시와 함께 JDBC 커넥션 역시 커넥션 풀로 반환된다. 즉, 트랜잭션이 커밋되면 영속성 컨텍스트 플러시와 JDBC 커넥션 반환이 동시에 일어나는 것이다. 따라서 트랜잭션이 완료되는 시점에 Session 과 데이터베이스와의 연결이 끊어지게 된다.

하이버네이트는 커넥션의 반환 시점 역시 커스터마이징할 수 있도록 확장 포인트를 제공한다. 선택 가능한 설정값은 org.hibernate.ConnectionReleaseMode 에 정의되어 있다.

- ON_CLOSE – Session이 닫힐 때 JDBC 커넥션을 반환한다.
- AFTER_TRANSACTION - 트랜잭션이 완료된 후에 JDBC 커넥션을 반환한다.
- AFTER_STATEMENT – 모든 SQL문 실행 후에 JDBC 커넥션을 반환한다.

이 값은 hibernate.connection.release_mode 프로퍼티로 설정 가능하며 기본 값은 "auto"로 JDBC 를 사용하는 비관리형 환경의 경우 트랜잭션이 종료되는 시점에 JDBC 커넥션을 반환하는 AFTER_TRANSACTION 이 기본값이다. 트랜잭션이 종료된 후 Session.close() 메소드를 호출하지 않고 다시 트랜잭션을 시작하면 하이버네이트는 커넥션 풀로부터 새로운 JDBC 커넥션을 얻어 온다. 기본 설정 하에서 JDBC 커넥션은 트랜잭션이 완료되는 시점에

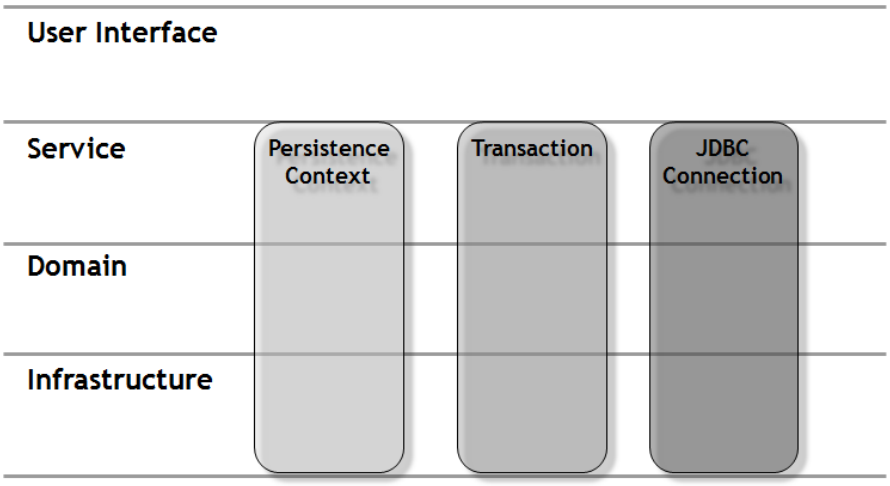
반환되며 영속성 컨텍스트를 포함하는 그 외의 자원들은 Session.close() 메소드를 호출하여 Session 을 닫은 후에 반환된다.

Persistent 상태의 객체들이 Detahced 상태로 전이되는 것은 Session.close() 메소드가 호출되는 시점이라는 점을 기억하자. 트랜잭션이 완료되고 JDBC 커넥션이 반환된 후라고 하더라도 Session.close()가 호출되지 않았다면 여전히 영속성 컨텍스트는 존재하며 캐시된 객체들은 Persistent 상태를 유지한다. 이 이슈는 Spring 프레임워크에서 사용되는 변형된 형식의 OPEN SESSION IN VIEW 필터의 단점을 이해하는데 있어 매우 중요하다.

이제 최종적으로 AlbumService 의 findAlbum()을 정리해 보자.

findAlbum() 메소드가 실행되면 Spring 프레임워크는 findAlbum() 메소드에 선언되어 있는 @Transactional 어노테이션 정의에 따라 하이버네이트 Session 을 열고 트랜잭션을 시작한다. 하이버네이트는 트랜잭션 시작 시에 JDBC 커넥션을 얻어와 Session 과 데이터베이스를 연결한다.

'현재 세션'을 전파하는 기본 방식은 현재의 스레드와 Session 을 연결하는 것이므로(hibernate.current_session_context_class 프로퍼티의 기본값이 thread 이므로) 하이버네이트는 현재 스레드에 새로 오픈된 Session 을 바인딩한다. Session 은 트랜잭션 시작 시에 풀로부터 얻은 JDBC 커넥션을 가지고 있으므로 동일한 스레드 내에서 SessionFactory.getCurrentSession() 메소드를 통한 모든 접근은 동일한 JDBC 커넥션을 사용하게 된다.



<그림 10> 일반적인 경우의 영속성 컨텍스트, 트랜잭션, 커넥션의 범위

findAlbum() 메소드 호출이 완료되면 Spring 프레임워크는 트랜잭션을 커밋하거나 롤백한다. 트랜잭션이 커밋될 때 하이버네이트는 영속성 컨텍스트를 플러시해서 데이터베이스와 동기화하고(FlushMode.AUTO 기본 설정에 의해) JDBC 커넥션을 반환한다(hibernate.connection.release_mode 기본설정에 의해). 최종적으로 Spring

프레임워크는 Session.close()를 호출하여 영속성 컨텍스트를 해제하며, 이 시점에 로드된 Album 인스턴스는 Persistent 상태에서 Detached 상태로 전이된 채 뷰로 전달된다.

지금까지 OPEN SESSION IN VIEW 패턴을 이해하기 위해 필요한 하이버네이트의 기본 메커니즘에 관해 살펴보았다. 이제 본론으로 돌아가 LazyInitializationException 예외를 방지할 수 있는 방법에 관해 살펴 보도록 하자. 먼저 뷰 렌더링에 필요한 객체 그래프를 모두 로드하도록 AlbumRepository.findAlbum() 메소드를 수정해서 문제를 해결하는 방법을 살펴본 후, 현재의 findAlbum() 구현을 그대로 유지한 채 문제를 해결할 수 있는 두 가지 패턴인 POJO FAÇADE 패턴과 OPEN SESSION IN VIEW 패턴에 관해 살펴보도록 하자.

뷰 렌더링에 필요한 객체 그래프 로드

LazyInitializationException 예외가 발생하는 이유는 AlbumRepository 의 findAlbum() 메소드에서 Album ENTITY 만을 로드하고 뷰 렌더링에 필요한 Artist 와 Song 의 목록은 로드하지 않기 때문이다. 따라서 문제를 해결할 수 있는 가장 간단한 방법은 Album 로드 시에 Artist 와 Song 도 함께 로드하는 것이다. Album 의 매핑 메타 데이터를 수정하지 않고 객체 그래프를 로드하는 방법은 HQL(Hibernate Query Language)를 사용하여 Album, Artist, Song 을 조인 페치(join fetch)로 한 번에 로드하는 것이다.

HibernateAlbumRepository.java

```
@Repository
public class HibernateAlbumRepository implements AlbumRepository {
  ......
  @Override
  public Album findAlbum(Long albumId) {
    return (Album)sessionFactory.getCurrentSession()
                .createQuery(
                   "from Album album " +
                   "left join fetch album.artist " +
                   "left join fetch album.songs " +
                   "where album.id = :id")
                .setParameter("id", albumId)
                .uniqueResult();
  }
}
```

이와 같이 REPOSITORY 에서 필요한 데이터를 한 번에 로드하는 방법은 부가적인 처리 없이도 LazyInitializationException 예외를 방지할 수 있지만 다음과 같은 단점이 존재한다.

- **REPOSITORY 의 재사용성 감소** – findAlbum()은 Album 과 Artist, Song 을 모두 로드하기 때문에 Album 만 필요한 SERVICE 를 구현하는 경우 해당 오퍼레이션을 호출하지 못 한다. 결국 findAlbum() 이외에 Album 만 로드하는 오퍼레이션이 추가될 것이므로 findAlbum()이 여러 문맥에서 재사용될 가능성은 줄어든다. 반면에

findAlbum()이 Album 만을 로드하는 원래의 구현의 경우 다양한 문맥에서 지연 로딩을 이용해 자신이 필요한 객체 그래프를 로드할 수 있으므로 재사용성이 증가한다.

- **REPOSITORY 복잡도 증가** – AlbumRepository 가 경우에 따라 서로 다른 객체 그래프를 반환할 경우 REPOSITORY 에 포함되는 오퍼레이션의 수가 늘어난다. 이것은 REPOSITORY 의 구현 복잡도를 증가시킬 뿐만 아니라 REPOSITORY 를 사용하는 클라이언트의 사용 복잡도 역시 증가시킨다.
- **뷰와 영속성 관심사의 강한 결합** – REPOSITORY 가 뷰가 필요로 하는 객체 그래프를 로드할 경우 영속성 관심사와 뷰 관심사가 강하게 결합된다. 뷰에서 렌더링할 데이터가 변경될 경우 REPOSITORY 에서 로드해야 하는 객체 그래프의 깊이가 달라지기 때문에 REPOSITORY 의 HQL 역시 변경해야 한다. 이것은 뷰의 관심사가 영속성 메커니즘으로 누수되었다는 것을 의미한다. "뷰를 변경하려면 반드시 REPOSITORY 를 변경해야 한다"라면 관심사의 분리 원칙을 위반하는 것이라고 생각해도 무방하다.

개인적으로 REPOSITORY 에서 뷰 렌더링에 필요한 모든 객체 그래프를 로드하는 방법은 관심사의 분리 원칙을 위반하기 때문에 선호하지 않는다. 객체 그래프를 조인 페치하는 것은 퍼포먼스 튜닝과 같이 특수한 문제를 해결하기 위한 제한적인 용도로만 사용해야 하며 일반적인 설계 이슈를 해결하기 위한 용도로 사용해서는 안 된다.

좋은 설계가 우선이며 성능은 그 다음이다.

POJO FACADE 패턴

현재의 findAlbum() 메소드의 구현을 유지하면서 LazyInitializationException 을 피하는 가장 간단한 방법은 영속성 컨텍스트가 열려 있는 동안 뷰 렌더링에 필요한 모든 프록시를 초기화하는 것이다. 앞의 예제에서 영속성 컨텍스트가 존재하게 될 작업 단위의 경계는 SERVICE 오퍼레이션에 의해 결정된다. 따라서 지연 로딩을 통해 프록시 객체를 초기화할 수 있는 가장 간단한 방법은 SERVICE 오퍼레이션 내부에서 프록시에 접근하는 것이다.

그러나 프록시를 초기화하는 로직을 SERVICE 에 포함할 경우 SERVICE 오퍼레이션이 뷰와 강하게 결합되므로 SERVICE 의 재사용성 및 캡슐화가 저해된다. 애플리케이션 레이어의 SERVICE 는 애플리케이션의 경계를 정의한다. SERVICE 오퍼레이션은 하나 이상의 뷰를 지원할 수 있어야 하며 리모트 클라이언트에 의한 원격 호출 역시 지원할 수 있어야 한다. SERVICE 를 뷰에 독립적인 상태로 유지하는 동시에 프록시의 초기화 문제를 해결할 수 있는 방법은 애플리케이션 레이어에 SERVICE 를 호출하는 새로운 객체를 추가하는 것이다. 이처럼 SERVICE 의 클라이언트로써 프록시를 초기화한 후 사용자 인터페이스로 반환하는 객체를 POJO FACADE 라고 한다[Richarson POJO].

AlbumFacade 인터페이스는 AlbumService 와 시그니처가 동일한 findAlbum() 오퍼레이션을 제공한다.

AlbumFacade.java

```java
public interface AlbumFacade {
  Album findAlbum(Long albumId);
}
```

AlbumFacade 는 AlbumService 대신 오퍼레이션에 대한 트랜잭션 경계가 된다. 따라서 AlbumFacade 의 오퍼레이션이 실행되는 동안 영속성 컨텍스트가 유지되므로 지연 로딩을 통해 필요한 프록시 객체들을 초기화할 수 있다. AlbumFacadeResultFactory 는 Album 의 프록시를 초기화할 수 있는 오퍼레이션을 제공한다.

AlbumFacadeResultFactory.java

```java
public interface AlbumFacadeResultFactory {
  Album make(Album album);
}
```

AlbumFacadeResultFactoryImpl 구현 클래스는 HibernateTemplate.initialize()를 사용하여 프록시를 초기화한다. HibernateTemplate.initialize()는 Hibernate.initialize()를 호출하며, HibernateException 이 발생할 경우 Spring 프레임워크의 DataAccessException 으로 변환해준다. AlbumFacadeResultFactoryImpl 은 HibernateTemplate 에 쉽게 접근하기 위해 SpringDaoSupprt 클래스를 상속받는다.

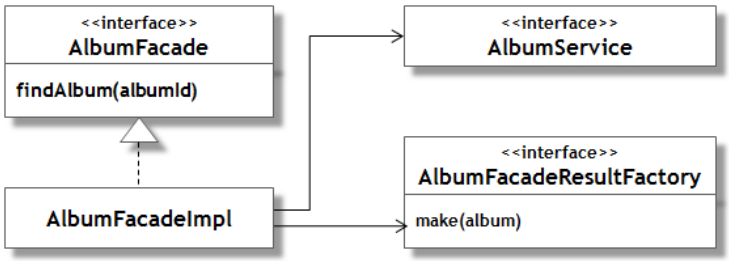
AlbumFacadeResultFactory.java

```java
@Component
public class HibernateAlbumFacadeResultFactory
              extends HibernateDaoSupport
              implements AlbumFacadeResultFactory {
  @Autowired
  private SessionFactory sessionFactory;

  @PostConstruct
  public void initialize() {
    setSessionFactory(sessionFactory);
  }

  @Override
  public Album make(Album album) {
    getHibernateTemplate().initialize(album.getArtist());
    getHibernateTemplate().initialize(album.getSongs());
    return album;
  }
}
```

AlbumFacadeImpl 구현 클래스는 트랜잭션 경계를 정의하며 AlbumService 를 호출해서 Album 을 조회한 후 AlbumFacadeResultFactory 를 사용하여 프록시를 초기화한다.

AlbumFacadeImpl.java

```java
@Service
public class AlbumFacadeImpl implements AlbumFacade {
  @Autowired
  private AlbumService albumService;

  @Autowired
  private AlbumFacadeResultFactory albumFacadeResultFactory;

  @Override
  @Transactional
  public Album findAlbum(Long albumId) {
    return albumFacadeResultFactory.make(albumService.findAlbum(albumId));
  }
}
```



<그림 11> POJO FACADE를 적용한 애플리케이션 레이어

POJO FAÇADE 패턴은 애플리케이션 레이어에 위치하며 데이터베이스에 대한 일관적이면서도 투명한 뷰를 제공할 수 있다는 장점을 제공하지만 뷰에서 렌더링할 모든 객체들을 지연 페치해야 하므로 에러가 발생하기 쉽고 개발이 복잡해 지는 담점이 있다. 또한 뷰에서 출력하는 데이터 포맷이 변경될 때마다 POJO FAÇADE 의 지연 페치 코드 역시 변경해야 하므로 유지보수가 어렵고 변경에 취약하다.

뷰에 필요한 객체 그래프를 REPOSITORY 에서 준비하는 방법이 뷰에 대한 관심사를 영속성과 관련된 관심사와 혼합하는 것이라면 POJO FAÇADE 패턴은 뷰에 대한 관심사를 애플리케이션 레이어의 흐름 관리와 관련된 관심사와 혼합하는 것이다. 비록 SERVICE 와 독립된 별도의 FAÇADE 에 프록시 초기화 로직을 위치시킨다고 해도 애플리케이션 레이어 개발 시에 렌더링될 뷰에 대한 존재와 렌더링과 관련된 요구사항을 고려해야 한다.

POJO FAÇADE 패턴의 가장 적절한 용도는 분산 환경에서 원격 통신을 지원하기 위한 REMOTE FAÇADE[Fowler PEAA]로 사용하는 것이다. REMOTE FACAE 는 원격 통신에 수반되는 라운드 트립 비용을 줄이기 위해 세밀한 단위(fine-grained)의 인터페이스 대신 큰 단위(coarse-grained)의 인터페이스를 제공하기 위해 사용한다. 물리적으로 단절되어 있는 원격 클라이언트의 경우 지연 로딩의 장점을 활용할 수 없기 때문에 REMOTE FAÇADE 는 원격 클라이언트가 원하는 모든 객체를 미리 로딩하여 전송해야 한다. 일반적으로 REMOTE FAÇADE 의 경우 도메인 객체를 전송하지 않고 원격 클라이언트가 필요로 하는 정보를 포함하는 추가적인 DTO(Data Transfer Object)[Fowler PEAA]를 전송한다.

따라서 분산 환경이 아닌 단일 JVM 상에서 뷰를 렌더링하기 위한 객체 그래프를 전달하는 경우에는 POJO FAÇADE 를 사용하는 것을 권하지 않는다.

POJO FAÇADE 패턴의 변형으로 EJB COMMAND 패턴[Marinescu EJB]을 사용할 수도 있다. EJB COMMAND 패턴은 COMMAND 패턴[GOF DP]의 변형으로 분산 환경에서 EJB Session Façade 의 복잡도를 낮추기 위해 고안된 패턴이다. 입력 파라미터와 리턴 값, 처리 로직을 함께 가지고 있는 COMMAND 객체를 원격으로 전송해서 처리한 후 COMMAND 객체를 다시 원격으로 반환하기 전에 뷰 렌더링에 필요한 모든 객체를 지연 페치하는 방법을 사용한다. EJB COMMAND 패턴에 대한 자세한 내용은 'EJB 디자인 패턴[Marinescu EJB]'을, EJB COMMAND 패턴에 지연 페치를 적용한 사례는 'Hibernate 완벽 가이드[Bauer JPAH]'를 참고하기 바란다.

POJO FAÇADE 패턴의 또 다른 변형은 관점-지향 프로그래밍(Aspect-Oriented Programming) 기법을 사용하여 뷰 렌더링에 필요한 객체 그래프를 지연 페치하는 애스펙트를 SERVICE 에 직조하는 것이다. 이와 관련해서는 스프링 포럼의 http://forum.springsource.org/showthread.php?15802-Alternative-to-open-session-in-view-pattern 을 참조하라.

OPEN SESSION IN VIEW 패턴

OPEN SESSION IN VIEW 패턴의 기본 아이디어는 단순하다. 뷰 렌더린 시점에 영속성 컨텍스트가 존재하지 않기 때문에 Detached 객체의 프록시를 초기화할 수 없다면 영속성 컨텍스트를 오픈된 채로 뷰 렌더링 시점까지 유지하자는 것이다. 즉, 작업 단위를 요청 시작 시점부터 뷰 렌더링 완료 시점까지로 확장하는 것이다.

OPEN SESSION IN VIEW 패턴을 구현하는 전통적인 방법은 서블릿 필터를 사용하는 것이다. 서블릿 필터를 사용하는 OPEN SESSION IN VIEW 패턴의 경우 요청이 도착하면 Session 을 오픈해서 새로운 영속성 컨텍스트를 준비하고 데이터베이스 트랜잭션을 시작한다. 컨트롤러가 실행을 마치고 뷰 렌더링이 완료되면 트랜잭션을 커밋하고 영속성 컨텍스트를 플러시한다. 만약 컨트롤러를 처리하거나 뷰를 렌더링하던 중에 예외가 발생한다면 트랜잭션을 롤백하고 Session 을 닫는다.

다음은 서블릿 필터를 사용하는 OPEN SESSION IN VIEW 패턴의 일반적인 구현 방법을 나타낸 것으로 "하이버네이트 완벽 가이드"에서 발췌한 것이다.

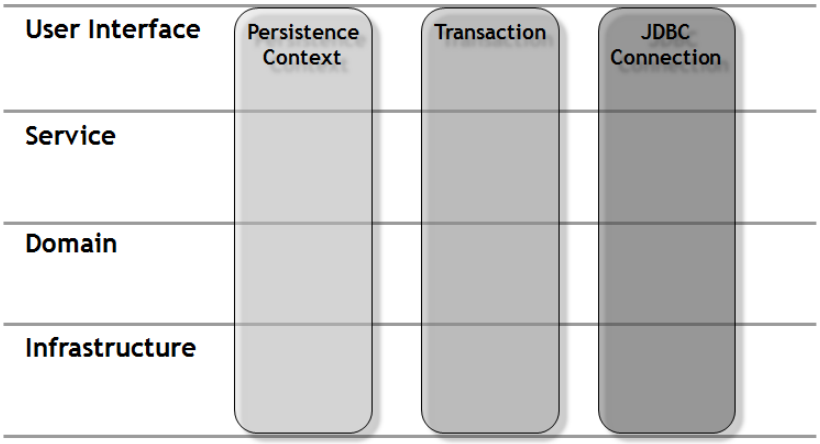
HibernateSessionRequestFilter.java

```
public class HibernateSessionRequestFilter implements Filter {
  private SessionFactory sessionFactory;

  public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
    try {
      // 데이터베이스 트랜잭션 시작
      sessionFactory.getCurrentSession().beginTransaction();
```

```
      // 다음 필터 호출(요청 처리 계속 진행)
      chain.doFilter(request, response);

      // 데이터베이스 트랜잭션 커밋
      sessionFactory.getCurrentSession().getTransaction().commit();
    } catch (Throwable ex) {

      // 무조건 롤백
      try {
        if (sessionFactory.getCurrentSession()
              .getTransaction().isActive()) {
          sessionFactory.getCurrentSession().getTransaction().rollback();
        }
      } catch (Throwable rbEx) {
        rbEx.printStackTrace();
      }

      // 다른 처리를 한다.
      throw new ServletException(ex);
    }
  }

  public void init(FilterConfig filterConfig) throws ServletException {
    sessionFactory = HibernateUtil.getSessionFactory();
  }

  public void destroy() {
  }
}
```

전통적인 OPEN SESSIO IN VIEW 패턴은 서블릿 필터 시작 시에 하이버네이트 Session 을 열고 트랜잭션을 시작한다. 이후 서블릿 필터는 컨트롤러에 요청을 위임하고 뷰 렌더링이 모두 완료된 후에 트랜잭션을 커밋한다(또는 롤백한다). 일반적으로 플러시 모드 기본값인 FlushMode.AUTO 를 사용하므로 영속성 컨텍스트에서 관리하고 있는 모든 Persistent 상태의 객체는 뷰의 렌더링이 모두 완료되고 서블릿 필터에서 트랜잭션을 커밋하는 순간 데이터베이스로 플러시된다. 또한 ConnectionReleaseMode 의 기본값인 AFTER_TRANSACTION 에 따라 JDBC 커넥션의 반환 역시 시점 역시 뷰가 모두 렌더링되고 서블릿 필터 내에서 트랜잭션이 커밋(또는 롤백)되는 시점에 이루어진다.

여기에서 중요한 것은 Session.close()가 호출되는 시점이다. 앞에서 설명한 바와 같이 하이버네이트는 Sesion.close()가 호출되면 모든 Persistent 상태의 객체를 Detached 상태로 변경하고 영속성 컨텍스트를 닫는다. OPEN SESSION IN VIEW 방식의 서블릿 필터에서는 모든 요청 처리가 완료되고 뷰 렌더링이 끝난 후에 Session.close()를 호출하므로 뷰가 렌더링 되는 시점에는 영속성 컨텍스트 내의 Persistent 객체에 접근하더라도 지연 로딩의 지원을 받을 수 있다. 따라서 OPEN SESSION IN VIEW 방식의 서블릿 필터를 사용하는 경우에는 애플리케이션 레이어에서 초기화하지 못한 프록시 객체에 접근하더라도 LazyInitializationException 예외가 발생하지 않는다.



<그림 12> 전통적인 방식에서의 영속성 컨텍스트, 트랜잭션, 커넥션의 범위

그러나 서블릿 필터 방식의 OPEN SESSION IN VIEW 패턴에는 다음과 같은 단점이 존재한다.

- **JDBC 커넥션 보유 시간 증가** – JDBC 커넥션은 뷰의 렌더링이 모두 완료된 후에야 커넥션 풀로 반환된다. 따라서 뷰의 렌더링 시간이 길어지면 길어질수록 개별 요청을 처리하기 위한 스레드가 JDBC 커넥션을 보유하는 시간이 길어진다.
- **모호한 트랜잭션 경계** – 일반적으로 애플리케이션의 트랜잭션 경계는 애플리케이션 레이어 SERVICE 를 경계로 한다. 즉, SERVICE 오퍼레이션이 호출 되기 전에 트랜잭션이 시작되고 SERVICE 오퍼레이션이 종료될 때 트랜잭션이 커밋되거나 롤백되는 것이 일반적이다. 이에 비해 OPEN SESSION IN VIEW 서블릿 필터의 트랜잭션 경계는 HTTP 요청 처리 시간의 거의 대부분을 아우른다. 결국 트랜잭션 경계에 대한 일관성 있는 뷰를 유지할 수 없으며 이로 인해 다양한 문제가 발생할 여지가 있다. 예를 들어 애플리케이션 레이어나 도메인 레이어가 아닌 사용자 인터페이스 레이어 컨트롤러에서 영속 객체를 변경하더라도 서블릿 필터에서 트랜잭션이 커밋되기 때문에 변경사항이 데이터베이스에 반영되게 된다. 이것은 사용자 인터페이스에서 객체를 변경하는 경우에는 데이터베이스로 플러시되지 않을 것이라는 일반적인 직관에 반하는 것이며 실제 프로젝트에서 다양한 이슈를 야기한다.

전통적인 서블릿 필터 방식의 가장 큰 단점은 사용자 인터페이스 레이어에서 트랜잭션 경계를 설정하는 것이다. 뷰 렌더링 시점의 지연 페치를 허용하면서도 일관성 있는 트랜잭션 경계를 유지하는 합리적인 절충안은 서블릿 필터에서 Session 을 오픈하되 트랜잭션 경계는 애플리케이션 레이어 범위로 한정하는 것이다. Spring 프레임워크에서는 FlushMode 와 ConnectionReleaseMode 의 조정을 통해 전통적인 OPEN SESSION IN VIEW 서블릿 필터의 단점을 보완한 OpenSessionInViewFliter 와 OpenSessionInViewInterceptor 를 제공한다. 두 클래스의 가장 큰 특징은 기존처럼 뷰에서 지연 로딩을 가능하게 하는 동시에 애플리케이션 레이어에 트랜잭션 경계를 선언할 수 있다는 점이다.

Spring 의 OpenSessionInViewFilter

Spring 프레임워크를 사용하고 있다면 직접 서블릿 필터를 개발할 필요가 없다. Spring 프레임워크는 추가적인 작업 없이도 OPEN SESSION IN VIEW 패턴의 장점을 취할 수 있는 org.springframework.orm.hibernate3.support.OpenSessionInViewFilter 클래스를 제공한다. 만약 웹 MVC 프레임워크로 SpringMVC 를 사용하고 있다면 인터셉터 방식의 org.springframework.orm.hibernate3.support.OpenSessionInViewInterceptor 를 사용할 것을 권장한다.

서블릿 필터에서 Session 을 오픈하고 트랜잭션을 시작하던 전통적인 방식의 OPEN SESSION IN VIEW 패턴과 달리 SpringMVC 에서 제공하는 OpenSessionInViewFilter 는 필터 내에서 Session 을 오픈하지만 트랜잭션은 시작하지 않는다. 따라서 서블릿 필터 안에서는 커넥션 풀로부터 JDBC 커넥션을 얻지 않는다(하이버네이트는 트랜잭션이 시작될 때 커넥션 풀로부터 JDBC 커넥션을 얻어 온다는 사실을 기억하자). 서블릿 필터 내에서 트랜잭션은 시작되지 않지만(따라서 JDBC 커넥션이 확보되지 않았으므로 Session 이 데이터베이스와 연결되지 않았지만) Session 은 열려 있기 때문에 영속 객체를 관리할 영속성 컨텍스트는 생성되어 있다는 사실에 주목하자.

서블릿 필터는 하이버네이트 Session 을 오픈한 후 Session 의 플러시 모드를 FlushMode.MANUAL 로 변경한다. FlushMode.MANUAL 로 설정할 경우 명시적으로 Session.flush()를 호출하지 않는 한 영속성 컨텍스트를 데이터베이스로 동기화하지 않는다. OpenSessionInViewFilter 자체는 요청이 완료된 시점에 명시적으로 Session.flush()를 호출하지 않으므로 요청 처리가 완료되는 시점에 영속성 컨텍스트에 존재하는 Persistent 상태의 객체들은 Session 이 닫힐 때 데이터베이스로 플러시되지 않는다.

필터는 Session 의 플러시 모드를 변경한 후에 Session 을 현재 스레드의 ThreadLocal 내에 저장한다. 이후 Spring 은 ThreadLocal 에 저장된 단일 Session 을 통해 모든 영속성 관련 처리를 수행한다.

영속성 컨텍스트를 통해 영속 객체를 관리할 준비를 마친 필터는 컨트롤러로 요청 처리를 위임한다. Spring 은 @Transactional 어노테이션이 명시된 메소드를 만나면 HibernateTransactionManager 를 통해 트랜잭션을 시작한다.

HibernateTransactionManager 는 ThreadLocal 에 저장된 Session 을 얻어와 서블릿 필터에서 FlushMode.MANUAL 로 설정했던 플러시 모드를 FlushMode.AUTO 로 변경한다. 이와 같이 Spring 프레임워크는 트랜잭션 시작 전에 플러시 모드를 변경함으로써 서블릿 필터에서 설정된 수동 플러시 모드를 트랜잭션이 커밋될 때 자동으로 플러시되도록 변경한다.

HibernateTransactionManager 는 플러시 모드를 변경한 후 하이버네이트 트랜잭션을 시작한다. 메소드 실행이 완료되면 HibernateTransactionManager 는 트랜잭션을 커밋(또는 롤백)하며 Session 의 플러시 모드가 FlushMode.AUTO 로 변경되었기 때문에 메소드 내에서 발생한 영속 객체에 대한 모든 변경 사항이 데이터베이스로 플러시된다.

HibernateTransactionManager 는 트랜잭션이 완료되면 플러시 모드를 다시 FlushMode.MANUAL 로 복구하여 이후의 수정 사항이 데이터베이스로 플러시되는 것을 방지한다(모든 수정 사항이 플러시되는 것을 막을 수는 없다. 이 이슈에 대해서는 뒤에서 좀 더 상세히 다루기로 한다).

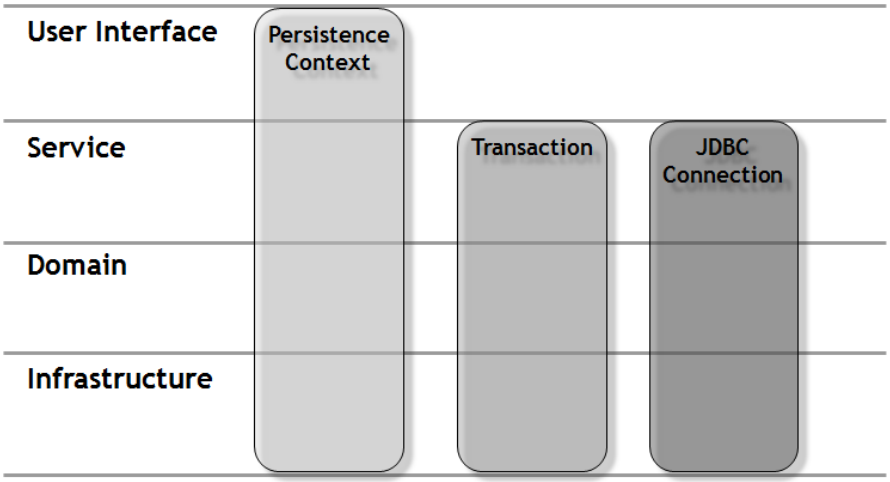
Spring 프레임워크를 사용할 경우 애플리케이션 컨텍스트에 LocalSessionFactory 를 선언하여 하이버네이트의 SessionFactory 인스턴스를 생성하게 된다. Spring 프레임워크의 LocalSessionFactory 는 ConnectionReleaseMode 의 값을 기본값인 AFTER_TRANSACTION 대신 ON_CLOSE 로 변경한다. 따라서 Spring 프레임워크의 지원 기능을 통해 하이버네이트를 사용하는 경우 JDBC 커넥션이 풀로 반환되는 시점은 Session.close()가 호출되는 시점이다. 따라서 OpenSessionInViewFilter 를 사용하는 경우 서블릿 필터가 완료되어야만 Session 이 닫히기 때문에 트랜잭션이 완료될 때 명시적으로 JDBC 커넥션을 풀로 반환해야 한다. 이를 위해 HibernateTransactionManager 는 트랜잭션이 완료된 후에 Session.disconnect() 메소드를 호출하여 강제로 Session 과 연결된 JDBC 커넥션을 커넥션 풀로 반환한다.

트랜잭션 완료 시점에 JDBC 커넥션은 커넥션 풀로 반환되어 Session 과 데이터베이스와의 연결이 단절되지만 Session.close()가 호출되지 않았으므로(Session.close()는 필터에서 호출된다) 영속성 컨텍스트는 그대로 존재한다. 따라서 트랜잭션 내에서 영속성 컨텍스트에 캐시된 모든 영속 객체는 Detached 상태로 전이되지 않으며 Persistent 상태를 유지하므로 뷰에서 초기화되지 않은 프록시에 접근하더라도 LazyInitializationException 예외가 발생하지 않고 정상적으로 지연 로딩 된다.

트랜잭션 완료 시에 JDBC 커넥션이 커넥션 풀로 반환되었음에도 불구하고 뷰에서 지연 로딩이 가능한 이유는 하이버네이트가 "트랜잭션 미적용 데이터 접근"을 허용하기 때문이다. 트랜잭션 미적용 데이터 접근이란 자동 커밋 모드(autocommit mode)를 사용해서 데이터에 접근하는 것을 의미한다. 자동 커밋 모드는 대화형 콘솔에서 SQL 문을 편하게 실행할 수 있도록 명시적인 트랜잭션 범위를 지정하지 않아도 내부적으로 개별 DML 문을 짧은 트랜잭션 내에서 실행할 수 있도록 해주는 모드를 말한다.

@Transactional 을 통해 명시적으로 트랜잭션이 지정되지 않은 곳에서 지연 로딩이 발생할 경우 하이버네이트는 내부적으로 트랜잭션 미적용 데이터 접근 방식으로 이를 처리한다. 즉, 프록시를 초기화할 때마다 커넥션 풀로부터 매번 새로운 JDBC 커넥션을 얻어와 암시적인 트랜잭션 내에서 지연 로딩을 처리한 후 JDBC 커넥션을 커넥션 풀로 반환한다.

마지막으로 뷰 렌더링이 완료되고 필터에서 Session.close()를 호출하면 모든 Persistent 상태의 객체들이 Detached 상태로 전이되며 영속성 컨텍스트가 닫히게 된다.



<그림 13> OpenSessionInViewFilter에서의 영속성 컨텍스트, 트랜잭션, 커넥션의 범위

OpenSessionInViewFilter 와 영속성 컨텍스트

Spring 에서 제공하는 OpenSessionInViewFilter 는 전체 요청 처리에 대해 트랜잭션 경계를 설정하는 전통적인 OPEN SESSION IN VIEW 필터 방식과 달리 애플리케이션 레이어 SERVICE 호출에 대해 트랜잭션 경계를 설정한다. 일반적으로 애플리케이션 레이어의 SERVICE 를 경계로 트랜잭션을 시작하고 종료하기 때문에 전통적인 방식에 비해 트랜잭션에 대한 투명하면서도 일관성 있는 뷰를 제공할 수 있다.

그러나 OpenSessionInViewFilte 는 트랜잭션 경계에 대한 일관성 있는 뷰는 보장하지만 영속성 컨텍스트에 대한 일관성 있는 뷰는 보장하지 않는다. Session 이 서블릿 필터에서 오픈된다는 것은 애플리케이션 레이어에 진입하기 전인 사용자 인터페이스 컨트롤러 실행 시점에 이미 영속성 컨텍스트가 활성화되어 있다는 것을 의미한다. 따라서 사용자 인터페이스 컨트롤러에서 영속 객체를 로드할 경우 해당 객체는 트랜잭션 경계 외부에서 로드되었음에도 불구하고 영속성 컨텍스트 내에 캐시된다.

이와 같은 방식의 문제점을 이해하기 위해 사용자 인터페이스 컨트롤러에서 영속 객체를 로드한 후 상태를 변경하는 케이스를 살펴 보자.

AlbumController.java

```
@Controller
public class AlbumController {
  @Autowired
  private AlbumService albumService;

  @Autowired
  private AlbumRepository albumRepository;

  @RequestMapping("/album")
  public String view(@RequestParam("albumId") long albumId, Model model) {
    Album controllerAlbum = albumRepository.findAlbum(albumId);
```

```
    controllerAlbum.addSong(new Song(3, "controllerAddedSong"));

    Album serviceAlbum = albumService.findAlbum(albumId);

    model.addAttribute("album", serviceAlbum);

    return "album";
  }
}
```

컨트롤러에서는 요청 파라미터로 전달된 albumId 에 해당하는 Album 을 로드하고 로드된 Album 에 새로운 Song 객체를 추가한다. AlbumService 의 findAlbum() 메소드는 앞에서 살펴본 것과 동일하며 파라미터로 전달된 albumId 에 해당하는 Album 객체를 다시 로드하고 조회수를 증가시킨 후 트랜잭션을 커밋해서 데이터베이스로 플러시한다.

AlbumServiceImpl.java

```
@Override
@Transactional
public Album findAlbum(Long albumId) {
  Album result = albumRepository.findAlbum(albumId);
  result.increaseViewCount();
  return result;
}
```

OpenSessionInViewFilter 를 사용하지 않는 일반적인 경우라면 AlbumController 에서 로드한 Album 과 AlbumService 에서 로드한 Album 은 서로 다른 객체여야 한다. 그러나 OpenSessionInViewFilter 를 사용하는 경우 AlbumController 에서 로드한 Album 객체와 AlbumService 에서 로드한 Album 객체는 동일한 객체이다. 즉, AlbumController 의 두 지역 변수인 controllerAlbum 과 findAlbum()에서 로드한 result 에 대해서 'controllerAlbum == result'가 된다.

앞서 설명한 바와 같이 Spring 의 OpenSessionInViewFilter 는 HTTP 요청을 처리하는 동안 하나의 Session 을 공유하며, 따라서 영속성 컨텍스트를 공유하게 된다. AlbumController 에서 albumId 에 해당하는 Album 객체를 로드하면 하이버네이트는 로드된 객체를 영속성 컨텍스트에 캐시한다. 이 후 AlbumService 에서 동일한 albumId 를 가진 Album 을 로드하려고 시도할 경우 하이버네이트는 데이터베이스에 접근하기 전에 우선 영속성 컨텍스트에 albumId 를 주키로 가진 객체가 캐시되어 있는 지 여부를 확인한다. 현재 AlbumController 에서 로드한 객체가 캐시되어 있으므로 하이버네이트는 SQL 문을 실행하지 않고 캐시에 저장되어 있는 Album 객체를 반환한다. 따라서 AlbumController 와 AlbumService 에서 로드한 Album 객체는 동일한 객체가 된다(하이버네이트는 이러한 영속성 컨텍스트 캐시를 통해 특별한 설정 없이도 반복 읽기(Repeatable Read) 레벨의 트랜잭션 격리 수준을 제공한다).

문제는 AlbumService 에 트랜잭션 경계가 설정되어 있기 때문에 메소드가 종료되어 트랜잭션이 커밋될 때 영속성 컨텍스트의 내용이 플러시된다는 것이다. findAlbum() 메소드를 호출한 원래의 의도는 albumId 에 해당하는 Album 을 로드한 후 조회수만을 증가시키는 것이었다. 그러나 AlbumController 에서 로드한 Album 과 AlbumService 에서 로드한 Album 이 동일한 객체이므로 트랜잭션 커밋 시 증가된 조회수뿐만 아니라 AlbumController 에서 추가된 Song 객체까지도 함께 데이터베이스에 플러시되게 된다.

여기에서의 문제는 AlbumController 와 AlbumService 간의 의존성이 암시적이라는 점이다. 만약 AlbumController 의 개발자와 AlbumService 의 개발자가 다른 사람이라면 동일한 Album 객체를 수정하고 있다는 사실을 알지 못할 수도 있다. OpenSessionInViewFilter 를 사용하지 않는 환경에서는 AlbumController 와 AlbumService 에서 로드된 객체는 서로 다른 트랜잭션 내에서 실행되기 때문에 값은 동일하더라도 객체 자체는 개별적으로 생성된다. 따라서 어느 한쪽에서 수정한 내용이 다른 쪽에 영향을 미치지 않는다. 그러나 OpenSessionInViewFilter 를 사용하는 경우에는 트랜잭션 경계를 넘어 동일한 영속성 컨텍스트를 공유하기 때문에 한쪽에서 로드한 객체를 수정할 경우 전혀 상관없는 영역에서 로드한 객체까지도 수정되는 문제가 발생한다. 이것은 OpenSessionInViewFilter 를 사용함으로써 동일한 객체에 대한 암시적인 별칭(aliasing)이 생성되었기 때문이다

컨트롤러에서 @Transactional 어노테이션이 붙은 SERVICE 의 메소드를 두 번 이상 호출할 경우에도 이와 유사한 문제가 발생한다. 다른 Album 에서 제목과 아티스트 정보만 복사해서 새로운 앨범을 만든 후 데이터베이스 저장하는 기능을 제공하는 copyAlbum() 메소드를 AlbumService 에 추가한다고 가정해 보자.

AlbumService.java

```
public interface AlbumService {
  ......
  void copyAlbum(Album source);
}
```

메소드에서는 파라미터로 전달된 Album 객체에서 제목과 아티스트 정보를 추출하고 이 정보를 이용해서 새로운 Album 인스턴스를 생성한 후 데이터베이스에 저장한다. copyAlubm() 메소드에는 새로운 Album 을 데이터베이스에 저장한 후 파라미터로 전달된 Album 의 조회수를 증가시킨다.

AlbumServiceImpl.java

```
@Service
public class AlbumServiceImpl implements AlbumService {
  ......

  @Override
  @Transactional
  public void copyAlbum(Album source) {
    Album result = new Album(source.getTitle(), source.getArtist());
```

```
    albumRepository.save(result);

    source.increaseViewCount();
  }
}
```

여기에서 트랜잭션 경계는 copyAlbum() 메소드가 된다. 따라서 파라미터로 전달된 Album 의 정보를 사용해서 새로 생성한 Transient 상태의 result 인스턴스는 AlbumRepository.save() 메소드 호출에 의해 Persistent 상태로 전이되며 영속성 컨텍스트에 추가된 후 트랜잭션이 커밋될 때 데이터베이스로 플러시된다. 파라미터로 전달된 Album 객체는 트랜잭션 경계 외부에서 전달된 객체이므로 copyAlbum() 메소드 내부에서 조회수를 증가시키는 increaseViewCount() 메소드를 호출하더라도 데이터베이스에 증가된 조회수가 반영되지 않을 것이다.

그러나 OpenSessionInViewFilter 를 사용하는 환경에서 다음과 같이 AlbumController 에서 copyAlbum()을 호출한다면 예상과는 다른 결과를 얻게 된다.

AlbumController.java

```
@Controller
public class AlbumController {
  @Autowired
  private AlbumService albumService;

  @RequestMapping("/album")
  public String view(@RequestParam("albumId") long albumId, Model model) {
    Album album = albumService.findAlbum(albumId);

    albumService.copyAlbum(album);

    model.addAttribute("album", album);

    return "album";
  }
}
```

AlbumController 는 AlbumService.findAlbum() 메소드를 통해 로드한 Album 객체를 copyAlbum() 메소드의 파라미터로 전달하고 있다. OpenSessionInViewFilter 를 사용하지 않는 환경일 경우 파라미터로 전달된 Album 객체는 영속성 컨텍스트와 무관한 Detached 상태이므로 copyAlbum() 메소드 내에서 조회수를 증가시키더라도 데이터베이스로 플러시 되지 않는다. 그러나 OpenSessionInViewFilter 를 사용하는 경우에는 파라미터로 전달된 Album 객체 가 동일한 영속성 컨텍스트에 의해 관리되는 Persistent 상태의 객체이므로 copyAlubm() 내에서 조회수를 증가시키는 경우 트랜잭션 커밋 시에 데이터베이스로 함께 플러시되게 된다.

따라서 OpenSessionInViewFilter 를 사용할 경우에는 영속성 컨텍스트 공유로 인해 발생하는 암시적인 별칭 문제를 염두에 두지 않는다면 연쇄적인 사이드 이펙트에 의해 이해하기 어렵고 애플리케이션의 상태를 예측하기 어렵게 될 수도 있다.

singleSession=false 설정

암시적인 별칭으로 인한 사이드 이펙트의 가장 큰 원인은 요청을 처리하는 동안 발생하는 모든 데이터베이스 접근이 동일한 영속성 컨텍스트를 공유하기 때문이다. 따라서 가장 간단한 해결방법은 각각의 트랜잭션 경계 단위로 독립적인 영속성 컨텍스트를 유지하도록 함으로써 트랜잭션 경계 간의 사이드 이펙트가 발생하지 않도록 하는 것이다.

Spring 의 OpenSessionInViewFilter 와 OpenSessionInViewInterceptor 의 singleSession 프로퍼티를 false 로 설정하면 OPEN SESSION IN VIEW 패턴을 사용하지 않는 경우와 동일하게 각각의 트랜잭션 별로 독립적인 Session 이 오픈된다. 이렇게 하면 트랜잭션 단위로 독립적인 영속성 컨텍스트가 생성되므로 별칭 문제로 인한 사이드 이펙트를 방지할 수 있다. singleSession=false 로 설정하더라도 뷰 렌더링 시점까지 각각의 Session 이 오픈된 상태를 유지해야 하므로 트랜잭션이 완료되더라도 Session 은 닫히지 않는다.

singleSession=false 의 경우 서블릿 필터 시작 시에 Session 을 오픈하지 않는다. 대신 SessionFactory 자체를 ThreadLocal 에 저장한 후 컨트롤러로 요청 처리를 위임한다.

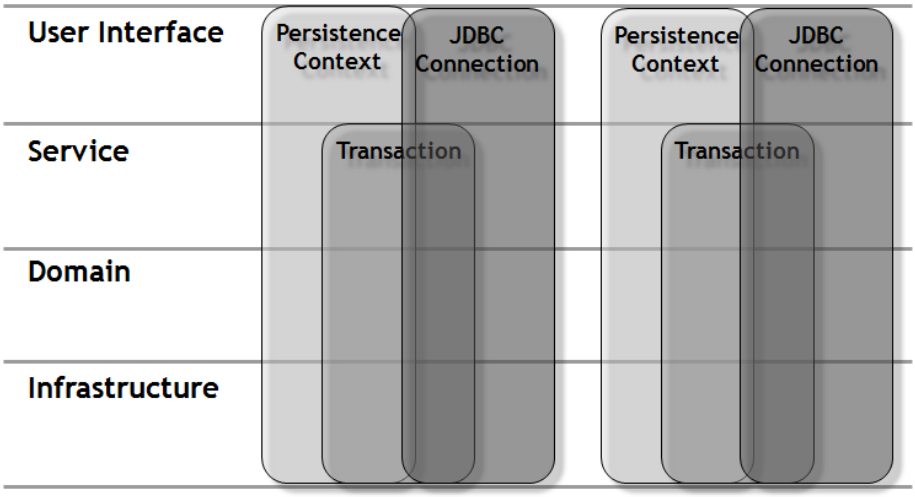
HibernateTransactionManager 는 @Transactional 어노테이션에 의해 새로운 트랜잭션이 시작될 때마다 ThreadLocal 에 저장되어 있는 SessionFactory 를 사용하여 새로운 Session 을 오픈한다. singleSession=true 인 경우에는 서블릿 필터에서 Session 을 오픈한 후 오픈된 Session 을 ThreadLocal 에 저장하여 모든 트랜잭션에서 전달된 Session 을 공유해서 사용하는데 비해 singleSession=false 인 경우에는 서블릿 필터에서 SessionFactory 자체를 ThreadLocal 에 저장하여 전달한 후 실제로 트랜잭션이 시작될 때마다 전달된 SessionFactory 를 사용하여 Session 을 오픈한다. 이 때 생성되는 Session 의 ConnectionReleaseMode 의 값은 ON_CLOSE 로, FlushMode 의 값은 AUTO 로 설정된다.

트랜잭션이 시작되면 Session 은 커넥션 풀로부터 JDBC 커넥션을 할당 받아 Session 과 데이터베이스를 연결한다. SERVICE 의 처리가 완료되고 트랜잭션이 커밋되면 Session 에 설정된 FlushMode.AUTO 설정에 의해 영속성 컨텍스트 내의 Persistent 객체들이 데이터베이스로 플러시된다. 그러나 singleSession=true 모드와 달리 JDBC 커넥션은 커넥션 풀로 반환되지 않는다. 따라서 Session 은 트랜잭션이 커밋되더라도 데이터베이스와의 연결 상태를 계속 유지한다. 이에 비해 singleSession=true 인 경우에는 ConnectionReleaseMode.ON_CLOSE 로 설정된 값을 무시하기 위해

HibernateTransactionManager 가 직접 Session.disconnect()를 호출하여 트랜잭션 커밋 시에 JDBC 커넥션을 반환한다.

이후 Spring 프레임워크는 트랜잭션이 시작될 때마다 별도의 Session 을 열어 트랜잭션 별로 독립적인 영속성 컨텍스트를 생성하며 각각의 Session 별로 별도의 JDBC 커넥션을 할당한다. 그리고 Session 별로 할당된 JDBC 커넥션은 뷰 렌더링이 완료되는 시점까지고 풀에 반환되지 않은 상태로 유지된다.

뷰 렌더링이 완료되면 서블릿 필터는 ThreadLocal 에 저장되어 있는 모든 열려있는 Session 들에 대해 close()를 호출한다. 이 시점에 각각의 Session 에 의해 생성된 영속성 컨텍스트가 제거되고 그와 동시에 Session 에 할당되어 있던 JDBC 커넥션이 커넥션 풀로 반환된다.



<그림 14> singleSession=false일 경우의 영속성 컨텍스트, 트랜잭션, 커넥션의 범위

지금까지 살펴 본 바와 같이 singleSession=false 인 경우 각 Session 별로 독립적인 JDBC 커넥션을 얻고 뷰 렌더링 시점까지 풀로 반환하지 않기 때문에 동시에 사용되는 JDBC 커넥션의 개수가 증가한다. singleSession=true 인 경우에는 트랜잭션이 커밋되거나 롤백되는 시점에 JDBC 커넥션을 곧장 반환하고 트랜잭션 외부에서는 자동 커밋 모드로 동작하기 때문에 singleSession=false 인 경우에 비해 상대적으로 사용 중인 전체 JDBC 커넥션의 수가 적고 커넥션을 보유하고 있는 시간 역시 짧다.

그러나 단일 Session 을 사용하는 경우의 문제점이었던 트랜잭션 경계의 일관성 문제와 영속성 컨텍스트 공유로 인한 사이드 이펙트 문제를 해결할 수 있다. 즉, singleSession=false 의 경우 뷰 렌더링 시점까지 Session 을 오픈된 상태로 유지할 수 있는 동시에 투명한 트랜잭션 경계와 영속성 컨텍스트 경계를 제공할 수 있는 것이다. 그러나 하나의 요청을 처리하기 위해 동시에 오픈되는 JDBC 커넥션의 수가 늘어나고 뷰 렌더링 이후까지도 커넥션 풀로 반환되지 않기 때문에 시스템의 가용성이 감소할 위험이 있다.

AlbumController 에서 Album 을 로드한 후 수정하고 동일한 Album 을 AlbumService 에서 로드하는 예제를 다시 살펴 보자.

AlbumController.java

```java
@Controller
public class AlbumController {
  @Autowired
  private AlbumService albumService;

  @Autowired
  private AlbumRepository albumRepository;

  @RequestMapping("/album")
  public String view(@RequestParam("albumId") long albumId, Model model) {
    Album controllerAlbum = albumRepository.findAlbum(albumId);
    controllerAlbum.addSong(new Song(3, "controllerAddedSong"));

    Album serviceAlbum = albumService.findAlbum(albumId);

    model.addAttribute("album", serviceAlbum);

    return "album";
  }
}
```

이 경우 AlbumController 에서의 로드한 Album 과 AlbumService 에서 로드한 Album 은 서로 다른 트랜잭션 내에서 로드된 것이므로 별개의 객체가 생성되며 서로 독립적인 영속성 컨텍스트 내에 캐시된다. 따라서 controllerAlbum 에 Song 을 추가하더라도 AlbumService 의 트랜잭션 커밋 시 플러시될 영속성 컨텍스트 내에 포함되어 있지 않기 때문에 데이터베이스에 반영되지 않는다.

이번에는 AlbumController 에서 두 개의 Service 메소드를 호출하는 경우를 다시 한번 살펴 보자.

AlbumController.java

```java
@Controller
public class AlbumController {
  @Autowired
  private AlbumService albumService;

  @RequestMapping("/album")
  public String view(@RequestParam("albumId") long albumId, Model model) {
    Album album = albumService.findAlbum(albumId);

    albumService.copyAlbum(album);

    model.addAttribute("album", album);

    return "album";
  }
}
```

이 경우 역시 findAlbum()과 copyAlbum() 메소드가 개별적인 트랜잭션으로 선언되어 있기 때문에 각각의 트랜잭션에 대해 서로 독립적인 Session 을 오픈한다. 따라서 두 트랜잭션은 서로 다른 영속성 컨텍스트에 객체를 캐시하게 된다. copyAlbum() 메소드는 자기 자신과 관련된 영속성 컨텍스트 이외에 또 다른 영속성 컨텍스트가 존재한다는 사실조차도 인식하지 못한다. copyAlbum() 입장에서는 파라미터로 전달된 Album 객체가 자신의 영속성 컨텍스트에 저장되어 있지 않기 때문에 트랜잭션 커밋 시에 데이터베이스로 플러시하지 않는다. 파라미터로 전달된 Album 객체를 트랜잭션 커밋 시에 함께 플러시하는 올바른 방법은 다음과 같이 명시적으로 현재의 영속성 컨텍스트에 해당 객체를 추가하는 것이다. 별도의 추가 작업 없이 파라미터로 전달된 객체가 데이터베이스에 반영되는 것은 OpenSessionInViewFilter 를 사용함으로써 발생되는 예기치 못한 사이드 이펙트라는 점에 주의하자.

AlbumServiceImpl.java

```
@Service
public class AlbumServiceImpl implements AlbumService {
  ......

  @Override
  @Transactional
  public void copyAlbum(Album source) {
    ......
    albumRepository.update(source);
  }
}
```

위 예제를 통해 singleSession=false 설정을 사용할 경우 동일한 영속 객체가 한 개 이상의 영속성 컨텍스트에 의해 관리될 수도 있다는 사실을 알 수 있다. copyAlbum() 메소드의 파라미터로 전달된 Album 인스턴스는 AlbumService.findAlubm() 메소드 실행 시에 오픈된 영속성 컨텍스트에 의해 이미 관리되고 있는 영속 객체이지만 AlbumService.copyAlubm() 메소드 실해 중에 새로 오픈된 또 다른 영속성 컨텍스트에도 추가된다. 따라서 동일한 Album 영속 객체가 서로 다른 두 개의 영속성 컨텍스트에 의해 관리되는 문제가 발생한다.

이보다 더 큰 문제는 서로 다른 영속성 컨텍스트에서 동일한 식별자를 가진 영속 객체를 로드하는 경우에 발생한다. 만약 findAlbum()과 copyAlbum()에서 동일한 식별자를 가진 Album 을 로드한다면 두 영속성 컨텍스트는 동일한 식별자를 가진 서로 다른 인스턴스를 관리하게 된다. 따라서 한 쪽의 영속성 컨텍스트에 의해 관리되는 Album 에 대한 수정 사항이 다른 영속성 컨텍스트에 포함된 Album 에는 반영되지 않게 된다. 이것은 영속성 컨텍스트와 영속 객체를 처리하는데 있어 발견하기 어려운 미묘한 문제를 야기할 수 있다.

지금까지 살펴 본 바와 같이 singleSession=false 의 경우 OPEN SESSION IN VIEW 패턴을 사용하지 않는 경우와 동일하게 각 트랜잭션 별로 독립적인 데이터베이스 연산을 처리하면서도 뷰에서의 LazyInitializationException 의 발생을 방지할 수 있다. 그러나 단일 요청 처리

중에 하나 이상의 JDBC 커넥션을 사용하고 뷰 렌더링이 완료된 이후까지 JDBC 커넥션을 풀로 반환하지 않기 때문에 커넥션 풀의 사이즈를 넉넉하게 할당하지 않으면 전체적인 시스템 가용성이 감소될 위험이 존재한다. 따라서 대용량 트래픽을 수용해야 하는 시스템의 경우 singleSession=false 설정으로 인해 발생할 수 있는 문제점을 인지하고 테스트를 통해 가용성 이슈가 발생하지 않는 지를 확인해야만 한다.

개인적인 관점에서 singleSession=false 의 장점보다는 단점으로 인한 위험성이 더 크다고 생각된다. 따라서 OpenSessionInViewFilter 나 OpenSessionInViewInterceptor 를 사용할 예정이라면 기본값인 singleSession=true 설정을 사용할 것을 권한다.

OPEN SESSION IN VIEW 는 안티패턴(Anti-Pattern)인가?

인터넷 상의 다양한 커뮤니티와 포럼을 방문하다 보면 OPEN SESSION IN VIEW 패턴에 대한 수많은 토론과 논쟁이 벌어지고 있다는 사실을 알 수 있다. 논쟁의 테이블 한 편에는 OPEN SESSION IN VIEW 패턴은 안티패턴(Anti-Pattern)이므로 사용해서는 안 된다는 진영이 포진하고 있고, 테이블의 반대편에는 OPEN SESSION IN VIEW 패턴은 분산되지 않는 환경에서 사용할 수 있는 최고의 선택사항이라는 진영이 자리하고 있다. OPEN SESSION IN VIEW 패턴에 대한 반대 입장을 표명하는 입장의 핵심 주장은 OPEN SESSION IN VIEW 패턴이 레이어 아키텍처를 위반한다는 것이다. 주장의 요지는 다음과 같다.

> OPEN SESSION IN VIEW 패턴을 사용할 경우 뷰 렌더링 중에 데이터베이스에 접근하게 되므로 REPOSITORY 나 DAO 에서 다루어져야 하는 영속성과 관련된 처리가 사용자 인터페이스 레이어에서 다루어지게 된다. 따라서 영속성과 관련된 관심사가 사용자 인터페이스 레이어로 누수되어 레이어 아키텍처의 기본 원칙인 관심사의 분리(Separation of Concerns) 원칙을 위반한다. 또한 사용자 인터페이스에서 데이터베이스 접근이 이루어지므로 사용자 인터페이스 레이어에서 영속성 레이어로의 의존 관계가 발생한다. 결론적으로 OPEN SESSION IN VIEW 는 레이어 아키텍처의 기본 원칙을 위반하는 안티패턴이다.

결론부터 말하자면 위 주장은 옳지 않다. 나는 OPEN SESSION IN VIEW 패턴이 레이어 아키텍처를 위반하지 않으며 오히려 각 레이어의 관심사를 좀 더 효과적으로 분리하도록 해준다고 생각한다. 이와 관련된 이슈를 하나씩 살펴 보기로 하자.

- **OPEN SESSION IN VIEW 패턴은 뷰 렌더링 중에 데이터베이스에서 객체를 로딩하기 때문에 사용자 인터페이스 레이어에서 영속성 메커니즘으로의 의존 관계를 추가한다. 따라서 레이어 아키텍처를 위반한 것이 아닌가?**

  소프트웨어 의존성(dependency)이란 하나의 소프트웨어 컴포넌트가 다른 소프트웨어 컴포넌트를 사용할 때 두 컴포넌트 간에 이루어지는 관계를 의미한다 일반적으로 A 가 B 에 의존한다는 것은 B 의 변경으로 인해 A 도 변경된다는 것을 의미한다. 따라서 뷰가 영속성

메커니즘에 의존한다는 것은 뷰에서 영속성과 관련된 로직이 나타나고 영속성과 관련된 로직이 변경될 경우 뷰도 함께 변경된다는 것을 의미한다.

이와 같은 사실을 기반으로 예제 프로그램의 뷰 코드를 살펴보자.

album.jsp

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
<title>앨범 정보</title>
</head>
<body>
  <h1>앨범 정보</h1>

  <h2>앨범명</h2>
  ${album.title}<br />
  <h2>가수명</h2>
  ${album.artist.name}<br/>
  <h2>수록곡</h2>
  <c:forEach var="song" items="${album.songs}">
    ${song.track} - ${song.name}<br />
  </c:forEach>
</body>
</html>
```

예제 프로그램의 뷰는 다음과 같은 두 가지 이유로 영속성 메커니즘에 의존하지 않는다.

첫 째, 예제 프로그램의 뷰를 생성하는 JSP 템플릿에는 화면 출력과 관련된 태그와 EL 표현식만이 존재할 뿐 영속성과 관련된 컴포넌트에 대한 어떤 코드도 존재하지 않는다. 따라서 현재의 뷰 로직은 영속성 메커니즘에 의존하지 않는다. OPEN SESSION IN VIEW 패턴을 사용하더라도 뷰에는 마크업이나 화면 출력과 관련된 로직만 존재해야 한다는 "깔끔한 뷰(Clean View)" 원칙은 여전히 만족시키고 있다.

둘 째, 영속성과 관련된 하부 메커니즘이 수정되더라도 뷰는 변경되지 않는다. REPOSITORY 에서 조인 페치를 통해 뷰 렌더링에 필요한 전체 객체 그래프를 로드하건, POJO FACADE 에서 지연 로딩을 통해 객체를 준비하건, OPEN SESSION IN VIEW 패턴을 사용하건, Native SQL 을 사용해 직접 쿼리를 실행해서 데이터를 준비하건 상관없이 뷰는 전달된 데이터를 이용해서 렌더링 작업을 수행할 뿐이다. 따라서 영속성 메커니즘이 변경되더라도 뷰는 변경되지 않기 때문에 뷰는 영속성 메커니즘에 의존하지 않는다.

■ **OEPN SESSION IN VIEW 패턴의 경우 사용자 인터페이스 레이어에 속하는 서블릿 필터나 인터셉터가 하이버네이트 API 에 의존하게 된다. 따라서 사용자 인터페이스로부터 영속성 메커니즘으로의 의존성이 존재하는 것이 아닌가?**

하이버네이트는 프레임워크이며 <그림 1>에 표시된 레이어 중 최하단에 위치하고 있는 인프라스트럭처 레이어에 속한다. 글 서두에서 언급한 바와 같이 현대적인 엔터프라이즈 애플리케이션은 각 레이어가 바로 하위 레이어에만 의존하는 것이 아니라 적절한 제어 속에서 하위에 위치한 어떤 레이어라도 자유롭게 접근할 수 있는 '완화된 레이어 시스템(relaxed layered system)' 방식을 채택하고 있다. 서블릿 필터와 인터셉터에서 하이버네이트에서 제공하는 SessionFactory 나 Session API 에 의존하는 것은 '완화된 레이어 시스템(relaxed layered system)'에 따라 상위 레이어에서 인프라스트럭처 레이어의 API 를 자유롭게 사용하는 것뿐이다.

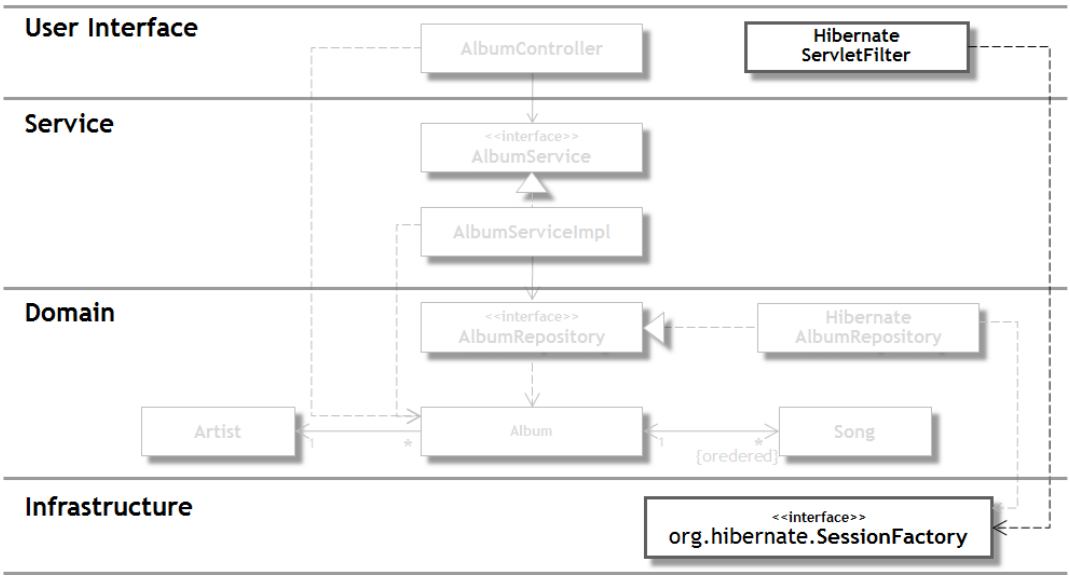
이것은 현대적인 엔터프라이즈 애플리케이션 아키텍처에서 광범위하게 사용되는 접근방법이다. Struts2 프레임워크를 사용하는 애플리케이션의 컨트롤러가 프레임워크의 Action 인터페이스를 사용하는 경우를 생각해 보라. JSP 템플릿에서 Struts2 에서 제공하는 커스텀 태그를 사용하는 경우를 생각해 보라. Spring 프레임워크에서 AOP 프록시를 통해 SERVICE 접근 전에 트랜잭션 매니저를 통해 JDBC 커넥션을 얻는 경우를 생각해 보라. 모든 레이어에서 인프라스트럭처 레이어에 속하는 전역 기능에 직접 접근하고 있다. '완화된 레이어 시스템(relaxed layered system)'은 엔터프라이즈 애플리케이션의 일반적인 패턴이며 OPEN SESSION IN VIEW 패턴에서의 서블릿과 인터셉터 역시 동일한 구조를 따르고 있을 뿐이다.

요약하면 서블릿 필터와 인터셉터는 영속성 관심사를 다루는 JDBC API 에 직접 접근하거나 REPOSITORY 에 위치해야 할 쿼리 코드를 포함하지 않으며 단지 전역 인프라스트럭처 기능을 사용할 뿐이다. 심지어 우리가 사용하는 필터와 인터셉터는 직접 작성한 것이 아닌 인프라스트럭처 레이어에 속하는 Spring 에서 제공하는 것이다. 따라서 서블릿 필터와 인터셉터는 무죄다.

서블릿 필터와 뷰와 관련된 의존성 이슈에 관심이 많다면 https://forum.hibernate.org/viewtopic.php?t=927886 을 방문해 보라. 아래 글은 논쟁을 읽을 시간이 없는 사람을 위해 결론에 해당하는 부분을 발췌한 것이다.

> 이것은 프리젠테이션 레이어와 영속성 레이어 간에 의존성을 추가하지 않는다. [Session 은] 리소스를 관리하는 전역 시스템 레이어(Global System Layer)의 기능이다. ... [뷰에서 프록시를 초기화하기 위해 데이터베이스에 접근하는 것은] 많은 사람들이 생각하는 악영향을 끼치는 의존성의 범주에 포함되지 않는다. ... 이것은 전통적인 설계 방식으로 모든 레이어에 의해 접근되는 시스템 레이어의 일부다. 여기에 실제적인 의존성이 존재한다고 생각하는 것은 모든 코드가 Thread, SecurityContext, 심지어 Object 와 같은 인프라스트럭처 구성물에 대해 의존성을 가진다고 생각하는 것과

동일하다. 이 문제를 해결할 수 있는 마법과도 같은 놀라운 방법이 등장할 가능성은 없어 보인다. 오로지 한가지 해결 방법이 존재한다: 인터셉터가 하이버네트 Session 을 관리하는 Listener 에게 이벤트를 전송할 시점을 알고 있는 것이다. 여기에는 "코드" 의존성 관점에서 실제적인 어떤 의존성도 존재하지 않으며 Observer 와 Observable 은 서로 분리되어 있다.



<그림 15> 완화된 레이어 아키텍처 패턴에 따르는 의존 관계

■ **명시적인 의존관계가 존재하지 않는다고 하더라도 사용자 인터페이스 레이어로 영속성과 관련된 관심사가 누수된 것이 아닌가?**

앞의 JSP 를 통해 확인할 수 있는 것처럼 뷰를 작성할 때 영속성과 관련된 어떤 관심사도 다루지 않는다는 것을 알 수 있다. 뷰는 제공된 모델간의 연관 관계를 통해 필요한 정보를 화면에 표시할 뿐이다.

뷰에서 지연 로딩을 통해 데이터베이스로 접근하는 것과 영속성과 관련된 관심사가 사용자 인터페이스 레이어로 누수되는 것은 전혀 상관이 없는 이슈이다. 영속성과 관련된 관심사는 ENTITY 를 어떻게 로드할 것인가, 또는 쿼리를 어떻게 작성할 것인가, 어떤 영속성 메커니즘을 사용할 것인가와 관련된 것이다. 이러한 이슈는 이미 REPOSITORY 구현 시점에 해결이 된 사항이다.

뷰는 전달된 데이터를 소비할 뿐이다. 탐욕스럽게 객체 그래프를 탐색하더라도 이것은 뷰 렌더링에 필요한 객체와 객체 간의 논리적인 연관 관계와 관련된 이슈이지 영속성 메커니즘과 관련된 이슈는 아니다. 앞에서 언급한 사실을 기억하라. 하부 레이어에서 데이터를 로드하는 방법을 변경하더라도 객체 그래프만 동일하다면, 즉 모델이 동일하다면 뷰는 변경되지 않는다.

■ **뷰를 작성할 때 영속성과 관련된 이슈를 고려해야 하기 때문에 관심사가 누수된 것 아닌가?**

앞서 말한 바와 같이 뷰를 작성할 때는 영속성에 대해 고민하지 않는다. OPEN SESSSION IN VIEW 패턴은 뷰에 대해 투명하다.

오히려 REPOSITORY 에서 뷰에 필요한 모든 객체 그래프를 준비하는 것이 뷰의 관심사가 영속성과 관련된 레이어로 누수된 것이라고 볼 수 있다. POJO FAÇADE 패턴 역시 뷰의 관심사가 애플리케이션 레이어로 누수된 것이라고 볼 수 있다. 두 가지 방법 모두 뷰의 관심사인 어떤 데이터를 어떻게 출력할 것인가에 대한 이슈를 올바르지 않은 레이어에서 해결하려는 잘 못된 시도일 뿐이다.

나는 OPEN SESSION IN VIEW 패턴이 DOMAIN MODEL 패턴[Fowler PEAA]에 따라 도메인 레이어를 구성할 수 있도록 해주는 효과적인 방법이라고 생각한다. 도메인 레이어로부터 뷰 렌더링과 관련된 관심사를 완전히 제거하고 순수하게 도메인 로직에 초점을 맞출 수 있기 때문이다. REPOSITORY 에서 뷰에 필요한 모든 객체 그래프를 준비하는 것은 데이터 중심의 아키텍처에 이르거나 잘 해봐야 TRANSACTION SCRIPT 패턴[Fowler PEAA]을 따르는 도메인 레이어를 구축할 수 있을 뿐이다. POJO FAÇADE 패턴을 사용하면 DOMAIN MODEL 패턴을 구현할 수는 있으나 구현 복잡도와 관리 복잡도라는 측면에서 득보다 실이 많다.

OPEN SESSION IN VIEW 패턴에 대해 의심이 든다면 자신에게 다음과 같은 질문을 해보라. 역으로 뷰와 관련된 관심사가 애플리케이션 레이어나 도메인 레이어로 누수되지 않도록 하려면 어떻게 해야 하는가?

■ **뷰에서는 도메인 객체의 프로퍼티를 반환하는 메소드를 자유롭게 호출할 수 있다. 만약 프로퍼티를 반환하는 메소드가 애플리케이션의 상태를 변경하는 중요한 로직을 포함하고 있을 경우 이 메소드를 뷰 렌더링 시점에 호출할 경우 문제가 발생하지 않겠는가?**

이것은 OPEN SESSION IN VIEW 패턴과 관련된 문제가 아니다. 이것은 소위 DTO(Data Transfer Object)[Fowler PEAA]나 TO(Transfer Object)[Alur CORE]라고 불리는 데이터 홀더 대신 도메인 객체를 뷰의 모델로 사용하는 애플리케이션 아키텍처와 관련된 이슈라고 보는 것이 타당하다.

이러한 문제를 방지하기 위해 도메인 객체 대신 DTO 를 사용하자는 주장도 있으나 이것은 앞에서 살펴 본 POJO FAÇADE 패턴처럼 뷰에 대한 관심사가 애플리케이션 레이어와 도메인 레이어로 누수되는 문제를 안고 있다. 뷰에 도메인 객체를 전달하는 것이 캡슐화의 원칙을 위반한다는 견해도 있으나 도메인 객체가 전달된다고 해서 반드시 캡슐화 위반이라고 볼 수 없으며 DTO 를 전달한다고 해서 반드시 캡슐화의 원칙이 지켜진다고 볼 수도 없다. 아키텍처적인 관점에서 뷰가 도메인 객체에 접근하는 것 역시 '완화된 아키텍처 시스템'의 일종일 뿐이다.

이와 관련해서 AOP 를 사용한 메커니즘적인 방법으로 위험을 방지하자는 의견도 있으나 나는 이 문제가 도메인 레이어의 설계와 관련된 문제라고 생각한다. 뷰에서는 도메인 객체의 상태를 조회하는 GETTER 메소드만 호출해야 하며 상태를 변경하는 메소드를 호출해서는 안된다. 이것은 문법적으로 "깔끔한 뷰(Clean View)"를 지향하면 달성할 수 있는데 엔터프라이즈 Java 환경에서 뷰를 생성하는 모든 템플릿 엔진들은 모델이 GETTER 메소드만 호출할 수 있기 때문이다(JSP 의 스크립틀릿과 같은 기능들은 "깔끔한 뷰" 원칙을 위반하므로 논외로 한다).

여기에서의 문제는 GETTER 메소드 내부에서 객체의 상태를 바꾸는 로직이 포함될 수도 있다는 것이다. 이것은 시스템의 상태를 변경시킬 수 있는 비즈니스 로직을 호출해서는 안된다는 "얇은 뷰(Thin View)" 원칙을 위배하는 것이다. 즉, 외적인 측면에서 "깔끔한 뷰"를 만족시키는 것은 스타일의 문제이지만 내적인 측면에서 "얇은 뷰"를 만족시키는 것은 설계의 문제이다.

서블릿 필터에서 트랜잭션이 시작되고 커밋되는 전통적인 방식의 OPEN SESSION IN VIEW 필터의 경우 뷰에서의 위험한 GETTER 호출로 인한 상태 변경이 데이터베이스로 플러시될 수 있기 때문에 문제가 될 여지가 있다. 그러나 Spring 프레임워크에서 제공하는 필터와 인터셉터를 사용하는 경우 뷰에서 이와 같은 메소드를 호출하더라도 객체의 상태 변경이 데이터베이스로 플러시되지 않기 때문에(FlushMode.MANUAL 로 설정되기 때문이다) 실제적인 문제로 이어지지는 않는다. 그럼에도 불구하고 GETTER 에서 객체의 상태를 변경하거나 비즈니스 적으로 중요한 처리를 수행하는 것은 올바른 접근 방법이 아니다.

여기에서 언급할만한 설계 원칙은 "CQS(Command-Query Separation)" 원칙[CQS]이다. 객체의 상태를 변경하는 Command 와 객체의 상태를 반환하는 Query 를 분리하라. CQS 원칙을 한 문장으로 줄여 표현하면 "질문이 답변을 수정해서는 안 된다"는 것이다. Command 는 상태를 변경하는 대신 객체의 상태를 반환해서는 안 된다. Query 는 객체의 상태를 반환하는 대신 값을 변경해서는 안 된다.

도메인 객체에 대해 Command 와 Query 를 분리하고 뷰에서는 사이드 이펙트가 없는 Query 만을 호출하라. 서두에서 언급한 레이어 아키텍처와 관련된 몇 가지 원칙을 기억하라. 항상 "얇은 뷰(Thin View)"를 만들기 위해 노력하라. 이것은 도메인 객체를 노출시킬 것인가 아닌가와 무관하게 좋은 API 를 만들기 위한 설계 원칙과 관련되어 있다. 도메인 레이어에 속하는 객체들이 항상 CQS 를 만족하도록 하라.

OPEN SESSIO IN VIEW 를 사용하라

OPEN SESSION IN VIEW 패턴은 안티패턴이 아니다. OPEN SESSION IN VIEW 패턴은 뷰와 관련된 관심사를 도메인 레이어로부터 분리시키기 위해 고려해 볼 수 있는 여러 가지 옵션 중의 하나이다. 나는 분산 환경이 아닌 단일 JVM 에서 실행되는 애플리케이션을 DOMAIN MODEL 패턴에 따라 개발해야 한다면 주저 없이 OPEN SESSION IN VIEW 패턴을 선택할 것이다.

개인적으로 선호하는 방법은 Spring 프레임워크에서 제공하는 OpenSessionInViewFilter 나 OpenSessionInViewInterceptor 를 사용하고 singleSession=true 로 설정하는 것이다.

그러나 앞에서 설명한 바와 같이 singleSession=true 로 설정하는 경우 영속성 컨텍스트가 전체 요청 처리 동안 오픈되어 있기 때문에 암시적인 별칭 문제로 인해 미묘한 문제가 발생할 수 있다. 이를 해결하는 방법은 효과적인 방법은 없으나 다음과 같은 몇 가지 지침을 통해 문제를 예방할 수는 있다.

- 애플리케이션 레이어 SERVICE 호출 전에 사용자 인터페이스 컨트롤러(또는 Action)에서 영속성 컨텍스트에 포함될 수 있는 ENTITY 를 로드하지 말라. 컨트롤러에서 로드한 ENTITY 가 영속성 컨텍스트에 포함되어 암시적인 별칭 문제를 야기할 수도 있다.
- 컨트롤러에서 ENTITY 를 로드해야 한다면 SERVICE 호출 전에 영속성 컨텍스트에 포함된 ENTITY 들을 클리어하라. 이 방법은 컨트롤러 구현이 OPEN SESSION IN VIEW 패턴을 사용한다는 사실과 설정에 영향을 받기 때문에 좋은 방법은 아니며 레이어 아키텍처의 원칙을 위반하기 때문에(사용자 인터페이스에서 영속성 메커니즘과 관련된 관심사를 해결하려고 한다!) 가급적 컨트롤러에서 SERVICE 호출 전에 ENTITY 를 로드하지 않는 방법을 선택하는 것이 좋다. 그러나 이러한 상황을 피할 수 없고 영속성 컨텍스트와 관련된 미묘한 문제로 인해 고통 받는 것보다는 코드가 조금 지저분해 지는 것이 정신건강에 이롭다는 입장이라면 고려해 볼 수 있는 방법이다(실제로 현재의 프로젝트에서 부분적으로 이 방법을 적용하고 있다).
- TRANSACTION SCRIPT 패턴을 피하라. 애플리케이션 레이어 SERVICE 는 최대한 얇아야 하며 중요한 책임은 SERVICE 가 아닌 도메인 객체에 할당되어야 한다. 아키텍처가 TRANSACTION SCRIPT 패턴을 향해 나아갈수록 SERVICE 에서 다른 SERVICE 를 호출하는 빈도가 늘어날 것이다. SERVICE 의 호출 스택 내에서 동일한 식별자를 가진 객체를 로드할 경우 암시적인 별칭 문제가 발생하게 되며 이것은 결국 디버깅의 악몽으로 이어질 것이다. SERVICE 호출 스택이 깊어지면 깊어질수록 문제를 해결하기가 어려워진다. 중요한 로직을 도메인 객체로 몰아 넣고 SERVICE 에는 흐름 제어 로직만 남겨 두도록 하라.
- 도메인 레이어를 객체 지향적으로 설계하라. CQS 원칙에 따라 상태 변경과 상태 조회를 분리하라. 도메인 레이어가 영리해 질수록 SERVICE 레이어는 얇아 질 것이다.
- 모든 요청 처리에 대해 하나의 트랜잭션 경계만을 유지하라. 이것은 컨트롤러에서 @Transactional 어노테이션이 명시된 SERVICE 오퍼레이션을 하나만 호출한다는 것을 의미한다. 이 원칙은 OPEN SESSION IN VIEW 패턴이 아니더라도 가급적 지키는 것이 좋다. 일반적으로 컨틀로러에서 동일한 영속성 저장소에 접근하는 두 개의 @Transactional 어노테이션이 존재할 경우 애플리케이션 레이어의 흐름 관리와 무관한 도메인 레이어의 비즈니스 로직이 컨트롤러에 위치하고 있는 것은 아닌지 확인해 보는 것이 좋다. 이 경우 컨트롤러 내부에 코드 중복이 발생할 가능성이 높다.

- 객체 간의 연관 관계를 파악하는데 많은 시간을 투자하라. 연관 관계로 연결된 객체가 항상 함께 사용된다면 매핑 메타 데이터를 수정하여 연관 객체 전체가 함께 로딩되도록 하라. 이러한 결정은 뷰의 요구사항이 아닌 도메인을 분석한 결과에 기반해서 내려져야 한다.

지금까지 OPEN SESSION IN VIEW 패턴과 관련된 아키텍처 및 설계 이슈에 관해 살펴보았다. 나는 OPEN SESSION IN VIEW 패턴이 DOMAIN MODEL 패턴의 원칙과 원리에 부합하는 객체-지향적인 도메인 레이어를 개발할 수 있도록 해주는 훌륭한 솔루션이라고 생각한다. 물론 OPEN SESSION IN VIEW 패턴 이외에도 뷰에서의 렌더링 이슈를 해결해 주는 다양한 솔루션이 존재한다. 이들 대부분은 기존의 상태 없는(Stateless) 애플리케이션 레이어 SERVICE 대신 상태를 가진(Stateful) SERVICE 나 플로우 기반으로 영속성 컨텍스트를 확장하여 컨버세이션(conversation)을 구현하는 방식을 사용하고 있다. 개인적으로 이러한 기술에 대한 깊은 지식이 없기 때문에 장단점을 논의하기는 어렵지만 분산되지 않은 환경에서 상태 없는 애플리케이션 레이어 SERVICE 를 사용하고 있다면 OPEN SESSION IN VIEW 패턴을 선택하는 것은 올바른 결정이라는 점을 강조하고 싶다.

마지막 조언은 다음과 같다. 도메인 레이어의 설계에 집중하고 훌륭한 객체 지향 원칙과 아키텍처 원칙을 따르도록 노력하라. OPEN SESSION IN VIEW 패턴은 애플리케이션이 훌륭한 원칙을 기반으로 구축되어 있을 때에 진가를 발휘할 것이다.

[Alur CORE] Deepak Alur et el, Core J2EE Patterns, 피어슨 에듀케이션 코리아, 2004

[Bauer JPAH] Christian Bauer, Gavin King, 하이버네이트 완벽 가이드, 위키북스, 2010

[Buschman POSA1] Frank Buschmann, 패턴 지향 소프트웨어 아키텍처: 패턴 시스템 Volume 1, 앤선, 2008

[CQS] Command-Query Separation(CQS) 원리, http://aeternum.egloos.com/1125381

[Evans DDD] Eric Evans, Domain-Driven Design, Addison-Wesley Professional, 2003

[Fowler PEAA] Martin Fowler, 엔터프라이즈 애플리케이션 아키텍처 패턴, 피어슨 에듀케이션 코리아, 2003

[GOF DP] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, GOF 의 디자인 패턴, 피어슨에듀케이션 코리아, 2007

[Cockburn HEXAGONAL] Alistair Cockburn, The Hexagonal Architecture (Ports & Adapters) Pattern, http://alistair.cockburn.us/Hexagonal+architecture

[HIBERNATE] Hibernate Reference, http://www.hibernate.org/docs.html

[Johnson J2EEDD] Ralph Johnson, Expert One-on-One J2EE Design and Development, Wrox, 2002

[Larman AUP] Craig Larman, UML 과 패턴의 적용 2/e, 홍릉과학출판사, 2003

[Nilsson ADDD] Jimmy Nilsson, Applying Domain-Driven Design and Patterns, Addison-Wesley Professional, 2006

[OSIV] OPEN SESSION IN VIEW, http://community.jboss.org/wiki/OpenSessionInView

[Richarson POJO] Chris Richardson, POJOS IN ACTION, Manning, 2006

[Marinescu EJB] Floyd Marinescu, EJB 디자인 패턴, 인사이트, 2002