



Hibernate EntityManager

사용자 안내서

버전: 3.1 베타 1

차례

| | |
|--|----|
| EJB3 영속 개요 | iv |
| 1. 아키텍처 | 1 |
| 1.1. 정의들 | 1 |
| 1.2. EJB 컨테이너 환경 | 1 |
| 1.2.1. 컨테이너에 의해 관리되는 엔티티 관리자 | 2 |
| 1.2.2. 어플리케이션에 의해 관리되는 엔티티 관리자 | 2 |
| 1.2.3. 영속 컨텍스트 영역 | 2 |
| 1.2.4. 영속 컨텍스트 보급(propagation) | 3 |
| 1.3. J2SE 환경들 | 4 |
| 2. 셋업과 구성 | 5 |
| 2.1. 셋업 | 5 |
| 2.2. 구성 | 5 |
| 2.3. J2SE 환경에서 하나의 EntityManagerFactory와 하나의 EntityManager 획득하기 .. | 7 |
| 3. 객체들로 작업하기 | 9 |
| 3.1. 엔티티 상태들 | 9 |
| 3.2. 객체들을 영속화 시키기 | 9 |
| 3.3. 객체를 로드시키기 | 9 |
| 3.4. 객체들을 질의하기 | 10 |
| 3.4.1. 질의들을 실행하기 | 10 |
| 3.4.1.1. 투영(Projection) | 11 |
| 3.4.1.2. 스칼라 결과들 | 11 |
| 3.4.1.3. 바인드 파라미터들 | 11 |
| 3.4.1.4. 쪽매김 | 12 |
| 3.4.1.5. 명명된 질의들을 외재화 시키기 | 12 |
| 3.4.1.6. Native 질의들 | 12 |
| 3.4.1.7. 질의 힌트들 | 13 |
| 3.5. 영속 객체들을 수정하기 | 13 |
| 3.6. 분리된 객체들(detached objects)을 수정하기 | 14 |
| 3.7. 자동적인 상태 검출 | 14 |
| 3.8. 관리되는 객체들을 삭제하기 | 15 |
| 3.9. 영속 컨텍스트를 Flush시키기 | 16 |
| 3.10. 전이 영속(Transitive persistence) | 17 |
| 4. 트랜잭션들과 동시성 | 19 |
| 4.1. 엔티티 관리자와 트랜잭션 영역들 | 19 |
| 4.1.1. 작업 단위 | 19 |
| 4.1.2. 장기간의 작업 단위 | 20 |
| 4.1.3. 객체 동일성(identity) 고찰 | 22 |
| 4.1.4. 공통된 동시성 제어 쟁점들 | 22 |
| 4.2. 데이터베이스 트랜잭션 경계설정 | 23 |
| 4.2.1. 관리되지 않는 환경 | 24 |
| 4.2.2. JTA 사용하기 | 24 |
| 4.2.3. 예외상황 처리 | 26 |
| 4.3. Optimistic 동시성 제어 | 27 |
| 4.3.1. 어플리케이션 버전 체크 | 27 |

| | |
|--|----|
| 4.3.2. 확장된 엔티티 관리자와 자동적인 버전 체크 | 27 |
| 4.3.3. Detached objects and automatic versioning | 28 |
| 5. 엔티티 리스너들과 콜백 메소드들 | 30 |
| 6. Batch 처리 | 32 |
| 6.1. 대용량 update/delete | 32 |
| 7. EJB-QL: 객체 질의 언어(Object Query Language) | 34 |
| 7.1. 대소문자 구분 | 34 |
| 7.2. from 절 | 34 |
| 7.3. 주해들과 조인들 | 34 |
| 7.4. select 절 | 36 |
| 7.5. 집계 함수들 | 37 |
| 7.6. 다형성 질의들 | 37 |
| 7.7. where 절 | 38 |
| 7.8. 표현식들 | 39 |
| 7.9. order by 절 | 42 |
| 7.10. group by 절 | 42 |
| 7.11. 서브질의들 | 43 |
| 7.12. EJB-QL 예제들 | 44 |
| 7.13. 대용량 UPDATE & DELETE 문장들 | 45 |
| 7.14. 팁들 & 트릭들 | 45 |
| 8. Native 질의 | 47 |
| 8.1. 결과셋을 표현하기 | 47 |
| 8.2. native SQL 질의들을 사용하기 | 47 |
| 8.3. 명명된 질의들 | 48 |
| A. 준수사항과 알려진 제한사항들 | 49 |

EJB3 영속 개요

WARNING ! 이 문서는 현재 3.1 베타 1 버전이며, 초역 상태입니다. 오역이나 오타가 존재하리라 생각되며, 이 점을 양해 바랍니다. 그리고 오타나 오역이 발견되면 역자의 전자 메일 [mailto:jdkim528@korea.com] 또는 블로그 [http://blog.naver.com/blog/jdkim528/]로 피드백을 주시기 바랍니다. 독자들의 관심으로 오역이나 오타가 정정되면 적절한 시점에 HibernateExt CVS에 정식으로 커밋시킬 수 있도록 할 예정입니다. 마찬가지로 현재 배포 중인 Hibernate Annotations 번역본 또한 이와 같은 과정을 거쳐 HibernateExt CVS 디렉토리에 커밋하도록 하겠습니다. 그럼 Hibernate EntityManager 참조문서와 함께 즐거운 시간을 보내시기를 바랍니다. [Translated By Jongdae Kim]

EJB3 명세서는 투명한 객체/관계형 매핑 패러다임의 중요성과 성공을 인정하고 있다. EJB3 명세서는 임의의 객체/관계형 영속 메커니즘에 필요한 기본 API들과 메타데이터를 표준화시킨다. Hibernate EntityManager는 EJB3 영속 명세서에 정의된 대로 프로그래밍 인터페이스들과 생명주기 규칙들을 구현한다. Hibernate Annotations와 함께, 이 포장 꾸러미(wrapper)는 성숙한 Hibernate 코어의 상단에 있는 하나의 완전한 (그리고 스탠드얼론) EJB3 영속 솔루션을 구현한다. 당신은 당신의 프로젝트의 비즈니스 및 기술적 요구에 따라 세 가지 모두의 조합, 또는 EJB3 프로그래밍 인터페이스들과 생명주기 없는 annotations, 또는 심지어 순수 있는 그대로의 Hibernate를 사용할 수 있다. 당신은 언제든지 필요하다면 Hibernate native API들, 심지어 native JDBC와 SQL로 되돌아갈 수 있다.

1장. 아키텍처

1.1. 정의들

EJB3는 J2EE 5.0 플랫폼의 부분이다. EJB3에서 영속성은 EJB3 컨테이너들에서 이용 가능할 뿐만 아니라 특정 컨테이너 외부에서 실행되는 J2SE 어플리케이션들에 대해서도 이용 가능하다. 다음 프로그래밍 인터페이스들과 산출물들은 두 환경들 모두에서 이용 가능하다.

EntityManagerFactory

엔티티 관리자 팩토리는 엔티티 관리자 인터페이스들을 제공하고, 모든 인스턴스들은 특정 구현 등에 의해 정의된 것과 동일한 디폴트 설정들을 사용하기 위해서, 동일한 데이터베이스에 연결하도록 구성된다. 당신은 여러 데이터 저장소들에 접근하기 위해서 여러 개의 엔티티 관리자 팩토리들을 준비할 수 있다. 이 인터페이스는 native Hibernate에서 `SessionFactory`와 유사하다.

EntityManager

`EntityManager` API는 하나의 특별한 작업 단위에서 하나의 데이터베이스에 접근하는데 사용된다. 그것은 영속 엔티티 인스턴스들을 생성시키고 제거시키는데, 그것들의 프라이머리 키 동일성 (identity)로서 엔티티들을 찾는데, 그리고 모든 엔티티들에 대해 질의하는데 사용된다. 이 인터페이스는 Hibernate에서 `Session`과 유사하다.

영속 컨텍스트

영속 컨텍스트는 유일 엔티티 인스턴스가 존재하는 임의의 영속 엔티티 identity에 대한 엔티티 인스턴스들의 집합이다. 영속 컨텍스트 내에서, 엔티티 인스턴스들과 그것들의 생명주기는 특별한 엔티티 관리자에 의해 관리된다. 이 컨텍스트의 영역은 트랜잭션일 수 있거나, 하나의 확장된 작업 단위일 수 있다.

영속 단위

하나의 주어진 엔티티 관리자에 의해 관리될 수 있는 엔티티 타입들의 집합은 하나의 영속 단위에 의해 정의된다. 하나의 영속 단위는 어플리케이션에 의해 관계되거나 그룹지어지는 모든 클래스들의 집합을 정의하고, 그것은 단일 데이터 저장소로 그것들의 매핑 속에 함께 위치 지워져야 한다.

컨테이너에 의해 관리되는 엔티티 관리자

그 생명주기가 컨테이너에 의해 관리되는 엔티티 관리자.

어플리케이션에 의해 관리되는 엔티티 관리자

그 생명주기가 어플리케이션에 의해 관리되는 엔티티 관리자.

JTA 엔티티 관리자

하나의 JTA 트랜잭션에 수반된 엔티티 관리자

리소스-로컬 엔티티 관리자

(JTA 트랜잭션이 아닌) 하나의 리소스 트랜잭션을 사용하는 엔티티 관리자.

1.2. EJB 컨테이너 환경

1.2.1. 컨테이너에 의해 관리되는 엔티티 관리자

J2EE 환경에서 가장 공통적이고 광범위하게 사용되는 엔티티 관리자는 컨테이너에 의해 관리되는 엔티티 관리자이다. 이 모드에서, 컨테이너는 엔티티 관리자를 열고 닫을 책임이 있다(이것은 어플리케이션에게는 투명하다). 그것은 또한 트랜잭션 경계들에 대해 책임을 진다. 컨테이너에 의해 관리되는 엔티티 관리자는 dependency injection을 통해 또는 JNDI 룩업을 통해 어플리케이션 내에서 획득되며, 컨테이너에 의해 관리되는 엔티티 관리자는 JTA를 필요로 한다.

1.2.2. 어플리케이션에 의해 관리되는 엔티티 관리자

어플리케이션에 의해 관리되는 엔티티 관리자는 어플리케이션 코드로 엔티티 관리자를 제어하는 것을 당신에게 허용해준다. 이 엔티티 관리자는 `EntityManagerFactory` API를 통해 검색된다. 하나의 어플리케이션에 의해 관리되는 엔티티 관리자는 현재의 JTA 트랜잭션(JTA 엔티티 관리자) 내에 수반될 수 있거나, 그 트랜잭션은 `EntityTransaction` API(리소스-로컬 엔티티 관리자)를 통해 제어될 수도 있다. 리소스-로컬 엔티티 관리자 트랜잭션은 하나의 직접적인 리소스 트랜잭션(예를 들어, Hibernate의 경우 하나의 JDBC 트랜잭션)으로 매핑된다. 엔티티 관리자 유형(JTA 또는 리소스-로컬)은 구성 시에, 즉 엔티티 관리자 팩토리를 설정할 때 정의된다.

1.2.3. 영속 컨텍스트 영역

하나의 엔티티 관리자는 영속 컨텍스트와 상호작용하는 API이다. 두 개의 공통된 방도들이 사용될 수 있다: 영속 컨텍스트를 트랜잭션 경계들에 바인딩 시키기, 또는 몇몇 트랜잭션들을 가로질러 영속 컨텍스트를 이용 가능하도록 유지하기.

가장 공통적인 경우는 영속 컨텍스트 영역을 현재의 트랜잭션 영역에 묶어두는(bind) 것이다. 이것은 JTA 트랜잭션들이 사용될 때 특히 편리하다: 영속 컨텍스트 영역은 JTA 트랜잭션 생명 주기와 연관된다. 하나의 엔티티 관리자가 호출될 때, 만일 현재의 JTA 트랜잭션과 연관된 영속 컨텍스트가 존재하지 않을 경우에, 영속 컨텍스트가 또한 열린다. 그 밖의 경우, 연관된 영속 컨텍스트가 사용된다. 영속 컨텍스트는 JTA 트랜잭션이 완료될 때 끝난다. 이것은 JTA 트랜잭션 동안에 어플리케이션이 동일한 영속 컨텍스트의 관리되는 엔티티들에 대해 작업하는 것이 가능할 것임을 의미한다. 달리 말해, 당신은 당신의 EJB 메소드 호출들을 가로질러 엔티티 관리자의 영속 컨텍스트를 전달하지 말아야 하지만, 당신이 하나의 엔티티 관리자를 필요로 할 때마다 간단하게 dependency injection 또는 JNDI 룩업을 사용하라. 리소스-로컬 관리자의 경우, 하나의 새로운 리소스 트랜잭션이 (`EntityTransaction.begin()`를 통해) 시작될 때 하나의 새로운 영속 컨텍스트가 시작되고 리소스 트랜잭션이 완료될 때 하나의 새로운 영속 컨텍스트가 끝난다. 만일 엔티티 관리자가 트랜잭션 영역의 외부에서 호출될 경우, 그 메소드 호출에만 서비스하기 위해 영속 컨텍스트가 생성되고 파괴되고, 데이터베이스로부터 로드된 모든 엔티티들은 그 메소드 호출의 끝에서 detach된다. 이것은 전통적인 JDBC에서 자동-커밋(auto-commit) 특징과 유사하다.

당신은 또한 하나의 확장된 영속 컨텍스트를 사용할 수 있다. 당신이 하나의 컨테이너에 의해 관리되는 엔티티 관리자를 사용할 경우, 이것은 상태있는 세션 빈즈와 결합될 수 있다: 영속 컨텍스트는 하나의 엔티티 관리자가 dependency injection 또는 JNDI 룩업으로부터 검색될 때 생성되고, 상태 있는 세션 빈(bean) 메소드 `Remove`의 완료 후에 컨테이너가 그것을 닫기 전까지 유지된다. 이것은 하나의 "긴" 작업 단위 패턴을 구현하는 하나의 완전한 메커니즘이다. 예를 들어, 만일 당신이 단일 작업 단위(예를 들면, 완전하게 완료되어야 하는 하나의 마법사 대화상자)로서 다중 사용자 상호작용 주기들을 다루어야 할 경우에, 당신은 대개 어플리케이션 사용자의 관점에서 이것을 하나의 작업 단위로 모형화 시키고, 하나의 확장된 영속 컨텍스트를 사용하여 그것을 구현한다. 이 패턴에 대한 추가 정보는 Hibernate 참조 매뉴얼 또는 Hibernate In Action 책을 참조하길 바란다. 어플리케이션에 의해

관리되는 엔티티 관리자의 경우에 영속 컨텍스트는 엔티티 관리자가 생성될 때 생성되고 엔티티 관리자가 닫히기 전까지 유지된다.

리 소 스 - 로컬 엔 티티 관 리자 또는 (어 플 리 케 이 션에 의해 관 리 되 는) `EntityManagerFactory.createEntityManager()`로 생성된 엔티티 관리자는 하나의 영속 컨텍스트와 하나의 일-대-일 관계를 갖는다. 다른 경우들에서는 영속 컨텍스트 전파 (persistence context propagation)가 일어난다.

1.2.4. 영속 컨텍스트 보급(propagation)

영속 컨텍스트 보급(propagation)은 컨테이너에 의해 관리되는 엔티티 관리자들에 대해 그리고 `EntityManagerFactory.getEntityManager()`를 통해 획득된 엔티티 관리자들에 대해 일어난다.

하나의 트랜잭션-영역의 컨테이너에 의해 관리되는 엔티티 관리자에서(J2EE 환경에서 공통적인 경우), JTA 트랜잭션 보급(propagation)은 영속 컨텍스트 리소스 보급(propagation)과 동일한 것이다. 달리 말해, 하나의 주어진 JTA 트랜잭션 내에서 검색된 모든 컨테이너-영역의 엔티티 관리자들은 모두 동일한 영속 컨텍스트를 공유한다. Hiberante 용어로, 이것은 모든 관리자들이 동일한 세션을 공유함을 의미한다.

하나의 JTA에 바인드된 (묶인) 그리고 어플리케이션에 의해 관리되는 엔티티 관리자가 `EntityManagerFactory.getEntityManager()` 내에서 검색될 때, 반환된 엔티티 관리자는 JTA 트랜잭션에 묶인 영속 컨텍스트와 연관된다. 만일 영속 컨텍스트가 아직 연관되어 있지 않다면, 하나의 새로운 것이 생성되고 연관된다.

중요: 영속 컨텍스트는 다른 JTA 트랜잭션들 사이에서 또는 동일한 엔티티 관리자 팩토리로부터 생겨나지 않은 엔티티 관리자 사이에서 결코 공유되지 않는다. 확장된 영속 컨텍스트들을 사용할 때 컨텍스트 보급(propagation)에 대한 몇몇 주목할 만한 예외상황들이 존재한다:

- 만일 하나의 트랜잭션 영역의 영속 컨텍스트를 가진 상태없는 세션 빈(bean), message-driven bean, 또는 상태있는 세션 빈(bean)이 동일한 JTA 트랜잭션 내에서 하나의 확장된 영속 컨텍스트를 가진 하나의 상태있는 세션 빈(bean)을 호출할 경우, `IllegalStateException`이 던져진다.
- 만일 하나의 확장된 영속 컨텍스트를 가진 하나의 상태있는 세션 빈(bean)이 동일한 JTA 트랜잭션 내에서 하나의 트랜잭션-영역의 영속 컨텍스트를 가진 상태없는 세션 빈(bean)으로서 또는 하나의 상태있는 세션 빈으로서 호출될 경우, 그 영속 컨텍스트가 보급된다.
- 만일 하나의 확장된 영속 컨텍스트를 가진 하나의 상태있는 세션 빈(bean)이 다른 JTA 트랜잭션 컨텍스트 내에 있는 하나의 상태 있는 세션 빈(bean) 또는 하나의 상태 없는 세션 빈(bean)을 호출할 경우, 그 영속 컨텍스트는 보급되지 않는다.
- 만일 하나의 확장된 영속 컨텍스트를 가진 하나의 상태있는 세션 빈(bean)이 하나의 확장된 영속 컨텍스트를 가진 또 다른 상태있는 세션 빈(bean)을 초기화 시킬 경우, 그 확장된 영속 컨텍스트는 두 번째 상태 있는 세션 빈(bean)에 의해 상속된다. 만일 두 번째 상태 있는 세션 빈(bean)이 첫 번째 상태 있는 세션 빈이 아닌 다른 트랜잭션 컨텍스트에 대해 호출될 경우, `IllegalStateException`이 던져진다.
- 만일 하나의 확장된 영속 컨텍스트를 가진 하나의 상태 있는 세션 빈(bean)이 동일한 트랜잭션 내에서 다른 확장된 영속 컨텍스트를 가진 하나의 상태 있는 세션 빈을 호출할 경우, `IllegalStateException`이 던져진다.

1.3. J2SE 환경들

J2SE 환경에서는 오직 어플리케이션에 의해 관리되는 엔티티 관리자들만이 이용 가능하다. 당신은 `EntityManagerFactory` API를 사용하여 하나의 엔티티 관리자를 검색할 수 있다. 오직 리소스-로컬 엔티티 관리자들만이 이용 가능하다. 달리 말해, JTA 트랜잭션들과 영속 컨텍스트 보급은 J2SE 환경에서는 지원되지 않는다(당신은 예를 들어 Hibernate 공동체에서 대중적인 스레드 로컬 세션 패턴을 사용하여, 당신 스스로 영속 컨텍스트를 보급해야 될 것이다).

하지만 당신은 두 개의 다른 엔티티 관리자 방도들 사이에서 여전히 선택할 수 있다. 첫 번째 것인 트랜잭션-영역의 엔티티 관리자는 `EntityManagerTransaction.begin()`이 매번 호출될 때마다 한 개의 영속 컨텍스트를 생성시킬 것이다. 이 영속 컨텍스트는 트랜잭션 완료 시에 닫혀질 것이다. 이용 가능한 두 번째 방도는 확장된 컨텍스트이다. 그 경우에, 하나의 영속 컨텍스트는 (`EntityManagerFactory.createEntityManager(EXTENDED)`를 사용하여) 엔티티 관리자가 검색될 때 생성되고 엔티티 관리자가 닫혀질 때 닫혀진다. 많은 리소스-로컬 트랜잭션은 이 경우에 동일한 영속 컨텍스트를 공유한다.

2장. 셋업과 구성

TODO: persistence.xml과 .pars에 대한 보다 깊은 지원이 전체적으로 완료될 때 이 절을 개정할 것.

2.1. 셋업

EJB3 호환 Hibernate EntityManager는 Hibernate 알맹이(core)와 Hibernate Annotations의 상단에서 빌드된다. 당신은 각각의 모듈에 대한 호환가능한 버전들을 사용해야 한다 - EntityManager의 배포본 패키지 내에 있는 README.TXT 파일을 보라. 다음 라이브러리들은 당신의 classpath 내에 있어야 한다: hibernate3.jar, hibernate-annotations.jar, hibernate-entity-manager.jar 그리고 각각의 패키지에 대해 모든 필요한 제 3의 라이브러리들.(ejb-persistence.jar를 포함하여).

2.2. 구성

어플리케이션 서버 내에서 그리고 스탠드얼론 어플리케이션 내에서 엔티티 관리자들에 대한 구성은 하나의 영속 아카이브(persistence archive, .par) 내에 존재한다. 하나의 영속 아카이브는 .jar 대신에 .par 접미사를 가진 하나의 JAR 파일이다. 당신은 또한 .par 파일의 META-INF 폴더 내에 존재하는 한 개의 persistence.xml 파일을 정의해야 한다. 다음은 persistence.xml 파일에 대한 예제이다.

```
<entity-manager>
  <name>manager1</name>
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>java:/DefaultDS</jta-data-source>
  <mapping-file>ormap.xml</mapping-file>
  <jar-file>MyApp.jar</jar-file>
  <class>org.acme.Employee</class>
  <class>org.acme.Person</class>
  <class>org.acme.Address</class>
  <properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
    <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
  </properties>
</entity-manager>
```

name

모든 엔티티 관리자는 한 개의 name을 가져야 한다. 만일 name이 지정되지 않을 경우, .par 파일의 이름에서 .par 접미사를 빼고 사용된다. Hiberante EntityManager의 현재 구현은 당신이 name 요소를 정의하는 것을 필요로 한다.

provider

provider는 EJB 영속 공급자에 대한 전체 수식된 클래스 이름이다. 당신은 그것이 Hibernate에 대한 디폴트가 될 것으로서 이것을 집어 넣지 말아야 한다. 이것은 당신이 EJB 영속에 대한 여러 벤더 구현들을 사용할 때 필요하다.

jta-data-source, non-jta-data-source

이것은 javax.sql.DataSource가 위치해 있는 JNDI 이름이다. 어플리케이션 서버 내에서 사용되지 않을 때 이것은 무시된다. 어플리케이션 외부에서 실행될 때, 당신은 Hibernate에 특정한 프로퍼티들을 가지고 JDBC 커넥션들을 지정해야 한다(아래를 보라).

mapping-file, jar-file, class

class 요소는 당신이 매핑하게 될 전체 수식된 클래스명을 지정한다. 현재, 당신은 당신이 JBoss 어플리케이션 서버 내에서 실행하고 있지 않을 경우 당신이 매핑시켰던 각각의 클래스에 대해 class XML 요소를 지정해야 한다. 명세서는 클래스 이름들, 매핑된 파일들에 대한 자동적인 검출을 허용하지만, 이것은 아직 구현되어 있지 않다. mapping-file과 jar-file은 아직 어느 것도 지원되지 않는다.

properties

properties 요소는 벤더에 특정한 프로퍼티들을 지정하는데 사용된다. 이곳이 당신이 당신의 Hibernate 지정적인 프로퍼티들을 정의하게 될 장소이다. 이곳은 또한 마찬가지로 당신이 JDBC 연결 정보를 지정해야 할 장소이다.

EJB3 명세서는 EntityManagerFactory와 EntityManager에 접근하기 위한 하나의 부트스트랩 절차를 정의하고 있다. 부트스트랩 클래스는 javax.persistence.Persistence이다. 예를 들면.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("manager1");
//or
Map configOverrides = new HashMap();
configuration.put("hibernate.hbm2ddl.auto", "create-drop");
EntityManagerFactory programmaticEmf =
    Persistence.createEntityManagerFactory("manager1", configOverrides);
```

첫 번째 버전은 하나의 공백의 map을 가진 두 번째 버전과 동등하다. map 버전은 당신의 persistence.xml 파일들 내에 정의된 임의의 프로퍼티들에 대해 우선하게 될 오버라이드들의 세트이다. Persistence.createEntityManagerFactory()이 호출될 때, 영속 구현은 ClassLoader.getResource("META-INF/persistence.xml") 메소드를 사용하여 임의의 META-INF/persistence.xml 파일들을 당신의 classpath에서 검색할 것이다. 리소스들에 대한 이 목록으로부터, 그것은 당신이 명령 라인에 지정한 이름이 persistence.xml 파일 내에 정의된 것과 일치하는 하나의 엔티티 관리자를 찾으려고 시도할 것이다.

Hibernate 시스템 레벨의 설정들과는 별개로, Hibernate에서 이용 가능한 모든 프로퍼티들이 persistence.xml 파일의 properties 요소 내에 설정될 수 있거나 당신이 createEntityManagerFactory()에 전달하는 map 내의 하나의 오버라이드로서 설정될 수 있다. 전체 목록에 대한 것은 Hibernate 참조 문서를 참조하길 바란다. 하지만 EJB3 공급자에서만 이용 가능한 쌍의 프로퍼티들이 존재한다.

표 2.1. Hibernate 엔티티 관리자에 특정에 프로퍼티들

| 프로퍼티 이름 | 설명 |
|--|---|
| hibernate.ejb.classcache.<classname> | 클래스에 대한 클래스 캐시 방도 [선택 캐시 영역]. 디폴트는 no cache이고, fully.qualified.classname에 대한 디폴트 영역 캐시(예를 들면. hibernate.ejb.classcache.com.acme.Cat read-write or hibernate.ejb.classcache.com.acme.Cat read-write, MyRegion). |
| hibernate.ejb.collectioncache.<collectionname> | 컬렉션에 대한 컬렉션 캐시 방도 [선택 캐시 영역]. 디폴트는 no cache이고, fully.qualified.classname.role에 대한 디폴트 영역 캐시(예를 들면. hibernate.ejb.classcache.com.acme.Cat read-write or hibernate.ejb.classcache.com.acme.Cat read-write, MyRegion). |

| 프로퍼티 이름 | 설명 |
|-----------------------|---|
| hibernate.ejb.cfgfile | Hibernate를 구성 하는데 사용할 XML 구성 파일 (예를 들면, /hibernate.cfg.xml) |

당신은 동일한 구성에서 xml <class> 선언과 hibernate.ejb.cfgfile 사용법을 혼합할 수 없음을 노트 하라. 한 개의 선언 방 도를 선택 하라. 만일 당신이 <class>를 선택할 경우, Hibernate는 hibernate.ejb.cfgfile파일을 사용하고자 시도할 것이다. 만일 당신이 <class>를 사용하고자 원할 경우, hibernate.ejb.cfgfile이 무시될 것이다. 당신은 당신의 클래스들을 명시적으로 선언해야 하고, 한 개의 .par 아카이브 내에서 클래스들에 대한 자동 검출은 스탠드얼론 방도에서는 구현되지 않음을 기억하라.

다음은 하나의 J2SE 환경에서의 전형적인 구성이다

```
<entity-manager>
  <name>manager1</name>
  <class>org.hibernate.ejb.test.Cat</class>
  <class>org.hibernate.ejb.test.Distributor</class>
  <class>org.hibernate.ejb.test.Item</class>
  <properties>
    <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
    <property name="hibernate.connection.driver_class" value="org.sqlldb.jdbcDriver"/>
    <property name="hibernate.connection.username" value="sa"/>
    <property name="hibernate.connection.password" value=""/>
    <property name="hibernate.connection.url" value="jdbc:sqlldb:."/>
    <property name="hibernate.max_fetch_depth" value="3"/>

    <!-- cache configuration -->
    <property name="hibernate.ejb.classcache.org.hibernate.ejb.test.Item" value="read-write"/>
    <property name="hibernate.ejb.collectioncache.org.hibernate.ejb.test.Item.distributors" value=

    <!-- alternatively to <class> and <property> declarations, you can use a regular hibernate.cfg
    <!-- property name="hibernate.ejb.cfgfile" value="/org/hibernate/ejb/test/hibernate.cfg.xml"/

  </properties>
</entity-manager>
```

2.3. J2SE 환경에서 하나의 EntityManagerFactory와 하나의 EntityManager 획득하기

하나의 엔티티 관리자 팩토리는 하나의 불변의 구성 소유자로서 간주되어야 하여, 그것은 단일 데이터소스를 가리키기 위해 그리고 엔티티들에 대한 하나의 정의된 집합을 매핑하기 위해 정의된다. 이것은 EntityManager들을 생성시키고 관리하기 위한 진입점이다. Persistence 클래스는 하나의 엔티티 관리자 팩토리를 생성시키기 위한 부트스트랩 클래스이다.

```
// Use persistence.xml configuration
EntityManagerFactory emf = Persistence.createEntityManagerFactory("manager1")
EntityManager em = emf.createEntityManager(); // Retrieve a transactional-scoped entity manager
// Work with the EM
em.close();
...
emf.close(); //close at application end
```

하나의 엔티티 관리자 팩토리는 일반적으로 어플리케이션 초기화 시에 생성되고 어플리케이션 종료 시에 닫혀진다. 그것의 생성은 값비싼 과정들이다. Hibernate에 익숙해져 있는 사용자들의 경우, 하나의 엔티티 관리자 팩토리는 한의 세션 팩토리와 매우 흡사하다. 실제로, 하나의 엔티티 관리자 팩토리는 하나의 세션 팩토리의 상단에 있는 하나의 포장 꾸러미(wrapper)이다.

두 가지 종류의 엔티티 관리자들이 존재한다. 트랜잭션-영역의 엔티티 관리자(디폴트)는 각각의 트랜잭션에 대해 하나의 영속 컨텍스트를 생성시키고 파괴시킨다: 달리 말해, 관리되는 엔티티들은 일단 그 트랜잭션이 종료되면 detach(분리)된다. 확장된 엔티티 관리자는 그 엔티티 관리자의 생명주기 동안에 동일한 영속 컨텍스트를 유지한다: 달리말해, 엔티티들은 여전히 두 개의 트랜잭션들 사이에서 관리된다. `emf.createEntityManager()`는 하나의 트랜잭션-영역의 엔티티 관리자를 생성시키고 `emf.createEntityManager(PersistenceContextType.EXTENDED)`는 하나의 확장된 엔티티 관리자를 생성시킨다. 당신은 하나의 Hibernate 세션의 상단에 있는 하나의 작은 포장 꾸러미(wrapper)로서 하나의 엔티티 관리자를 볼 수 있다.

3장. 객체들로 작업하기

3.1. 엔티티 상태들

Hibernate(괄호 내에 있는 비교 가능한 용어들)에서처럼, 하나의 엔티티 인스턴스는 다음 방도들 중 하나이다:

- **New (transient)**: 하나의 엔티티가 `new` 연산자를 사용하여 방금 초기화 되었다면, 그 엔티티는 새로운 것(new)이고, 그것은 하나의 영속 컨텍스트와 연관되어 있지 않다. 그것은 데이터베이스 내에 영속 표상을 갖지 않으며 식별자 값이 할당되지 않았다.
- **Managed (persistent)**: 하나의 관리되는 엔티티 인스턴스는 현재 하나의 영속 컨텍스트와 연관되어 있는 하나의 영속 동일성(identity)을 가진 하나의 인스턴스이다.
- **Detached**: 대개 영속 컨텍스트가 영속 컨텍스트가 달렸거나 인스턴스가 그 컨텍스트로부터 되겨 되었기 때문에, 엔티티 인스턴스는 더 이상 하나의 영속 컨텍스트와 연관되어 있지 않은 하나의 영속 동일성(identity)을 가진 하나의 인스턴스이다.
- **Removed**: 하나의 제거된 엔티티 인스턴스는 하나의 영속 컨텍스트와 연관되어 있지만, 데이터베이스로부터 제거되도록 일정이 잡혀져 있는 하나의 영속 동일성(identity)을 한 개의 인스턴스이다.

`EntityManager` API는 엔티티의 상태를 변경시키는 것, 달리 말해, 객체들을 로드시키고 저장시키는 것을 당신에게 허용해준다. 만일 당신이 SQL 문장들을 관리하는 것이 아니라, 객체 상태 관리에 대해 생각할 경우에 이해하는 것이 더 쉬운 EJB3에 대한 영속성을 발견하게 될 것이다.

3.2. 객체들을 영속화 시키기

일단 당신이 (공통 `new` 연산자를 사용하여) 하나의 새로운 엔티티 인스턴스를 생성시켰다면 그것은 `new` 상태에 있다. 당신은 그것을 하나의 엔티티 관리자에 연관시켜서 그것을 영속화 시킬 수 있다:

```
DomesticCat fritz = new DomesticCat();
frtiz.setColor(Color.GINGER);
frtiz.setSex('M');
frtiz.setName("Fritz");
em.persist(fritz);
```

만일 `DomesticCat` 엔티티 타입이 하나의 산출된 식별자를 갖고 있다면, 그 값은 `persist()`이 호출될 때 그 인스턴스에 연관된다. 만일 그 식별자가 자동적으로 산출되지 않는다면, 어플리케이션에 의해 할당된(대개 `natural`) 키 값이 `persist()`이 호출되기 전에 그 인스턴스에 설정되어야 한다.

3.3. 객체를 로드시키기

엔티티 관리자의 `find()` 메소드로서 엔티티 인스턴스의 식별자 값으로 하나의 엔티티 인스턴스를 로드시켜라:

```
cat = em.find(Cat.class, catId);
```

```
// You may need to wrap the primitive identifiers
long catId = 1234;
em.find( Cat.class, new Long(catId) );
```

몇몇 경우들에서, 당신은 객체 상태를 로드시키는 것을 진정 원하지 않고, 단지 그것에 대한 하나의 참조를 갖고자 원한다(예를 들면 하나의 프락시). 당신은 `getReference()` 메소드를 사용하여 이 참조를 얻을 수 있다. 이것은 부모를 로드시키지 않고서 하나의 자식을 그것의 부모에 링크시키는 것에 특히 유용하다.

```
child = new Child();
child.SetName("Henry");
Parent parent = em.getReference(Parent.class, parentId); //no query to the DB
child.setParent(parent);
em.persist(child);
```

당신은 `em.refresh()` 연산을 사용하여 아무때든지 하나의 엔티티 인스턴스와 그것의 컬렉션들을 다시 로드시킬 수 있다. 이것은 데이터베이스 트리거들이 그 엔티티의 프로퍼티들 중 몇몇을 초기화 시키는데 사용될 때 유용하다. 당신이 임의의 연관들에 대한 하나의 케이스케이드 스타일로서 `REFRESH`를 지정하지 않는 한 오직 엔티티 인스턴스와 그것의 컬렉션들 만이 재생(refresh)된다는 점을 노트하라 :

```
em.persist(cat);
em.flush(); // force the SQL insert and triggers to run
em.refresh(cat); //re-read the state (after the trigger executes)
```

3.4. 객체들을 질의하기

만일 당신이 당신이 찾고 있는 객체들의 식별자 값들을 알지 못할 경우, 당신은 질의를 필요로 한다. Hibernate EntityManager 구현은 HQL (그리고 기타)에 의해 영감을 받았던 사용이 쉽지만 강력한 객체-지향 질의 언어(EJB3-QL)를 지원한다. 두 질의 언어들 양자들 데이터베이스들 사이에 이식 가능하고, (테이블 이름과 컬럼 이름 대신에) 식별자들로서 엔티티 이름과 프로퍼티 이름을 사용한다. 당신은 또한 Java 비즈니스 객체들로의 결과 셋 변환을 위한 EJB3에 있는 옵션 지원으로서, 당신의 데이터베이스의 native SQL로 당신의 질의를 표현할 수도 있다.

3.4.1. 질의들을 실행하기

EJB3QL 질의와 SQL 질의들은 `javax.persistence.Query`의 인스턴스에 의해 표현된다. 이 인스턴스는 파라미터 바인딩, 결과 셋 핸들링, 그리고 질의의 실행을 위한 메소드들을 제공한다. 질의들은 항상 현재의 엔티티 관리자를 사용하여 생성된다:

```
List cats = em.createQuery(
    "select cat from Cat as cat where cat.birthdate < ?")
    .setParameter(0, date, TemporalType.DATE)
    .getResultList();

List mothers = em.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setParameter(0, name)
    .getResultList();

List kittens = em.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .getResultList();
```

```
Cat mother = (Cat) em.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setParameter(0, izi)
    .getSingleResult();
```

하나의 질의는 대개 `getResultList()`를 호출하여 실행된다. 이 메소드는 질의의 귀결되는 인스턴스들을 전체 메모리 내로 로드시킨다. 하나의 질의에 의해 검색된 엔티티 인스턴스들은 영속 상태에 있다. 만일 당신의 질의가 한 개의 객체 만을 반환할 것임을 당신이 알고 있다면 `getSingleResult()` 메소드는 단축형을 제공한다.

3.4.1.1. 투영(Projection)

만일 투영(projection)이 사용될 경우에 EJB3QL query 질의들은 객체들의 튜플들을 반환할 수 있다. 각가의 결과 튜플은 하나의 객체 배열로서 반환된다:

```
Iterator kittensAndMothers = sess.createQuery(
    "select kitten, mother from Cat kitten join kitten.mother mother")
    .getResultList()
    .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten = tuple[0];
    Cat mother = tuple[1];
    ....
}
```

3.4.1.2. 스칼라 결과들

질의들은 하나의 엔티티 alias 가 아닌, select 절 내에 하나의 엔티티에 대한 하나의 특별한 프로퍼티를 지정할 수 있다. 당신은 SQL 집계 함수들도 호출할 수 있다. 반환된 비-트랜잭션 객체들이나 집계 결과들은 "스칼라" 결과들로서 간주되며 영속 상태 내에 있는 엔티티들이 아니다 (달리 말해 그것들은 "읽기 전용"으로 간주된다):

```
Iterator results = em.createQuery(
    "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
    "group by cat.color")
    .list()
    .iterator();

while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    ....
}
```

3.4.1.3. 바인드 파라미터들

명명된 질의 파라미터와 위치 질의 파라미터 모두 지원되며, Query API는 아규먼트들을 바인드 시키는 몇몇 메소드들을 제공한다. JDBC와는 대조적으로, EJB3 명세서는 위치 파라미터들을 1이 아닌, 0에서 시작한다. 명명된 파라미터들은 질의 문자열 내에서 `:paramname` 형식의 식별자들이다. 명명된 파라미터들이 선호되며, 그것들은 읽고 이해하는데 보다 강점을 갖고 보다 쉽다:

```
// Named parameter (preferred)
Query q = em.createQuery("select cat from DomesticCat cat where cat.name = :name");
q.setParameter("name", "Fritz");
List cats = q.getResultList();
```

```
// Positional parameter
Query q = em.createQuery("select cat from DomesticCat cat where cat.name = ?");
q.setParameter(0, "Izi");
List cats = q.getResultList();

// Named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = em.createQuery("select cat from DomesticCat cat where cat.name in (:namesList)");
q.setParameter("namesList", names);
List cats = q.list();
```

3.4.1.4. 쪽매김

만일 당신이 당신의 결과 셋에 경계점들(당신이 검색하고자 원하는 행들의 최대 개수 그리고/또는 당신이 검색하고자 원하는 첫 행의 최대 개수)를 지정할 필요가 있다면, 다음 메소드들을 사용하라:

```
Query q = em.createQuery("select cat from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list(); //return cats from the 20th position to 29th
```

Hibernate는 이 limit 질의를 당신의 DBMS의 native SQL로 변환시키는 방법을 알고 있다.

3.4.1.5. 명명된 질의들을 외재화 시키기

당신은 또한 주해들(annotations)을 통해 명명된 질의들을 정의할 수도 있다:

```
@javax.persistence.NamedQuery(name="eg.DomesticCat.by.name.and.minimum.weight",
    queryString="select cat from eg.DomesticCat as cat where cat.name = ? and cat.weight > ?")
```

Parameters are bound programatically to the named query, before it is executed:

```
Query q = em.createNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

실제 프로그램 코드는 사용되는 질의 언어에 독립적이며, 당신은 또한 메타 데이터로 native SQL 질의들을 정의할 수도 있거나 그것들을 XML 매핑 파일들 내에 위치지움으로써 Hibernate의 native 편의설비들을 사용할 수도 있음을 노트하라.

3.4.1.6. Native 질의들

당신은 `createNativeQuery()`를 사용하여 하나의 질의를 표현할 수도 있고 Hibernate로 하여금 JDBC 결과 셋들로부터 비즈니스 객체들로의 매핑을 처리하도록 할 수 있다. `@SqlResultSetMapping` (SQL 결과셋 매핑을 매핑시키는 방법에 대해서는 Hibernate Annotations 참조 문서를 보길 바란다) 또는 엔티티 매핑(질의 결과의 컬럼 이름들이 엔티티 매핑 내에 선언된 이름들과 동일할 경우; 모든 엔티티 컬럼들이 이 메커니즘이 동작하도록 반환 되어야 함을 기억하라)을 사용하라:

```
@SqlResultSetMapping(name="getItem", entities =
    @EntityResult(name="org.hibernate.ejb.test.Item", fields= {
        @FieldResult(name="name", column="itemname"),
        @FieldResult(name="descr", column="itemdescription")
    })
)
```

```

Query q = em.createNativeQuery("select name as itemname, descr as itemdescription from Item", "getItem");
item = (Item) q.getSingleResult(); //from a resultset

Query q = em.createNativeQuery("select * from Item", Item.class);
item = (Item) q.getSingleResult(); //from a class columns names match the mapping

```

참고

native 질의들에 대한 현재 구현은 스칼라 결과들을 지원하지 않으며, 오직 트랜잭션적인 엔티티들만을 지원한다.

3.4.1.7. 질의 힌트들

(대개 퍼포먼스 최적화를 위한) 질의 힌트들은 구현 지정적이다. 힌트들은 `query.setHint(String name, Object value)`를 사용하여 선언된다. 이것들은 SQL 질의 힌트들이 아님을 노트하라! Hibernate EJB3 구현은 다음 질의 힌트들을 제공한다:

표 3.1. Hibernate 질의 힌트들

| 힌트 | 설명 |
|--|--|
| <code>org.hibernate.timeout</code> | 초 단위의 질의 타임아웃 (예를 들면, <code>new Integer(10)</code>) |
| <code>org.hibernate.fetchSize</code> | 라운드트립 당 JDBC 드라이버에 의해 페치되는 행들의 개수 (예를 들면, <code>new Integer(50)</code>) |
| <code>org.hibernate.comment</code> | DBA에게 유용한 주석을 SQL 질의에 추가한다 (예를 들면, <code>new String("fetch all orders in 1 statement")</code>) |
| <code>org.hibernate.cacheable</code> | 질의가 캐시 가능한지 여부 (예를 들면, <code>new Boolean(true)</code>), 디폴트는 <code>false</code> |
| <code>org.hibernate.cacheMode</code> | 이 질의에 대한 캐시 모드를 오버라이드 시킨다 (예를 들면, <code>CacheMode.REFRESH</code>) |
| <code>org.hibernate.cacheRegion</code> | 이 질의의 캐시 영역 (예를 들면, <code>new String("regionName")</code>) |
| <code>org.hibernate.readOnly</code> | Hibernate가 그것들에 대해 <code>dirty-check</code> 를 결코 행하지 않거나 변경들을 결코 영속화 시키지 않는 곳에서 이 질의에 의해 검색된 엔티티들은 읽기 전용 모드로 로드될 것이다(예를 들면, <code>new Boolean(true)</code>), 디폴트는 <code>false</code> |

추가 정보는 Hibernate 참조 문서를 참조하길 바란다.

3.5. 영속 객체들을 수정하기

트랜잭션 관리되는 인스턴스들(예를 들면, 엔티티 관리자에 의해 로드되고, 저장되고, 생성되거나 질

의되는 객체들)은 어플리케이션에 의해 처리될 수 있고 영속 상태에 대한 임의의 변경들은 Entity 관리자가 flush될 때 영속화 될 것이다(이 장의 뒷 부분에서 논의됨). 당신의 수정들을 영속화 시키기 위한 어떤 특별한 메소드 호출이 필요하지 않다. 하나의 엔티티 인스턴스의 상태를 업데이트시키는 간단한 방법은 영속 컨텍스트가 열려져 있는 동안에 그것을 find() 하고 나서, 그것을 직접 처리하는 것이다:

```
Cat cat = em.find( Cat.class, new Long(69) );
cat.setName("PK");
em.flush(); // changes to cat are automatically detected and persisted
```

때때로 이 프로그래밍 모형은 불충분하다. 왜냐하면 그것은 동일 세션 내에서 (하나의 객체를 로드시키는) 하나의 SQL SELECT와 (그것의 업데이트된 상태를 영속화 시키기 위한) 하나의 SQL UPDATE 양자를 필요로 할 것이기 때문이다. 그러므로 Hibernate는 분리된 인스턴스들(detached instances)를 사용하여 대안적인 접근법을 제공한다.

3.6. 분리된 객체들(detached objects)을 수정하기

많은 어플리케이션들은 하나의 트랜잭션 내에서 하나의 객체를 검색하고 그것을 처리를 위한 프리젠테이션 계층으로 전송하고, 나중에 새로운 트랜잭션 내에서 변경들을 저장하는 것을 필요로 한다. 두 트랜잭션들 사이의 사용자 생각 시간과 대기 시간이 중요해질 수 있다. 고도의 동시성 환경에서 이런 종류의 접근법을 사용하는 어플리케이션들은 대개 "장기간의" 작업 단위에 대한 격리를 보장하는데 버전화 된 데이터를 사용한다.

EJB3 명세서들은 EntityManager.merge() 메소드를 사용하여 분리된 인스턴스들(detached instances)에 대해 행해진 수정들의 영속화를 제공함으로써 이 개발 모형을 지원한다:

```
// in the first entity manager
Cat cat = firstEntityManager.find(Cat.class, catId);
Cat potentialMate = new Cat();
firstEntityManager.persist(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new entity manager
secondEntityManager.merge(cat); // update cat
secondEntityManager.merge(mate); // update mate
```

merge() 메소드는 영속 컨텍스트의 상태를 고려하지 않고서 분리된 인스턴스들(detached instances)에 대해 행해진 수정들을 대응하는 관리되는 인스턴스(managed instance) 속으로 병합시킨다. 달리 말해, 병합된 객체들 상태는 그것이 존재할 경우에 영속 컨텍스트 내에 있는 영속 엔티티 상태를 오버라이드 시킨다. 어플리케이션은 그것이 분리된 인스턴스들의 상태가 또한 영속화 되기를 원하는 경우에만 주어진 분리된 인스턴스로부터 도달가능한 분리된 인스턴스들(detached instances)을 하나씩 merge() 시킬 것이다. 이것은 전이 영속(transitive persistence)을 사용하여 연관된 엔티티들과 콜렉션들로 케이스케이드 될 수 있으며, Transitive persistence를 보라.

3.7. 자동적인 상태 검출

merge 연산은 분리된 인스턴스에 대한 병합이 insert로 귀결되어야 하는지 update로 귀결되어야 하는지 여부를 자동적으로 검출해낼 만큼 충분히 영리하다. 달리 말해, 당신은 하나의 새로운 인스턴스를 merge()에 전달하는 것(과 하나의 분리된 인스턴스를 전달하지 않는 것)에 대해 걱정하지 않아도

되며, 엔티티 관리자가 당신을 위해 이것을 해석할 것이다:

```
// In the first entity manager
Cat cat = firstEntityManager.find(Cat.class, catID);

// In a higher layer of the application, detached
Cat mate = new Cat();
cat.setMate(mate);

// Later, in a new entity manager
secondEntityManager.merge(cat); // update existing state
secondEntityManager.merge(mate); // save the new instance
```

`merge()`의 사용법 및 의미는 신규 사용자들에게는 혼동스러워 보인다. 먼저 당신이 하나의 엔티티 관리자 내에서 로드된 객체 상태를 또 다른 엔티티 관리자 내에서 사용하려고 시도하는 한, 당신은 `merge()`를 전혀 사용할 필요가 없을 것이다. 몇몇 전체 어플리케이션들은 이 메소드들을 전혀 사용하지 않을 것이다.

대개 `merge()`는 다음 시나리오들 내에서 사용된다:

- 어플리케이션이 첫 번째 엔티티 관리자 내에 한 개의 객체를 로드시킨다
- 그 객체는 프리젠테이션 계층으로 전달된다
- 몇몇 수정들이 그 객체에 대해 행해진다
- 그 객체는 다시 비즈니스 로직 계층으로 전달된다
- 어플리케이션은 두 번째 엔티티 관리자 내에서 `merge()`를 호출하여 이들 수정들을 영속화 시킨다

다음은 `merge()`의 정확한 의미이다:

- 만일 영속 컨텍스트에 현재 연관된 동일한 식별자를 가진 하나의 관리되는 인스턴스가 존재한다면, 주어진 객체의 상태를 관리되는 인스턴스 상으로 복사하라
- 만일 영속 컨텍스트와 현재 연관된 관리되는 인스턴스가 존재하지 않을 경우, 데이터베이스로부터 그것을 로드시키거나 하나의 새로운 관리되는 인스턴스를 생성시키려고 시도하라
- 관리되는 인스턴스가 반환된다
- 주어진 인스턴스는 영속 컨텍스트에 연관되지 않으며, 그것은 분리된 채로 (`detached` 상태로) 유지되고 대개 폐기된다

병합 대 `saveOrUpdate/saveOrUpdateCopy`

EJB3에서 병합(Merging)은 native Hibernate에서 `saveOrUpdateCopy()` 메소드와 유사하다. 하지만 그것은 `saveOrUpdate()` 메소드와 동일하지 않으며, 주어진 인스턴스는 영속 컨텍스트에 다시첨부되지 않지만, 하나의 관리되는 인스턴스가 `merge()` 메소드에 의해 반환된다.

3.8. 관리되는 객체들을 삭제하기

`EntityManager.remove()`는 데이터베이스로부터 객체들 상태를 제거할 것이다. 물론 당신의 어플리케이션은 여전히 하나의 삭제된 객체에 대한 하나의 참조를 소유하고 있을 수 있다. 당신은 `remove()`를

하나의 영속 인스턴스를 다시 새롭게(별칭은 transient로) 만드는 것으로서 간주할 수 있다. 그것은 분리된(detached) 것이 아니고, 병합은 하나의 insert로 귀결될 것이다.

3.9. 영속 컨텍스트를 Flush시키기

시간이 지남에 따라 엔티티 관리자는 메모리 내에 유지되는 객체들의 상태를 데이터 저장소와 동기화 시키는데 필요한 SQL DML 문장들을 실행시킬 것이다. 이 과정, 즉 flush는 디폴트로(이것은 Hibernate에 특징적인 것이고 명세서에 정의된 것은 아니다) 다음 시점들에서 일어난다:

- 질의 실행 전
- `javax.persistence.EntityTransaction.commit()` 시에
- `EntityManager.flush()`가 호출될 때

SQL 문장들은 다음 순서로 일어난다

- `EntityManager.persist()`를 사용하여 저장되었던 대응하는 객체들과 동일한 순서로 모든 엔티티 삽입들
- 모든 엔티티 업데이트들
- 모든 컬렉션 삭제들
- 모든 컬렉션 요소 삭제들, 업데이트들, 삽입들
- 모든 컬렉션 삽입들
- `EntityManager.remove()`를 사용하여 삭제되었던 대응하는 객체들과 동일한 순서로 모든 엔티티 삭제들

(예외: 어플리케이션에 의해 할당된 식별자들을 사용하는 엔티티 인스턴스들은 그것들이 저장될 때 insert된다.)

당신이 명시적으로 `flush()` 시키는 때를 제외하면, 엔티티 관리자가 JDBC 호출들을 실행시키는 시점에 대한 절대적인 보장은 존재하지 않으며, 오직 그것들이 실행되는 순서 만이 보장된다. 하지만 Hibernate는 `Query.getResultList()/Query.getSingleResult()`가 손상된 데이터를 결코 반환하지 않을 것임을 보장할 것이거나; 그것들은 잘못된 데이터를 반환하지 않을 것이다.

flush가 자주 발생하지 않도록 하기 위해 디폴트 특징을 변경시키는 것이 가능하다. 엔티티 관리자에 대한 `FlushModeType`은 세 개의 다른 모드들을 정의한다: 약속(commit) 시에만 flush 시키는 모드, 설명된 루틴을 사용한 자동적인 flush 모드, 또는 `flush()`가 명시적으로 호출되지 않는 한 결코 flush 시키지 않는 모드. 마지막 모드는 장기간 실행되는 확장된 영속 컨텍스트들에 유용하며, 여기서 컨텍스트와 그것의 엔티티 관리자가 열려진 채로 유지된다(그러나 JDBC 데이터소스로부터 연결 해제된다). TODO: 연결해제에 대한 링크를 추가할 것이지만, 연결해제는 구현 지정적이다.

```
em = emf.createEntityManager();
Transaction tx = em.getTransaction().begin();
em.setFlushMode(FlushModeType.COMMIT); // allow queries to return stale state

Cat izi = em.find(Cat.class, id);
izi.setName(iznizi);
```

```
// might return stale data
em.createQuery("from Cat as cat left outer join cat.kittens kitten").getResultList();

// change to izi is not flushed!
...
em.getTransaction().commit(); // flush occurs
```

flush 동안에, 하나의 예외상황이 발생할 수도 있다 (예를 들어, 하나의 DML 연산이 제약(컨스트레인트)을 위반할 경우). TODO: 예외상황 핸들링에 대한 링크 추가할 것.

3.10. 전이 영속(Transitive persistence)

개별 객체들을 저장하고, 삭제하고, 또는 재첨부하는 것은 특히 당신이 연관 객체들의 그래프를 다룰 때 꽤 귀찮다. 하나의 공통된 경우는 하나의 부모/자식 관계이다. 다음 예제를 검토하자:

만일 부모/자식 관계에서 자식이 값 타입일 경우(예를 들면, 주소 또는 문자열을 가진 하나의 컬렉션), 그것들의 생명주기는 부모에 의존할 것이고 상태 변경들에 대한 편리한 "케스케이딩"에 더 이상의 액션이 없을 것이다. 부모가 영속화 될 때, 값-타입의 자식 객체들도 마찬가지로 영속화 되고, 부모가 제거될 때, 자식이 제거될 것이다. 기타 마찬가지로. 이것은 심지어 컬렉션으로부터 하나의 자식을 제거하는 것과 같은 연산들에 대해서도 동작한다; Hibernate는 이것을 검출해낼 것이고, 값-타입의 객체들이 공유된 참조들을 가질 수 없기 때문에, 데이터베이스로부터 자식을 제거할 것이다.

이제 값-타입이 아닌, 엔티티들인 부모 객체와 자식 객체에 대해 동일한 시나리오를 검토하자(예를 들면, 카테고리들과 아이템들, 또는 부모 고양이와 자식 고양이). 엔티티들은 그것들 자신의 생명주기를 갖고, 공유된 참조들을 지원하고(따라서 컬렉션으로부터 하나의 엔티티를 제거하는 것은 그것이 삭제될 수 있음을 의미하지 않는다), 하나의 엔티티로부터 임의의 다른 연관된 엔티티들로의 상태에 대한 케스케이딩이 디폴트로 존재하지 않는다. EJB3 명세서는 도달가능성(reachability)에 따른 영속성을 요구하지 않는다. 그것은 Hibernate에서 처음 보았듯이, 전이 영속에 대한 보다 유연한 모형을 지원한다.

엔티티 관리자의 각각의 기본 연산들- `persist()`, `merge()`, `remove()`, `refresh()`을 포함하여 - 의 경우, 하나의 대응하는 케스케이드 스타일이 존재한다. 케스케이드 스타일들은 각각 PERSIST, MERGE, REMOVE, REFRESH로 명명된다. 만일 당신이 하나의 연산이 연관된 엔티티 (또는 엔티티들을 가진 컬렉션)에 대해 케스케이드되기를 원한다면, 당신은 연관 주해(association annotation) 속에 그것을 진술해야 한다:

```
@OneToOne(cascade=CascadeType.PERSIST)
```

Cascading options can be combined:

```
@OneToOne(cascade= { CascadeType.PERSIST, CascadeType.REMOVE, CascadeType.REFRESH } )
```

당신이 하나의 특별한 연관에 대해 모든 연산들이 케스케이드되어야 함을 지정하는데 `CascadeType.ALL`을 사용할 수도 있다. 디폴트로 연산은 케스케이드되지 않음을 기억하라.

Hibernate는 보다 많은 native 케스케이딩 옵션들을 제공하는데, 추가 정보들은 Hibernate Annotations 매뉴얼과 Hibernate 참조 안내서를 참조하길 바란다.

권장사항들:

- `@ManyToOne` 또는 `@ManyToMany` 연관에 대해 케스케이드를 사용 가능하도록 하는 것은 대개 의미가

없다. 케스케이드는 흔히 @OneToOne 연관과 @OneToMany 연관에 유용하다.

- 만일 자식 객체의 생명주기가 부모 객체의 생명주기에 의해 제한되어 있다면, `CascadeType.ALL`과 `org.hibernate.annotations.CascadeType.DELETE_ORPHAN`을 지정하여 부모를 하나의 완전한 생명주기 객체로 만들어라. (orphan delete의 의미에 대해서는 Hibernate 참조 안내서를 참조하길 바란다)
- 그 밖의 경우, 당신은 케스케이드를 전혀 필요로 하지 않을 수 있다. 그러나 당신이 자주 동일 트랜잭션 내에서 부모와 자식에 대해 함께 동작되어야 할 것이라고 당신이 생각할 경우, 그리고 당신 스스로 어떤 타입화를 저장하고자 원한다고 당신이 생각할 경우, `cascade={PERSIST, MERGE}` 사용을 고려하라. 이들 옵션들은 many-to-many 연관에 대해서조차 의미있게 만들 수 있다.

4장. 트랜잭션들과 동시성

Hibernate 엔티티 관리자와 동시성 제어에 대해 가장 중요한 점은 이해하기가 매우 쉽다는 점이다. Hibernate 엔티티 관리자는 어떤 추가적인 잠금 행위 없이 JDBC 연결들과 JTA 리소스들을 직접 사용한다. 우리는 당신이 당신의 데이터베이스 관리 시스템의 JDBC, ANSI, 그리고 트랜잭션 격리에 약간의 시간을 소비하기를 매우 권장한다. Hibernate 엔티티 관리자는 오직 자동적인 버전을 추가시키지만 메모리 내에서 객체들을 잠그거나 당신의 데이터베이스 트랜잭션들의 격기 레벨을 변경시키지 않는다. 기본적으로 당신이 당신의 데이터베이스 리소스들에 대해 직접적인 JDBC(또는 JTA/CMT)를 사용하는 것처럼 Hibernate 엔티티 관리자를 사용하라.

우리는 `EntityManagerFactory`, 그리고 `EntityManager`, 뿐만 아니라 데이터베이스 트랜잭션들과 장기 간에 걸친 작업 단위들의 입장으로 Hibernate에서의 동시성 제어에 대한 논의를 시작한다.

이 장에서, 그리고 명시적으로 표현되지 않는 한, 우리는 엔티티 관리자와 영속 컨텍스트의 개념을 섞어 쓰고 동이랴게 사용할 것이다. 엔티티 관리자는 하나의 API이고 프로그래밍 객체이고, 영속 컨텍스트는 영역(scope)에 대한 정의이다. 하지만 본질적으로 차이가 있음을 염두에 두라. 영속 컨텍스트는 대개 J2EE에서 하나의 JTA 트랜잭션에 묶이고, 하나의 영속 컨텍스트는 당신이 하나의 확장된 엔티티 관리자를 사용하지 않는 한, 트랜잭션 경계들에서(트랜잭션 영역에서) 시작되고 끝난다. 추가 정보는 1.2.3절. “영속 컨텍스트 영역”를 참조하길 바란다.

4.1. 엔티티 관리자와 트랜잭션 영역들

하나의 `EntityManagerFactory`는 생성하기가 값비싸고, 모든 어플리케이션 쓰레드들에 의해 공유되도록 고안된 쓰레드안전한 객체이다. 그것은 대개 어플리케이션의 시작 시에 한 번 생성된다.

하나의 `EntityManager`는 비용이 들지 않는, 한번만 사용될 쓰레드 안전하지 않은 객체이며, 하나의 단일 작업 단위이고, 그런 다음 폐기된다. 하나의 `EntityManager`는 그것이 필요하지 않는 한 하나의 JDBC Connection(또는 하나의 `Datasource`)를 획득하지 않을 것이어서, 당신이 반드시 데이터 액세스가 하나의 특별한 요청에 서비스하는데 필요하게 될 것임을 확실히 하지 않은 경우조차도 하나의 `EntityManager`를 안전하게 열고 닫을 수 있다. (이것은 당신이 요청 인터셉트를 사용하여 다음 패턴들 중 몇몇을 구현하자마자 중요하게 된다.)

이 그림을 완성하기 위해 당신은 또한 데이터베이스 트랜잭션들에 대해 생각해야 한다. 하나의 데이터베이스 트랜잭션은 데이터베이스에서의 잠금 쟁탈을 감소시키기 위해 가능한 짧아야 한다. 긴 데이터베이스 트랜잭션들은 당신의 어플리케이션이 고도의 동시성 로드로 비례조정하는 것을 방해할 것이다.

작업 단위의 영역이란 무엇인가? 하나의 Hibernate `EntityManager`는 여러 개의 데이터베이스 트랜잭션들에 걸칠 수 있는가? 또는 이것은 하나의 one-to-one 관계의 영역들인가? 당신은 언제 하나의 `Session`을 열고 닫는가? 그리고 당신은 데이터베이스 트랜잭션 경계들을 어떻게 구별하는가?

4.1.1. 작업 단위

먼저, `entitymanager-per-operation` 안티패턴을 사용하지 말라. 즉 하나의 단일 쓰레드 내에서 모든 간단한 데이터베이스 호출에 대해 하나의 `EntityManager`를 열고 닫지 말라! 물론 동일한 것이 데이터베이스 트랜잭션들에 대해서도 참이다. 하나의 어플리케이션 내에서 데이터베이스 호출들은 하나의 계획된 순서를 사용하여 행해지며, 그것들은 자동적인 작업 단위들 속으로 그룹지어진다. (이것은 또한 어플리케이션 내에 있지 않는 한 모든 단일 SQL 문장 후에 자동-확약(auto-commit)됨을 의미하

며, 이 모드는 특별한 목적의 SQL 콘솔 작업으로 고안되어 있다.)

하나의 다중-사용자 클라이언트/서버 어플리케이션에서 가장 공통된 패턴은 entitymanager-per-request(요청 별 엔티티관리자)이다. 이 모형에서 클라이언트로부터의 하나의 요청은 (EJB3 영속 계층이 실행되는) 서버로 전송되고, 하나의 새로운 EntityManager가 열리고, 모든 데이터베이스 연산들이 이 작업 단위 내에서 실행된다. 일단 그 작업이 완료되었다면(그리고 클라이언트를 위한 응답이 준비되었다면), 영속 컨텍스트가 flush되고 닫히지며, 마찬가지로 엔티티 객체도 닫혀진다. 당신은 또한 클라이언트 요청에 서비스하는데 하나의 단일 데이터베이스 트랜잭션을 사용할 것이다. 둘 사이의 관계는 one-to-one이고 이 모형은 많은 어플리케이션들에 대해 완전하게 들어맞는다.

이것은 J2EE 환경(JTA 경계지워진, 트랜잭션-영역의 영속 컨텍스트)에서는 디폴트 EJB3 영속 모형이다; 삽입된(또는 록업된) 엔티티 관리자들은 하나의 특별한 JTA 트랜잭션에 대해 동일한 영속 컨텍스트를 공유한다. EJB3의 아름다움은 당신이 더 이상 그것을 돌보지 않아도 되고 단지 엔티티 관리자를 통한 데이터 접근과 완전하게 직교하는 것으로서 세션 빈즈 상의 트랜잭션 영역에 대한 경계설정을 지켜보는 것이다.

난제는 EJB3 컨테이너의 외부에서 이(그리고 다른)행위에 대한 구현이다: EntityManager와 리소스-로컬 트랜잭션이 정확하게 시작되고 종료되어야 할 뿐만 아니라, 그것들은 또한 데이터 접근 연산들에 대해 접근가능해야 한다. 단위 작업에 대한 경계설정은 이론적으로 하나의 요청이 비-EJB3 컨테이너 서버에 도달할 때 실행되고 응답이 전송되기 전에 인터셉터 (예를 들어, 당신이 하나의 스탠드얼론 서블릿 컨테이너를 사용하고 있을 경우에는 ServletFilter)를 사용하여 구현되어야 한다. 우리는 하나의 ThreadLocal 변수를 사용하여, 요청에 서비스하는 스레드에 EntityManager를 묶을 것을 권장한다. 이것은 이 스레드 내에서 실행되는 모든 코드 내에서 (하나의 static 변수에 접근하는 것과 같은) 쉬운 접근을 허용해준다. 당신이 선택하는 데이터베이스 트랜잭션 경계설정 메커니즘에 따라, 당신은 또한 ThreadLocal 변수 내에 트랜잭션 컨텍스트를 유지할 수도 있다. 이것을 위한 구현 패턴들은 Hibernate 공동체 내에서 ThreadLocal Session 그리고 Open Session in View 으로 알려져 있다. 당신은 이 패턴을 구현하기 위해서 Hibernate 참조 문서 내에 보여진 HibernateUtil을 쉽게 확장할 수 있으며, 당신은 어떤 외부적인 소프트웨어를 필요로 하지 않는다(사실 그것은 매우 평범하다). 물론 당신은 하나의 인터셉터를 구현하고 그것을 당신의 환경에 셋업하는 방법을 찾아야 한다. 팁들과 예제들은 Hibernate 웹 사이트를 보라. 다시금 당신의 첫 번째 선택은 당연히 EJB3 컨테이너-아마 JBoss 어플리케이션 서버와 같은 하나의 경량급의 모듈 컨테이너-임을 염두에 두라.

4.1.2. 장기간의 작업 단위

entitymanager-per-request(요청 별 엔티티관리자) 패턴은 당신이 작업 단위들을 설계하는데 사용할 수 있는 유일하게 유용한 개념은 아니다. 많은 비즈니스 프로세스들은 사용자와 데이터베이스 접근들 사이를 등가적으로 고속화 시키는(interleave) 하나의 전체 일련의 상호작용들을 필요로 한다. 웹과 엔터프라이즈 어플리케이션들에서 하나의 데이터베이스 트랜잭션이 장기간의 대기 시간을 가진 사용자 상호작용을 요청들 사이에 걸치도록 하는 것을 수용하기가 불가능하다. 다음 예제를 검토하자:

- 대화상자의 첫 번째 화면이 열리고, 사용자에게 보여진 데이터가 하나의 특별한 EntityManager와 리소스-로컬 트랜잭션 내에 로드되었다. 사용자가 분리된(detached) 객체들을 수정하는 것이 자유롭다.
- 사용자는 5분 후에 "저장"을 클릭하고 그의 수정들이 영속화 되기를 기대한다; 사용자는 또한 그가 이 정보를 편집하는 유일한 개인이었기를 그리고 충돌하는 수정이 발생할 수 없기를 기대한다.

우리는 사용자의 관점에서 이 작업 단위, 하나의 장기간 실행되는 어플리케이션 트랜잭션을 호출한다. 당신이 당신의 어플리케이션에서 이것을 구현할 수 있는 많은 방법들이 존재한다.

첫 번째 native 구현은 동시 수정을 방지하고, 격리와 원자성을 보장하기 위해 데이터베이스 내에서 잠금을 유지한채로 사용자 생각 시간 동안에 EntityManager와 데이터베이스 트랜잭션을 유지시킬 수 있다. 이것은 잠금 쟁투가 어플리케이션이 많은 동시 사용자들에 대해 비례조정 하는 것을 어플리케이션에게 허용하지 않기 때문에, 이것은 물론 하나의 안티-패턴이고, 하나의 pessimistic 접근이다.

명백하게, 우리는 어플리케이션 트랜잭션을 구현하기 위해 몇몇 데이터베이스 트랜잭션들을 사용해야 한다. 이 경우에, 비즈니스 프로세스들에 대한 격리를 유지하는 것은 어플리케이션 tier의 부분적인 책임이 된다. 하나의 단일 어플리케이션 트랜잭션은 대개 여러 개의 데이터베이스 트랜잭션들에 걸친다. 그것은 이들 데이터베이스 트랜잭션들 중 하나(마지막 트랜잭션) 만이 업데이트된 데이터를 저장하며, 모든 다른 트랜잭션들은 단순히 데이터를 읽는다(예를 들면, 여러 개의 요청/응답 주기에 걸치는 마법사 스타일의 대화상자에서). 이것은 특히 당신이 EJB3 영속 관리자와 영속 컨텍스트 특징들을 사용할 경우에, 들리는 것보다 구현하기가 더 쉽다:

- 자동적인 버전화 - 하나의 엔티티 관리자는 당신을 위해 자동적인 optimistic 동시성 제어를 행할 수 있으며, 그것은 사용자 생각 시간 동안 하나의 동시적인 수정이 일어났는지 여부를 (대개 최종 리소스-로컬 트랜잭션 내에서 데이터를 업데이트할 때 버전 번호들이나 timestamp들을 비교함으로써)자동적으로 검출할 수 있다.
- 분리된(Detached) 엔티티들 - 만일 당신이 이미 논의했던 entity-per-request 패턴을 사용하기로 결정할 경우, 모든 로드된 인스턴스들은 사용자 생각 시간 동안에 분리된(detached) 상태에 있을 것이다. 엔티티 관리자는 분리된(detached) 상태(수정된 상태)를 병합시키고 수정들을 영속화 시키는 것을 당신에게 허용해 주며, 그 패턴은 entitymanager-per-request-with-detached-entities (분리된 엔티티들을 가진 요청 별 엔티티 관리자)로 명명된다. 자동적인 버전화는 동시적인 수정들을 격리시키는데 사용된다.
- 확장된 엔티티 관리자 - Hibernate 엔티티 관리자는 데이터베이스 트랜잭션이 확약된 후에 기본 JDBC 연결로부터 연결해제될 수 있고, 하나의 새로운 클라이언트 요청이 발생할 때 다시 연결될 수 있다. 이 패턴은 entitymanager-per-application-transaction (어플리케이션 트랜잭션 별 엔티티관리자)로 알려져 있고 심지어 불필요하게 병합시킨다. 하나의 확장된 영속 컨텍스트는 임의의 행해진 수정을 수집하고 기억할 책임이 있다. 자동적인 버전화는 동시적인 수정들을 격리시키는데 사용된다.

Both entitymanager-per-request-with-detached-objects 그리고 entitymanager-per-application-transaction 양자는 장점들과 단점들을 갖고 있으며, 우리는 이 장의 뒷 부분에서 optimistic 동시성 제어 문맥에서 그것들을 논의한다.

TODO: 이 노트는 아마 나중에 들어올 것이다.

권장되는 접근법

entitymanager-per-application-transaction은 확장된 엔티티 관리자와 그것의 영속 컨텍스트의 상태를 유지하는 하나의 상태있는 세션 빈(bean)을 사용하여 EJB3 컨테이너 내에서 상자밖에서 이용 가능하다. 이 확장된 영속 컨텍스트를 이용 가능하도록 만들기 위해, 당신의 빈(bean)에 @PersistenceContext(EXTENDED)로서 주해를 달아라.

TODO: 내가 행하고자 가정한 것은 정확히 무엇인가? 이것은 불완전하다...

4.1.3. 객체 동일성(identity) 고찰

하나의 어플리케이션은 두 개의 다른 영속 컨텍스트들 내에서 동일한 영속 상태에 동시적으로 접근할 수도 있다. 하지만 하나의 관리되는 클래스의 인스턴스는 결코 두 개의 영속 컨텍스트들 사이에서 공유되지 않는다. 그러므로 동일성(identity)에 대한 두 개의 다른 개념들이 존재한다:

데이터베이스 Identity

```
foo.getId().equals( bar.getId() )
```

JVM Identity

```
foo==bar
```

하나의 특별한 영속 컨텍스트에 첨부된 객체들의 경우(예를 들면 `EntityManager`의 영역에서) 두 개의 개념들은 같고, 데이터베이스에 대한 JVM identity는 Hibernate Entity Manager에 의해 보장된다. 하지만, 어플리케이션이 두 개의 다른 영속 컨텍스트에서 "동일한" (영속 동일성(identity)) 비즈니스 객체에 동시에 접근하는 동안, 두 개의 인스턴스들은 실제로 "다름" 것이다(JVM 동일성(identity)). 충돌은 optimistic 접근법을 사용하여 flush/commit 시에 (자동적인 버전화)를 사용하여 해결된다.

이 접근법은 동시성을 처리하는 것을 Hibernate와 데이터베이스에게 위임시킨다; 그것은 또한 최상의 비례조정가능성을 제공하는데, 왜냐하면 단일 스레드의 작업 단위에서 동일성(identity)를 보장하는 것은 값비싼 잠금이나 다른 동기화 수단들을 필요로 하지 않기 때문이다. 어플리케이션이 `EntityManager`에 의한 하나의 단일 스레드에 달라붙어 있는 한, 어플리케이션은 결코 어떤 비즈니스 객체 상에 동기화 될 필요가 없다. 하나의 영속 컨텍스트 내에서, 어플리케이션은 엔티티들을 비교하는데 안전하게 `==`을 사용할 수 있다.

하지만 하나의 영속 컨텍스트 외부에서 `==`를 사용하는 어플리케이션은 예기치 않은 결과들을 보게 될 수 있다. 이것은 심지어 어떤 예기치 않은 장소들에서, 예를 들면 당신이 두 개의 분리된 (detached) 객체들을 동일한 set 속에 집어 넣을 경우에 일어날 수 있다. 둘 다 동일한 데이터베이스 동일성(identity)을 갖지만(예를 들면, 그것들은 동일한 행을 나타낸다), JVM 동일성(identity)은 정의를 상 분리된(detached) 상태에서는 보장되지 않는다. 개발자는 영속 클래스들 내에서 `equals()`와 `hashCode()` 메소드들을 오버라이드 시켜야 하고, 객체 서로 같음(equality, 동등성)에 대한 그 자신의 개념을 구현해야 한다. 한 가지 단서가 존재한다: 서로같음(equality)을 구현하는데 데이터베이스 식별자를 결코 사용하지 말 것이며, 하나의 비즈니스 키, 유일 속성성들, 대개 불변 속성들의 조합을 사용하라. 하나의 전이(transient) 엔티티가 영속화 될 경우에 데이터베이스 식별자가 변경될 것이다(`persist()` 연산에 대한 계약을 보라). (대개 분리된(detached) 인스턴스들과 함께) 전이(transient) 인스턴스가 하나의 set 내에 보관될 경우에, hashcode 변경은 set에 대한 계약을 깨뜨린다. 좋은 비즈니스 키들을 위한 속성들은 데이터베이스 프라이머리 키들 만큼 영속적이지 않아야 하며, 당신은 오직 객체들이 동일한 set 내에 존재하는 동안만 안정성을 보장해야 한다. 또한 이것은 Hibernate 쟁점이 아니라, 단순히 Java 객체 동일성(identity)와 서로 같음(equality, 동등성)이 구현되는 방식임을 노트하라.

4.1.4. 공통된 동시성 제어 쟁점들

안티-패턴들 `entitymanager-per-user-session` 또는 `entitymanager-per-application`을 결코 사용하지 말라 (물론 이 규칙에 대한 드문 예외들이 존재한다. 예를 들어 `entitymanager-per-application`(어플리케이션 별 엔티티관리자)는 영속 컨텍스트에 대한 수동적인 flushing을 가진, 하나의 데스크탑 어플리케이션에서 허용가능할 수도 있다.) 다음 쟁점들 중 몇몇은 또한 권장되는 패턴들로 나타날 수 있음을 노트하고, 당신이 설계 결정을 내리기 전에 확실하게 그 관련 함축을 이해하도록 하라:

- 하나의 엔티티 관리자는 스레드-안전하지 않다. HTTP request들, 세션 빈즈, 또는 Swing worker 들과 같이, 동시적으로 동작하도록 지원되는 것들은 EntityManager가 공유될 경우에 경쟁 조건들을 불러 일으킬 것이다. 만일 당신이 당신의 HttpSession 내에 당신의 Hibernate EntityManager를 유지시킬 경우(나중에 논의됨), 당신은 당신의 Http 세션에 대한 접근을 동기화 시키는 것을 고려해야 한다. 그 밖의 경우, 충분히 빠르게 새로고침을 클릭하는 사용자는 두 개의 동시적으로 실행되는 스레드들 내에서 같은 EntityManager를 사용할 수도 있다. 당신은 이 경우, 다른 스레드 안전하지 않지만 세션-영역의 객체들의 경우를 미리 대비하는 것을 매우 좋아할 것이다.
- 엔티티 관리자에 의해 던져진 예외상황은 당신이 당신의 데이터베이스 트랜잭션을 롤백시키고 즉시 EntityManager를 닫아야 함을 의미한다(상세한 것은 나중에 논의됨). 만일 당신의 EntityManager가 어플리케이션에 묶여져 있다면, 당신은 어플리케이션을 중지시켜야 한다. 데이터베이스 트랜잭션을 롤백시키는 것은 당신의 비즈니스 객체들을 그것들이 트랜잭션의 시작 시에 존재했던 상태로 집어넣지 않는다. 이것은 데이터베이스 상태와 비즈니스 객체들이 동기화를 벗어남을 의미한다. 예외상황들은 복구가 불가능하기 때문에, 대개 이것은 문제가 아니고 당신은 아무튼 롤백 후에 당신의 작업 단위를 다시 시작해야 한다.
- 영속 컨텍스트는 관리되는 상태(Hibernate에 의해 dirty 상태를 관찰하거나 체크된 상태)에 있는 모든 객체들을 캐시시킨다. 이것은 만일 당신이 영속 컨텍스트를 오랜 시간 동안 열어 두거나 단순히 너무 많은 데이터를 로드시키게 할 경우에 당신이 OutOfMemoryException을 얻을 때까지 그것은 끝없이 성장함을 의미한다. 이것에 대한 한 가지 해결책은 영속 컨텍스트에 대한 정기적인 flushing을 가진 어떤 종류의 batch 처리이지만, 만일 당신이 대용량 데이터 연산들을 필요로 할 경우에 당신은 데이터베이스 내장 프로시저를 사용하는 것을 고려해야 한다. 이 문제에 대한 몇 가지 해결책들이 6장, Batch 처리에 예시되어 있다. 하나의 사용자 세션 동안에 하나의 영속 컨텍스트를 열린채로 유지하는 것은 손상된 데이터의 높은 확률을 의미하며, 당신은 그것을 알아야 하고 적절하게 제어해야 한다.

4.2. 데이터베이스 트랜잭션 경계설정

데이터베이스(또는 시스템) 트랜잭션 경계들은 항상 필수적이다. 데이터베이스와의 통신은 데이터베이스 트랜잭션 외부에서 일어날 수 없다(이것은 자동-확약(auto-commit) 모드를 사용하는 많은 개발자들에게 혼동스러워 보인다). 항상 명료한 트랜잭션 경계들을 사용하라. 심지어 읽기 전용 오퍼레이션들에 대해서도. 당신의 격리 레벨과 데이터베이스 가용성에 따라 이것은 요구되지 않을 수도 있지만 당신이 명시적으로 트랜잭션들을 경계지을 경우에 결점이 존재하지 않는다.

EJB3 어플리케이션은 관리되지 않는 J2EE 환경(예를 들면, 스프링, 단순 웹 어플리케이션 또는 Swing 어플리케이션)과 관리되는 J2EE 환경에서 실행될 수 있다. 관리되지 않는 환경에서, EntityManagerFactory는 대개 그것 자신의 데이터베이스 연결 풀을 책임진다. 어플리케이션 개발자는 수작업으로 트랜잭션 경계들을 설정해야 하는데, 달리 말해 개발자 자신이 데이터베이스 트랜잭션들을 시작시키고, 확약(commit)시키고, 또는 롤백시켜야 한다. 관리되는 환경은 예를 들어 대개 EJB 세션 빈즈의 주해들(annotations)을 통해 선언적으로 정의된 트랜잭션 어셈블리를 가진 컨테이너에 의해 관리되는 트랜잭션들을 제공한다. 그때 프로그램 상의 트랜잭션 경계 설정은 더 이상 필요하지 않으며, 심지어 EntityManager를 flush 시키는 것이 자동적으로 행해진다.

대개 작업 단위를 끝내는 것은 네 개의 구별되는 단계들을 수반한다:

- (리소스-로컬 또는 JTA) 트랜잭션을 확약(commit) 시킨다 (이것은 엔티티 관리자와 영속 컨텍스트를 flush 시킨다)
- (EJB3 컨테이너 외부에서 실행 중일 경우) 엔티티 관리자를 닫는다
- 예외상황들을 처리한다

우리는 이제 관리되는 환경과 관리되지 않는 두 환경들에서 트랜잭션 경계설정과 예외상황 처리를 보다 깊이 살펴볼 것이다.

4.2.1. 관리되지 않는 환경

EJB3 영속 계층이 관리되지 않는 환경에서 실행될 경우, 데이터베이스 연결들은 대개 보이지 않게 Hibernate의 풀링 메커니즘에 의해 처리된다. 공통된 엔티티 관리자와 트랜잭션 처리 관용구는 다음과 같다:

```
// Non-managed environment idiom
EntityManager em = emf.createEntityManager();
EntityTransaction tx = null;
try {
    tx = em.getTransaction();
    tx.begin();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    em.close();
}
```

당신은 명시적으로 EntityManager를 flush() 시켜지 말아야 한다 - commit()에 대한 호출은 동기화를 자동적으로 트리거 시킨다.

close()에 대한 호출은 EntityManager의 끝을 정한다. close()에 대한 주된 함축은 자원들에 대한 해제이다 - 당신은 반드시 닫도록 하고 보장된 finally 블록 외부에서 결코 닫지 말라.

당신은 통상의 어플리케이션의 비즈니스 코드 속에서 이 관용구가 보이지 않는 것을 매우 좋아할 것이다; 치명적인 예외상황들은 "상단"에서 잡혀져야 할 것이다. 달리 말해, (영속 계층에서) 엔티티 관리자 호출을 실행시키는 코드와 RuntimeException을 처리하는(그리고 대개 오직 제거되고 빠져나갈 수만 있는) 코드는 다른 계층들 속에 있다. 이것은 당신 자신이 설계하는 난제일 수 있고 당신은 J2EE/EJB 컨테이너 서비스들이 이용 가능할 때마다 J2EE/EJB 컨테이너 서비스들을 사용해야 한다. 예외상황 처리는 이 장의 뒷 부분에서 논의된다.

4.2.2. JTA 사용하기

만일 당신의 영속 계층이 하나의 어플리케이션 서버(예를 들면 비하인드 EJB3 세션 빈즈) 내에서 실행된다면, 엔티티 관리자에 의해 내부적으로 획득된 모든 데이터소스 연결은 자동적으로 전역 JTA 트랜잭션의 부분이 될 것이다. Hibernate는 이 통합을 위한 두 개의 방도들을 제공한다.

만일 당신이 bean-managed transactions (BMT)를 사용하는 경우에, 코드는 다음과 같을 것이다:

TODO: UserTransaction이 이제 실제로 rollback()인지 그리고 setRollbackOnly()가 아닌지를 체크할 것.

```
// BMT idiom
@Resource public UserTransaction utx;
@Resource public EntityManagerFactory factory;
```

```

public void doBusiness() {
    EntityManager em = factory.createEntityManager();
    try {

        // do some work
        ...

        utx.commit();
    }
    catch (RuntimeException e) {
        if (utx != null) utx.rollback();
        throw e; // or display error message
    }
    finally {
        em.close();
    }
}

```

EJB3 컨테이너에서 Container Managed Transactions (CMT)의 경우, 트랜잭션 경계설정은 프로그램 상이 아닌, 세션 빈(bean) 주해들 또는 배치 디스크립터들로 행해진다. EntityManager는 트랜잭션 완료 시에 자동적으로 flush 될 것이다(그리고 당신이 EntityManager를 끼워넣었거나 록업했다면 그것은 또한 자동적으로 닫혀질 것이다). 만일 EntityManager 사용 동안에 예외상황이 일어난다면, 당신이 그 예외상황을 잡지 못하는 경우에 트랜잭션 롤백이 자동적으로 일어난다. EntityManager 예외 상황들이 RuntimeException들이기 때문에 그것들은 EJB 명세서(시스템 예외상황 대 어플리케이션 예외상황)에 따라 트랜잭션을 롤백시킬 것이다.

TODO: 이 구성이 여전히 EJB3 엔티티 관리자 팩토리에 대해 동일한지 여부를 체크할 것

당신이 Hibernate의 트랜잭션 팩토리를 구성할 때, 당신은 하나의 BMT 세션 빈(bean)에서 org.hibernate.transaction.JTATransactionFactory를 선택해야 하고, 하나의 CMT 세션 빈(bean)에서 org.hibernate.transaction.CMTTransactionFactory를 선택해야 함을 노트하라. 또한 org.hibernate.transaction.manager_lookup_class를 설정하는 것을 기억하라.

만일 당신이 하나의 CMT 환경에서 작업할 경우, 당신은 또한 당신의 코드의 다른 부분들에서 동일한 엔티티 관리자를 사용하고자 원할 수 있다. 전형적으로 관리되지 않는 환경에서 당신은 엔티티 관리자를 보관할 하나의 ThreadLocal 변수를 사용할 것이지만, 어떤 하나의 EJB 요청은 다른 쓰레드들 내에서 실행될 수도 있다(예를 들면 또 다른 세션 빈(bean)을 호출하는 세션 빈(bean)). EJB3 컨테이너는 당신을 위해 영속 컨텍스트 보급을 처리한다. 끼워넣기를 사용하거나 EntityManagerFactory.getEntityManager()를 사용하면, EJB3 컨테이너는 아무튼 JTA 컨텍스트에 묶인 동일한 영속 컨텍스트를 가진 하나의 엔티티 관리자를 반환할 것이거나, 하나의 새로운 영속 컨텍스트를 생성시키고 엔티티 관리자에게 바인드 시킬 것이다(1.2.4절, “영속 컨텍스트 보급(propagation)”를 보라).

CMT 용도 및 EJB3 컨테이너 용도를 위한 우리의 엔티티 관리자/트랜잭션 관리 관용구는 다음으로 줄여진다:

```

// CMT idiom through factory
EntityManager em = factory.getEntityManager();

// do some work
...

//CMT idiom through injection
@Resource EntityManager em;

```

달리 말해, 관리되는 환경에서 당신이 행해야 하는 모든 것은 EntityManagerFactory.getEntityManager()를 호출하거나 EntityManager를 끼워넣고, 당신의 데이터

접근 작업을 행하고, 나머지를 컨테이너에게 위임시키는 것이다. 트랜잭션 경계설정들은 당신의 세션 빈즈의 주해들 또는 배치 디스크립터들 내에 선언적으로 설정된다. 엔티티 관리자와 영속 컨텍스트의 생명주기는 컨테이너에 의해 완전하게 관리된다.

TODO: 다음 단락은 매우 혼동스럽다. 특히 초심자들에게는...

When using particular Hibernate native APIs, one caveat has to be remembered: `after_statement` connection release mode. Due to a silly limitation of the JTA spec, it is not possible for Hibernate to automatically clean up any unclosed `ScrollableResults` Or `Iterator` instances returned by `scroll()` or `iterate()`. You must release the underlying database cursor by calling `ScrollableResults.close()` Or `Hibernate.close(Iterator)` explicitly from a `finally` block. (Of course, most applications can easily avoid using `scroll()` or `iterate()` at all from the CMT code.)

4.2.3. 예외상황 처리

만일 `EntityManager`가 (어떤 `SQLException`을 포함하여) 예외상황을 던질 경우, 당신은 즉시 데이터베이스 트랜잭션을 롤백시키고, (`createEntityManager()`가 호출되었을 경우에) `EntityManager.close()`를 호출하고 `EntityManager` 인스턴스를 폐기시켜야 한다. `EntityManager`의 어떤 메소드들은 영속 컨텍스트를 하나의 일관된 상태로 남겨두지 않을 것이다. 하나의 엔티티 관리자에 의해 던져진 예외상황은 복구 가능한 것으로 다루어질 수 없다. `EntityManager`는 하나의 `finally` 블록 내에서 `close()`를 호출하여 닫혀질 것이다. 하나의 컨테이너에 의해 관리되는 엔티티 관리자는 당신을 위해 그것을 행할 것임을 노트하라. 단지 당신은 `RuntimeException`을 컨테이너에까지 보급되도록 해야 한다.

Hibernate 엔티티 관리자는 일반적으로 Hibernate 알맹이(`core`) 예외상황을 캡슐화 시키는 예외상황들을 발생킨다. `EntityManager` API에 의해 발생하는 공통된 예외상황들은 다음이다

- `IllegalArgumentException`: 잘못된 어떤 것이 발생함
- `EntityNotFoundException`: 기대된 엔티티였지만 사양과 일치하지 않음
- `TransactionRequiredException`: 이 연산은 트랜잭션 내에 있어야 함
- `IllegalStateException`: 엔티티 관리자가 잘못된 방식으로 사용되고 있음

Hibernate 영속 계층에서 일어날 수 있는 대부분의 오류들을 포장하는, `HibernateException`은 체크되지 않은 예외상황이다. Hibernate가 또한 하나의 `HibernateException`이 아닌 다른 예기치 않은 예외상황들을 던질 수 있음을 노트하라. 다시 말하지만 이것들은 복구가능하지 않고 적절한 조치가 취해여야 한다.

Hibernate는 데이터베이스와 상호작용하는 동안에 던져진 `SQLException`들을 하나의 `JDBCException` 속에 포장한다. 사실, Hibernate는 그 예외상황을 `JDBCException`의 보다 의미 있는 서브클래스로 변환시키려 시도할 것이다. 기본 `SQLException`은 `JDBCException.getCause()`를 통해 항상 이용 가능하다. Hibernate는 `SessionFactory`에 첨부된 `SQLExceptionConverter`를 사용하여 `SQLException`을 하나의 적절한 `JDBCException`으로 변환시킨다. 디폴트로, `SQLExceptionConverter`는 구성된 `dialect`에 의해 정의된다; 하지만 그것은 또한 하나의 맞춤 구현을 플러그 인 시키는 것이 가능하다(상세한 것은 `SQLExceptionConverterFactory` 클래스에 대한 javadocs를 보라). 표준 `JDBCException` 서브타입들은 다음과 같다:

- `JDBCConnectionException` - 기본 JDBC 통신에 대한 오류를 나타낸다.
- `SQLGrammarException` - 실행 명령이 내려진 SQL에 대한 문법 또는 구문 문제점을 나타낸다.

- `ConstraintViolationException` - 어떤 형식의 무결성 제약 위반을 나타낸다.
- `LockAcquisitionException` - 요청된 연산을 수행하는데 필수적인 하나의 잠금 레벨을 획득하는 오류를 나타낸다.
- `GenericJDBCException` - 어떤 다른 카테고리들 속으로 분류되지 않은 일반적인 예외상황.

4.3. Optimistic 동시성 제어

고도의 동시성과 고 가용성을 일관되게 유지하는 유일한 접근법은 버전을 가진 optimistic 동시성 제어이다. 버전 체크는 충돌하는 업데이트들을 검출하는데(그리고 업데이트들이 손실되는 것을 방지하는데) 버전 번호들, 또는 타임스탬프들을 사용한다. Hibernate는 optimistic 동시성을 사용하는 어플리케이션 코드를 작성하는 세 가지 가능한 접근법들을 제공한다. 우리가 예시하는 쓰임새들은 긴 어플리케이션 트랜잭션들의 컨텍스트 내에 있지만 버전 체크는 또한 단일 데이터베이스 트랜잭션들 내에서 업데이트들이 손실되는 것을 방지하는 이점을 갖고 있다.

4.3.1. 어플리케이션 버전 체크

영속 메커니즘으로부터 많은 도움 없는 구현에서, 데이터베이스와의 각각의 상호작용은 하나의 새로운 `EntityManager` 내에서 일어나고 개발자는 모든 영속 인스턴스들을 처리하기 전에 데이터베이스로부터 그것들을 모두 다시 로드시킬 책임이 있다. 이 접근법은 어플리케이션 트랜잭션 격리를 보장하기 위해 그것 자신의 버전 체크를 수행하는 것을 어플리케이션에게 강제시킨다. 이 접근법은 데이터베이스 접근의 관점에서 보면 가장 효과적이지 않다. 그것은 EJB2 엔티티들과 가장 유사한 접근법이다:

```
// foo is an instance loaded by a previous entity manager
em = factory.createEntityManager();
EntityTransaction t = em.getTransaction();
t.begin();
int oldVersion = foo.getVersion();
Foo dbFoo = em.find( foo.getClass(), foo.getKey() ); // load the current state
if ( dbFoo.getVersion() != foo.getVersion() ) throw new StaleObjectStateException();
dbFoo.setProperty("bar");
t.commit();
em.close();
```

`version` 프로퍼티는 `@Version`을 사용하여 매핑되고, 엔티티 관리자는 그 엔티티가 `dirty`일 경우에 `flush` 동안에 그것(`@Version`)을 자동적으로 증가시킬 것이다.

물론 당신이 낮은 데이터 동시성 환경에서 운영하고 있고 버전 체크를 필요로 하지 않을 경우에, 당신은 이 접근법을 사용할 수 있고 버전 체크를 단순히 생략할 수 있다. 그 경우에, `last commit wins`(최종 확약 성공)이 당신의 긴 어플리케이션 트랜잭션들을 위한 디폴트 방도일 것이다. 이것은 어플리케이션의 사용자들을 혼동스럽게 할 수도 있음을 명심하라. 왜냐하면 그들은 오류 메시지 또는 충돌하는 변경들을 병합시키는 기회 없이 업데이트 손실을 경험할 수도 있기 때문이다.

명료하게, 수동적인 버전 체크는 매우 사소한 환경들에서만 적절하며 대부분의 어플리케이션들의 경우에는 실용적이지 못하다. 종종 단일 인스턴스들 뿐만 아니라 수정된 객체들의 전체 그래프들이 체크되어야 한다. Hibernate는 분리된(`detached`) 인스턴스들 또는 하나의 확장된 엔티티 관리자 그리고 영속 컨텍스트에 대해 자동적인 버전화 체크를 설계 패러다임으로서 제공한다.

4.3.2. 확장된 엔티티 관리자와 자동적인 버전 체크

하나의 영속 컨텍스트가 전체 어플리케이션 트랜잭션에 사용된다. 엔티티 관리자는 `flush` 시점에서

인스턴스 버전들을 체크하며, 동시적인 수정이 검출될 경우에 하나의 예외상황을 던진다. 이 예외상황을 잡고 처리하는 것은 개발자의 몫이다(공통 옵션들은 사용자가 그의 변경들을 병합하거나 손상되지 않은 데이터를 가지고 비즈니스 프로세스를 재시작할 기회이다).

Entity Manager는 사용자 상호작용을 기다릴 때 임의의 기본 JDBC 연결로부터 연결해제된다. 어플리케이션에 의해 관리되는 확장된 엔티티 관리자에서, 이것은 트랜잭션 완료 시에 자동적으로 일어난다. 하나의 컨테이너에 의해 관리되는 확장된 엔티티 관리자를 소유하는 하나의 상태 있는 세션 빈(bean)(예를 들면 @PersistenceContext(EXTENDED) 주해를 가진 SFSB)에서, 이것은 마찬가지로 투명하게 일어난다. 이 접근법은 데이터베이스 접근 수단으로서 가장 효과적이다. 어플리케이션은 버전 체크링으로 또는 분리된(detached) 인스턴스들을 병합시키는 것으로 그 자체에 관계할 필요가 없거나, 그것은 모든 데이터베이스 트랜잭션 내에 인스턴스들을 다시 로드시켜야 할 필요가 없다. 많은 열려진 연결들과 닫혀진 연결들에 관계될 수 있는 사람들의 경우, 연결 공급자는 하나의 연결 풀일 것이어서, 퍼포먼스 영향이 존재하지 않음을 기억하라. 다음 예제들은 관리되지 않는 호나경에서 관용구를 보여준다:

```
// foo is an instance loaded earlier by the extended entity manager
em.getTransaction.begin(); // new connection to data store is obtained and tx started
foo.setProperty("bar");
em.getTransaction().commit(); // End tx, flush and check version, disconnect
```

foo 객체는 그것이 로드시켰던 ## ####가 어느 것인지를 여전히 알고 있다. getTransaction.begin(); 으로 엔티티 관리자는 하나의 새로운 연결을 획득하고 영속 컨텍스트를 다시 시작시킨다. getTransaction().commit() 메소드는 flush 시키고 버전들을 체크할 뿐만 아니라, 또한 JDBC 연결로부터 엔티티 관리자를 연결해제시키고 연결을 풀(pool)로 반환시킬 것이다.

영속 컨텍스트가 사용자 생각 시간 동안에 저장되기에 너무 클 경우에, 그리고 당신이 그것을 저장할 장소를 알지 못할 경우에 이 패턴은 문제성이 있다. 예를 들어, HttpSession은 가능하면 작게 유지되어야 한다. 영속 컨텍스트는 또한 (필수적인) 첫번째-레벨의 캐시이고 모든 로드된 객체들을 포함하므로, 우리는 아마 적은 요청/응답 주기들에 대해서만 이 방법을 사용할 수 있다. 영속 컨텍스트가 또한 골장 손상된 데이터를 갖게 될 때 이것이 진정으로 권장된다.

당신이 요청들 동안에 확장된 엔티티 관리자를 저장하는 장소는 당신에게 달려 있으며, EJB3 컨테이너 내에서 당신은 위에 설명된 대로 하나의 상태 있는 세션 빈(bean)을 간단히 사용한다. HttpSession 내에 그것을 저장하기 위해 웹 계층으로 그것을 전송하지 말라(심지어 그것을 하나의 별도의 계층으로 직렬화 시키지 말라). 관리되지 않는 two-tier 환경에서 HttpSession은 그것을 저장하는 진정한 올바른 장소 일 수도 있다.

4.3.3. Detached objects and automatic versioning

이 패러다임의 경우, 데이터 저장소와의 각각의 상호작용은 하나의 새로운 영속 컨텍스트 내에서 일어난다. 하지만 동일한 영속 인스턴스들은 데이터베이스와의 각각의 상호작용에 대해 재사용된다. 어플리케이션은 또 다른 영속 컨텍스트 내에 애초에 로드된 분리된(detached) 인스턴스들의 상태를 처리하고 그런 다음 EntityManager.merge()를 사용하여 변경들을 병합시킨다:

```
// foo is an instance loaded by a non-extended entity manager
foo.setProperty("bar");
entityManager = factory.createEntityManager();
entityManager.getTransaction().begin();
managedFoo = session.merge(foo); // discard foo and from now on use managedFoo
entityManager.getTransaction().commit();
entityManager.close();
```

다시, 엔티티 관리자는 flush 동안에 인스턴스들을 던질 것이고, 충돌하는 업데이트들이 일어날 때 하나의 예외상황을 던질 것이다.

5장. 엔티티 리스너들과 콜백 메소드들

양식 메커니즘 내부에서 발생하는 어떤 이벤트들에 반응하는 것은 어플리케이션에 자주 유용하다. 이것은 어떤 종류의 일반적 기능, 그리고 미리 만들어진 기능의 확장에 대한 구현을 허용해준다. EJB3 명세서는 이 용도를 위한 두 개의 관련된 메커니즘들을 제공한다.

엔티티의 메소드는 하나의 특정한 엔티티 생명 주기 이벤트의 통보를 수신받기 위한 하나의 콜백 메소드로서 고안될 수 있다. 당신은 또한 엔티티 클래스 내부에 직접 정의된 콜백 메소드들 대신에 사용될 하나의 엔티티 리스너 클래스를 정의할 수 있다. 하나의 엔티티 리스너는 아규먼트 없는 생성자를 가진 상태없는 클래스이다. 엔티티 리스너는 엔티티 클래스를 `@EntityListener` 주해로 엔티티 클래스를 주해함으로써 정의된다:

```
@Entity(access=FIELD)
@EntityListener(class=Audit.class)
public class Cat {
    @Id private Integer id;
    private String name;
    private Calendar dateOfBirth;
    @Transient private int age;
    private Date lastUpdate;
    //getters and setters

    /**
     * Set my transient property at load time based on a calculation,
     * note that a native Hibernate formula mapping is better for this purpose.
     */
    @PostLoad
    public void calculateAge() {
        Calendar birth = new GregorianCalendar();
        birth.setTime(dateOfBirth);
        Calendar now = new GregorianCalendar();
        now.setTime( new Date() );
        int adjust = 0;
        if ( now.get(Calendar.DAY_OF_YEAR) - birth.get(Calendar.DAY_OF_YEAR) < 0 ) {
            adjust = -1;
        }
        age = now.get(Calendar.YEAR) - birth.get(Calendar.YEAR) + adjust;
    }
}

public class LastUpdateListener {
    /**
     * automatic property set before any database persistence
     */
    @PreUpdate
    @PrePersist
    public void setLastUpdate(Cat o) {
        o.setLastUpdate( new Date() );
    }
}
```

동일한 콜백 메소드 또는 엔티티 리스너 메소드는 하나 이상의 콜백 주해들로서 주해될 수 있다. 주어진 하나의 엔티티에 대해, 당신은 그것이 하나의 콜백 메소드이든지 하나의 엔티티 리스너 메소드 이든지 간에 동일한 콜백 주해에 의해 주해되는 두 개의 메소드들을 가질 수 없다. 하나의 콜백 메소드는 반환 타입 없이 어떤 임의의 이름을 가진 하나의 아규먼트 없는 메소드이다. 하나의 엔티티 리스너는 시그니처 `public void <METHOD>(Object)`을 가지며 여기서 Object는 실제 엔티티 타입이다 (Hibernate 엔티티 관리자는 이 생성자를 완화시키고 (몇몇 엔티티들을 가로질러 리스너들을 공유하는 것을 허용하는) `java.lang.Object` 타입의 `object`를 허용함을 노트하라.)

하나의 콜백 메소드는 `RuntimeException`을 일으킨다. 아무튼 현재 트랜잭션은 롤백되어야 한다. 다음

콜백들이 정의된다:

표 5.1. Callbacks

| 타입 | 설명 |
|--------------|--|
| @PrePersist | 엔티티 관리자 persist 연산이 실제로 실행되거나 캐스케이드 되기 전에 실행된다. 이 호출은 persist 연산과 동기적이다. |
| @PreRemove | 엔티티 관리자 remove 연산이 실제로 실행되거나 캐스케이드 되기 전에 실행된다. 이 호출은 remove 연산과 동기적이다. |
| @PostPersist | 엔티티 관리자 persist 연산이 실제로 실행되거나 캐스케이드 된 후에 실행된다. 이 호출은 데이터베이스 INSERT가 실행된 후에 호출된다. |
| @PostRemove | 엔티티 관리자 remove 연산이 실제로 실행되거나 캐스케이드 된 후에 실행된다. 이 호출은 remove 연산과 동기적이다. |
| @PreUpdate | 데이터베이스 UPDATE 연산 전에 실행된다. |
| @PostUpdate | 데이터베이스 UPDATE 연산 후에 실행된다. |
| @PostLoad | 하나의 엔티티가 현재의 영속 컨텍스트 속으로 로드된 후에 또는 하나의 엔티티가 갱신된 후에 실행된다. |

하나의 콜백 메소드는 EntityManager 메소드나 query 메소드를 호출하지 말아야 한다!

6장. Batch 처리

Batch 처리는 전통적으로 전체 객체/관계형 매핑에서 어려웠다. ORM은 객체 상태 관리에 관한 모든 것이고, 그것은 객체 상태가 메모리 내에서 이용 가능하다는 점을 의미한다. 하지만 Hibernate는 Hibernate 참조 안내서에서 논의되어 있는 batch 처리를 최적화 시키는 몇몇 특징들을 갖고 있지만, EJB3 영속은 약간 다르다.

6.1. 대용량 update/delete

이미 논의했듯이, 자동적인 그리고 투명한 객체/관계형 매핑이 객체의 상태를 관리하는 것에 관계된다. 이것은 객체 상태가 메모리 내에서 이용 가능하고, 그러므로 (SQL UPDATE와 DELETE를 사용하여) 데이터베이스 내에서 직접 데이터를 업데이트하거나 삭제하는 것은 메모리 내 상태에 영향을 주지 않을 것이다. 하지만 Hibernate는 EJB-QL(7장. EJB-QL: 객체 질의 언어(Object Query Language))을 통해 수행되는 대용량 SQL-스타일의 UPDATE 및 DELETE 문장 실행을 위한 메소드들을 제공한다.

UPDATE와 DELETE 문장들을 위한 유사-구문은 다음과 같다: (UPDATE | DELETE) FROM? ClassName (WHERE WHERE_CONDITIONS)?. 다음을 노트하라:

- from-절에서, FROM 키워드는 옵션이다.
- from-절 내에 명명된 한 개의 클래스 만이 존재할 수 있고, 그것은 하나의 alias를 가질 수 없다(이것은 현재 Hibernate 제약점이고 곧 제거될 것이다).
- (함축적이든 명시적이든) 조인들은 대용량 EJB-QL 질의 내에 지정될 수 없다. 서브-질의들은 where-절 속에 사용될 수 있다.
- where-절은 또한 옵션이다.

하나의 EJB-QL UPDATE를 실행하기 위한 하나의 예제로서, `Query.executeUpdate()` 메소드를 사용하라:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();

String ejbqlUpdate = "update Customer set name = :newName where name = :oldName"
int updatedEntities = entityManager.createQuery( hqlUpdate )
    .setParameter( "newName", newName )
    .setParameter( "oldName", oldName )
    .executeUpdate();

entityManager.getTrasnaction().commit();
entityManager.close();
```

하나의 EJB-QL DELETE를 실행하기 위해, 같은 `Query.executeUpdate()` 메소드를 사용하라(그 메소드는 JDBC의 `PreparedStatement.executeUpdate()`에 익숙한 사람들을 위해 명명된다):

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();

String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = entityManager.createQuery( hqlDelete )
    .setParameter( "oldName", oldName )
    .executeUpdate();

entityManager.getTrasnaction().commit();
entityManager.close();
```

`Query.executeUpdate()` 메소드에 의해 반환된 int 값은 그 연산에 의해 영향받은 엔티티들의 개수를 나타낸다. 이것은 데이터베이스내에서 영향받은 많은 행들에 관련될 수 있거나 관계되지 않을 수 있

다. 하나의 EJB-QL 대용량 연산은 예를 들어 조인된-서브클래스(joined-subclass)에 대해 실행되는 여러 개의 실제 SQL 문장들로 귀결될 수 있다. 반환되는 개수는 그 문장에 의해 영향받은 실제 엔티티들의 개수를 나타낸다. 조인된-서브클래스 예제로 되돌아가면, 서브클래스들 중 하나에 대한 삭제는 실제로 단지 그 서브클래스가 매핑되어 있는 테이블에 대한 것이 아니라, 또한 "루트" 테이블과 상속 계층을 더 내려간 잠정적으로 조인된-서브클래스 테이블들에 대한 삭제들로 귀결될 수 있다.

장래의 배포본들에서 제기될 대용량 연산들에 대해 가진 몇몇 제약점들이 존재함을 노트하라; 상세한 것은 JIRA 로드맵을 참조하라.

7장. EJB-QL: 객체 질의 언어(Object Query Language)

EJB3-QL은 HQL, native Hibernate Query Language에 의해 크게 영감을 받았다. 그러므로 둘 다 SQL에 아주 가깝지만, 데이터베이스 스키마에 이식 가능하고 독립적이다. HQL에 익숙한 사람들은 EJB-QL을 사용하는데 별다른 문제점을 갖지 않는다. 실제로 당신은 EJB-QL과 HQL 질의들에 대해 동일한 질의 API를 사용한다. 하지만 이식가능한 EJB3 어플리케이션들은 EJB-QL에 충실해야 하거나 유사한 벤더 확장들이 필요하다.

7.1. 대소문자 구분

질의들은 자바 클래스들과 프로퍼티들의 이름들을 제외하면 대소문자를 구분하지 않는다. 따라서 `seLeCT`가 `sELEct`과 동일하며 `SELECT`과 동일하지만 `org.hibernate.eg.FOO`은 `org.hibernate.eg.Foo`과 같지 않고 `foo.barSet`은 `foo.BARSET`과 같지 않다.

이 매뉴얼은 소문자 EJBQL 키워드들을 사용한다. 몇몇 사용자들은 보다 가독성이 있는 대문자 키워드들을 가진 질의들을 사용하지만, 우리는 자바 코드 속에 삽입시킬 때 이 컨벤션이 추하다는 점을 발견한다.

7.2. from 절

가능 간단한 가능한 EJB-QL 질의는 다음 형식이다:

```
select c from eg.Cat c
```

이것은 단순히 `eg.Cat` 클래스의 모든 인스턴스들을 반환한다. HQL과는 달리, `select` 절은 EJB-QL에서 옵션이 아니다. 우리는 클래스 이름을 수식할 필요가 없다. 왜냐하면 엔티티 이름이 디폴트로 수식되지 않은 클래스 이름(`@Entity`)이기 때문이다. 따라서 우리는 거의 항상 단지 다음과 같이 작성한다:

```
select c from Cat c
```

당신이 인지했을 수도 있듯이 당신은 클래스들에 `alias`들을 할당할 수 있으며, `as` 키워드는 옵션이다. 한 개의 `alias`는 질의의 다른 부분들 내에서 `cat`을 참조하는 것을 당신에게 허용해준다.

```
select cat from Cat as cat
```

여러 개의 클래스들이 하나의 카티전 값 또는 "크로스" 조인으로 귀결되어 나타날 수도 있다.

```
select form, param from Formula as form, Parameter as param
```

로컬 변수들에 대한 자바 명명법 표준과 일치되게끔 초문자 첫 글자를 사용하여 질의 `alias`를 명명하는 것은 좋은 실례로 간주된다(예를들면, `domesticCat`).

7.3. 주해들과 조인들

당신은 또한 `join`을 사용하여, `alias`들을 연관된 엔티티들에 할당할 수 있거나, 심지어 값들을 가진 하나의 컬렉션의 요소들에 할당할 수도 있다.

```
select cat, mate, kitten from Cat as cat
       inner join cat.mate as mate
       left outer join cat.kittens as kitten
```

```
select cat from Cat as cat left join cat.mate.kittens as kittens
```

지원되는 join 타입들은 ANSI SQL로부터 빌려왔다

- inner join
- left outer join

inner join, left outer join 구조체는 생략하여 쓰일수도 있다.

```
select cat, mate, kitten from Cat as cat
       join cat.mate as mate
       left join cat.kittens as kitten
```

게다가, 하나의 "fetch" join은 연관들 또는 값들을 가진 컬렉션들이 한 개의 select를 사용하여 그것들의 부모 객체들에 따라 초기화 되는 것을 허용해준다. 이것은 컬렉션의 경우에 특히 유용하다. 그것은 연관들과 컬렉션 매핑 메카데이터에서 폐칭 옵션들을 효과적으로 오버라이드 시킨다. 추가 정보는 Hibernate 참조 안내서의 퍼포먼스 장을 보라.

```
select cat from Cat as cat
       inner join fetch cat.mate
       left join fetch cat.kittens
```

하나의 fetch join은 대개 한 개의 alias를 할당할 필요가 없다. 왜냐하면 연관된 객체들이 where 절(또는 어떤 다른 절) 속에서 사용되지 않을 것이기 때문이다. 또한 연관된 객체들은 질의 결과 셋들 속에서 직접 반환되지 않는다. 대신에, 그것들은 부모 객체를 통해 접근될 수도 있다. 우리가 alias를 필요로 하는 유일한 이유는 우리가 추가적인 컬렉션을 재귀적으로 조인 폐칭할 경우이다:

```
select cat from Cat as cat
       inner join fetch cat.mate
       left join fetch cat.kittens child
       left join fetch child.kittens
```

fetch 구조체는 scroll() 또는 iterate()를 사용하여 호출되는 질의들 내에 사용되지 않음을 노트하라. 또한 fetch는 setMaxResults() 또는 setFirstResult()과 함께 사용되지 않을 것이다. (위의 예제에서 처럼) 한 개의 질의 내에 하나 이상의 컬렉션을 조인 폐칭시켜서 한 개의 카티전 곱을 생성시키는 것이 가능하며, 이 곱의 결과가 당신이 예상하는 것보다 더 크지 않을 것임을 주의하라. 여러 개의 컬렉션 role들을 조인 폐칭시키는 것은 또한 때때로 bag 매핑들에 대해 예기치 않은 결과들을 초래하므로, 당신이 이 경우에 당신의 질의들을 처방하는 방법을 주의하라.

TODO: 마지막 문장은 쓸모없고 일반적인 개발자 생각이며, 퇴고하길 바란다. 단어 "때로는 (sometimes)"는 어떤 기술 문서 내에 결코 나타나지 말아야 한다.

만일 당신이 (바이트코드 수단으로) 프로퍼티 레벨의 lazy 폐칭을 사용할 경우에, Hibernate로 하여금 (fetch all properties를 사용하여)(첫 번째 질의 내에)lazy 프로퍼티들을 즉시 폐치시키는 것을 강제시키는 것이 가능하다. 이것은 Hibernate에 특징적인 옵션이다:

```
select doc from Document doc fetch all properties order by doc.name
```

```
select doc from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

7.4. select 절

select 절은 어느 객체들과 프로퍼티들이 질의 결과 셋 내에 반환될 것인지를 골라낸다. 다음을 검토 하자:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

질의는 다른 Cat들 중에서 mate들을 select할 것이다. 실제로 당신은 다음과 같이 보다 축약적으로 이 질의를 표현할 수도 있다:

```
select cat.mate from Cat cat
```

질의들은 컴포넌트 타입인 프로퍼티들을 포함하는 임의의 값 타입의 프로퍼티들을 반환할 수 있다:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

질의들은 여러 개의 객체들 과/또는 프로퍼티들을

Object[] 타입의 하나의 배열로서,

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

또는 하나의 List로서 (HQL에 특정한 특징),

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

또는 Family 클래스가 하나의 적절한 생성자를 갖고 있음을 가정하면 하나의 실제 타입안전한 Java 객체로서,

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

반환할 수 있다

당신은 as를 사용하여 select된 표현식들에 alias들을 할당할 수도 있다:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

이것은 select new map(HQL에 특정한 특징)과 함께 사용될 때 가장 유용하다:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

이 질의는 alias들로부터 select된 값들로의 하나의 Map을 반환한다.

7.5. 집계 함수들

HQL 질의들은 심지어 프로퍼티들에 대한 집계 함수들의 결과들을 반환할 수도 있다:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

지원되는 집계 함수들은 다음과 같다

- avg(...), avg(distinct ...), sum(...), sum(distinct ...), min(...), max(...)
- count(*)
- count(...), count(distinct ...), count(all...)

당신은 (구성된 dialect에 따라, HQL에 특정한 특징임) select 절 내에 산술 연산자들, concatenation, 그리고 인지된 SQL 함수들을 사용할 수 있다:

```
select cat.weight + sum(kitten.weight)
from Cat cat
      join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

distinct 키워드와 all 키워드가 사용될 수 있고 SQL에서와 같은 의미를 갖는다.

```
select distinct cat.name from Cat cat
select count(distinct cat.name), count(cat) from Cat cat
```

7.6. 다형성 질의들

다음과 같은 질의:

```
select cat from Cat as cat
```

은 cat의 인스턴스들 뿐만 아니라 DomesticCat과 같은 서브클래스들의 인스턴스들을 반환한다. Hibernate 질의들은 from 절 내에서 임의의 자바 클래스 또는 인스턴스를 명명할 수 있다(이식 가능한 EJB-QL 질의들은 오직 매핑된 엔티티들만을 명명해야 한다). 질의는 그 클래스를 확장하거나 그 인터페이스를 구현하는 모든 영속 클래스들의 인스턴스들을 반환할 것이다. 다음 질의는 모든 영속 객체들을 반환할 것이다:

```
from java.lang.Object o // HQL only
```

인터페이스 Named는 여러 영속 클래스들에 의해 구현될 수 있다:

```
from Named n, Named m where n.name = m.name // HQL only
```

이들 마지막 두 개의 질의들이 하나 이상의 SQL SELECT를 필요로 할 것임을 노트하라. 이것은 order by 절이 전체 결과 셋을 정확하게 순서(order) 지우지 않음을 의미한다. (그것은 또한 당신이

Query.scroll()을 사용하여 이들 질의들을 호출할 수 없음을 의미한다.)

7.7. where 절

where 절은 반환되는 인스턴스들의 목록을 좁히는 것을 당신에게 허용해준다. 만일 alias가 존재하지 않을 경우, 당신은 이름으로 프로퍼티들을 참조할 수 있다:

```
select cat from Cat cat where cat.name='Fritz'
```

는 'Fritz'로 명명된 cat의 인스턴스들을 반환한다.

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

는 Foo의 startDate 프로퍼티와 같은 date 프로퍼티를 가진 bar의 인스턴스가 존재하는 Foo의 모든 인스턴스들을 반환할 것이다. 합성된 경로 표현식들은 where 절을 극히 강력하게 만든다. 다음을 검토하자:

```
select cat from Cat cat where cat.mate.name is not null
```

이 질의는 하나의 테이블 (inner) 조인을 가진 하나의 SQL 질의로 변환된다. 만일 당신이 다음과 같은 것을 작성했다면

```
select foo from Foo foo
where foo.bar.baz.customer.address.city is not null
```

당신은 SQL로된 네 개의 테이블 조인들을 필요로 하는 하나의 질의로 끝낼 것이다.

= 연산자는 프로퍼티들 뿐만 아니라 인스턴스들을 비교하는데 사용될 수 있다:

```
select cat, rival from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

(소문자로 된) 특별한 프로퍼티 id는 객체의 유일 식별자를 참조하는데 사용될 수 있다. (당신은 또한 그것의 매핑된 식별자 프로퍼티 이름을 사용할 수도 있다.). 이 키워드가 HQL에 특정한 것임을 노트하라.

```
select cat from Cat as cat where cat.id = 123

select cat from Cat as cat where cat.mate.id = 69
```

두 번째 질의가 효과적이다. 테이블 조인들이 필요하지 않다!

합성(composite) 식별자들의 프로퍼티들이 또한 사용될 수 있다. person이 country와 medicareNumber로 구성된 하나의 합성 식별자를 갖는다고 가정하자.

```
select person from bank.Person person
where person.id.country = 'AU'
and person.id.medicareNumber = 123456
```

```
select account from bank.Account account
where account.owner.id.country = 'AU'
and account.owner.id.medicareNumber = 123456
```

다시 한번, 두 번째 질의는 테이블 조인을 필요로 하지 않는다.

마찬가지로 특별한 프로퍼티 class는 다형성 상속의 경우에 하나의 인스턴스의 판별자 값에 접근한다. where 절 내에 삽입된 Java 클래스 이름은 그것의 판별자 값으로 변환될 것이다. 다시 한번, 이것은 HQL에 특정한 것이다.

```
select cat from Cat cat where cat.class = DomesticCat
```

당신은 또한 컴포넌트 사용자 타입들 또는 합성(composite) 사용자 타입들(그리고 컴포넌트들을 가진 컴포넌트 타입)인 프로퍼티들을 지정할 수도 있다. (하나의 컴포넌트의 하나의 프로퍼티와는 대조적으로) 컴포넌트 타입인 하나의 프로퍼티로 끝나는 하나의 경로-표현식을 결코 사용하려고 시도하지 말라. 예를 들어, 만일 store.owner이 하나의 컴포넌트 address을 가진 엔티티일 경우에

```
store.owner.address.city // okay
store.owner.address // error!
```

"any" 타입은 다음 방법으로 하나의 조인을 표현하는 것을 우리에게 허용해주는 특별한 프로퍼티들 id와 class를 갖는다(여기서 AuditLog.item은 <any>로서 매핑된 하나의 프로퍼티이다). Any는 Hibernate에 특징적인 것이다

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

log.item.class와 payment.class는 위의 질의에서 완전히 다른 데이터베이스 컬럼들의 값들을 참조할 것임을 주지하라.

7.8. 표현식들

where 절 내에 허용되는 표현식들은 당신이 SQL로 작성할 수 있는 거의 대부분의 종류의 것들을 포함한다:

- 수학 연산자들 +, -, *, /
- 바이너리 비교 연산자들 =, >=, <=, <>, !=, like
- 논리 연산자들 and, or, not
- 그룹핑을 나타내는 괄호들 ()
- in, not in, between, is null, is not null, is empty, is not empty, member of 그리고 not member of
- "간단한" case, case ... when ... then ... else ... end, 그리고 "검색된(searched)" case, case when ... then ... else ... end (HQL# ####)
- 문자열 연결 ...||... 또는 concat(..., ...) (##### EJB-QL ### ## concat()# #####)
- current_date(), current_time(), current_timestamp()
- second(...), minute(...), hour(...), day(...), month(...), year(...), (HQL에 특징적임)
- EJB-QL 3.0에 의해 정의된 이므이의 함수 또는 연산자: substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length()
- coalesce() 그리고 nullif()
- cast(... as ...), 여기서 두 번째 아규먼트는 하나의 Hibernate 타입의 이름이고, 만일 ANSI

`cast()`과 `extract()`가 기본 데이터베이스에 의해 지원될 경우에는 `extract(... from ...)`

- `sign()`, `trunc()`, `rtrim()`, `sin()`과 같이 임의의 데이터베이스-지원되는 SQL 스칼라 함수
- JDBC IN 파라미터들 ?
- 명명된 파라미터들 `:name`, `:start_date`, `:x1`
- SQL 리터럴들 `'foo'`, `69`, `'1970-01-01 10:00:01.0'`
- Java `public static final` 상수들 eg. `Color.TABBY`

`in` 과 `between`은 다음과 같이 사용될 수도 있다:

```
select cat from DomesticCat cat where cat.name between 'A' and 'B'
```

```
select cat from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

and the negated forms may be written

```
select cat from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
select cat from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

마찬가지로, `is null`과 `is not null`이 null 값들을 테스트하는데 사용될 수 있다.

Boolean들은 Hibernate 구성에서 HQL 질의 치환을 선언함으로써 표현식들 내에 쉽게 사용될 수 있다:

```
hibernate.query.substitutions true 1, false 0
```

이것은 이 HQL로부터 변환된 SQL 내에서 `true` 및 `false` 키워드들을 리터럴들 1과 0으로 바꾸게 될 것이다:

```
select cat from Cat cat where cat.alive = true
```

당신은 특별한 프로퍼티 `size` 또는 특별한 `size()` 함수로서 컬렉션의 사이즈를 테스트할 수 있다 (HQL에 특징적임).

```
select cat from Cat cat where cat.kittens.size > 0
```

```
select cat from Cat cat where size(cat.kittens) > 0
```

인덱싱 된 컬렉션들의 경우, 당신은 `minindex` 함수와 `maxindex` 함수를 사용하여 최소 인덱스와 최대 인덱스를 참조할 수 있다. 유사하게 당신은 `minelement` 함수와 `maxelement` 함수를 사용하여 기본 타입을 가진 하나의 컬렉션의 최소 요소와 최대 요소를 참조할 수 있다. 이것들은 HQL에 특정한 특징들이다.

```
select cal from Calendar cal where maxelement(cal.holidays) > current date
```

```
select order from Order order where maxindex(order.items) > 100
```

```
select order from Order order where minelement(order.items) > 10000
```

SQL 함수들 `any`, `some`, `all`, `exists`, `in`은 하나의 컬렉션의 요소 세트 또는 인덱스 세트 (`elements` 함수와 `indices` 함수) 또는 하나의 서브질의 결과를 전달했을 때 지원된다. 서브질의들이 EJB-QL에 의해 지원되는 반면에, `elements`와 `indices`는 특정한 HQL 특징들이다.

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
select cat from Cat cat where exists elements(cat.kittens)
```

```
select cat from Player p where 3 > all elements(p.scores)
```

```
select cat from Show show where 'fizard' in indices(show.acts)
```

이들 구조체들 - size, elements, indices, minindex, maxindex, minelement, maxelement - 은 오직 Hibernate3에서 where 절 내에서만 사용될 수 있음을 노트하라.

HQL에서, 인덱싱 된 콜렉션들(배열들, 리스트들, map들)의 요소들은 (오직 where 절 내에서만) index에 의해 참조될 수 있다:

```
select order from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

[] 내의 표현식은 심지어 산술 표현식일 수 있다.

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL은 또한 하나의 one-to-many 연관 의 요소들 또는 값들을 가진 콜렉션의 요소들에 대해 미리 빌드된 index() 함수를 제공한다.

```
select item, index(item) from Order order
join order.items item
where index(item) < 5
```

기본 데이터베이스에 의해 지원되는 스칼라 SQL 함수들이 사용될 수 있다

```
select cat from DomesticCat cat where upper(cat.name) like 'FRI%'
```

만일 당신이 아직 이 모든 것이 확인되지 않을 경우, 다음 질의가 SQL에서 얼마나 더 가독적이 있는지 덜 가독적인지를 생각하라:

```
select cust
from Product prod,
Store store
inner join store.customers cust
where prod.name = 'widget'
and store.location.name in ( 'Melbourne', 'Sydney' )
and prod = all elements(cust.currentOrder.lineItems)
```

힌트: 다음과 같은 것

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
     stores store,
     locations loc,
     store_customers sc,
     product prod
WHERE prod.name = 'widget'
     AND store.loc_id = loc.id
     AND loc.name IN ( 'Melbourne', 'Sydney' )
     AND sc.store_id = store.id
     AND sc.cust_id = cust.id
     AND prod.id = ALL(
         SELECT item.prod_id
         FROM line_items item, orders o
         WHERE item.order_id = o.id
             AND cust.current_order = o.id
     )
```

7.9. order by 절

하나의 질의에 의해 반환되는 리스트는 반환된 클래스 또는 컴포넌트들의 임의의 프로퍼티에 의해 순서지워질 수 있다:

```
select cat from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

선택적인 `asc` 또는 `desc`는 각각 오름차순 또는 내림차순을 나타낸다.

7.10. group by 절

집계 값들을 반환하는 하나의 질의는 반환된 클래스 또는 컴포넌트들의 임의의 프로퍼티에 의해 그룹지워질 수 있다:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

하나의 `having` 절이 또한 허용된다.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

SQL 함수들과 집계 함수들이 기본 데이터베이스에 의해 지원될 경우에, `having` 절과 `order by` 절 속에 허용된다. (예를 들어 MySQL에서는 지원되지 않음).

```
select cat
from Cat cat
     join cat.kittens kitten
group by cat
```

```
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

group by 절이든 order by 절이든 어느 것도 산술 표현식들을 포함할 수 없음을 노트하라.

7.11. 서브질의들

subselect들을 지원하는 데이터베이스들에 대해, EJB-QL은 질의들 내에 서브질의들을 지원한다. 하나의 서브질의는 (흔히 SQL 집계함수 호출에 의해) 괄호들에 의해 둘러싸워져야 한다. 심지어 서로 관련된 서브질의들(outer 질의 내에서 하나의 alias를 참조하는 서브질의들)이 허용된다.

```
select fatcat from Cat as fatcat
where fatcat.weight > (
  select avg(cat.weight) from DomesticCat cat
)
```

```
select cat from DomesticCat as cat
where cat.name = some (
  select name.nickName from Name as name
)
```

```
select cat from Cat as cat
where not exists (
  from Cat as mate where mate.mate = cat
)
```

```
select cat from DomesticCat as cat
where cat.name not in (
  select name.nickName from Name as name
)
```

select list 내에 하나 이상의 표현식을 가진 서브질의들의 경우에, 당신은 하나의 튜플(tuple) 생성자를 사용할 수 있다:

```
select cat from Cat as cat
where not ( cat.name, cat.color ) in (
  select cat.name, cat.color from DomesticCat cat
)
```

(Oracle이나 HSQLDB가 아닌) 몇몇 데이터베이스들 상에서, 당신은 예를 들어 컴포넌트들이나 합성(composite) 사용자 타입들을 질의할 때 다른 컨텍스트들 내에서 튜플(tuple) 생성자들을 사용할 수 있음을 노트하라:

```
select cat from Person where name = ('Gavin', 'A', 'King')
```

Which is equivalent to the more verbose:

```
select cat from Person where name.first = 'Gavin' and name.initial = 'A' and name.last = 'King')
```

당신이 이런 종류의 것을 행하고자 원하자 않는 두 가지 좋은 이유들이 존재한다: 첫 번째로, 그것은 데이터베이스 플랫폼들 사이에서 완전하게 이식가능하지 않다; 두 번째로 그 질의는 이제 매핑 문서 내에 있는 프로퍼티들의 순서에 종속된다.

7.12. EJB-QL 예제들

Hibernate 질의들은 꽤 강력하고 복잡해질 수 있다. 사실, 질의 언어의 힘은 Hibernate(그리고 이제 EJB-QL)의 주요 판매 포인트들 중 하나이다. 다음은 내가 최근의 프로젝트에서 사용했던 질의들과 매우 유사한 몇몇 예제 질의들이다. 당신이 작성하게 될 대부분의 질의들은 이것들과 훨씬 유사함을 노트하라!

다음 질의는 합계값에 따라 결과들을 순서지우는, 주문 id, 아이템들의 개수, 그리고 특정 고객에 대해 모든 지불되지 않은 주문들에 대한 합계 값, 그리고 주어진 최소 합계를 반환한다. 가격 결정에 있어, 그것은 현재의 카다록을 사용한다. ORDER, ORDER_LINE, PRODUCT, CATALOG 그리고 PRICE 테이블들에 대해 귀결되는 SQL 질의는 네 개의 inner join들과 한 개의 (상관되지 않은) subselect를 갖는다.

```
select order.id, sum(price.amount), count(item)
from Order as order
  join order.lineItems as item
  join item.product as product,
  Catalog as catalog
  join catalog.prices as price
where order.paid = false
  and order.customer = :customer
  and price.product = product
  and catalog.effectiveDate < sysdate
  and catalog.effectiveDate >= all (
    select cat.effectiveDate
    from Catalog as cat
    where cat.effectiveDate < sysdate
  )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

괴물 같은 것! 실제로 실 세계에서, 나는 서브질의들을 매우 좋아하지 않아서, 나의 질의는 실제로 다음과 같았다:

```
select order.id, sum(price.amount), count(item)
from Order as order
  join order.lineItems as item
  join item.product as product,
  Catalog as catalog
  join catalog.prices as price
where order.paid = false
  and order.customer = :customer
  and price.product = product
  and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

다음 질의는 가장 최근의 상태 변경이 현재 사용자에게 의해 행해졌던 AWAITING_APPROVAL 상태에 있지 않는 모든 지불들을 제외한, 각각의 상태에 있는 지불들의 개수를 카운트한다. 그것은 PAYMENT, PAYMENT_STATUS 그리고 PAYMENT_STATUS_CHANGE 테이블들에 대해 두 개의 inner 조인들과 한 개의 상관된 subselect를 가진 하나의 SQL로 변환된다.

```
select count(payment), status.name
from Payment as payment
  join payment.currentStatus as status
  join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
  or (
    statusChange.timeStamp = (
      select max(change.timeStamp)
```

```

        from PaymentStatusChange change
        where change.payment = payment
    )
    and statusChange.user <> :currentUser
)
group by status.name, status.sortOrder
order by status.sortOrder

```

만일 내가 `statusChanges` 콜렉션을 하나의 set이 아닌, 하나의 리스트로서 매핑했다면, 그 질의는 작성하기가 훨씬 더 간단했을 것이다.

```

select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder

```

하지만 그 질의는 HQL 특징적인 것이었다.

다음 질의는 현재 사용자가 속해 있는 조직에 대한 모든 계정들과 지불되지 않은 지불들을 반환하는데 MS SQL Server `isNull()` 함수를 사용한다. 그것은 `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` 그리고 `ORG_USER` 테이블들에 대해 세 개의 inner 조인과, 한 개의 outer 조인과 한 개의 subselect를 가진 한 개의 SQL로 변환된다.

```

select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate

```

7.13. 대용량 UPDATE & DELETE 문장들

Hibernate는 이제 HQL/EJB-QL 내에 UPDATE 문장과 DELETE 문장을 지원한다. 상세한 것은 6.1절 “대용량 update/delete”를 보라.

7.14. 팁들 & 트릭들

하나의 콜렉션의 사이즈에 따라 결과를 순서지우는데 다음 질의를 사용하라:

```

select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)

```

만일 당신의 데이터베이스가 subselect들을 지원할 경우, 당신은 당신의 질의의 where 절 속에 select 사이즈에 따른 조건을 위치시킬 수 있다:

```

from User usr where size(usr.messages) >= 1

```

만일 당신의 데이터베이스가 subselect들을 지원하지 않을 경우, 다음 질의를 사용하라:

```
select usr.id, usr.name
from User usr.name
      join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

이 해결책이 inner 조인 때문에 0개의 메시지들을 가진 한 개의 User를 반환할 수 없으므로, 다음 형식이 또한 유용하다:

```
select usr.id, usr.name
from User as usr
      left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

8장. Native 질의

당신은 또한 당신의 데이터베이스의 native SQL dialect 내에 질의들을 표현할 수 있다. 당신이 Oracle에 있는 질의 힌트들 또는 CONNECT BY 옵션과 같은 데이터베이스에 특정한 특징들을 활용하고자 원할 경우에 이것이 유용하다. 그것은 또한 하나의 직접적인 SQL/JDBC 기반의 어플리케이션으로부터 Hibernate로의 하나의 명료한 이전 경로를 제공한다. Hibernate3은 모든 create, update, delete, 그리고 load 연산들에 대해 (내장 프로시저를 포함한) 손으로 작성된 SQL을 지정하는 것을 당신에게 허용해줌을 노트하라(추가 정보는 참조 안내서를 참조하길 바란다.)

8.1. 결과셋을 표현하기

하나의 SQL 질의를 사용하기 위해, 당신은 SQL 결과셋을 설명할 필요가 있고, 이 설명은 EntityManager으로하여금 당신의 컬럼들을 엔티티 프로퍼티들 상으로 매핑시키기는 것을 도와줄 것이다. 이것은 @SqlResultSetMapping 주해를 사용하여 행해진다. 각각의 @SqlResultSetMapping은 EntityManager 상에 한 개의 SQL 질의를 생성시킬 때 사용되는 한 개의 이름을 갖는다.

```
@SqlResultSetMapping(name="GetNightAndArea", entities={
    @EntityResult(name="org.hibernate.test.annotations.query.Night", fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id")
    }),
    @EntityResult(name="org.hibernate.test.annotations.query.Area", fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
})
//or
@SqlResultSetMapping(name="defaultSpaceShip", entities=@EntityResult(name="org.hibernate.test.annotations.query.SpaceShip", fields = {
    @FieldResult(name="id", column="id"),
    @FieldResult(name="name", column="name")
}))
```

@SqlResultSetMapping.에 대한 추가 정보는 Hibernate Annotations 참조 안내서를 참조하길 바란다.

참고

현재의 구현은 native SQL 질의들 내에 스칼라 결과들을 지원하지 않는다.

8.2. native SQL 질의들을 사용하기

TODO: 이것은 사기처럼 들린다...

결과 셋이 설명되고, 우리가 native SQL 질의를 실행하는 가용성이 있다는 점을 노트하라. EntityManager는 모든 필요한 API들을 제공한다. 첫 번째 메소드는 바인딩을 행하는데 한 개의 SQL 결과셋 이름을 사용하는 것이고, 두 번째 메소드는 엔티티 디폴트 매핑을 사용한다(반환된 컬럼은 매핑에서 사용된 컬럼 이름과 동일한 이름을 가져야 한다). (아직 Hibernate 엔티티 관리자에 의해 지원되지 않는) 세 번째 메소드는 순수 스칼라 결과들을 반환한다.

```
String sqlQuery = "select night.id nid, night.night_duration, night.night_date, area.id aid, "
    + "night.area_id, area.name from Night night, Area area where night.area_id = area.id "
    + "and night.night_duration >= ?";
```

```
Query q = entityManager.createNativeQuery(sqlQuery, "GetNightAndArea");
q.setParameter( 1, expectedDuration );
q.getResultList();
```

이 native 질의는 GetNightAndArea 결과 셋에 기반하여 night들과 area를 반환한다.

```
String sqlQuery = "select * from tbl_spaceship where owner = ?";
Query q = entityManager.createNativeQuery(sqlQuery, SpaceShip.class);
q.setParameter( 1, "Han" );
q.getResultList();
```

두 번째 버전은 당신의 SQL 질의가 메타데이터 내에 매핑된 컬럼들과 동일한 컬럼들을 재사용하여 한 개의 엔티티를 반환할 때 유용하다.

8.3. 명명된 질의들

native 명명된 질의들은 EJB-QL 명명된 질의들과 동일한 호출 API를 공유한다. 당신의 코드는 둘 사이의 차이점을 알 필요가 없다. 이것은 SQL로부터 EJB-QL로의 이관에 매우 유용하다:

```
Query q = entityManager.createNamedQuery("getSeasonByNativeQuery");
q.setParameter( 1, name );
Season season = (Season) q.getSingleResult();
```

부록 A. 준수사항과 알려진 제한사항들

native 질의들 내에서 스칼라 결과들을 지원한다