# JPA- Spring Data JPA and Hibernate

-- Hibernate uses a sequence to insert values when inserting data with @generatedValue for ID.

-- Entity needs to have adefault constructor with no parameters

-- Entitymanager.flush would send all the changes at that point to the DB , i.e. the records updated till that point would be sent to DB even before the txn is committed



Under the @Transactional attribute, the object is under the purview of Entitymanager, so even after saving, like em.save(course),

if the course entity is updated,

like course.setName("1"), this would be updated in the DB.

This is because the entity is attached to EM still , and any changes would be reflected in the DB.

To detach the entity from EM after saving the obj, use

-- em.detach(course);

-- em.clear() -- this would clear everything that is attached to the Entitymanager

all objects being monitored by it would be detached

-- em.refresh() would read the latest value from DB

All entitied saved via Entitymanager are saved to PersistenceContext

This context keeps track of all entities updated / saved by EM.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------

JPQL :

JPQL queries are written using entities.

They interact with entities, internally converted to sql queries.

select c from course // course is our entity.

c is the alias for \*.

With named query, u can give name to a query and use it later

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

Relationships :

onetoOne

Student - Passport :

If, passport object included in student class, then iby default it is loaded when student object is loaded, this is called EAGER FETCH.

By default, in one to one relationships, there is Eager fetch.

Even if you do not want to read the passport object by default, it would be fetched with student by default. This can result in performance issues, because passport would be fetched even if not required.

SLF4J: Failed toString() invocation on an object of type [org.hibernate.collection.internal.PersistentBag]

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

Many to many relationship :

When there is many to many reln,

it does not matter which side is the owning side of the relationship. We use join table

// in many to many, there is no owning side of the relationship, so we can

// define mapped by on any side

// A separate join table is created with student id and course id mapping

reference is not stored in same tables, boc there can be multiple references to one object

Hence, a separate table is created to store both mappings.

After we define mappedby on any of the sides of many to many, the other one can then technically be called the owning side.

We can put the @jointable annotation there.

Using @jointable annotation, we can give the table our own name, (the table which contains mappings for all records of the many to many relationship).

Then we also specify the join column and the inverse join column.

In the joincolumn annotation, we will give the id of the owning side of the relationship, i.e studentid (coz student is the owning side if mapped by is defined in Course table).

In inverse join column annotation, the column which is specified is the column on the other side

Many to many is lazy fetch by default

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

Hibernate :

Create a session factory - single object per application

Session factory is a very heavy object and requires lots of resources, so it is loaded only once.

Create a session from the session factory.

Use the session to save the model objects

SessionFactory factory = new configuration().configure().buildSessionFactory("hibernate.cfg.xml");

Session session = factory.openSession();

session.beginTransaction();

session.save(modelObject);

session.getTransaction().commit();

session.close();

to read objects, open another session and get object from it

session = factory.openSession();

session.beginTransaction();

session.get(UserDetails.class, id);

session.close();

`

If you have @transient or @static property, it would not be persisted

So if you dont want a field in the entity to be saved to DB, make it static or mark as @transient

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

Hibernate waits until the last point in the transaction before firing the queries

If we are using @Transactional, queries are inserted / fired at the end of the method's execution.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------

Inheritance in JPA :

A.

@Inheritance(stratery = InheritanceType.SingleTable)

: default type

: When you define classes in inheritance way,

like class Employee is parent abstract class, and class FullTimeEmployee and PartTimeEmployee extend that parent class, and all of these are @entity

Then if we want all the data from these entities to be persisted into the same table in the DB, we use this strategy.

Good for performance perspective, it is good because data can be fetched from one table.

But, data integegrity and management becomes difficult when there is lots of data, then there are lots of null values also.

An additional column gets added into the table, dType (discriminatory type), which defines which type of value it is, from both the child classes.

B. @Inheritance(stratery = InheritanceType.TablePerClass)

Here, there is a table for all the classes, not the abstract classes

So there is a table for all child classes, but the columns belonging to the parent class are present in all tables belonging to subclasses.

This would mean duplicate data in many cases.

To fetch data from all tables,

a union is executed in all tables.

From performance point of view it is good still, but data duplicacy is lot.

C. @Inheritance(stratery = InheritanceType.JOINED)

Parent class will have its own table

Sub classes have their own table

Common data is saved in parent table

Specific data for all subclasses is saved in their own tables, also a column (like id), as a foreign key from the parent table to help in joins.

to retreive data joins are executed.

this is really good in terms of DB design, but from performance it is not very good, as their are joins executed to fetch data and when there are lots of tables it would become really big.

D. @MappedSuperClass

Put this to the parent class.

NOTE : @MappedSuperClass cannot be an entity

No table exists for the mapped superclass.

So, you cannot query on the parent class using it's entity name.

You can only query on the child classes, which have entities.

Actually, there is not inheritance between the tables here.

There is just some common data between the 2 tables.

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

ACID properties

Atomicity :

Basic thing that a transaction should do. Either all steps of the transaction should be successful, or all should be rolled back / failed.

Consistency / Concurrency:

There can be many transactions happening in parallel.

Any transaction should leave the system in a consistent state.

E.g bank transaction : If the txn where the money is being transferred from Account A to Account B -> If the money being added to account B fails, then money should not be deducted from account A as well.

If it is deducted from A but not added to B, it leaves the system in an inconsistent state.

Isolation:

If there are many transactions running in parallel.

If a transaction 1 is updating account A's value, in between another txn tries to read the value of A, will it get the correct value of A updated.

This is determined by Isolation.

Changes within a txn are visible to other ones. What level of changes in a transactionu want to be visible outside the transaction at that moment ?

Durability:

Any change that you make is durable. If the transaction was completed, but the system crashed after that or something like that happened, the changes made by the transaction should be persisted and durable.

When data is being updated by one transaction and being read by another transaction, how do we manage it ?

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

Below are 3 types of problems which occur with transactions :

Dirty Reads , Non Repeatable Reads and Phanthom Reads :

Dirty Reads :

Dirty read is another transaction reading the modified value of one transaction before the transaction is committed.

Txn 2 is reading value of transaction 1 before 1 has been committed.

Non Repeatable Reads :

Consider a table Person with 3 rows : id, name and age

We execute 3 queries :

select \* from person where age = 10;

update person set name = "B" where age = 10;

select \* from person where age = 10;

Here we execute a query initially, where we execute a query to read values from person table, but in the 2nd step we update the row.

So when we read this again in the next step, we dont get the same value.

This is called non-repeatable read, i.e. when we are reading the same value twice in a transaction, but in between the value has been updated by some other transaction, so we dont get the same value the second time.

Phanthom Reads :

select \* from person where between 10 and 20;

insert into person --> age =15, name = "X"

select \* from person where between 10 and 20;

A query is executed at the first step,

but in between there is a record inserted, so when the query is executed again, it gives new results.

This is called phanthom read

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

Now, we discuss 4 types of isolation levels, and how they help solve the problems :

1. Read uncommitted :

It does not wait for transaction to be committed for other transactions to read it.

As soon as it changes the value, it is readable by the other transaction.

It does not have any locks at all.

It does not solve any problems of the 3 defined above .

2. Read Committed :

A transaction would be able to read data of another transaction,only if that data has been committed by the other transaction.

any data changed during txn will be available only after it has been committed.

So, there will be no DIRTY READS.

non repeatable and phanthom reads are still possible.

3. Repeatable Reads :

This, does not only lock the modified values in the transaction, but also locks the values that were read during the transaction.

If the query : select \* from person where age = 10;

was executed during the transaction, and the row was read, then the row/object would be locked.

Until the transaction is completed, it remains locked. Once the txn is completed, the lock is released and it's value can be modified by other transactions.

Here, both DIRTY READS and Non repeatable Reads are solved.

Not the phanthom read -> because data can be inserted.

4. Serializable :

Solves all problems.

When a query is executed : select \* from person where between 10 and 20;

A lock is created for any row which matches this constraint.

Whether a txn is trying to insert or delete or modify data which matches this constraint (10-20), it wont be allowed to do so.

Thisi solves all problems.

Choosing between isolation levels :

When we use Serializable, we wont have any of the dirty reads, phanthom reads, or non repeatable reads. However, we will have very bad performance.

If we are reading a table in a txn, and there are 5000 txns running in parallel, then each txn would have to wait for the first one to complete.

so, we have to consider our application before trying to put the transaction level.

Typically, the level used is Read Committed.

It is a balance between the integrity of data and the performance of the system.

You dont really lock a lot of stuff, and system data is decently consistent.

you can also have different isolation levels in the same application for different scenarios.

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

There is a @transactional annotation in JPA and in Spring.

When we are talking to just 2 DB, JPa annotation is sufficient.

When trying to communicate with multiple DBs, queues etc, we should use the one provided by Spring.

When using Spring @transactional annotation, you can define isolation level.

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

4. Pagination and sorting techniques

4.1. Paging WITHOUT sorting

To apply only pagination in result set, we shall create Pageable object without any Sort information.

Pageable paging = PageRequest.of(pageNo, pageSize);

Page<EmployeeEntity> pagedResult = repository.findAll(paging);

4.2. Paging WITH sorting

To apply only pagination in result set, we shall create Pageable object with desired Sort column name.

Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by("email"));

Page<EmployeeEntity> pagedResult = repository.findAll(paging);

By default, records are ordered in DESCENDING order. To choose ASCENDING order, use .ascending() method.

Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by("email").ascending());

Page<EmployeeEntity> pagedResult = repository.findAll(paging);

4.3. Sorting only

If there is no need to page, and only sorting is required, we can create Sort object for that.

Sort sortOrder = Sort.by("email");

List<EmployeeEntity> list = repository.findAll(sortOrder);

If we wish to apply sorting on multiple columns or group by sort, then that is also possible by creating Sort using simple builder pattern steps.

Sort emailSort = Sort.by("email");

Sort firstNameSort = Sort.by("first\_name");

Sort groupBySort = emailSort.and(firstNameSort);

List<EmployeeEntity> list = repository.findAll(groupBySort);

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

5. Difference between Page and Slice

5.1. Page

The findAll(Pageable pageable) method by default returns a Page object. A Page object provides lots of extra useful information other than just list of employees in current page.

E.g. A Page object has the number of total pages, number of the current page and well as whether current page is first page or last page.

Finding total pages invokes an additional count() query causing an extra overhead cost. Be sure when you are using it.

5.2. Slice

Slice is very much similar to Page, except it does not provide the number of total pages in database. It helps in improving performance when we do not need to display total number pages in UI.

Generally, Slice is used in case navigation is consist of Next page and Previous page links.

To use Slice, we have implement our own custom methods.

EmployeeRepository.java

public interface EmployeeRepository extends CrudRepository<EmployeeEntity, Long>

{

public Slice<EmployeeEntity> findByFirstName(String firstName, Pageable pageable);

}

Remember that is we use PagingAndSortingRepository, default return type is Page.

Pageable paging = PageRequest.of(pageNo, pageSize, Sort.by("email").descending());

Slice<EmployeeEntity> slicedResult = repository.findByFirstName("alex", paging);

List<EmployeeEntity> employeeList = slicedResult.getContent();

---------------------------------------------------------------------------------------------------------------------------------------------------------------------

6. Spring boot paging and sorting demo

In this demo, default page number is 0, page size is 10 and default sort column is ‘id’.

Now here are differnt types of URLs, which can be used for different types of paging and sorting options.

http://localhost:8080/employees?pageSize=5

http://localhost:8080/employees?pageSize=5&pageNo=1

http://localhost:8080/employees?pageSize=5&pageNo=2

http://localhost:8080/employees?pageSize=5&pageNo=1&sortBy=email

http://localhost:8080/employees?pageSize=5&pageNo=1&sortBy=firstName

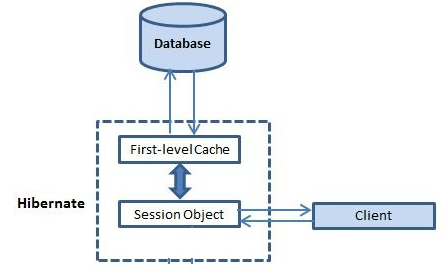
--------------------------------------------------------------------------------------

# Understanding Hibernate First Level Cache with Example

Caching is a facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate achieves the second goal by implementing first level cache.

**Fist level cache** in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you can not disable it even forcefully.

Its easy to understand the first level cache if we understand the fact that**it is associated with Session object**. As we know session object is created on demand from session factory and**it is lost, once the session is closed**. Similarly, first level cache associated with session object is available only till session object is live. It is available to session object only and is **not accessible to any other session object** in any other part of application.



## **Important facts**

1. First level cache is associated with “session” object and other session objects in application can not see it.
2. The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
3. First level cache is enabled by default and you can not disable it.
4. When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
5. If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
6. The loaded entity can be removed from session using evict() method. The next loading of this entity will again make a database call if it has been removed using evict() method.
7. The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.