Hibernate

1. Domain model and meta data

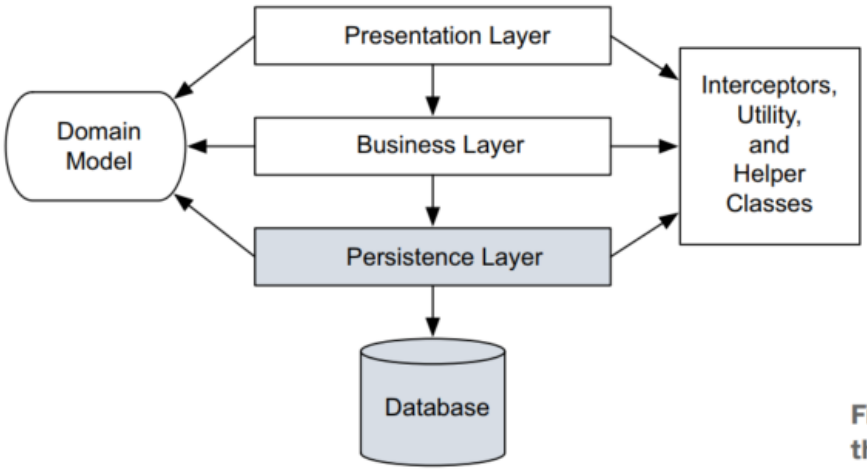


Figure 3.1 A persistence layer is the basis in a layered architecture.

Is my class now dependent on JPA?

Yes, but it's a compile-time only dependency. You need JPA libraries on your classpath when compiling the source of your domain model class. The Java Persistence API isn't required on the classpath when you create an instance of the class: for example, in a desktop client application that doesn't execute any JPA code. Only when you access the annotations through reflection at runtime (as Hibernate does internally when it reads your metadata) will you need the packages on the classpath.

Applying Bean Validation rules

```
@Entity
public class Item {

    @NotNull
    @Size(
        min = 2,
        max = 255,
        message = "Name is required, maximum 255 characters."
    )
    protected String name;
```



```
    @Future
    protected Date auctionEnd;
}
```

2. Mapping persistent classes

@Id

```
@Entity
public class Item {

    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    protected Long id;
    public Long getId() {
        return id;
    }
}
```

A *candidate key* is a column or set of columns that you could use to identify a particular row in a table. To become the primary key, a candidate key must satisfy the following requirements:

- The value of any candidate key column is never null. You can't identify something with data that is unknown, and there are no nulls in the relational model. Some SQL products allow defining (composite) primary keys with nullable columns, so you must be careful.
- The value of the candidate key column(s) is a unique value for any row.
- The value of the candidate key column(s) never changes; it's immutable.

`GenerationType.AUTO`—Hibernate picks an appropriate strategy, asking the SQL dialect of your configured database what is best. This is equivalent to `@GeneratedValue()` without any settings.

`GenerationType.SEQUENCE`—Hibernate expects (and creates, if you use the tools) a sequence named `HIBERNATE_SEQUENCE` in your database. The sequence will be called separately before every `INSERT`, producing sequential numeric values.

`GenerationType.IDENTITY`—Hibernate expects (and creates in table DDL) a special auto-incremented primary key column that automatically generates a numeric value on `INSERT`, in the database.

`GenerationType.TABLE`—Hibernate will use an extra table in your database schema that holds the next numeric primary key value, one row for each entity class. This table will be read and updated accordingly, before `INSERT`s. The default table name is `HIBERNATE_SEQUENCES` with columns `SEQUENCE_NAME` and `SEQUENCE_NEXT_HI_VALUE`. (The internal implementation uses a more complex but efficient hi/lo generation algorithm; more on this later.)

2.1 Entity-mapping options

Controlling names

Listing 4.3 `@Table` annotation overrides the mapped table name

PATH: /model/src/main/java/org/jpwh/model/simple/User.java

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {

    // ...
}
```

Enforcing naming convention:

Listing 4.4 PhysicalNamingStrategy, overriding default naming conventions

PATH: /shared/src/main/java/org/jpwh/shared/CENamingStrategy.java

```
public class CENamingStrategy extends
    org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl {

    @Override
    public Identifier toPhysicalTableName(Identifier name,
                                          JdbcEnvironment context) {
        return new Identifier("CE_" + name.getText(), name.isQuoted());
    }

}
```

Create view:

```
@Entity
@org.hibernate.annotations.Immutable
@org.hibernate.annotations.Subselect(
    value = "select i.ID as ITEMID, i.ITEM_NAME as NAME, " +
            "count(b.ID) as NUMBEROFBIDS " +
            "from ITEM i left outer join BID b on i.ID = b.ITEM_ID " +
            "group by i.ID, i.ITEM_NAME"
)
@org.hibernate.annotations.Synchronize({"Item", "Bid"})
public class ItemBidSummary {

    @Id
    protected Long itemId;

    protected String name;

    protected long numberOfBids;

    public ItemBidSummary() {
    }

    // Getter methods...

    // ...
}
```

TODO: table names are case sensitive (Hibernate bug HHH-8430)

2.2 Mapping value types

Overriding basic property defaults

```
@Column(name = "START_PRICE", nullable = false)
BigDecimal initialPrice;
```

Generated and default property values

```
@Temporal(TemporalType.TIMESTAMP)
@Column(insertable = false, updatable = false)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
protected Date lastModified;

@Column(insertable = false)
@org.hibernate.annotations.ColumnDefault("1.00")
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
protected BigDecimal initialPrice;
```

Temporal properties

Listing 5.5 Property of a temporal type that must be annotated with `@Temporal`

```
@Temporal(TemporalType.TIMESTAMP)
@Column(updatable = false)
@org.hibernate.annotations.CreationTimestamp
protected Date createdOn;

// Java 8 API
// protected Instant reviewedOn;
```

JPA says @Temporal is required, but Hibernate defaults to TIMESTAMP without it.

Mapping enumerations

```
@NotNull
@Enumerated(EnumType.STRING)
protected AuctionType auctionType = AuctionType.HIGHEST_BID;
```

Defaults to ORDINAL

Mapping embeddable components

```java
@Embeddable
public class Address {

    protected String street;

    protected String zipcode;

    protected String city;

    protected Address() {}
}


@Entity
@Table(name = "USERS")
public class User implements Serializable {
    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;
    public Long getId() {
        return id;
    }
    protected Address homeAddress;
    public Address getHomeAddress() {
        return homeAddress;
    }
    public void setHomeAddress(Address homeAddress) {
        this.homeAddress = homeAddress;
    }
    // ...
}
```

Override embedded:

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {

    @Embedded                                          <-- Not necessary
    @AttributeOverrides({
            @AttributeOverride(name = "street",
                    column = @Column(name = "BILLING_STREET")),
            @AttributeOverride(name = "zipcode",
                    column = @Column(name = "BILLING_ZIPCODE", length = 5)),
            @AttributeOverride(name = "city",
                    column = @Column(name = "BILLING_CITY"))
    })
    protected Address billingAddress;

    public Address getBillingAddress() {
        return billingAddress;
    }
```

NULLable! -->

Creating custom JPA converters

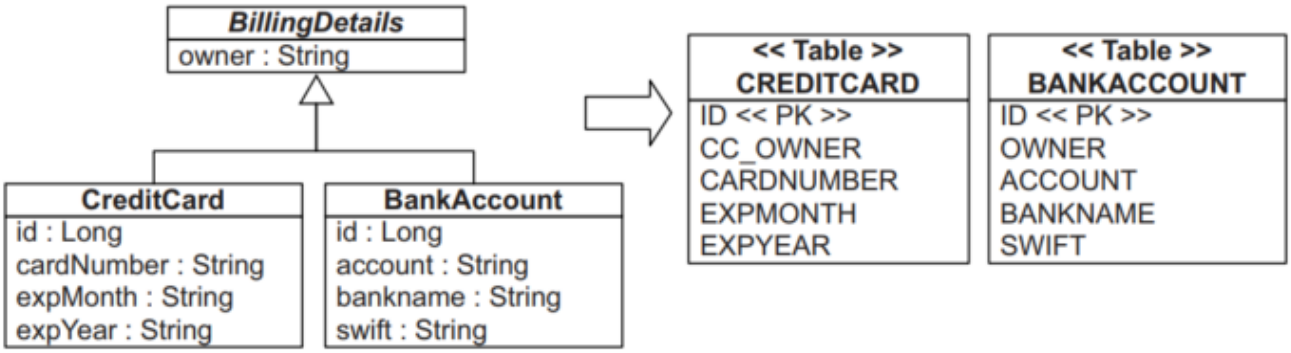
```
public class MonetaryAmount implements Serializable {      <-- 1 Value-typed class is java.io.Serializable

    protected final BigDecimal value;
    protected final Currency currency;                      <-- 2 No special constructor needed

    public MonetaryAmount(BigDecimal value, Currency currency) {
        this.value = value;
        this.currency = currency;
    }
```

```
@Converter(autoApply = true)                                <-- Default for MonetaryAmount properties
public class MonetaryAmountConverter
    implements AttributeConverter<MonetaryAmount, String> {

    @Override
    public String convertToDatabaseColumn(MonetaryAmount monetaryAmount) {
        return monetaryAmount.toString();
    }

    @Override
    public MonetaryAmount convertToEntityAttribute(String s) {
        return MonetaryAmount.fromString(s);
    }
}
```

```java
@Entity
public class Item {

    @NotNull
    @Convert(
        converter = MonetaryAmountConverter.class,
        disableConversion = false)
    @Column(name = "PRICE", length = 63)
    protected MonetaryAmount buyNowPrice;

    // ...
}
```

Optional: autoApply is enabled.

2.3 Mapping inheritance




```java
@MappedSuperclass
public abstract class BillingDetails {

    @NotNull
    protected String owner;

    // ...
}
```

```java
@Entity
@AttributeOverride(
        name = "owner",
        column = @Column(name = "CC_OWNER", nullable = false))
public class CreditCard extends BillingDetails {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @NotNull
    protected String cardNumber;

    @NotNull
    protected String expMonth;

    @NotNull
    protected String expYear;

    // ...
}
```

2.4 Mapping collections and entity associations

```java
@Entity
public class Item {

    @ElementCollection
    @CollectionTable(
            name = "IMAGE",
            joinColumns = @JoinColumn(name = "ITEM_ID"))
    @Column(name = "FILENAME")
    protected Set<String> images = new
            HashSet<String>();

    // ...
}
```

Defaults to ITEM_IMAGES

Default

Defaults to IMAGES

Initialize the field here.

ITEM

ID	NAME
1	Foo
2	B
3	C

IMAGE

ITEM_ID	FILENAME
1	foo.jpg
1	bar.jpg
1	baz.jpg
2	b.jpg

Figure 7.2 Table structure and example data for a set of strings

Mapping entity associations

- **@ManyToOne**: example one Item have many bids

The simplest possible association

We call the mapping of the `Bid#item` property a *unidirectional many-to-one association*. Before we discuss this mapping, look at the database schema in figure 7.14.

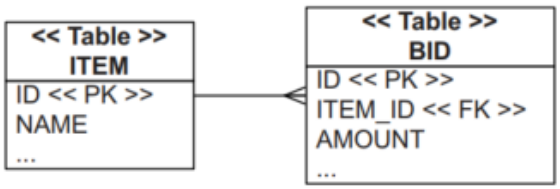


Figure 7.14 A *many-to-one* relationship in the SQL schema

Listing 7.14 `Bid` has a single reference to an `Item`

PATH: /model/src/main/java/org/jpwh/model/associations/onetomany/bidirectional/Bid.java

```
@Entity
public class Bid {

    @ManyToOne(fetch = FetchType.LAZY)                    <-- Defaults to EAGER
    @JoinColumn(name = "ITEM_ID", nullable = false)
    protected Item item;

    // ...
}
```

Making it bidirectional

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item",                         <-- Required for bidirectional association
               fetch = FetchType.LAZY)                    <-- Default
    protected Set<Bid> bids = new HashSet<>();
    // ...
}
```

Cascading state:

ENABLING TRANSITIVE PERSISTENCE

With the current mapping of `@ManyToOne` and `@OneToMany`, you need the following code to save a new `Item` and several `Bid` instances.

Listing 7.16 Managing independent entity instances separately

PATH: /examples/src/test/java/org/jpwh/test/associations/OneToManyBidirectional.java

```
Item someItem = new Item("Some Item");
em.persist(someItem);

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);                          <-- Don't forget!
em.persist(someBid);

Bid secondBid = new Bid(new BigDecimal("456.00"), someItem);
someItem.getBids().add(secondBid);
em.persist(secondBid);

tx.commit();                                              <-- Dirty checking; SQL execution
```

It can be simplified with **cascade**:

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item", cascade = CascadeType.PERSIST)
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

```
Item someItem = new Item("Some Item");
em.persist(someItem);                                     <-- Saves the bids automatically (later, at flush time)

Bid someBid = new Bid(new BigDecimal("123.00"), someItem);
someItem.getBids().add(someBid);

Bid secondBid = new Bid(new BigDecimal("456.00"), someItem);
someItem.getBids().add(secondBid);

tx.commit();                                              <-- Dirty checking; SQL execution
```

Cascade deletion:

- Without cascade

```
Item item = em.find(Item.class, ITEM_ID);

for (Bid bid : item.getBids()) {
    em.remove(bid);                    <-- 1 Removes bids
}

em.remove(item);                       <-- 2 Removes owner
```

```
@Entity
public class Item {

    @OneToMany(mappedBy = "item",
               cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

PATH: /examples/src/test/java/org/jpwh/test/associations/OneToManyCascadeRemove.java

```
Item item = em.find(Item.class, ITEM_ID);
em.remove(item);
```

Deletes bids one by one after loading them

2.5 Advanced entity association mappings

One-to-one associations

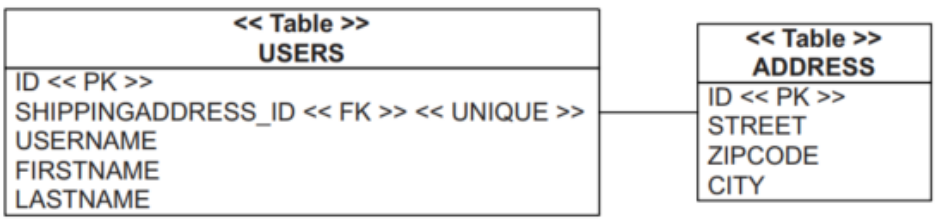


Figure 8.4 A *one-to-one* join column association between the USERS and ADDRESS tables

```
@Entity
@Table(name = "USERS")
public class User {

    @Id
    @GeneratedValue(generator = Constants.ID_GENERATOR)
    protected Long id;

    @OneToOne(
        fetch = FetchType.LAZY,
        optional = false,                    <-------  NOT NULL
        cascade = CascadeType.PERSIST
    )
    @JoinColumn(unique = true)               <-------  Defaults to SHIPPINGADDRESS_ID
    protected Address shippingAddress;

    // ...
}
```

Many-to-many and ternary associations

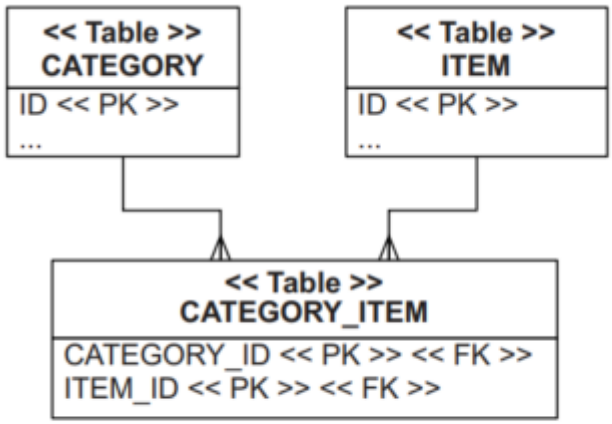


Figure 8.14 A *many-to-many* relationship with a link table

```
@Entity
public class Category {

    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(
        name = "CATEGORY_ITEM",
        joinColumns = @JoinColumn(name = "CATEGORY_ID"),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID")
    )
    protected Set<Item> items = new HashSet<Item>();

    // ...
}
```

```java
@Entity
public class Item {

    @ManyToMany(mappedBy = "items")
    protected Set<Category> categories = new HashSet<Category>();

    // ...
}
```

Many-to-many with an intermediate entity

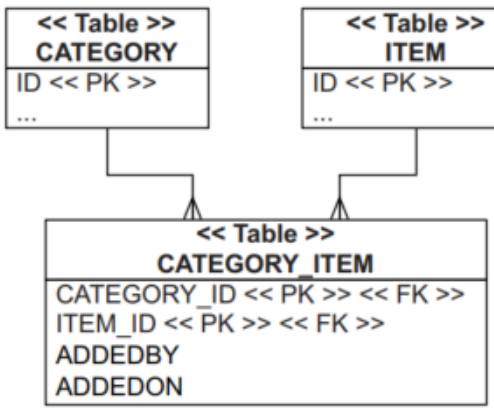


Figure 8.16 Additional columns on the join table in a *many-to-many* relationship

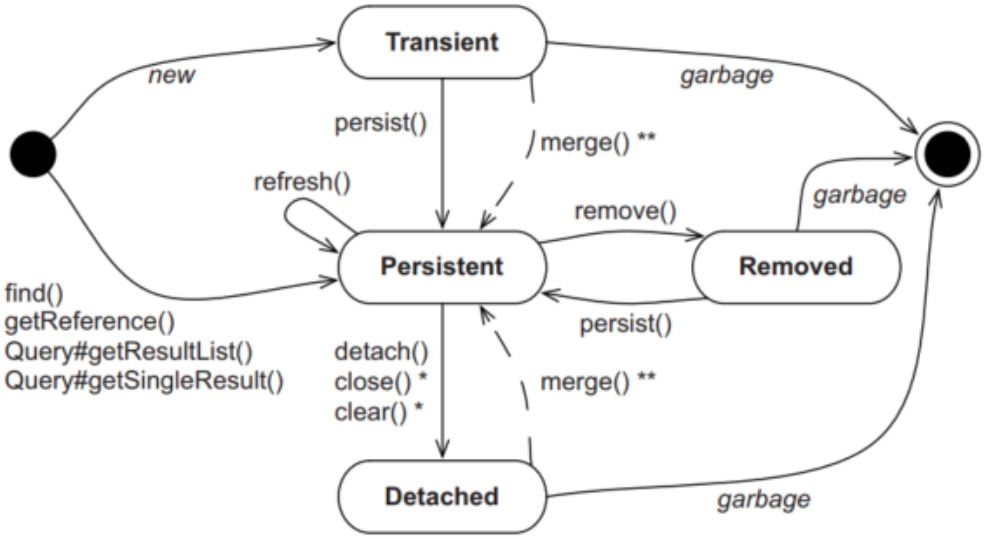
```java
@Entity
@Table(name = "CATEGORY_ITEM")
@org.hibernate.annotations.Immutable
public class CategorizedItem {

    @Embeddable
    public static class Id implements Serializable {

        @Column(name = "CATEGORY_ID")
        protected Long categoryId;

        @Column(name = "ITEM_ID")
        protected Long itemId;
```

1 Declares class immutable

2 Encapsulates composite key

2.6 Managing data

Entity instance states



State	
TRANSIENT	create with new() key word, in order to make it to persistence state, use merge or persist
PERSISTENT	have a representation in database, by calling **persist**. Persistent instances are always associated with a persistence context. This instance can be called from persist, or query look up
REMOVED	Hibernate delete with .remove keyword, after call remove at the end of unit of work.
DETACHED	

The persistence context

In a Java Persistence application, an `EntityManager` has a persistence context. You create a persistence context when you call `EntityManagerFactory#createEntityManager()`. The context is closed when you call `EntityManager#close()`. In JPA terminology, this is an *application-managed* persistence context; your application defines the scope of the persistence context, demarcating the unit of work.

The persistence context allows the persistence engine to perform *automatic dirty checking*, detecting which entity instances the application modified. The provider then synchronizes with the database the state of instances monitored by a persistence context, either automatically or on demand. Typically, when a unit of work completes, the provider propagates state held in memory to the database through the execution of SQL `INSERT`, `UPDATE`, and `DELETE` statements (all part of the Data Modification Language [DML]). This *flushing* procedure may also occur at other times. For example, Hibernate may synchronize with the database before execution of a query. This ensures that queries are aware of changes made earlier during the unit of work.

The canonical unit of work

```
EntityManager em = null;
UserTransaction tx = TM.getUserTransaction();
try {
    tx.begin();
    em = JPA.createEntityManager();        <-- Application-managed
```



```
    // ...
    tx.commit();                           <-- Synchronizes/flushes persistence context
} catch (Exception ex) {
    // Transaction rollback, exception handling
    // ...
} finally {
    if (em != null && em.isOpen())
        em.close();                        <-- You create it, you close it!
}
```

Everything between `tx.begin()` and `tx.commit()` occurs in one transaction. For now, keep in mind that all database operations in transaction scope, such as the SQL statements executed by Hibernate, completely either succeed or fail. Don't worry too much about the transaction code for now; you'll read more about concurrency control in the next chapter. We'll look at the same example again with a focus on the transaction and exception-handling code. Don't write empty `catch` clauses in your code, though—you'll have to roll back the transaction and handle exceptions.

```
Item item = new Item();
item.setName("Some Item");          <-- Item#name is NOT NULL.

em.persist(item);

Long ITEM_ID = item.getId();        <-- Has been assigned
```

You can see the same unit of work and how the `Item` instances changes state in figure 10.2.

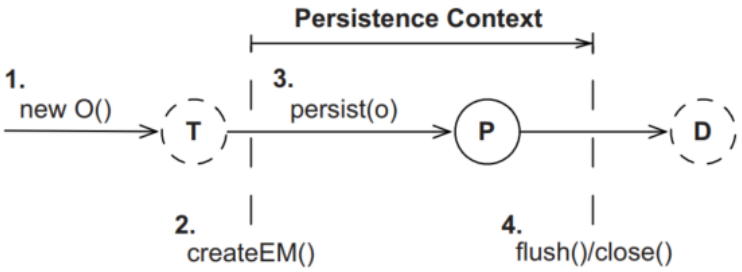


Figure 10.2 Making an instance persistent in a unit of work

Exception:

- Hibernate includes subtypes of `JDBCException` and an internal converter that tries to translate the vendor-specific error code thrown by the database driver into something more meaningful. The built-in converter can produce `JDBCConnectionException`, `SQLGrammarException`, `LockAcquisitionException`, `DataException`, and `ConstraintViolationException` for the most important database dialects supported by Hibernate. You can either manipulate or enhance the dialect for your database or plug in a `SQLExceptionConverterFactory` to customize this conversion.

As usual, this isn't the whole picture. Some standardized exceptions aren't fatal:

- `javax.persistence.NoResultException`—Thrown when a `Query` or `TypedQuery` is executed with `getSingleResult()` and no result was returned from the database. You can wrap the query call with exception-handling code and continue working with the persistence context. The current transaction won't be marked for rollback.
- `javax.persistence.NonUniqueResultException`—Thrown when a `Query` or `TypedQuery` is executed with `getSingleResult()` and several results were returned from the database. You can wrap the query call with exception handling code and continue working with the persistence context. Hibernate won't mark the current transaction for rollback.
- `javax.persistence.QueryTimeoutException`—Thrown when a `Query` or `TypedQuery` takes too long to execute. Doesn't mark the transaction for rollback. You may want to repeat the query, if appropriate.
- `javax.persistence.LockTimeoutException`—Thrown when a pessimistic lock couldn't be acquired. May occur during flushing or explicit locking (more on this topic later in this chapter). The transaction isn't marked for rollback, and you may want to repeat the operation. Keep in mind that endlessly hammering on a database system that is already struggling to keep up won't improve the situation.

Pemisstic locking:

Issue of concurrency:

- Lost Update: transaction commited but not read
- Dirty read: transaction not yet commited
- Unrepeatable read
- Phantom read

Solution:

- Read uncommited
- Read commited: solve lost update
- Repeatable read: solve unrepeatable read
- Serializable isolation

Note that we refer to *optimistic locking* (with versioning) in the following explanation, a concept explained later in this chapter. You may want to skip this section for now and come back to it later when it's time to pick an isolation level for your application. Choosing the correct isolation level is, after all, highly dependent on your particular scenario. Read the following discussion as recommendations, not dictums carved in stone.

Optimistic concurrency control

ENABLING VERSIONING

You enable versioning with an `@Version` annotation on a special additional property of your entity class, as shown next.

Listing 11.2 Enabling versioning on a mapped entity

PATH: /model/src/main/java/org/jpwh/model/concurrency/version/Item.java

```
@Entity
public class Item implements Serializable {

    @Version
    protected long version;

    // ...
}
```

```
tx.begin();
em = JPA.createEntityManager();

Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?

assertEquals(item.getVersion(), 0);

item.setName("New Name");

em.flush();
// update ITEM set NAME = ?, VERSION = 1 where ID = ? and VERSION = 0
```

1 Retrieves by identifier

2 Instance version: 0

3 Flushes persistence context

VERSIONING WITH TIMESTAMPS

If your database schema already contains a timestamp column such as `LASTUPDATED` or `MODIFIED_ON`, you can map it for automatic version checking instead of using a numeric counter.

Listing 11.4 Enabling versioning with timestamps

PATH: /model/src/main/java/org/jpwh/model/concurrency/versiontimestamp/Item.java

```
@Entity
public class Item {

    @Version
    // Optional: @org.hibernate.annotations.Type(type = "dbtimestamp")
    protected Date lastUpdated;

    // ...
}
```

```
@Entity
@org.hibernate.annotations.OptimisticLocking(
    type = org.hibernate.annotations.OptimisticLockType.ALL)
```



```
@org.hibernate.annotations.DynamicUpdate
public class Item {

    // ...
}
```

Avoiding deadlocks

```
tx.begin();
EntityManager em = JPA.createEntityManager();

Item itemOne = em.find(Item.class, ITEM_ONE_ID);
itemOne.setName("First new name");

Item itemTwo = em.find(Item.class, ITEM_TWO_ID);
itemTwo.setName("Second new name");

tx.commit();
em.close();
```

Hibernate executes two SQL `UPDATE` statements when the persistence context is flushed. The first `UPDATE` locks the row representing `Item` one, and the second `UPDATE` locks `Item` two:

```
update ITEM set ... where ID = 1;      <-- Locks row 1
update ITEM set ... where ID = 2;      <-- Attempts to lock row 2
```

A deadlock may (or it may not!) occur if a similar procedure, with the opposite order of `Item` updates, executes in a concurrent transaction:

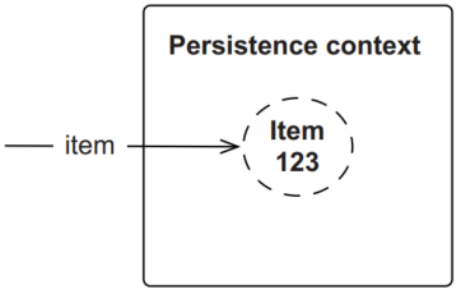
```
update ITEM set ... where ID = 2;      <-- Locks row 2
update ITEM set ... where ID = 1;      <-- Attempts to lock row 1
```

An alternative pragmatic optimization that significantly reduces the probability of deadlocks is to order the `UPDATE` statements by primary key value: Hibernate should always update the row with primary key `1` before updating row `2`, no matter in what order the data was loaded and modified by the application. You can enable this optimization for the entire persistence unit with the configuration property `hibernate.order_updates`. Hibernate then orders all `UPDATE` statements it executes in ascending order by primary key value of the modified entity instances and collection elements detected during flushing. (As mentioned earlier, make sure you fully understand the transactional and locking behavior of your DBMS product. Hibernate inherits most of its transaction guarantees from the DBMS; for example, your MVCC database product may avoid read locks but probably depends on exclusive locks for writer isolation, and you may see deadlocks.)

2.7 Fetch plans, strategies and profiles

2.7.1 Proxy, hibnerate only create a proxy object

```
Item item = em.getReference(Item.class, ITEM_ID);      <-- No SELECT
assertEquals(item.getId(), ITEM_ID);      <-- Calling the identifier getter (no field access!) doesn't trigger initialization.
```



2.7.2 Lazy vs eager

Lazy persistent collections

The `find()` operation loads the `Item` entity instance into the persistence context, as you can see in figure 12.2. The `Item` instance has a reference to an uninitialized `User` proxy: the `seller`. It also has a reference to an uninitialized `Set` of `bids` and an uninitialized `List` of `images`.

Hibernate implements lazy loading (and dirty checking) of collections with its own special implementations called *collection wrappers*. Although the `bids` certainly look like a `Set`, Hibernate replaced the implementation with an `org.hibernate.collection.internal.PersistentSet` while you weren't looking.

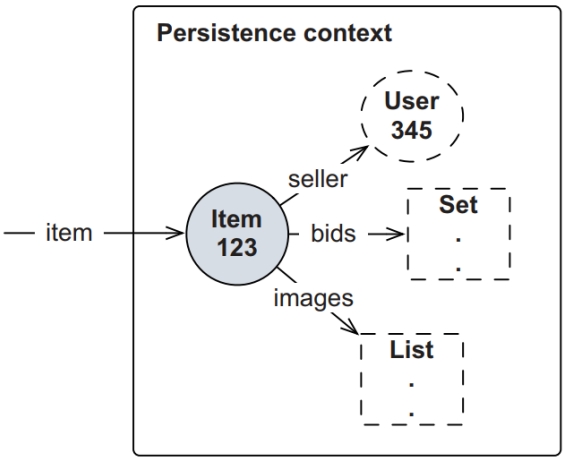


Figure 12.2 Proxies and collection wrappers represent the boundary of the loaded graph.

Eager loading of associations and collections

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.EAGER)          <-- The default
    protected User seller;

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)  <-- Not recommended
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

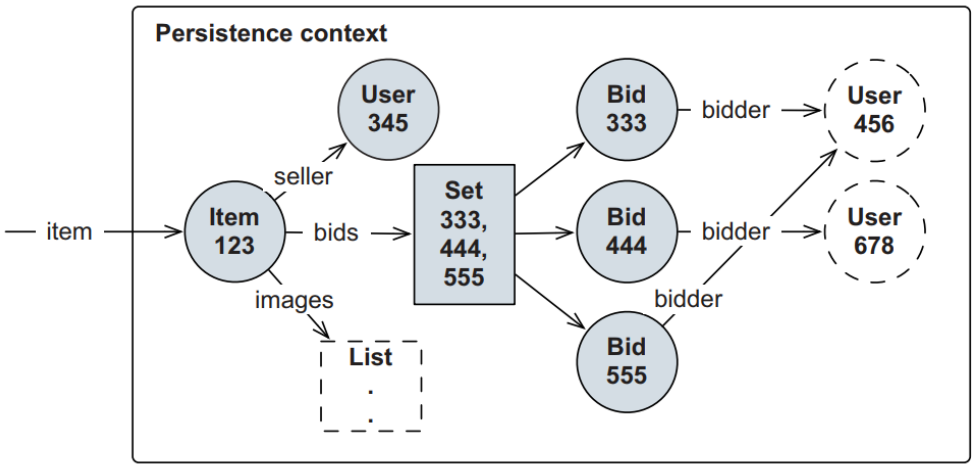
```
Item item = em.find(Item.class, ITEM_ID);
// select i.*, u.*, b.*
//  from ITEM i
//   left outer join USERS u on u.ID = i.SELLER_ID
//   left outer join BID b on b.ITEM_ID = i.ID
//  where i.ID = ?

em.detach(item);

assertEquals(item.getBids().size(), 3);
assertNotNull(item.getBids().iterator().next().getAmount());

assertEquals(item.getSeller().getUsername(), "johndoe");
```

Done fetching: no more lazy loading

In detached state, bids are available …

… and so is the seller

The em.find(Item.class) will gen 2 left join query.



2.7.3 The n+1 selects problem

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
    // select * from USERS where ID = ?
}
```

Each seller must be loaded with an additional SELECT.

Iterate through items increase the number of queries n + 1.

```
List<Item> items = em.createQuery("select i from Item i").getResultList();
// select * from ITEM

for (Item item : items) {
    assertTrue(item.getBids().size() > 0);
    // select * from BID where ITEM_ID = ?
}
```

Each bids collection has to be loaded with an additional SELECT.

Again, if you know you'll access each `bids` collection, loading only one at a time is inefficient. If you have 100 items, you'll execute 101 SQL queries!

With what you know so far, you might be tempted to change the default fetch plan in your mappings and put a `FetchType.EAGER` on your `seller` or `bids` associations. But doing so can lead to our next topic: the *Cartesian product* problem.

2.7.4 The Cartesian product problem

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.EAGER)
    protected User seller;

    // ...
}
```

```
item = em.find(Item.class, ITEM_ID);
// select i.*, u.*
//   from ITEM i
//    left outer join USERS u on u.ID = i.SELLER_ID
//   where i.ID = ?
```

The result set contains one row with data from the `ITEM` table combined with data from the `USERS` table, as shown in figure 12.4.

i.ID	i.NAME	i.SELLER_ID	...	u.ID	u.USERNAME	...
1	One	2	...	2	johndoe	...

Figure 12.4 Hibernate joins two tables to eagerly fetch associated rows.

2.7.5 Solution

Eager fetching with multiple SELECTs

```
@Entity
public class Item {

    @ManyToOne(fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
       org.hibernate.annotations.FetchMode.SELECT
    )
    protected User seller;

    @OneToMany(mappedBy = "item", fetch = FetchType.EAGER)
    @org.hibernate.annotations.Fetch(
       org.hibernate.annotations.FetchMode.SELECT
    )
    protected Set<Bid> bids = new HashSet<>();

    // ...
}
```

Defaults to JOIN

```
Item item = em.find(Item.class, ITEM_ID);
// select * from ITEM where ID = ?
// select * from USERS where ID = ?
// select * from BID where ITEM_ID = ?

em.detach(item);
```

Dynamic eager fetching

PATH: /examples/src/test/java/org/jpwh/test/fetching/EagerQuery.java

```
List<Item> items =
    em.createQuery("select i from Item i join fetch i.seller")
        .getResultList();
// select i.*, u.*
//  from ITEM i
//   inner join USERS u on u.ID = i.SELLER_ID
//  where i.ID = ?

em.close();                                          <-- Detach all

for (Item item : items) {
    assertNotNull(item.getSeller().getUsername());
}
```

2.8 Filtering data

2.8.1 Cascading state transitions

Option	Description
`CascadeType.PERSIST`	When an entity instance is stored with `EntityManager#persist()`, at flush time any associated entity instance(s) are also made persistent.
`CascadeType.REMOVE`	When an entity instance is deleted with `EntityManager#remove()`, at flush time any associated entity instance(s) are also removed.
`CascadeType.DETACH`	When an entity instance is evicted from the persistence context with `EntityManager#detach()`, any associated entity instance(s) are also detached.
`CascadeType.MERGE`	When a transient or detached entity instance is merged into a persistence context with `EntityManager#merge()`, any associated transient or detached entity instance(s) are also merged.
`CascadeType.REFRESH`	When a persistent entity instance is refreshed with `EntityManager#refresh()`, any associated persistent entity instance(s) are also refreshed.
`org.hibernate.annotations.CascadeType.REPLICATE`	When a detached entity instance is copied into a database with `Session#replicate()`, any associated detached entity instance(s) are also copied.
`CascadeType.ALL`	Shorthand to enable all cascading options for the mapped association.

2.8.2 Events

```
public class PersistEntityListener {

    @PostPersist
    public void notifyAdmin(Object entityInstance) {

        User currentUser = CurrentUser.INSTANCE.get();
        Mail mail = Mail.INSTANCE;

        mail.send(
            "Entity instance persisted by "
                + currentUser.getUsername()
                + ": "
                + entityInstance
        );
    }

}
```

1 Entity listener constructor

2 Makes notfyAdmin() a callback method

3 Gets user info and email access

Annotation	Description
`@PostLoad`	Triggered after an entity instance is loaded into the persistence context, either by identifier lookup, through navigation and proxy/collection initialization, or with a query. Also called after refreshing an already-persistent instance.
`@PrePersist`	Called immediately when `persist()` is called on an entity instance. Also called for `merge()` when an entity is discovered as transient, after the transient state is copied onto a persistent instance. Also called for associated entities if you enable `CascadeType.PERSIST`.
`@PostPersist`	Called after the database operation for making an entity instance persistent is executed and an identifier value is assigned. This may be at the time when `persist()` or `merge()` is invoked, or later when the persistence context is flushed if your identifier generator is *pre-insert* (see section 4.2.5). Also called for associated entities if you enable `CascadeType.PERSIST`.
`@PreUpdate, @PostUpdate`	Executed before and after the persistence context is synchronized with the database: that is, before and after flushing. Triggered only when the state of the entity requires synchronization (for example, because it's considered dirty).

An entity listener class must be enabled for any entity you'd like to intercept, such as this `Item`:

PATH: /model/src/main/java/org/jpwh/model/filtering/callback/Item.java

```
@Entity
@EntityListeners(
    PersistEntityListener.class
)
public class Item {

    // ...
}
```

2.9 Creating queries

JPA represents a query with a `javax.persistence.Query` or `javax.persistence.TypedQuery` instance. You create queries with the `EntityManager#createQuery()` method and its variants. You can write the query in the Java Persistence Query Language (JPQL), construct it with the `CriteriaBuilder` and `CriteriaQuery` APIs, or use plain SQL. (There is also `javax.persistence.StoredProcedureQuery`, covered in section 17.4.)

```
CriteriaBuilder cb = em.getCriteriaBuilder();
// Also available on EntityManagerFactory:
// CriteriaBuilder cb = entityManagerFactory.getCriteriaBuilder();
CriteriaQuery criteria = cb.createQuery();
criteria.select(criteria.from(Item.class));
Query query = em.createQuery(criteria);
```

Native query:

```
Query query = em.createNativeQuery(
"select * from ITEM", Item.class
);
```

Type query:

```
Query query = em.createQuery(
"select i from Item i where i.id = :id"
).setParameter("id", ITEM_ID);
Item result = (Item) query.getSingleResult();
```

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Item> criteria = cb.createQuery(Item.class);
Root<Item> i = criteria.from(Item.class);
criteria.select(i).where(cb.equal(i.get("id"), ITEM_ID));
TypedQuery<Item> query = em.createQuery(criteria);
Item result = query.getSingleResult();
```

Protecting against SQL injection attacks

Without runtime parameter binding, you're forced to write bad code:

```
String searchString = getValueEnteredByUser();      <-- Never do this!
Query query = em.createQuery(
    "select i from Item i where i.name = '" + searchString + "'"
);
```

You should never write code like this, because a malicious user could craft a search string to execute code on the database you didn't expect or want—that is, by entering the value of `searchString` in a search dialog box as `foo' and callSomeStoredProcedure()` and `'bar' = 'bar`.

Paging through large result sets

```
Query query = em.createQuery("select i from Item i");
query.setFirstResult(40).setMaxResults(10);
```

Calling a named query

```
Query query = em.createNamedQuery("findItems");
TypedQuery<Item> query = em.createNamedQuery("findItemById", Item.class);
```

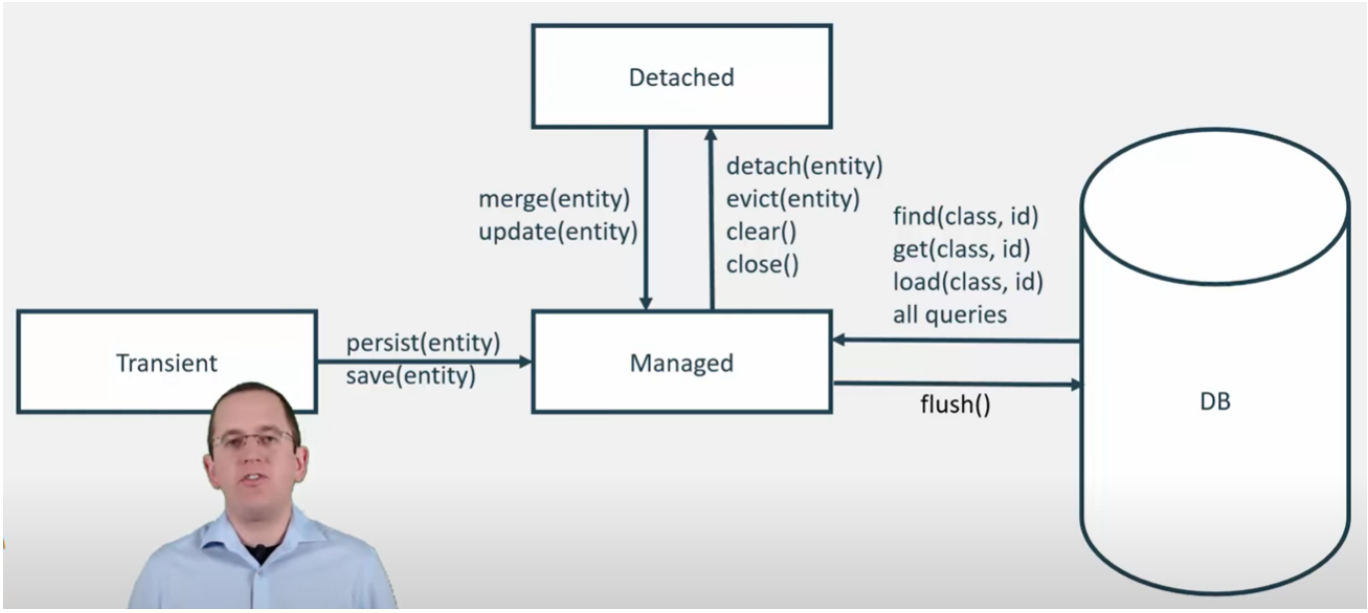
Query hints

Name	Value	Description
`javax.persistence.query.timeout`	(Milliseconds)	Sets the timeout for query execution. This hint is also available as a constant on `org.hibernate.annotations.QueryHints.TIMEOUT_JPA`.
`javax.persistence.cache.retrieveMode`	`USE` | `BYPASS`	Controls whether Hibernate tries to read data from the second-level shared cache when marshaling a query result, or bypasses the cache and only reads data from the query result.
`javax.persistence.cache.storeMode`	`USE` | `BYPASS` | `REFRESH`	Controls whether Hibernate stores data in the second-level shared cache when marshaling a query result.

Name	Value	Description
`org.hibernate.flushMode`	`org.hibernate.FlushMode` (Enum)	Controls whether and when the persistence context should be flushed before execution of the query
`org.hibernate.readOnly`	`true` | `false`	Enables or disables dirty checking for the managed entity instances returned by the query
`org.hibernate.fetchSize`	(JDBC fetch size)	Calls the JDBC `PreparedStatement#setFetchSize()` method before executing the query, an optimization hint for the database driver
`org.hibernate.comment`	(SQL comment string)	A comment to prepend to the SQL, useful for (database) logging

```
Query query = em.createQuery(queryString)
                .setHint("javax.persistence.query.timeout", 60000);
```

State of entity



- Specification vs. proprietary feature
  - Persist = JPA
  - Save = Hibernate

Hibernate return with Key

@Query

```java
public interface AuthorRepository extends JpaRepository<Author, Long> {

    @Query("FROM Author WHERE firstName = ?1")
    List<Author> findByFirstName(String firstName);

    @Query("SELECT a FROM Author a WHERE a.firstName = ?1 AND a.lastName = ?2")
    List<Author> findByFirstNameAndLastName(String firstName, String lastName);

    @Query("FROM Author WHERE firstName = ?1 ORDER BY lastName ASC")
    List<Author> findByFirstNameOrderByLastname(String firstName);

    @Query("FROM Author WHERE firstName = ?1")
    List<Author> findByFirstName(String firstName, Sort sort);

    @Query("FROM Author WHERE UPPER(firstName) LIKE CONCAT('%', UPPER(?1), '%')")
    List<Author> findByFirstNameContainingIgnoreCase(String firstName);

    @Query(value = "SELECT * FROM author WHERE first_name = :firstName", nativeQuery = true)
    List<Author> findAuthorsByFirstName(@Param("firstName") String firstName);

}
```

Dynamic enityName

```java
@Query("FROM Author WHERE firstName = ?1 ORDER BY lastName ASC")
List<Author> findByFirstNameOrderByLastname(String firstName);

@Query("FROM Author WHERE firstName = ?1")
List<Author> findByFirstName(String firstName, Sort sort);

@Query("FROM Author WHERE firstName = ?1")
List<Author> findByFirstName(String firstName, Pageable pageable);

@Query("FROM #{#entityName} WHERE firstName = ?1")
List<Author> findByFirstName(String firstName);

@Query("FROM Author WHERE UPPER(firstName) LIKE %?#{[0].toUpperCase()}%")
List<Author> findByFirstNameContainingIgnoreCase(String firstName);

@Query(value = "SELECT * FROM author WHERE first_name = :firstName", nativeQuery = true)
List<Author> findAuthorsByFirstName(@Param("firstName") String firstName);
```

Export Entity DTO ?

- Reasons to use DTO:
    - Hide internal implementation, for example, Enum vs Type in DB
    - Sensitive Data

- Handle Change with sepecific Third party
- Don't worry about association
- Loose couplingC

• 1 Entity

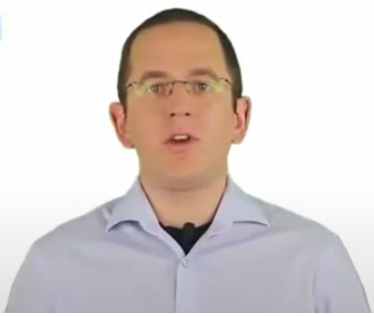
```
1 | Author a = em.find(Author.class, id);
```

• Multiple Entities

```
1 | MultiIdentifierLoadAccess<Book> multi = session.byMultipleIds(Book.class);
2 | List<Book> books = multi.multiLoad(1L, 2L, 3L);
```

```
@Entity
@NamedQuery(name = "Author.findAuthorByFirstname", query = "SELECT a FROM Author a WHERE a.firstname = :firstname")
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;
```

```
public void getJqlAuthorByFirstname() {
    log.info("... getJqlAuthorByFirstname ...");

    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    TypedQuery<Author> q = em.createNamedQuery("Author.findByFirstname", Author.class);
    q.setParameter("firstname", "Thorben");
    List<Author> authors = q.getResultList();

    for (Author a : authors) {
        log.info(a);
    }

    em.getTransaction().commit();
    em.close();
}
```



```
Run Test | Debug Test
public void getAuthorByFirstname() {
    log.info("... getAuthorByFirstname ...");

    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Author> cq = cb.createQuery(Author.class);
    Root<Author> root = cq.from(Author.class);

    ParameterExpression<String> paramFirstName = cb.parameter(String.class);
    cq.where(cb.equal(root.get(Author_.firstName), paramFirstName));

    TypedQuery<Author> query = em.createQuery(cq);
    query.setParameter(paramFirstName, "Thorben");
    List<Author> authors = query.getResultList();

    for (Author a : authors) {
        log.info(a);
    }

    em.getTransaction().commit();
    em.close();
}
```

This is the same query as I showed you in the previous JPQL example.

```
Run Test | Debug Test
public void selectPojo() {
    log.info("... selectPojo ...");

    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<AuthorValue> q = cb.createQuery(AuthorValue.class);
    Root<Author> root = q.from(Author.class);
    q.select(cb.construct(AuthorValue.class, root.get(Author_.firstName), root.get(Author_.lastName)));

    TypedQuery<AuthorValue> query = em.createQuery(q);
    List<AuthorValue> authors = query.getResultList();

    for (AuthorValue author : authors) {
        log.info(author.getFirstName() + " "
                + author.getLastName());
    }

    em.getTransaction().commit();
    em.close();
}
```

```java
@Entity
@SqlResultSetMapping(name = "BookValueMapping",
        classes = @ConstructorResult(
                targetClass = BookValue.class,
                columns = { @ColumnResult(name = "title"),
                            @ColumnResult(name = "date")}
                )
)
public class Book {
```

```java
@Test
Run Test | Debug Test
public void explicitMapping() {
    log.info("... explicitMapping ...");

    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();

    BookValue b = (BookValue) em.createNativeQuery(
        "SELECT b.publishingDate as date, b.title, b.id FROM book b WHERE b.id = 1",
        "BookValueMapping").getSingleResult();
    Assert.assertTrue(b instanceof BookValue);
    Assert.assertEquals("Effective Java", b.getTitle());

    em.getTransaction().commit();
    em.close();
}
```

- Entities
  - Only for write operations
- DTO
  - Best option for read-only operations

Get current EntityManager in spring

```
public class ProductDaoImpl implements ProductDao {

    private EntityManagerFactory emf;

    @PersistenceUnit
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }

    public Collection loadProductsByCategory(String category) {
        EntityManager em = this.emf.createEntityManager();
        try {
             Query query = em.createQuery("from Product as p where p.category = ?1");
             query.setParameter(1, category);
             return query.getResultList();
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

The main problem with such a DAO is that it always creates a new `EntityManager` through the factory. You can avoid this by requesting a transactional `EntityManager` (also called "shared EntityManager" because it is a shared, thread-safe proxy for the actual transactional EntityManager) to be injected instead of the factory:

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceContext
    private EntityManager em;

    public Collection loadProductsByCategory(String category) {
       Query query = em.createQuery("from Product as p where p.category = :category");
       query.setParameter("category", category);
       return query.getResultList();
    }
}
```

Getting hibnerate with JPA:

```
Session session = em.unwrap(Session.class);
SessionFactory sessionFactory = em.getEntityManagerFactory()
                                  .unwrap(SessionFactory.class);
```

Explicit type in hibernate

```
    @org.hibernate.annotations.Type( type = "nstring" )
    private String name;

    @org.hibernate.annotations.Type( type = "materialized_nclob" )
    private String description;
```

Using UUID

```
@Entity
public class Author {
   @Id
   @GeneratedValue
   @Column(name = "id", updatable = false, nullable = false)
   private UUID id;
   ...
}
```

```
@Entity
public class Book {

    @Id
    @GeneratedValue(generator = "UUID")
    @GenericGenerator(
        name = "UUID",
        strategy = "org.hibernate.id.UUIDGenerator",
        parameters = {
            @Parameter(
                name = "uuid_gen_strategy_class",
                value =
                "org.hibernate.id.uuid.CustomVersionOneStrategy"
            )
        }
    )
    @Column(name = "id", updatable = false, nullable = false)
    private UUID id;
```

Many to one:

How to map a bidirectional many-to-one association

Problem

My table model contains a many-to-one association. How do I model it with Hibernate so that I can navigate it in both directions?

Solution

You need to model the association on both entities if you want to be able to navigate it in both directions. Consider this example. A book in an online bookstore can have multiple reviews. In your domain model, the `Book` entity has a one-to-many association to the `Review` entity, and the `Review` entity has a many-to-one relationship to the `Book` entity.

Birectional require additional step to persit:

```
Book b = em.find(Book.class, 1L);

Review r = new Review();
r.setComment("This is a comment");
r.setBook(b);

b.getReviews().add(r);
em.persist(r);
```

```
@ManyToOne
@JoinColumn(name = "fk_book")
private Book book;

@OneToMany(mappedBy = "book")
private List<Review> reviews = new ArrayList<Review>();
```

@OneToMany

```
@OneToMany
private List<Review> reviews = new ArrayList<Review>();
```

Modeling like this is okay, but require mapping two entity at the same times

```
Book b = em.find(Book.class, 1L);
Review r = new Review();
r.setComment("This is a comment");
b.getReviews().add(r);
em.persist(r);
```

More importantly, Hibernate uses a join table to persist the association.

```
05:50:37,181 DEBUG [org.hibernate.SQL] -
    insert
    into
        Review
        (comment, id)
    values
05:50:37,198 DEBUG [org.hibernate.SQL] -
    insert
    into
        Book_Review
        (Book_id, reviews_id)
    values
        (?, ?)
```

Avoid such situation:

You can avoid that with a `@JoinColumn` annotation. It allows you to define the name of the foreign key column in the table mapped by the associated entity. In the following example, it tells Hibernate to use the `fk_book` column on the `review` table as the foreign key.

```
@Entity
public class Book {

    @OneToMany
    @JoinColumn(name = "fk_book")
    private List<Review> reviews = new ArrayList<Review>();

    ...
}
```

This will fix the join table update

```java
@Entity
public class Book {

    @OneToMany
    @JoinColumn(name = "fk_book")
    private List<Review> reviews = new ArrayList<Review>();

    ...
}
```

```
05:52:01,118 DEBUG [org.hibernate.SQL] -
    insert
    into
        Review
        (comment, id)
    values
        (?, ?)
05:52:01,125 DEBUG [org.hibernate.SQL] -
    update
        Review
    set
        fk_book=?
    where
        id=?
```

Best practice to insert many to one

```java
Post post = entityManager.getReference(Post.class, 1L);

entityManager.persist(
    new PostComment()
        .setId(1L)
        .setReview("Amazing book!")
        .setPost(post)
);
```

Fetch many to one

```
PostComment comment = entityManager.find(PostComment.class, 1L);

LOGGER.info(
    "The post '{}' got the following comment '{}'",
    comment.getPost().getTitle(),
    comment.getReview()
);
```

Hibernate is going to trigger a secondary SELECT statement:

```
SELECT
    pc.id AS id1_1_0_,
    pc.post_id AS post_id3_1_0_,
    pc.review AS review2_1_0_
FROM post_comment pc
WHERE pc.id = 1

SELECT
    p.id AS id1_0_0_,
    p.title AS title2_0_0_
FROM post p
WHERE p.id = 1

The post 'High-Performance Java Persistence' got the following comment 'Amazing book!'
```

Using SQL fetch join to avoid addtional queries

```
PostComment comment = entityManager.createQuery("""
    select pc
    from PostComment pc
    join fetch pc.post
    where pc.id = :id
    """, PostComment.class)
.setParameter("id", 1L)
.getSingleResult();

LOGGER.info(
    "The post '{}' got the following comment '{}'",
    comment.getPost().getTitle(),
    comment.getReview()
);
```

Now, Hibernate executes a single SQL query to fetch both the child and parent entities:

```
SELECT
    pc.id AS id1_1_0_,
    p.id AS id1_0_1_,
    pc.post_id AS post_id3_1_0_,
    pc.review AS review2_1_0_,
    p.title AS title2_0_1_
FROM post_comment pc
INNER JOIN post p ON pc.post_id = p.id
WHERE pc.id = 1

The post 'High-Performance Java Persistence' got the following comment 'Amazing book!'
```

PrePersit

```
@Entity
public class Author {

        ...

        @PrePersist
        private void initializeCreatedAt() {
                this.createdAt = LocalDateTime.now();
                log.info("Set createdAt to "+this.createdAt);
        }
}
```

Nautarl ID

How to map natural IDs

Problem

My domain model contains several natural IDs which I need to use to find objects in my business logic. What's the best way to models these IDs with Hibernate?

Solution

Hibernate provides proprietary support for natural IDs. It allows you to model them as a natural identifier of an entity and provides an additional API for retrieving them from the database.

The only thing you have to do to model an attribute as a natural id is to add the `@NaturalId` annotation to it. If your natural id consists of multiple attributes, you have to add this annotation to each of the attributes. You can see a simple example of such a mapping in the following code snippet. The ISBN number is a common natural id that is often used in the business logic to identify a book.

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;

    @NaturalId
    private String isbn;

    ...
}
```