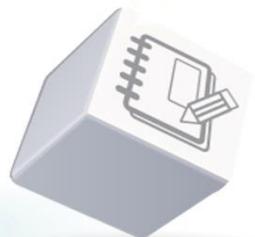


데이터 접근프레임워크 (JPA와 Hibernate)

교육기간 : 2014.05.26 ~ 05.30

강사 : 박석재, 임병인

넥스트리소프트(주)
demonpark@nextree.co.kr
byleem@nextree.co.kr



교육 일정표

✓ 교육은 매 회 3 시간씩 총 5회에 걸쳐 진행합니다.

1 일차	2 일차	3 일차	4 일차	5 일차
5월 26일(월)	5월 27일(화)	5월 28일(수)	5월 29일(목)	5월 30일(금)
ORM 이해하기 <ul style="list-style-type: none">• 영속성• 패러다임 불일치• 계층형 아키텍처• 객체/관계형 매핑• Quick Start	<ul style="list-style-type: none">- Hibernate• Hibernate 개요• Hibernate 기본• OR Mapping I	<ul style="list-style-type: none">- Hibernate• Spring과 Hibernate 통합• OR Mapping II• 트랜잭션 관리• Hibernate와 EJB	<ul style="list-style-type: none">- JPA• 도메인 모델과 JPA• JPA 이용 도메인 객체 구현• 엔티티 관계• 엔티티 매핑• 엔티티 관계 매핑	<ul style="list-style-type: none">- JPA• 상속 매핑• Entity Manager• 질의 API• JPQL



4일차 – JPA

4.1 도메인 모델과 JPA

4.2 JPA이용 도메인 객체 구현

4.3 엔티티 관계

4.4 엔티티 매핑

4.5 엔티티 관계 매핑

- 도메인 모델 소개
- 문제 도메인
- 도메인 모델의 액터
- EJB3 자바 지속성 API
- 자바 클래스로서의 도메인 객체

도메인 모델 소개

□ 도메인 모델

- 시스템을 통해 해결하려고 하는 문제의 개념적인 이미지
- 객체와 객체 들의 연관관계(association)로 구성함
- 도메인 모델의 객체는 물리적인 객체가 아니라 시스템에서 사용하는 개념임
- 비즈니스 룰과 비즈니스 로직은 도메인 모델 위에서 실행됨(act on the domain model)

□ Martin Fowler 가 도메인 모델의 개념을 처음 사용

문제 도메인 [1/3]

□ 분석 방향

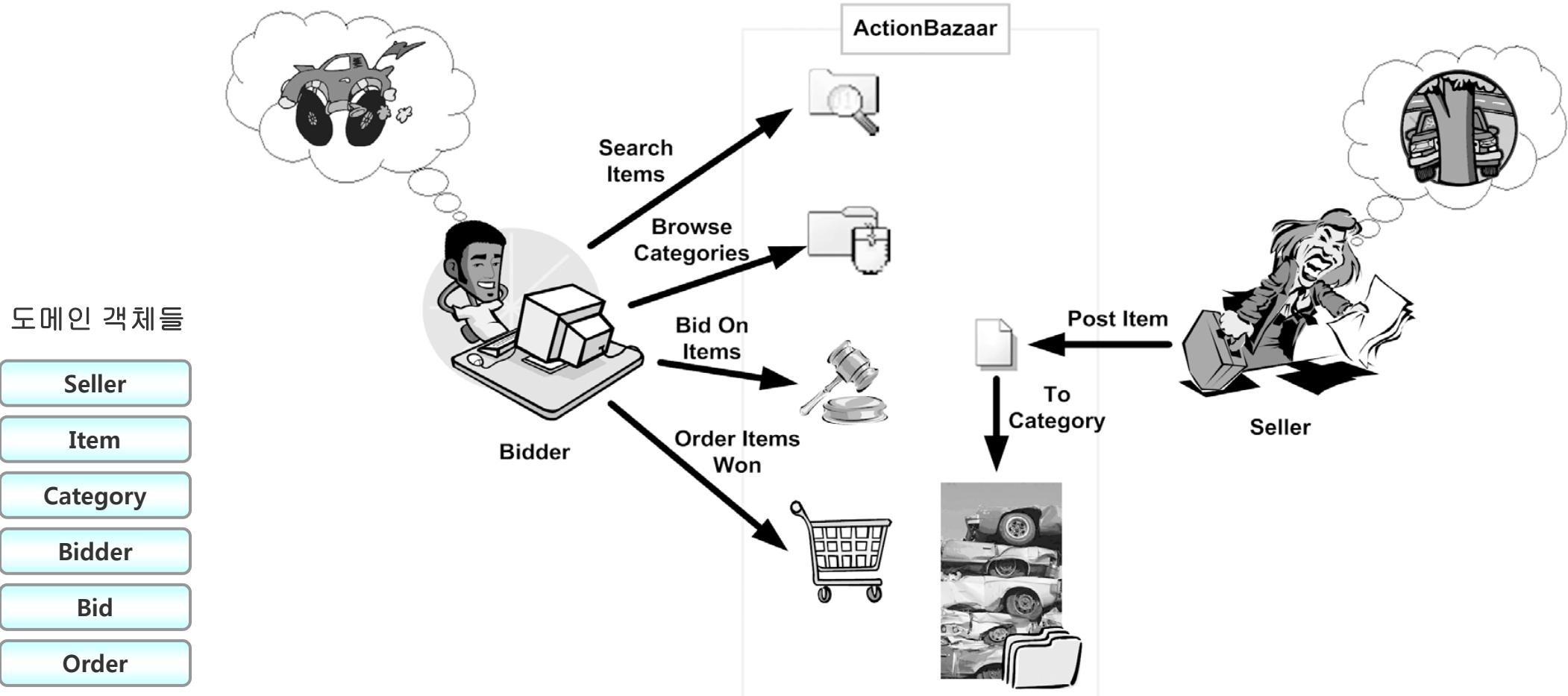
- JPA를 설명하는 꼭 필요한 도메인 부분만 포함
- 온라인 옵션에서 품목을 사고 파는데 직접 관련된 기능만 대상으로 함
- 현실적인 예와는 약간의 거리를 두고 핵심 개념은 정확히 반영

□ 도메인 내용

- 판매자(Seller)는 품목(item)을 ActionBazaar 사이트에 올린다.
- 품목은 검색가능(searchable) 카테고리와 항해가능(navigable) 카테고리로 구성된다
-
- 경매 참여자(Bidder)는 품목에 입찰(bid)한다.
- 가장 높은 금액을 써낸 사람이 이긴다.

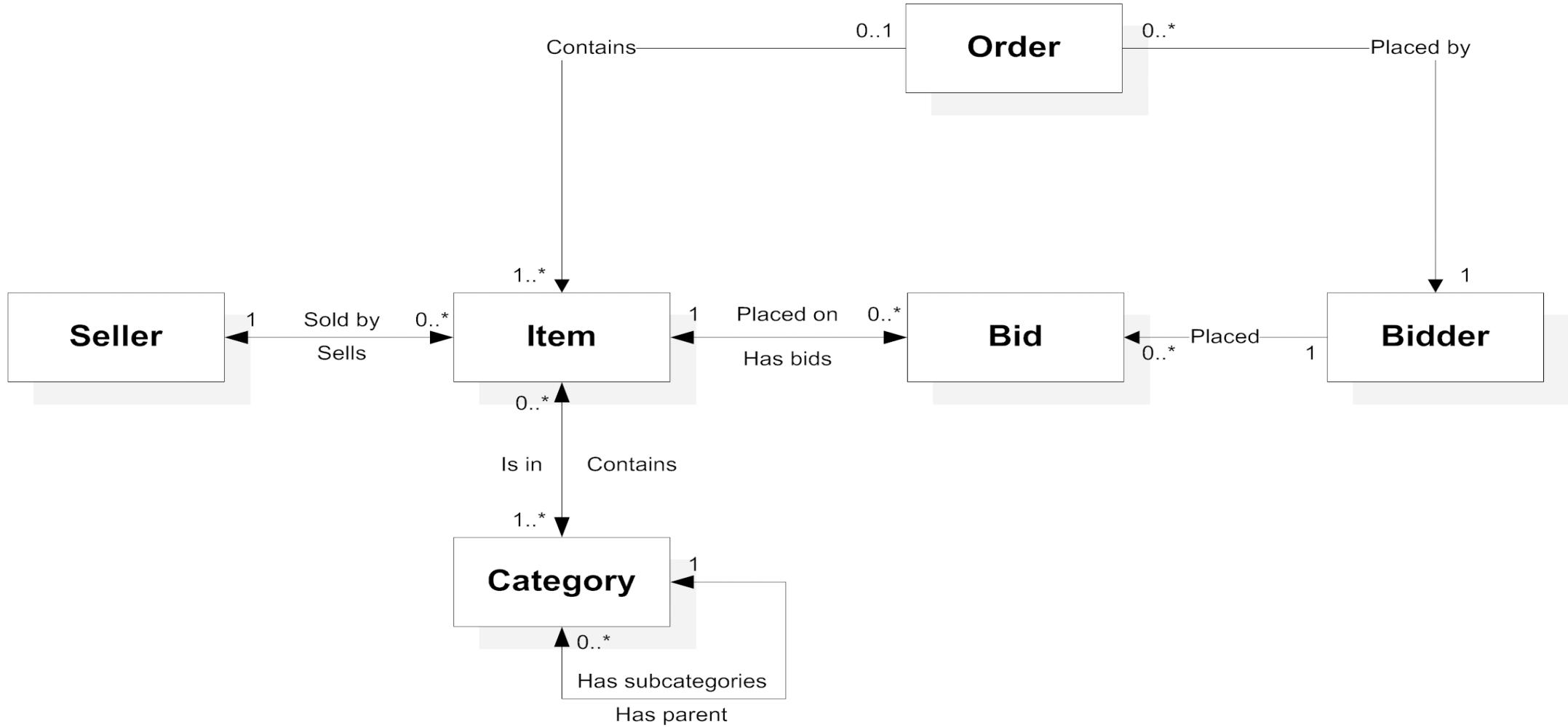
문제 도메인 (2/3)

- 핵심기능-판매자는 품목을 검색가능한 카테고리로 등록하고, 입찰자는 품목에 입찰하고, 가장 높은 입찰금액을 쓴 입찰자가 이긴다.



문제 도메인 (3/3)

□ 엔티티와 관계로 구성된 도메인 모델



도메인 모델의 액터 (1/2)

□ 관계를 통해 파악할 수 있는 정보들

- 품목은 판매자에 의해 판매되고, 판매자는 여러 품목을 팔 수 있다.
- 품목은 하나 이상의 카테고리에 속하고, 각 카테고리는 한 부모 카테고리를 가진다.
- 입찰자는 품목에 입찰한다.

□ 도메인 모델은, 객체가 다루어지는 방식을 서술하지는 않음

- 주문이 하나 이상의 품목으로 구성되며, 입찰자가 주문한다는 사실은 알 수 있지만,
- 언제 어떻게 이 관계가 형성되는 지 알 수 없다.
- 비즈니스 로직과 비즈니스 규칙에 의해 밝혀진다.

도메인 모델의 액터 (2/2)

□ 객체(object)

- 도메인 객체는 자바의 클래스와 동일한 개념
- 도메인 객체 Category의 인스턴스는 수백개가 될 수 있음

□ 관계 (relationship)

- 한 도메인 객체가 다른 도메인 객체를 참조하고 있음
- 관계의 방향은 일방향이거나 양방향임

□ 개수(multiplicity, or cardinality)

- 1:1, 1:N, N:N 등의 개수가 존재함

□ 선택사항(optionality)

- cardinality가 0..1, 0..n, 0..2 등으로 관계의 대상이 존재하지 않을 수도 있음
- 예, 사용자(User)는 청구정보(BillingInfo)를 가질 수도 있고 없을 수도 있음

□ JPA(Java Persistence API)

- 메타 데이터 – 기반 POJO 기술임, 따라서 Java 객체 저장을 위해 특정 인터페이스 구현, 클래스 상속, 프레임워크 패턴 준수 등을 하지 않음
- 저장되는 객체는 JPA 관련 코드를 한 줄도 가지지 않음
- POJO로 구성한 도메인 모델과 annotation 또는 Xml만 있으면 됨

□ Annotation/Xml 포함 정보

- 도메인 객체가 무엇인가? (@Entity, @Embeded)
- 도메인 객체를 어떻게 식별하는가? (@Id)
- 객체들 간의 관계유형은 무엇인가?
(@OneToOne, @OneToMany, @ManyToMany)
- 도메인 객체가 어떻게 DB에 매핑되는가? (@Table, @Column, @JoinColumn)

□ JPA를 사용한 O/R 매핑이 EJB2의 엔티티나 JDBC에 비해 많이 개선된 것이긴 하지만 여전히 매핑 관련 복잡한 작업을 해야 함

자바 클래스로서의 도메인 객체

□ POJO 구성요소

- id – 객체 식별자
- name,modificationDate – 객체 속성
- items,
parentCategory,subCategories – 관계를 위한 객체 속성
- 객체 속성을 위한 getter/setter
- 관계 객체 속성을 위한 getter/setter

```
public class Category {  
    protected Long id;  
    protected String name;  
    protected Date modificationDate;  
    protected Set<Item> items;  
    protected Category parentCategory;  
    protected Set<Category> subCategories;  
  
    public Category() {}  
    public Long getId() {  
        return this.id;  
    }  
    public void setId(Long id) {  
        this.id = id;  
    }  
    public String getName() {  
        return this.name;  
    }  
    ...  
    ...  
    public Category getParentCategory() {  
        return this.parentCategory;  
    }  
    public void setParentCategory(Category parentCategory) {  
        this.parentCategory = parentCategory;  
    }  
}
```



4일차 – JPA

4.1 도메인 모델과 JPA

4.2 JPA이용 도메인 객체 구현

4.3 엔티티 관계

4.4 엔티티 매핑

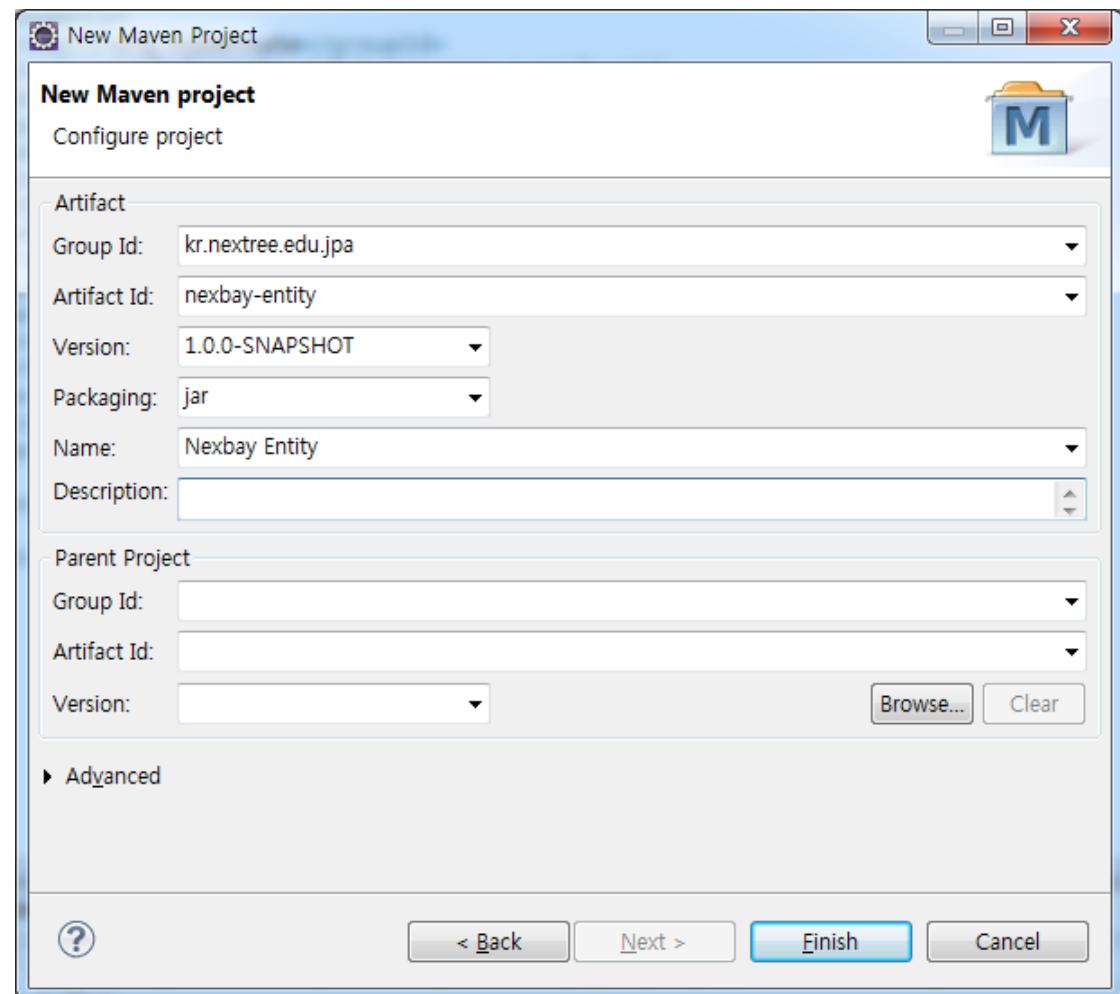
4.5 엔티티 관계 매핑

- 테스트 프로젝트 생성
- @Entity annotation
- 엔티티 데이터 저장
- 엔티티 식별자(identity) 명세
- @Embeddable annotation

테스트 프로젝트 생성 (1/3)

□ Maven Project

- Group Id: kr.nextree.edu.jpa
- Artifact Id: nexbay-entity
- Version: 1.0.0-SNAPSHOT
- Packaging: jar
- Name: Nexbay Entity



테스트 프로젝트 생성 (2/3)

□ 의존관계 생성

```
<dependencies>
    .....
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-annotations</artifactId>
        <version>3.5.5-Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.3.3.Final</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.7</version>
        <scope>test</scope>
    </dependency>
    .....
```

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.7</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.3.2</version>
</dependency>
.....
</dependencies>
```

테스트 프로젝트 생성 (3/3)

□ HSQLDB Server Mode

- 참고: <http://hsqldb.org/doc/guide/ch01.html>
- hsqldb-server-run.cmd 파일 생성

```
java -cp ./lib/hsqldb-2.3.2.jar org.hsqldb.Server -database.0 file:/z-nexbay-data/nexbaydb -dbname.0 nexbaydb
```

- hsqldb-manager-run.cmd 파일 생성

```
java -cp ./lib/hsqldb-2.3.2.jar org.hsqldb.util.DatabaseManager
```

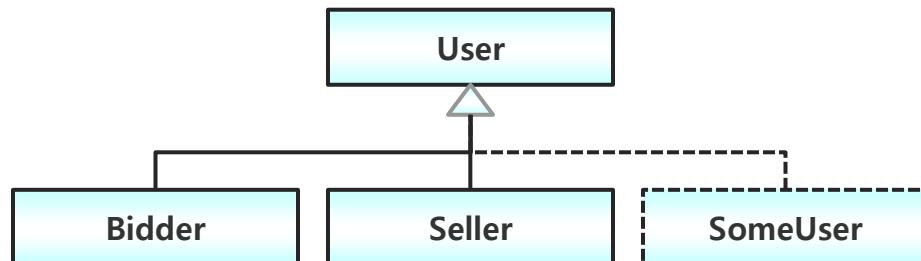
- jdbc 연결정보
 - jdbc:hsqldb:hsqldb://localhost:9001/nexbaydb

@Entity 주석(annotation) [1/2]

- @Entity는 POJO가 고유하게 식별되는 도메인 객체임을 알림

```
@Entity  
public class Category {  
    ...  
    public Category() { ... }  
    public Category(String name) { ... }  
    ...  
}
```

- 엔티티는 완전한 POJO이므로 객체지향 특성을 100% 활용할 수 있음
 - 다형성(polymorphism)
 - 상속(inheritance)



@Entity 주석(annotation) [2/2]

□ 상속과 지속성(persistence)

- User가 엔티티이므로 User의 모든 속성(userId, userName, email, ...)은 Seller나 Bidder가 저장될 때 함께 저장됨
- User 자체가 직접 인스턴스가 되거나 저장되어서는 안될 경우, abstract class로 선언함

```
@Entity  
public class User {  
    ...  
    String userId;  
    String userName;  
    String email;  
    ...  
}  
  
@Entity  
public class Seller extends User {  
    ...  
}  
  
@Entity  
public class Bidder extends User {  
    ...  
}
```

엔티티 데이터 저장 [1/3]

□ 접근 방식 정의

- 필드-기반 접근: 필드나 인스턴스 변수를 이용하여 O/R 매핑을 정의
- 속성-기반 접근: 속성(property)을 사용하여 O/R 매핑을 정의(getter/setter 사용)

□ 필드 기반 접근

- POJO의 저장 대상 데이터 필드를 public이나 protected로 선언
- Persistence Provider에게 getter/setter를 무시하도록 함
- 인스턴스 변수에 @XXX annotation을 사용

```
@Entity  
public class Category {  
    @Id  
    public Long id;  
    public String name;  
    public Date modificationDate;  
    ...  
    public Category() { ... }  
}
```

엔티티 데이터 저장 [2/3]

□ @Transient annotation

- 해당 필드는 저장하지 않음
- 주로 실행 시점에 사용하는 정보이거나, 파생 속성(derived attribute) 임

```
@Entity  
public class Category {  
    @Transient  
    protected Long activeUserCount;  
    transient public String generatedName;  
    ...  
    public Category() { ... }  
}
```

- 속성(property) 기반 접근일 경우, @Transient 태그를 getter에 붙임
- transient 키워드는 @Transient 태그와 동일한 효과를 가짐

엔티티 데이터 저장 (3/3)

□ 데이터 타입의 상이함

- POJO의 인스턴스 변수 타입과 테이블의 타입이 일치하지 않음
- 따라서, 둘 간에 약간의 제약조건이 있음

□ Persistence 필드/속성이 가능한 데이터 타입

- Java 원시 타입: int, double, long 등
- Java 원시 타입 래퍼: Integer, Double
- String 타입: String
- Java API Serializable 타입: BigInteger, Date
- 사용자 정의 Serializable 타입: Serializable
- 배열 타입: byte[], char[]
- 열거 타입: {SELLER, bIDDER, CSR, ADMIN}
- 엔티티의 집합: Set<Category>
- 임베디드 클래스: @Embeddable 태그를 붙여 정의한 클래스

엔티티 식별자(identity) 명세 (1/4)

□ 모든 엔티티는 유일하게 식별이 되어야 함 (uniquely identifiable)

- 어느 시점에 엔티티는 유일하게 식별되는 테이블의 row로 저장되어야 하기 때문
- 테이블은 일차 키(primary key)로 유일하게 식별함

□ 객체 구분

- 서로 다른 객체: 객체의 식별자(identity)가 다름
- 서로 같은 객체: 객체의 식별자(identity)가 같음

□ 객체 비교

- Object 클래스의 equals 메소드 이용
- 예, name 이 유일성 식별자 인 경우,

```
public boolean equals (Object other) {  
    if (other instanceof Category) {  
        return this.name.equals(((Category)other).name)  
    } else {  
        return false;  
    }  
}
```

- 식별자로는 Long id와 같이 Long 타입이 더 경제적 임

엔티티 식별자(identity) 명세 [2/4]

□ Persistence Provider에게 identity가 저장된 위치를 알려주는 방법

- @Id (javax.persistence.Id)
- @IdClass (javax.persistence.IdClass)
- @EmbeddedId (javax.persistence.EmbeddedId)

□ @Id annotation

- 필드나 속성에 지정하고 이는 테이블의 일차 키(primary key)가 됨
- 속성(property) 기반 접근의 예

```
@Entity  
public class Category {  
    ...  
    protected Long id;  
    ...  
    @Id  
    public Long getId() {  
        return this.id;  
    }  
    public void setId(Long id) {  
        this.id = id;  
    }  
    ...  
}
```

**피해야 할 타입

*float, Float, double 등을 피해야 함
TimeStamp 타입을 피해야 함*

엔티티 식별자(identity) 명세 (3/4)

□ @IdClass

- 하나 이상의 @Id를 사용할 경우 @IdClass를 사용
- 즉, 복합(composite) 키가 필요할 경우 사용

```
public class CategoryPK implements Serializable {  
    String name;  
    Date createDate;  
    public CategoryPK() {}  
  
    public boolean equals(Object other) {  
        if (other instanceof CategoryPK) {  
            final CategoryPK otherCategoryPK = (CategoryPK)other;  
            return (otherCategory.name.equals(name) &&  
                    otherCategory.createDate.equals(createDate));  
        }  
        return false;  
    }  
    public int hashCode() {  
        return super.hashCode();  
    }  
}
```

```
@Entity  
@IdClass(CategoryPK.class)  
public class Category {  
    public Category() {}  
    @Id  
    protected String name;  
    @Id  
    protected Date createDate;  
    ...  
}
```

엔티티 식별자(identity) 명세 (4/4)

□ @EmbeddedId

- IdClass를 엔티티 안으로 가져와서 내포된 유일성 필드를 사용하는 것과 같음

```
@Embeddedable  
public class CategoryPK implements Serializable {  
    String name;  
    Date createDate;  
    public CategoryPK() {}  
  
    public boolean equals(Object other) {  
        if (other instanceof CategoryPK) {  
            final CategoryPK otherCategoryPK = (CategoryPK)other;  
            return (otherCategory.name.equals(name) &&  
                    otherCategory.createDate.equals(createDate));  
        }  
        return false;  
    }  
    public int hashCode() {  
        return super.hashCode();  
    }  
}
```

```
@Entity  
public class Category {  
    public Category() {}  
    @EmbeddedId  
    protected CategoryPK categoryPK;  
  
    ...  
}
```

@Embeddable 주석(annotation)

□ Embeddable

- 자체 identity는 필요없고 다른 엔티티에 포함되는 객체
- 자체로 저장되거나 접근되지 않음
- @Embeddable 객체는 identity를 가질 수 없음
- @Embeddable 객체는 엔티티 객체와 같은 테이블에 저장됨

```
@Embeddedable  
public class Address {  
    protected String streetLine1;  
    protected String streetLine2;  
    protected String city;  
    protected String state;  
    protected String zipCode;  
    protected String country;  
    ...  
}
```

```
@Entity  
public class User {  
    @Id  
    protected Long id;  
    protected String username;  
    protected String firstName;  
    protected String lastName;  
    @Embedded  
    protected Address address;  
    protected String email;  
    protected String phone;  
    ...  
}
```



4일차 – JPA

4.1 도메인 모델과 JPA

4.2 JPA이용 도메인 객체 구현

4.3 엔티티 관계

4.4 엔티티 매팅

4.5 엔티티 관계 매팅

- @OneToOne
- @OneToMany, @ManyToOne
- @ManyToMany

@OneToOne (1/3)

□ @OneToOne

- 단방향 일대일
- 양방향 일대일

□ 단방향

- 예, 사용자는 청구정보를 가질 수 있다.
- 필드 기반 접근

```
@Entity  
public class User {  
    @Id  
    protected String userId;  
    protected String email;  
    @OneToOne  
    protected BillingInfo billingInfo;  
}
```



```
@Entity  
public class BillingInfo {  
    @Id  
    protected Long billingId;  
    protected String creditCardType;  
    protected String creditCardNumber;  
    protected String nameOnCreditCard;  
    protected Date creditCardExpiration;  
    protected String bankAccountNumber;  
    protected String bankName;  
    protected String routingNumber;  
}
```

@OneToOne [2/3]

- 속성 기반 접근 단방향 관계

```
@Entity  
public class User {  
    @Id  
    protected String userId;  
    protected String email;  
    protected BillingInfo billing;  
  
    @OneToOne  
    public BillingInfo getBilling() {  
        return this.billing;  
    }  
    public void setBilling(BillingInfo billing) {  
        this.billing = billing;  
    }  
}
```

```
@Entity  
public class BillingInfo {  
    @Id  
    protected Long billingId;  
    protected String creditCardType;  
    protected String creditCardNumber;  
    protected String nameOnCreditCard;  
    protected Date creditCardExpiration;  
    protected String bankAccountNumber;  
    protected String bankName;  
    protected String routingNumber;  
}
```

@OneToOne [3/3]

□ 양방향 일대일

- 두 개의 @OneToOne 태그를 사용함

```
@Entity  
public class User {  
    @Id  
    protected String userId;  
    protected String email;  
    @OneToOne  
    protected BillingInfo billingInfo;  
}
```

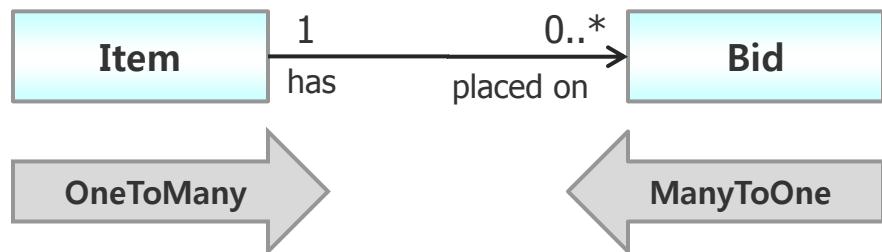
```
@Entity  
public class BillingInfo {  
    @Id  
    protected Long billingId;  
    protected String creditCardType;  
    ...  
    protected String routingNumber;  
    @OneToOne(mappedBy="billingInfo", optional="false");  
    protected User user;  
}
```

- mappedBy="billingInfo" 의 의미
 - 관계 소유자가 User 임
 - User 클래스 안의 billingInfo 인스턴스 변수로 매팅됨
- optional="false"의 의미
 - BillingInfo 객체는 User 객체없이 단독으로 저장될 수 없음

@OneToMany, @ManyToOne

□ @OneToMany, @ManyToOne

- 가장 흔한 관계임
- Set이나 List를 사용



```
@Entity  
public class Item {  
    @Id  
    protected Long itemId;  
    protected String title;  
    protected String description;  
    protected Date postdate;  
    ...  
    @OneToMany(mappedBy="item")  
    protected Set<Bid> bids;  
    ...  
}
```

```
@Entity  
public class Bid {  
    @Id  
    protected Long bidId;  
    protected Double amount;  
    protected Date timestamp;  
    ...  
    @ManyToOne  
    protected Item item;  
    ...  
}
```

@ManyToMany

□ @ManyToMany

- 흔하지 않고 가끔씩 나타나는 관계 유형으로 기본적으로 양방향임
- 양쪽 모두 관련 엔티티 다수를 참조하는 관계



- Category 는 여러 개의 Item을 포함하고 있음, Item은 여러 Category에 속함

```
@Entity  
public class Category {  
    @Id  
    protected Long categoryId;  
    protected String name;  
    ...  
    @ManyToMany  
    protected Set<Item> items;  
    ...  
}
```

```
@Entity  
public class Item {  
    @Id  
    protected Long itemId;  
    protected String title;  
    ...  
    @ManyToMany(mappedBy="items")  
    protected Set<Category> categories;  
    ...  
}
```



4일차 – JPA

4.1 도메인 모델과 JPA

4.2 JPA이용 도메인 객체 구현

4.3 엔티티 관계

4.4 엔티티 매핑

4.5 엔티티 관계 매핑

- 개요
- 테이블 지정
- 컬럼 매핑
- @Enumerated 사용
- CLOB와 BLOB 매핑
- 날짜 타입 매핑
- 단일 엔티티 복수 테이블 매핑
- 일차 키 생성
- 내장 가능 클래스 매핑

□ 다양한 annotation을 이용한 매팅

```
@Entity  
@Table(name="USERS")  
@SecondaryTable(name="USER_PICTURES",  
pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))  
public class User implements Serializable {  
    @Id  
    @Column(name="USER_ID", nullable=false)  
    protected Long userId;  
    @Column(name="USER_NAME", nullable=false)  
    protected String username;  
    @Column(name="FIRST_NAME", nullable=false, length=1)  
    protected String firstName;  
    @Column(name="LAST_NAME", nullable=false)  
    protected String lastName;  
    @Enumerated(EnumType.ORDINAL)  
    @Column(name="USER_TYPE", nullable=false)  
    protected UserType userType;  
    @Column(name="PICTURE", table="USER_PICTURES")  
    @Lob  
    @Basic(fetch=FetchType.LAZY)  
    protected byte[] picture;
```

```
        @Column(name="CREATION_DATE", nullable=false)  
        @Temporal(TemporalType.DATE)  
        protected Date creationDate;  
        @Embedded  
        protected Address address;  
        public User() {}  
    }  
  
    @Embeddable  
    public class Address implements Serializable {  
        @Column(name="STREET", nullable=false)  
        protected String street;  
        @Column(name="CITY", nullable=false)  
        protected String city;  
        @Column(name="STATE", nullable=false)  
        protected String state;  
        @Column(name="ZIP_CODE", nullable=false)  
        protected String zipCode;  
        @Column(name="COUNTRY", nullable=false)  
        protected String country;  
    }
```

테이블 지정

□ @Table 주석

- 엔티티가 매핑될 테이블을 지정하는 주석(annotation)
- name, catalog, schema 등의 파라미터를 가짐 (catalog,schema는 거의 사용하지 않음)
- @Table은 선택사항임, 지정되지 않을 경우 클래스과 같은 이름의 테이블로 매핑
- 대부분의 Persistence 제공자들은 스키마 자동 생성 기능을 제공함, 하지만 단순한 스키마에만 적용해야 함
- User 클래스를 USERS 테이블로 매핑하는 예

```
@Table(name="USERS", schema="ACTIONBAZAAR")  
public class User
```

- uniqueConstraints 파라미터 사용 예

```
@Table(name="CATEGORIES",  
uniqueConstraints= {@UniqueConstraint(columnNames={"CATEGORY_ID"})})
```

□ @Column 주석

- 저장될 필드나 속성을 테이블의 컬럼으로 매핑
- userId 속성을 USER_ID 컬럼으로 매핑하고자 할 경우,

```
@Column(name="USER_ID")
protected Long userId;
```
- 한 엔티티를 여러 테이블에 분산하여 저장할 경우,

```
@Column(name="PICTURE", table="USER_PICTURES")
protected byte[] picture;
```
- 파라미터: insertable, updatable

```
@Column(name="USER_ID", insertable=false, updatable=false)
protected Long userId;
```
- @Column은 선택사항임, 지정되지 않을 경우 속성과 같은 이름의 컬럼으로 매핑됨

@Enumerated 사용

□ @Enumerated - 열거형 데이터 매팅

- 열거형 UserType 정의
 - public enum UserType {SELLER, BIDDER, SCR, ADMIN};
- 배열과 마찬가지로, 위 네 값은 ordinal이라 부르는 인덱스 값과 연관지어짐
- 사용예1, UserType.SELLER의 경우 0이 저장됨
 - @Enumerated(EnumType.ORDINAL) *← default 값임*
 - UserType userType;
- 사용예2, UserType.SELLER의 경우 "SELLER"가 저장됨
 - @Enumerated(EnumType.STRING)
 - UserType userType;

CLOB과 BLOB 매핑

□ RDBMS에서 대규모 데이터를 저장하는 방법

- BLOB - Binary Large Object 타입
- CLOB – Character Large Object 타입

□ @Lob

- BLOB이나 CLOB 필드를 매핑함
- 매핑 예,

@Lob

@Basic(fetch=FetchType.LAZY) ← 필드로 직접 매핑(처음 접근할 때 로드됨)
protected byte[] picture;

- CLOB/BLOB 여부는 타입에 의해 결정됨
- char[], String → CLOB 컬럼에 저장
- 지연(lazy)로딩은 선택사항이며 EJB 벤더별로 차이가 있음

날짜(temporal) 타입 매팅

□ 날짜(temporal) 데이터 타입

- 대부분 DBMS 벤더가 서로 다른 입자(granularity)를 가진 날짜 데이터 타입을 지원
 - DATE(day, month, year 저장)
 - TIME(time만 저장, day, month, year는 저장하지 않음)
 - TIMESTAMP(time, day, month, year 저장)
- 사용 예,

```
@Temporal(TemporalType.DATE)  
protected Date creationDate;
```
- @Temporal로 지정하지 않을 경우, TIMESTAMP가 기본값(가장 작은 입도)

단일 엔티티 복수 테이블 매팅

□ @SecondaryTable: 복수 테이블 매팅

- 흔히 사용되지 않지만, 특정 상황에서 매우 유용한 방법
- 예, User 도메인 객체는 User의 대용량 이미지 정보를 가지고 있다. 이 경우, USERS 테이블에 대용량 이미지를 저장하면 수행성능을 현저하게 떨어뜨린다. 이 경우, 대용량 이미지를 별도의 USER_PICTURES란 테이블에 저장을 하면 효율적이다.

```
@Entity  
@Table(name="USERS")  
@SecondaryTable(name="USER_PICTURES",  
    pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))  
public class User implements Serializable {  
    ..}
```

- 위에서, USER_ID 가 USER_PICTURES 테이블의 일차키이면서 동시에 외부키 임.

일차 키 생성 (1/4)

□ 대행키(Surrogate key) 권장

- CATEGORY_ID, EMPLOYEE_ID 같은 대행 키
- 대행 키는 일차 키로 사용하기 위해 만듬
- 복합 키보다는 대행 키 사용을 권장함

□ @GeneratedValue : 일차 키 생성

- Identity
- Sequence
- 테이블

□ Identity 컬럼 이용

- MS-SQL 서버처럼 여러 DBMS가 지원함
- 예,

```
@Id  
@GeneratedValue(strategy=GenerationType.IDENTITY)  
@Column(name="USER_ID")  
protected Long userId;
```

일차 키 생성 [2/4]

□ 데이터베이스 시퀀스 사용 절차

- 데이터베이스 안에 시퀀스 정의

```
CREATE SEQUENCE USER_SEQUENCE START WITH 1 INCREMENT BY 10;
```

- SequenceGenerator 생성

```
@SequenceGenerator(name="USER_SEQUENCE_GENERATOR",  
sequenceName="USER_SEQUENCE", initialValue=1, allocationSize=10)
```

- @GeneratedValue 태크 지정

```
@Id  
@GeneratedValue(strategy=GenerationType.SEQUENCE,  
generator="USER_SEQUENCE_GENERATOR")  
@Column(name="USER_ID")  
protected Long userId;
```

일차 키 생성 (3/4)

□ 시퀀스 테이블 사용 절차

- 한 테이블로 여러 시퀀스 값을 사용할 수 있음
- 시퀀스 테이블 정의(오라클의 예)

```
CREATE TABLE SEQUENCE_GENERATOR_TABLE  
  (SEQUENCE_NAME VARCHAR2(80) NOT NULL, SEQUENCE_VALUE NUMBER(15) NOT NULL,  
   PRIMARY KEY (SEQUENCE_NAME));
```

- 시퀀스 테이블 초기화

```
INSERT INTO SEQUENCE_GENERATOR_TABLE (SEQUENCE_NAME, SEQUENCE_VALUE)  
VALUES ('USER_SEQUENCE', 1);
```

- TableGenerator 생성

```
@TableGenerator (name="USER_TABLE_GENERATOR",  
  table="SEQUENCE_GENERATOR_TABLE", pkColumnName="SEQUENCE_NAME",  
  valueColumnName="SEQUENCE_VALUE", pkColumnValue="USER_SEQUENCE")
```

- @GeneratedValue 사용

```
@Id  
@GeneratedValue(strategy=GenerationType.TABLE,  
  generator="USER_TABLE_GENERATOR")  
@Column(name="USER_ID")  
protected Long userId;
```

일차 키 생성 (4/4)

□ 기본 일차키 생성 전략

- 자동 선택

 @Id

 @GeneratedValue(strategy=GenerationType.AUTO)

 @Column(name="USER_ID")

 protected Long userId;

- JPA 공급자에 따라 기본 값이 다름

- Oracle 데이터베이스를 사용할 경우, SEQUENCE가 기본 전략

- SQL Server 데이터베이스를 사용할 경우, IDENTITY가 기본 전략

내장 가능 (embeddable) 클래스 맵핑

- 내장가능 객체는 엔티티를 위한 편리한 데이터 저장 공간이며 identity가 없음

```
@Table(name="USERS")
...
public class User implements Serializable {
    @Id
    @Column(name="USER_ID", nullable=false)
    protected Long userId;
    ...
    @Embedded
    protected Address address;
    ...
}
```

```
@Embeddable
public class Address implements Serializable {
    @Column(name="STREET", nullable=false)
    protected String street;
    ...
    @Column(name="ZIP_CODE", nullable=false)
    protected String zipCode;
    ...
}
```

- Address의 street는 USERS.STREET 컬럼으로 맵핑됨
- Address의 zipCode는 USERS.ZIP_CODE 컬럼으로 맵핑됨



4일차 – JPA

4.1 도메인 모델과 JPA

4.2 JPA이용 도메인 객체 구현

4.3 엔티티 관계

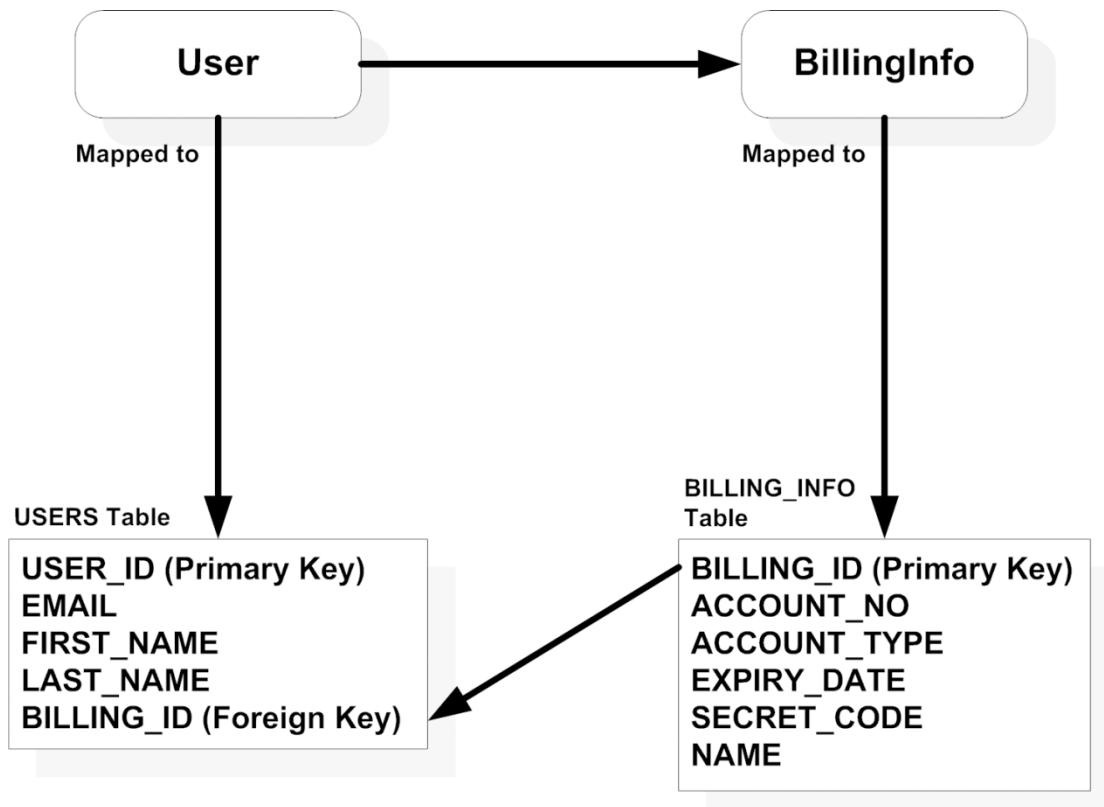
4.4 엔티티 매핑

4.5 엔티티 관계 매핑

- 일대일 관계 매핑
- 일대다, 다대일 관계 매핑
- 다대다 관계 매핑

일대일 관계 매핑 (1/2)

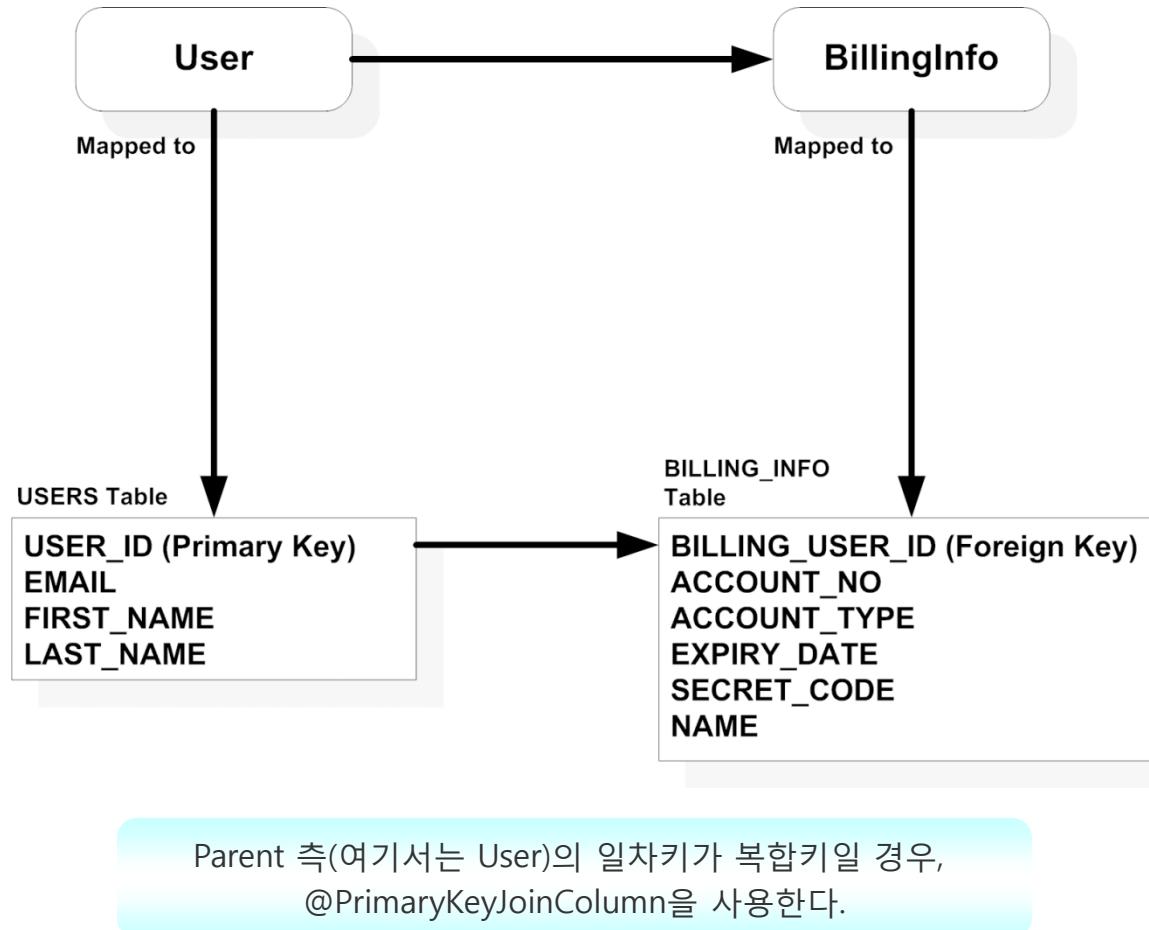
□ @JoinColumn을 이용한 매핑



```
@Entity  
@Table(name="USERS")  
public class User {  
    @Id  
    @Column(name="USER_ID")  
    protected String userId;  
    ...  
    @OneToOne  
    @JoinColumn(name="USER_BILLING_ID",  
    referencedColumnName="BILLING_ID",  
    updatable=false)  
    protected BillingInfo billingInfo;  
}  
@Entity  
@Table(name="BILLING_INFO")  
public class BillingInfo {  
    @Id  
    @Column(name="BILLING_ID")  
    protected Long billingId;  
    ...  
}
```

일대일 관계 매핑 (2/2)

□ @PrimaryKeyJoinColumn을 이용한 매핑

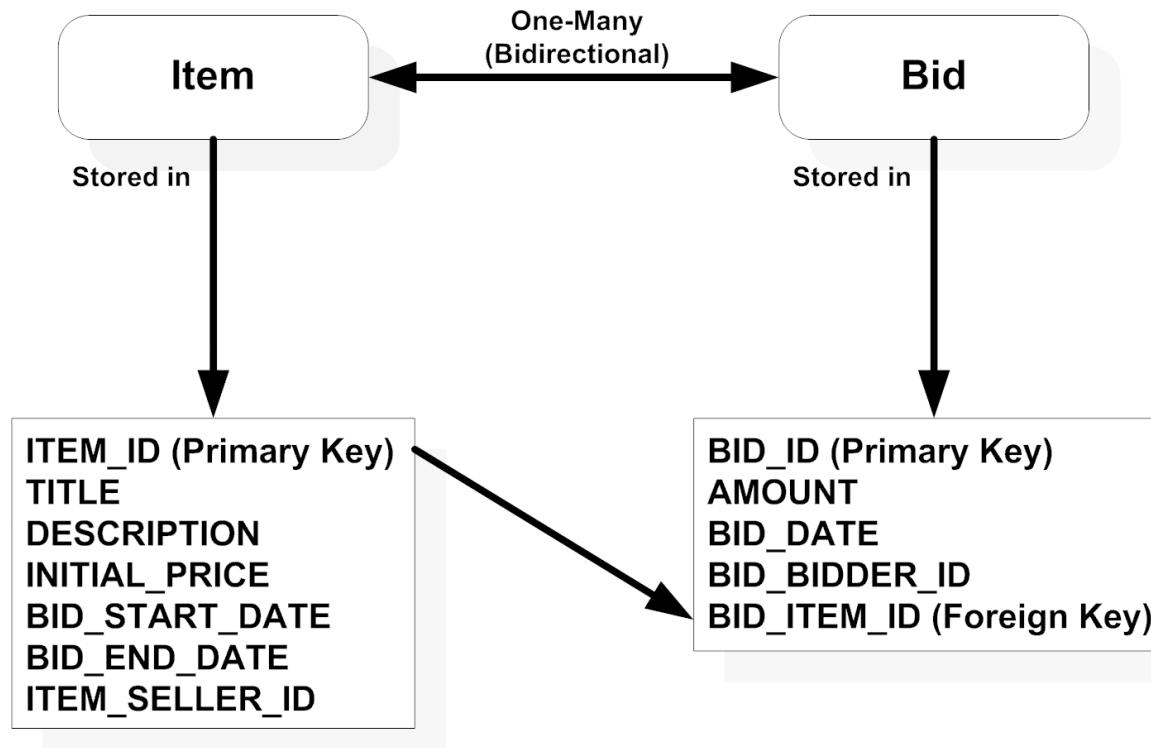


```
@Entity
@Table(name="USERS")
public class User {
    @Id
    @Column(name="USER_ID")
    protected Long userId;
    ...
    @OneToOne
    @PrimaryKeyJoinColumn(name="USER_ID",
    referencedColumnName="BILLING_USER_ID")
    protected BillingInfo billingInfo;
}

@Entity
@Table(name="BILLING_INFO")
public class BillingInfo {
    @Id
    @Column(name="BILLING_USER_ID")
    protected Long userId;
    ...
}
```

일대다, 다대일 관계 매팅

□ Item-Bid 관계

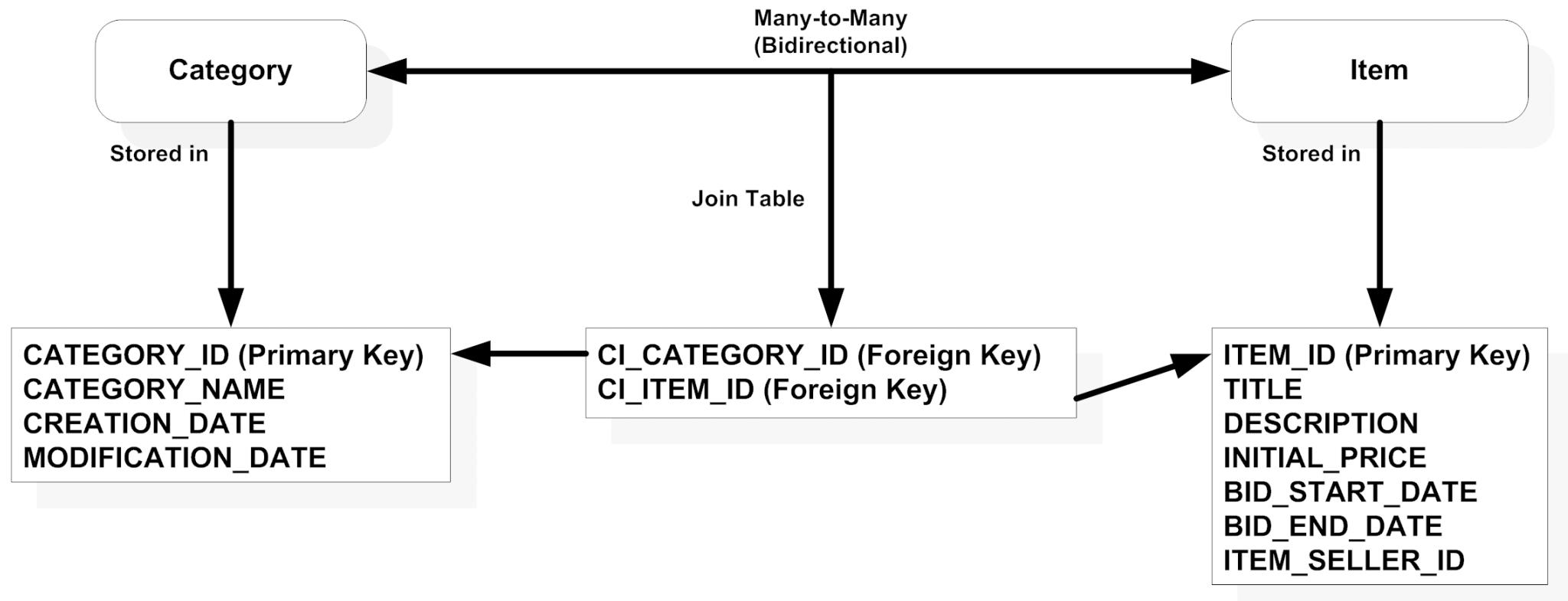


```
@Entity  
@Table(name="ITEMS")  
public class Item {  
    @Id  
    @Column(name="ITEM_ID")  
    protected Long itemId;  
    ...  
    @OneToMany(mappedBy="item")  
    protected Set<Bid> bids;  
    ...  
}  
@Entity  
@Table(name="BIDS")  
public class Bid {  
    @Id  
    @Column(name="BID_ID")  
    protected Long bidId;  
    ...  
    @ManyToOne  
    @JoinColumn(name="BID_ITEM_ID",  
    referencedColumnName="ITEM_ID")  
    protected Item item;  
    ...  
}
```

다대다 관계 매핑 (1/2)

□ 다대다 관계

- 두 개의 일대다 관계로 나눌 수 있음
- 조인 테이블이 필요함



다대다 관계 매팅 (2/2)

□ 다대다 관계 (계속)

- 소유 다대다: 1번
- 종속 다대다: 2번

```
@Entity  
@Table(name="CATEGORIES")  
public class Category implements Serializable {  
    @Id  
    @Column(name="CATEGORY_ID")  
    protected Long categoryId;  
  
    1 @ManyToMany  
    @JoinTable(name="CATEGORIES_ITEMS",  
    joinColumns=  
        @JoinColumn(name="CI_CATEGORY_ID",  
        referencedColumnName="CATEGORY_ID"),  
    inverseJoinColumns=  
        @JoinColumn(name="CI_ITEM_ID",  
        referencedColumnName="ITEM_ID"))  
    protected Set<Item> items;  
  
    ...  
}
```

```
@Entity  
@Table(name="ITEMS")  
public class Item implements Serializable {  
    @Id  
    @Column(name="ITEM_ID")  
    protected Long itemId;  
  
    ...  
    2 @ManyToMany(mappedBy="items")  
    protected Set<Category> categories;  
  
    ...  
}
```

감사합니다....

- ❖ 넥스트리소프트(주)
- ❖ 박석재 수석 (**demonpark@nextree.co.kr**)
- ❖ 임병인 수석 (**byleem@nextree.co.kr**)