1. How to make the field unmodifiable ?

There are two ways to handle this:

**Apply Hibernate’s *@Immutable* annotation on the collection.**

@OneToMany(cascade = CascadeType.ALL)

@JoinColumn(name = "receipt\_id")

@org.hibernate.annotations.Immutable

private Set lineItems;

With this setup, calling the *merge()* method in above test will result in an exception.

org.hibernate.HibernateException:

changed an immutable collection instance:

[com.allaroundjava.model.Receipt.lineItems#1]

Instead of silently skipping the persistence of additional items in a set, Hibernate is throwing an exception. If that wasn’t the case, Receipt contents in application memory and in database would not match.

**Return a defensive, immutable copy.**

public Set getLineItems() {

return Collections.unmodifiableSet(lineItems);

}

This *Set* implementation will disallow any modifications made to its contents. Attempt to do so, will result in *UnsupportedOperationException* at the moment of calling *add()* on *Set*. This approach is also very verbose, allowing to discover issues early. It also brings a good habit of returning a defensive copy of a referenced collection property.

1. NEFT / RTGS / IMPS / UPI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Description** | **NEFT** | **RTGS** | **IMPS** | **UPI** |
| Full-Form | National Electronic Fund Transfer | Real-Time Gross Settlement | Immediate Mobile Payment Service | Unified Payment Interface |
| Regulated by | RBI | RBI | NPCI | NPCI |
| Settlement Type | Half Hourly Batches | Immediate Settlement | Immediate Settlement | Immediate Settlement |
| Minimum Transfer | Rs.1 | Rs.2 lakh | Rs.5 | Rs.1 |
| Maximum Transfer | 50000 | No limit | Rs.2 lakh | Rs. 1 lakh |
| Payment options | Online and Offline | Online and Offline | Online | Online |
| Timings | In Batches- 24/7 | Weekdays: 9:00 AM - 4:30 PM | 24/7, 365 days | 24/7, 365 days |
| Charges | No Charges | Charges differ with bank and amount | Difffers | NO charges |

1. PessimisticLockException ?

* javax.persistence
* Class PessimisticLockException
* java.lang.Object
* java.lang.Throwable
* java.lang.Exception
* java.lang.RuntimeException
* javax.persistence.PersistenceException
* javax.persistence.PessimisticLockException

We want to lock it for ourselves for further processing .so nobody interrupts our actions.

There are 2 ways

1. setting proper Isolation Level
2. setting a lock on data that we need at the moment.

For isolation level, it affects every statement within the connection.

So we use pessimistic locking which uses database mechanism for reserving more granular access to data and no other transactions can modify.

There are 2 types of locks.

1. Exclusive lock - We can do both read and write on the data.
2. Shared lock – We can read the data but can’t write.

JPA Specification defines 3 pessimistic lock modes. Static members of LockModeType class.

We can obtain only one lock at a time. Never throw PersistenceException.

1. PESSIMISTIC\_READ - allows obtaining a shared lock. Data can’t be updated or deleted.
2. PESSIMISTIC\_WRITE- allows obtaining a exclusive lock and prevent read,update or delete .
3. PESSIMISTIC\_FORCE\_INCREMENT- works like PESIMISTIC\_WRITE and it additionally increatements a version attribute of versioned entity . @Version Entities. It updates the version column.

Exception :

PessimisticLockException: indicates obtaining a lock or converting a shared to exclusive lock fails then results in transaction-level rollback.

LockTimeoutException : Indicates that obtaining a lock or converting a shared to exclusive timeout , then results in statement level rollback.

PersistanceException : indicates that persistence problem occurred.Makes active transaction to rollback.

**Find**

It's probably the most straightforward way. It's enough to pass a *LockModeType*object as a parameter to the *find* method:

entityManager.find(Student.class, studentId, LockModeType.PESSIMISTIC\_READ);

**3.2. Query**

Additionally, we can use a *Query* object as well and call the *setLockMode* setter with a lock mode as a parameter:

Query query = entityManager.createQuery("from Student where studentId = :studentId");

query.setParameter("studentId", studentId);

query.setLockMode(LockModeType.PESSIMISTIC\_WRITE);

query.getResultList()

**3.3. Explicit Locking**

It's also possible to lock manually the results retrieved by the find method:

Student resultStudent = entityManager.find(Student.class, studentId);

entityManager.lock(resultStudent, LockModeType.PESSIMISTIC\_WRITE);

### Refresh

If we want to overwrite the state of the entity by the refresh method, we can also set a lock:

Student resultStudent = entityManager.find(Student.class, studentId);

entityManager.refresh(resultStudent, LockModeType.PESSIMISTIC\_FORCE\_INCREMENT);

### NamedQuery

@NamedQuery annotation allows us to set a lock mode as well:

@NamedQuery(name="lockStudent",

query="SELECT s FROM Student s WHERE s.id LIKE :studentId",

lockMode = PESSIMISTIC\_READ)

## 4. Lock Scope

**Lock scope parameter defines how to deal with locking relationships of the locked entity.** It's possible to obtain a lock just on a single entity defined in a query or additionally block its relationships.

To configure the scope we can use PessimisticLockScope enum. It contains two values: NORMAL and EXTENDED.

We can set the scope by passing a parameter ‘javax.persistance.lock.scope‘ with PessimisticLockScope value as an argument to the proper method of EntityManager, Query, TypedQuery or NamedQuery:

Map<String, Object> properties = **new** HashMap<>();

map.put("javax.persistence.lock.scope", PessimisticLockScope.EXTENDED);

entityManager.find(

Student.class, 1L, LockModeType.PESSIMISTIC\_WRITE, properties);

### 4.1. PessimisticLockScope.NORMAL

**We should know that the PessimisticLockScope.NORMAL is the default scope.** With this locking scope, we lock the entity itself. When used with joined inheritance it also locks the ancestors.

Let's look at the sample code with two entities:

@Entity

@Inheritance(strategy = InheritanceType.JOINED)

**public** **class** **Person** {

@Id

**private** Long id;

**private** String name;

**private** String lastName;

// getters and setters

}

@Entity

**public** **class** **Employee** **extends** **Person** {

**private** BigDecimal salary;

// getters and setters

}

When we want to obtain a lock on the Employee, we can observe the SQL query which spans over those two entities:

**SELECT** t0.ID, t0.DTYPE, t0.LASTNAME, t0.NAME, t1.ID, t1.SALARY

**FROM** PERSON t0, EMPLOYEE t1

**WHERE** ((t0.ID = ?) **AND** ((t1.ID = t0.ID) **AND** (t0.DTYPE = ?))) **FOR** **UPDATE**

### 4.2. PessimisticLockScope.EXTENDED

The EXTENDED scope covers the same functionality as NORMAL. In addition, **it's able to block related entities in a join table**.

Simply put, it works with entities annotated with @ElementCollection or @OneToOne, @OneToMany etc. with @JoinTable.

Let's look at the sample code with the @ElementCollection annotation:

@Entity

**public** **class** **Customer** {

@Id

**private** Long customerId;

**private** String name;

**private** String lastName;

@ElementCollection

@CollectionTable(name = "customer\_address")

**private** List<Address> addressList;

// getters and setters

}

@Embeddable

**public** **class** **Address** {

**private** String country;

**private** String city;

// getters and setters

}

Let's analyze some queries when searching for the Customer entity:

**SELECT** CUSTOMERID, LASTNAME, NAME

**FROM** CUSTOMER **WHERE** (CUSTOMERID = ?) **FOR** **UPDATE**

**SELECT** CITY, COUNTRY, Customer\_CUSTOMERID

**FROM** customer\_address

**WHERE** (Customer\_CUSTOMERID = ?) **FOR** **UPDATE**

We can see that there are two ‘FOR UPDATE‘ queries which lock a row in the customer table as well as a row in the join table.

**Another interesting fact we should be aware of is that not all persistence providers support lock scopes.**

## 5. Setting Lock Timeout

Besides setting lock scopes, we can adjust another lock parameter – timeout.**The timeout value is the number of milliseconds that we want to wait for obtaining a lock until the LockTimeoutException occurs.**

We can change the value of timeout similarly to lock scopes, by using property ‘javax.persistence.lock.timeout' with the proper number of milliseconds.

It's also possible to specify ‘no wait' locking by changing timeout value to zero. However, we should keep in mind that there are database drivers which **don't support setting a timeout value this way.**

Map<String, Object> properties = **new** HashMap<>();

map.put("javax.persistence.lock.timeout", 1000L);

entityManager.find(

Student.class, 1L, LockModeType.PESSIMISTIC\_READ, properties);

## 6. Conclusion

When setting the proper isolation level is not enough to cope with concurrent transactions, JPA gives us pessimistic locking. It enables us to isolate and orchestrate different transactions so they don't access the same resource at the same time.

To achieve that we can choose between discussed types of locks and consequently modify such parameters as their scopes or timeouts.

On the other hand, we should remember that understanding database locks is as important as understanding the mechanisms of underlying database systems. It's also important to have in mind that the behavior of pessimistic locks depends on persistence provider we work with.

* **Eager Loading** is a design pattern in which data initialization occurs on the spot.
* **Lazy Loading** is a design pattern that we use to defer initialization of an object as long as it's possible.

### ****6.1. Lazy Loading****

Advantages:

* Much smaller initial load time than in the other approach
* Less memory consumption than in the other approach

Disadvantages:

* Delayed initialization might impact performance during unwanted moments.
* In some cases we need to handle lazily initialized objects with special care, or we might end up with an exception.

### ****6.2. Eager Loading****

Advantages:

* No delayed initialization-related performance impacts

Disadvantages:

* Long initial loading time
* Loading too much unnecessary data might impact performance

## **7. Lazy Loading in Hibernate**

**Hibernate applies lazy loading approach on entities and associations by providing a proxy implementation** of classes.

Hibernate intercepts calls to an entity by substituting it with a proxy derived from an entity’s class. In our example, missing requested information will be loaded from a database before control is ceded to the User class implementation.

We should also note that when the association is represented as a collection class (in the above examples, it is represented as Set<OrderDetail> orderDetailSet), a wrapper is created and substituted for an original collection.

To know more about proxy design pattern, refer [here](https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html).

@Entity

@Table(name = "USER")

**public** **class** **UserLazy** **implements** **Serializable** {

@Id

@GeneratedValue

@Column(name = "USER\_ID")

**private** Long userId;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "user")

**private** Set<OrderDetail> orderDetail = **new** HashSet();

// standard setters and getters

// also override equals and hashcode

}

Next, we'll see the OrderDetail class:

@Entity

@Table (name = "USER\_ORDER")

**public** **class** **OrderDetail** **implements** **Serializable** {

@Id

@GeneratedValue

@Column(name="ORDER\_ID")

**private** Long orderId;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name="USER\_ID")

**private** UserLazy user;

// standard setters and getters

// also override equals and hashcode

}

Pagination:

Implement pagination in Spring Boot is quite easy only you need to follow basic steps -

1 - Extends PagingAndSortingRepository in repository interface

public interface UserRepository extends PagingAndSortingRepository <User, Long>

2 - Method declaration should be like below example

Page<User> userList(Pageable pageable);

3 - Method implementation in Service class should be like below example

@Override

public Page<User> userList(Pageable pageable) {

return userRepository.findAll(pageable);

}

4 - Controller class code should be like below

@GetMapping("/list")

public String userList(Model model, Pageable pageable) {

Page<User> pages = userService.userList(pageable);

model.addAttribute("number", pages.getNumber());

model.addAttribute("totalPages", pages.getTotalPages());

model.addAttribute("totalElements",

pages.getTotalElements());

model.addAttribute("size", pages.getSize());

model.addAttribute("users", pages.getContent());

return "/user/list";

}

From front-end call should be like below

http://localhost:8080/application/user/list?page=0&size=5

http://localhost:8080/application/user/list?page=1&size=5

http://localhost:8080/application/user/list?page=2&size=5