One-To-Many Relationship

Association mapping

While modeling a database using JPA (Java Persistence API) for mapping a one-to-many relationship, the first thing we should know is that JPA allows us to define this type of relationship with two different annotations differing by perspective:

- **@ManyToOne:** The Foreign Key is mapped in the many-related entity object so that this has an entity object reference to its one-related correspondent.
- **@OneToMany:** The relationship is mapped as a collection of many-related entity objects inside the one-related entity object.

Unidirectional Relationship: an object knows about the object it is related to, but the other knows nothing about the first one.

Bidirectional Relationship: objects in a relationship know of each other. Is made by using both annotations in the respective objects.

Unidirectional @ManyToOne

This is the most natural and efficient way to map a one-to-many relationship. Each entity has a lifecycle of its own. Once the **@ManyToOne** association is set, Hibernate will set the associated database foreign key column.

```java
@Entity(name = "Person")
public static class Person
{
    @Id
    @GeneratedValue
    private Long id;
}

@Entity(name = "Phone")
public static class Phone
{
    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @ManyToOne
    @JoinColumn(name = "person_id",
            foreignKey = @ForeignKey(name = "PERSON_ID_FK")
    )
    private Person person;
}
```

```sql
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    person_id BIGINT ,
    PRIMARY KEY ( id )
 )

ALTER TABLE Phone
ADD CONSTRAINT PERSON_ID_FK
FOREIGN KEY (person_id) REFERENCES Person
```

This is the best practice for mapping one-to-many relationship if we don't require lots of accesses to the collection of many-side objects starting from a one-side one or when we deal with huge associations. We can always get the list of many-side objects (Phone) related to a one-side (Person) with a simple query.

Unidirectional @OneToMany

Using this method should be avoided because Hibernate will create an additional table dealing with the relation like a Many-To-Many one. This could be partially fixed with the use of **@JoinColumn** but more SQL statements than expected are executed anyways.

```java
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();
}

@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;
}
```

```sql
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Person_Phone (
    Person_id BIGINT NOT NULL ,
    phones_id BIGINT NOT NULL
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    PRIMARY KEY ( id )
)

ALTER TABLE Person_Phone
ADD CONSTRAINT UK_9uhc5itwc9h5gcng944pcaslf
UNIQUE (phones_id)

ALTER TABLE Person_Phone
ADD CONSTRAINT FKr38us2n8g5p9rj0b494sd3391
FOREIGN KEY (phones_id) REFERENCES Phone

ALTER TABLE Person_Phone
ADD CONSTRAINT FK2ex4e4p7w1cj310kg2woisjl2
FOREIGN KEY (Person_id) REFERENCES Person
```

When persisting the Person entity, the cascade will propagate the persist operation to the underlying Phone children as well. Upon removing a Phone from the phones collection, the association row is deleted from the link table, and the **orphanRemoval** attribute will trigger a Phone removal as well.

Bidirectional @OneToMany

This also requires a @**ManyToOne** association on the many side. Every bidirectional association must have one owning side only (the many side), the other one being referred to as the **mappedBy** side. This method should be avoided while dealing with big associations as Hibernate loads all associated entities when it initializes the association. This operation can take a long time while dealing with lots of entities.

Helper methods to update bidirectional associations should always be implemented to synchronize both ends whenever a many side element is added or removed.

```java
@Entity(name = "Person")
public static class Person {
    @Id
    @GeneratedValue
    private Long id;
    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Phone> phones = new ArrayList<>();
    public void addPhone(Phone phone) {
        phones.add( phone );
        phone.setPerson( this );
    }
    public void removePhone(Phone phone) {
        phones.remove( phone );
        phone.setPerson( null );
    }
}

@Entity(name = "Phone")
public static class Phone {
    @Id
    @GeneratedValue
    private Long id;
    @NaturalId
    @Column(name = "`number`", unique = true)
    private String number;
    @ManyToOne
    private Person person;
    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Phone phone = (Phone) o;
        return Objects.equals( number, phone.number );
    }
    @Override
    public int hashCode() {
        return Objects.hash( number );
    }
}
```

```sql
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    person_id BIGINT ,
    PRIMARY KEY ( id )
)

ALTER TABLE Phone
ADD CONSTRAINT UK_l329ab0g4c1t78onljnxmbnp6
UNIQUE (number)

ALTER TABLE Phone
ADD CONSTRAINT FKmw13yfsjypiiq0i1osdkaeqpg
FOREIGN KEY (person_id) REFERENCES Person
```

Many-To-Many Relationship

The annotation to use with this kind of relationship is **@ManyToMany**. Just like we have seen for the one-to-many relationship, we can have a unidirectional or bidirectional relationship.

Unidirectional @ManyToMany

Just like with unidirectional **@OneToMany** associations, the link table is controlled by the owning side. When an entity is removed from the **@ManyToMany** collection, Hibernate simply deletes the joining record in the link table. Unfortunately, this operation requires removing all entries associated with a given parent and recreating the ones that are listed in the current running persistent context.

We will get an **exception** if we try to **cascade a remove action** as we could have more objects of the same type linked with the same objects of the other entity type (other rows associated with the same rows of the other table).

Bidirectional @ManyToMany

A bidirectional **@ManyToMany** association has an owning and a mappedBy side. To preserve synchronicity between both sides, we should provide **helper methods** like in the **Bidirectional @OneToMany**.

If a **bidirectional @OneToMany** association performs better when removing or changing the order of child elements, the **@ManyToMany** relationship cannot benefit from such an optimization because the **foreign key side is not in control**. To overcome this limitation, the link table must be directly exposed and the **@ManyToMany** association split into two **bidirectional @OneToMany** relationships.

Bidirectional many-to-many with a link entity

Follows the same logic employed by the database schema, and the link table has an associated entity which controls the relationship for both sides that need to be joined. This is more natural and efficient because the link entity will be in control of the relationships from the **@ManyToOne** side.

In the entities provided as an example getter/setter methods and equality/hashcode overrides are omitted for brevity. In the following pages a set of CRUD operations extracted from our project and operating on these entities are proposed as a practical example of the usage of Hibernate with JPA.

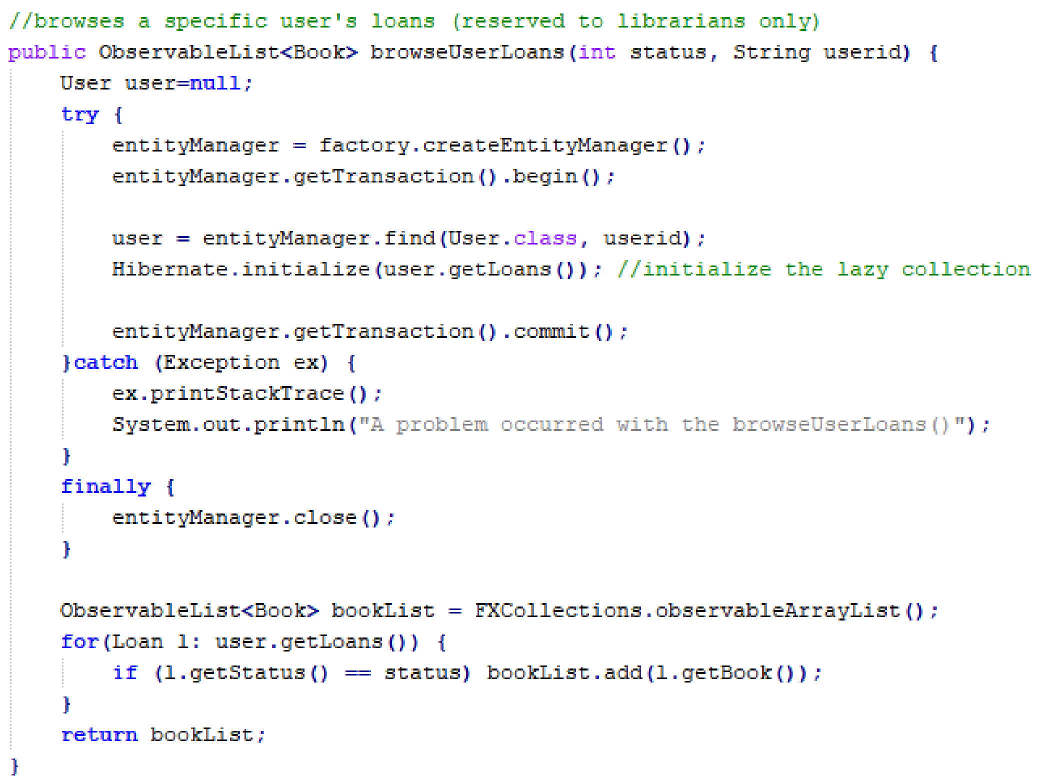
```java
@Entity
public class Book implements Serializable {

    @Id
    //The id variable is linked to the column of the table with the name bookId
    //therefore we must use the following annotation.
    @Column(name = "ISBN")
    private long id;

    private String title;
    private String author;
    private String category;
    private int numCopies;

    @OneToMany(mappedBy = "book", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Loan> loans= new ArrayList<>();
}
```

```java
@Entity
public class User implements Serializable {

    @Id
    //The id variable is linked to the column of the table with the name userId
    //therefore we must use the following annotation.
    @Column(name="idUser")
    private String id;

    private String name;
    private String surname;

    @Column(columnDefinition = "tinyint(4) default 0")
    private int privilege;


    @OneToMany( mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Loan> loans = new ArrayList<>();

    //helper method to sync a loan addition
    public void addLoan(Book book) {
        Loan loan = new Loan(this, book);
        loans.add(loan);
        book.getLoans().add(loan);
    }

    //helper method to sync a loan removal
    public void removeLoan(Book book) {
        for (Iterator<Loan> iterator = loans.iterator(); iterator.hasNext(); )
        {
            Loan loan = iterator.next();
                if (loan.getUser().equals(this) &&
                        loan.getBook().equals(book)) {
                    iterator.remove();
                    book.getLoans().remove(loan);
                    loan.setUser( null );
                    loan.setBook( null );
                }
        }
    }
}
```

```java
@Entity
public class Loan implements Serializable {
    @EmbeddedId
    private LoanId id;

    @ManyToOne(fetch = FetchType.LAZY)
    @MapsId("userId")
    private User user;

    @ManyToOne
    @MapsId("bookId")
    private Book book;

    @Column(columnDefinition = "int default 0")
    private int status;
}
```

Read operations

While reading one or more table lines in a one-to-many or many-to-many relationship we don’t have any significant difference depending on the JPA relationship annotation. We should only care to initialize the child objects if we are using getter methods on a collection of objects related to a found object outside the transaction.



```
//browses a specific user's loans (reserved to librarians only)
public ObservableList<Book> browseUserLoans(int status, String userid) {
    User user=null;
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();

        user = entityManager.find(User.class, userid);
        Hibernate.initialize(user.getLoans()); //initialize the lazy collection

        entityManager.getTransaction().commit();
    }catch (Exception ex) {
        ex.printStackTrace();
        System.out.println("A problem occurred with the browseUserLoans()");
    }
    finally {
        entityManager.close();
    }

    ObservableList<Book> bookList = FXCollections.observableArrayList();
    for(Loan l: user.getLoans()) {
        if (l.getStatus() == status) bookList.add(l.getBook());
    }
    return bookList;
}
```

Please notice that in this example the loop to browse the book objects related to each loan of the user’s collection is outside the transaction, making that atomic transaction have constant time complexity. The collection is initialized inside the transaction with the **Hibernate.initialize()** function as we are using a lazy fetching strategy for efficiency (we don’t have to fetch all the loans every time we fetch a user).

Hibernate will execute the following queries:

```
    select
        user0_.idUser as idUser1_2_0_,
        user0_.name as name2_2_0_,
        user0_.privilege as privileg3_2_0_,
        user0_.surname as surname4_2_0_
    from
        User user0_
    where
        user0_.idUser=?

    select
        loans0_.user_idUser as user_idU3_1_0_,
        loans0_.book_ISBN as book_ISB2_1_0_,
        loans0_.book_ISBN as book_ISB2_1_1_,
        loans0_.user_idUser as user_idU3_1_1_,
        loans0_.status as status1_1_1_,
        book1_.ISBN as ISBN1_0_2_,
        book1_.author as author2_0_2_,
        book1_.category as category3_0_2_,
        book1_.numCopies as numCopie4_0_2_,
        book1_.title as title5_0_2_
    from
        Loan loans0_
    inner join
        Book book1_
            on loans0_.book_ISBN=book1_.ISBN
    where
        loans0_.user_idUser=?

    select
        user0_.idUser as idUser1_2_0_,
        user0_.name as name2_2_0_,
        user0_.privilege as privileg3_2_0_,
        user0_.surname as surname4_2_0_
    from
        User user0_
    where
        user0_.idUser=?

    select
        loans0_.user_idUser as user_idU3_1_0_,
        loans0_.book_ISBN as book_ISB2_1_0_,
        loans0_.book_ISBN as book_ISB2_1_1_,
        loans0_.user_idUser as user_idU3_1_1_,
        loans0_.status as status1_1_1_,
        book1_.ISBN as ISBN1_0_2_,
        book1_.author as author2_0_2_,
        book1_.category as category3_0_2_,
        book1_.numCopies as numCopie4_0_2_,
        book1_.title as title5_0_2_
    from
        Loan loans0_
    inner join
        Book book1_
            on loans0_.book_ISBN=book1_.ISBN
    where
        loans0_.user_idUser=?
```

Create operations

Insert SQL queries are executed with JPA and Hibernate by creating a new object and using its setter methods, then persisting the object in a transaction. As we already stated while discussing the different annotations, while building a relationship between tuples, if the relationship is bidirectional, we should use helper methods to synchronize the related collections. What follows is an example of a bidirectional many-to-many relationship with a link entity.

```java
public String borrowBook(long bookId) {
    String response = "";
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        LoanId loanid = new LoanId(loggedUser,bookId);
        Loan loan = entityManager.find(Loan.class, loanid);
        if(loan != null)
            return "You already borrowed this book.";
        else {
            Book book = entityManager.find(Book.class, bookId);
            User user = entityManager.find(User.class, loggedUser);
            if(available(book) > 0) {
                user.addLoan(book);
                response ="Successfully requested the book: " + book.getTitle() +".";
            }
            else
                response = "This book is not available.";
        }
        entityManager.getTransaction().commit();
    }catch (Exception ex) {
        ex.printStackTrace();
        response = "A problem occurred with the loan request.";
    }
    finally {
        entityManager.close();
        return response;
    }
}
```

Hibernate will execute the following queries:

```sql
    select
        loan0_.book_ISBN as book_ISB2_1_0_,
        loan0_.user_idUser as user_idU3_1_0_,
        loan0_.status as status1_1_0_,
        book1_.ISBN as ISBN1_0_1_,
        book1_.author as author2_0_1_,
        book1_.category as category3_0_1_,
        book1_.numCopies as numCopie4_0_1_,
        book1_.title as title5_0_1_
    from
        Loan loan0_
    inner join
        Book book1_
            on loan0_.book_ISBN=book1_.ISBN
    where
```

```
        loan0_.book_ISBN=?
        and loan0_.user_idUser=?

    select
        book0_.ISBN as ISBN1_0_0_,
        book0_.author as author2_0_0_,
        book0_.category as category3_0_0_,
        book0_.numCopies as numCopie4_0_0_,
        book0_.title as title5_0_0_
    from
        Book book0_
    where
        book0_.ISBN=?

    select
        user0_.idUser as idUser1_2_0_,
        user0_.name as name2_2_0_,
        user0_.privilege as privileg3_2_0_,
        user0_.surname as surname4_2_0_
    from
        User user0_
    where
        user0_.idUser=?

    select
        loans0_.book_ISBN as book_ISB2_1_0_,
        loans0_.user_idUser as user_idU3_1_0_,
        loans0_.book_ISBN as book_ISB2_1_1_,
        loans0_.user_idUser as user_idU3_1_1_,
        loans0_.status as status1_1_1_
    from
        Loan loans0_
    where
        loans0_.book_ISBN=?

    insert into Loan (status, book_ISBN, user_idUser)
    values
        (?, ?, ?)
```

Update operations

Update SQL queries are executed with JPA and Hibernate by finding the object we want to modify and using its setter methods to update the attribute we want to change, doing this inside a transaction, in order to avoid a merge. What follows is an example of a bidirectional many-to-many relationship with a link entity.

```
public void validateBorrow(String userid, long bookId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();

        LoanId loanid = new LoanId(userid, bookId);
        Loan loan = entityManager.find(Loan.class, loanid);

        if(loan.getStatus() == 0)
            loan.setStatus(1);

        entityManager.getTransaction().commit();
    }catch (Exception ex) {
        ex.printStackTrace();
        System.out.println("A problem occurred with the loan validation!");
    }
    finally {
        entityManager.close();
    }
}
```

Hibernate will execute the following queries:

```
    select
        loan0_.book_ISBN as book_ISB2_1_0_,
        loan0_.user_idUser as user_idU3_1_0_,
        loan0_.status as status1_1_0_,
        book1_.ISBN as ISBN1_0_1_,
        book1_.author as author2_0_1_,
        book1_.category as category3_0_1_,
        book1_.numCopies as numCopie4_0_1_,
        book1_.title as title5_0_1_
    from
        Loan loan0_
    inner join
        Book book1_
            on loan0_.book_ISBN=book1_.ISBN
    where
        loan0_.book_ISBN=?
        and loan0_.user_idUser=?

    update Loan
    set
        status=?
    where
        book_ISBN=?
        and user_idUser=?
```

Delete operations

Delete SQL queries are executed with JPA and Hibernate by finding the object we want to delete and using his remove methods to delete the target object, doing this inside a transaction. Also, in this type of operation, if the relationship is bidirectional, we should use helper methods to synchronize the related collections. What follows is an example of a bidirectional many-to-many relationship with a link entity.

```
public void validateReturn(String userid, long bookId) {
    try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();

        LoanId loanid = new LoanId(userid, bookId);
        Loan loan = entityManager.find(Loan.class, loanid);

        if(loan.getStatus() == 2)
            loan.getUser().removeLoan(loan.getBook());

        entityManager.getTransaction().commit();
    }catch (Exception ex) {
        ex.printStackTrace();
        System.out.println("A problem occurred with the loan validation!");
    }
    finally {
        entityManager.close();
    }
}
```

Hibernate will execute the following queries:

```
    select
        loan0_.book_ISBN as book_ISB2_1_0_,
        loan0_.user_idUser as user_idU3_1_0_,
        loan0_.status as status1_1_0_,
        book1_.ISBN as ISBN1_0_1_,
        book1_.author as author2_0_1_,
        book1_.category as category3_0_1_,
        book1_.numCopies as numCopie4_0_1_,
        book1_.title as title5_0_1_
    from
        Loan loan0_
    inner join
        Book book1_
            on loan0_.book_ISBN=book1_.ISBN
    where
        loan0_.book_ISBN=?
        and loan0_.user_idUser=?

    select
        user0_.idUser as idUser1_2_0_,
        user0_.name as name2_2_0_,
        user0_.privilege as privileg3_2_0_,
        user0_.surname as surname4_2_0_
    from
        User user0_
    where
        user0_.idUser=?
```

```
select
    loans0_.user_idUser as user_idU3_1_0_,
    loans0_.book_ISBN as book_ISB2_1_0_,
    loans0_.book_ISBN as book_ISB2_1_1_,
    loans0_.user_idUser as user_idU3_1_1_,
    loans0_.status as status1_1_1_,
    book1_.ISBN as ISBN1_0_2_,
    book1_.author as author2_0_2_,
    book1_.category as category3_0_2_,
    book1_.numCopies as numCopie4_0_2_,
    book1_.title as title5_0_2_
from
    Loan loans0_
inner join
    Book book1_
        on loans0_.book_ISBN=book1_.ISBN
where
    loans0_.user_idUser=?

select
    loans0_.book_ISBN as book_ISB2_1_0_,
    loans0_.user_idUser as user_idU3_1_0_,
    loans0_.book_ISBN as book_ISB2_1_1_,
    loans0_.user_idUser as user_idU3_1_1_,
    loans0_.status as status1_1_1_
from
    Loan loans0_
where
    loans0_.book_ISBN=?

delete
from
    Loan
where
    book_ISBN=?
    and user_idUser=?
```