Hibernate Summary

Author : Ankit Panda
Mail : akpanda.ds@gmail.com
Sample implementation

https://github.com/akpandads/HibernateImpl
https://github.com/akpandads/AdvanceJpa

Accessing DB

ORM model which maps tables to classes and fields to variables
Use inmemory database like H2 for demo purposes
JPA is the interface while hibernate is the implementation

If using spring boot, autoconfiguration feature configures all the database connections
There are different ways of retrievig data from DB

1. Spring JDBC teamplate

- Autowired by spring boot
- takes parametred
- Advantageous over regular JDBC , no bboiler plate code required to map rows and manage trnsactions
- Uses a BeanPropertyRowMapper to map the result to a entity class

```
      jdbcTemplate.query("select * from person", new BeanPropertyRowMapper<Person>(Person.class));
---- direct maoping
            jdbcTemplate.queryForObject("select * from person where id=?",new Object[]{id}, new
BeanPropertyRowMapper<Person>(Person.class)); --paramter passing via Object Array

            class PersonRowMapper implements RowMapper < Person > {     -------Inner class in the
entity object to for custom row mapper. Note the default can also be used if the table cloumns and
class variables name matches

                @Override
                 public Person mapRow(ResultSet resultSet, int i) throws SQLException {
                      Person person = new Person();
                      person.setId(resultSet.getInt("id"));
                      person.setName(resultSet.getString("name"));
                      person.setLocation(resultSet.getString("location"));
                      person.setBirthDate(resultSet.getDate("birth_date"));
                      return person;
                 }
        }
```

2. Entity Manager

- Persistence context provides a bean entity manager
- Persistence context mnatains all the beans that are associated with the rows

```
@PersistenceContext
EntityManager entityManager;

entityManager.find(Person.class,id);
entityManager.merge(person);

TypedQuery <Person> typedQuery =  entityManager.createNamedQuery("find_all_personss",Person.class); ----- the
typed query name  is defined at the entity class. here just the name of the query is used
return typedQuery.getResultList();
```

3. Spring Data Jpa

- No need for creating Entity Manager and respository
- Implement the Jpa Repository or Crud Repository and use availabale functions
- Not suitable for complex queries or the one which require performance optimization
- Supports out of box sorting and pagination functionalities
- inbuilt methods like find(), findAll(), findBy<ElementName>() are available

JPA annotations

We will look at some common JPA annotations and methiods on this section

@Entity to Denote Entity class
@Table(name= <table name>) used if the entity name is different from table name
@Column(name =<name>) denote mapping between column and entity variable
@NamedQuery(name="get_all_courses",query="select c from Course c") = Query to be used with name. Defined at top of entity class

@UpdationTimeStamp = java type can be mapped to LocalDateTime. Used to store timestamp of last updation on the row
@CreationTimeStamp = java type can be mapped to LocalDateTime. Used to store timestamp of creation on the row
@Id = Denote primary key
@GeneratedValue = denote that the value is generated from a generator

@Transactional = When used on a method or class, all the db operatsions on that method is condered as one transaction
@DirtiesContext = Used on test methods usually to denote that the method is making changes on the persitence context. And the changes made as part of this method will be reverted back

@Enumerated = Use this whenever there are enums being used. This will ensure that string values are stored in DB and not the positional values of the enums

When are the methods called

If the fetch type is eager, then all the associated data are called at once.
if fetch type is s lazy the queries are fired when the associated data are fetched
Hibernate usually holds on persisting data till the last moment. All the changes done on entity objects are stored in the persistence context.
At end of the method all perstent context objects are persisted in the db, that is yupdates are executed
If to force the changes to DB at regular intervals, the flush() can be used to force execute the the updates , instead of waiting for hibernate

The detach() is used to detach an object from the persistence context. once detached, the object any changes to the object are not stored in DB and it is not associated with any transactions
The clear() clears all the changes done to the entity object and not yet persisted in DB. In other words, the state of the enity is synced to the values in DB. All local changes in persistence context are lost

Relationships

there are 4 different types of rellationships that can be represented.
OneToOne, OneToMany, ManyToOne, ManyToMany
*ToOne are Eager fetch by default
*toMany are Lazy fetch by default

Bitdirectional Mapping, is where there exists a relation on boths sides from two entity classes . Changes in one entity object will make changes in the associated entity oobject of the other class
Unidirectiona Mapping is when there is only one way relationship between 2 entities. Chenges in the owning entity reflect in the owned entoty but not the other way round.l

@OneToOne
Consider 2 tables Student and passport. One student can have only one passport

Entity Student definition

```
@OneToOne(fetch = FetchType.LAZY)   --no mappedBy denotes owning side. here stident is the owning side
private Passport passport;
```

Entity Passport Definition

```
@OneToOne(fetch = FetchType.LAZY, mappedBy = "passport")  --mappedBy is not present on owning side of the relationship
private Student student;
```

Save Student with passport

```
    public void saveStudentWothPassport(){
        Passport passport = new Passport("G176251276");
        entityManager.persist(passport);
        Student student = new Student("Mike");
        student.setPassport(passport); entityManager.persist(student);  ---note both the entities need to be oersisted
separately
    }
```

@ManyToOne

Many reviews can be associated to one course

Review Entity Dfinition

```
    @ManyToOne
    private Course course;
```

@OneToMany

Each course entity can have many reviews

Entity Course definition

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

@ManyToMany

One course can have many students

```
   @ManyToMany(mappedBy = "courseList")
   private List <Student> students = new ArrayList<>();
```

One student can be in many courses

```
    @ManyToMany
    @JoinTable(name = "STUDENT_COURSE", joinColumns = @JoinColumn(name = "STUDENT_ID"),
                                inverseJoinColumns = @JoinColumn(name = "COURSE_ID") )
     private List <Course> courseList = new ArrayList<>();
```

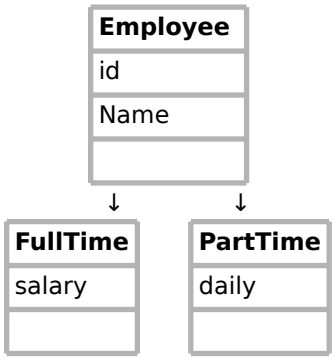
The joinTable creates another table which contains just the student id and course id. Each entry will correspond to a student id who is enrolled on the course

Inserting a row with student and course

```
    public void enterStudentAndCourse(Student student,Course course){
        entityManager.persist(student);
        entityManager.persist(course);
        course.getStudents().add(student);    ---- this is required to make the mappinf from course to student
        student.getCourseList().add(course); ----- this is required to make the mapping from student to course
    }
```

Inheritance Strategy

In this section we will discuss the 4 inheritance stratgy and Embeddable concpet
Consider the scenario where there is an abstract class Employee
2 classes FullTimeEmployee and PartTimeEmployee extend the Employee class. We will look at how this can be represented in hibernate and DB



1. InheritanceType.SINGLE_TABLE

The variables from Employee, PartTimeEmployee and FullTimeEmployee are combined into one table and all the variables are present as coums in a single table
This type is great for performance because only one table is present but poses a high risjk for data integrity
A single method on the superclass can be used to retrieve data. Since only one table, no joins are involved
The columns which do not correspond to a a particular table , are filled up as null nfor that particular row
we can use a @DiscrimantorColumn to create a column to detrmine the type of a particular row

Id	name	salary	hourly
1	x	null	12
2	y	1200	null

2. InheritanceType.TABLE_PER_CLASS

A separate table per class is created containing only the common columns of Employee and the specific columns of the subclass

id	name	salary

FullTimeEmployee Table

id	name	salary

Part Time Employee

3. InheritanceType.JOINED

This is great model where data integrity is more important than performance.
3 tables will be created. One for each type of the class. And the id of the super calss will act as foriegn key in the sublass tables

id	name

Employee table

salary	id (this the same entry as that of employee table)

FullTimeEmployee Table

salary	id (this the same entry as that of employee table)

Part Time EMployee Table

4. MappedSuperClass

Mapped super class removes all the relation between the super class and sublcass while forming table
A separate method needs tobe wrtitten for accessing data from each of the subtables as the super class object will not be considered as the generic one.
The tbales created will be similar o table per class strategy with the dofference that the anstract or supper class is not an entity.
So polymorphic queries cannot be written

Embeddable and Embedded

A composite object is one which contains another object within itself.
The Composite class need to have the @Embedded on the enclosing membering variable.
The smaller class need to have the @Embeddable annotation on it to denote that this can be embeeded on a composite class

In the DB representation, all the attributes of Embeddable class are expanded as represented as columns, within the composite class table

Transaction Management and Caching

Transaction Management is made easier with teh @Transactional annotation provided by Spring.

But there are a lot of issues that needs to be handled while managing a transaction.
Each transaction must adhere to the ACID properties to mantain data consistency

A- Atomicity : the whole transaction be considred as a single unit. EIther it should all persist or rolled back
C-Consistency : SUccessive reads within a transaction must retrun the same number of rows and same value
I-Isolation : Any number of transactions operating parallely nust not affect the other
D-Durable : A transaction once succesful must be persisted permanantly

Problems in transactions

Dirty Read : 2 simutaneous transactions. Transaction 1 changes value of x. Transaction 2 reads x. Transaction 1fails and reverts x. But transaction 2 updates on the modified value . Basically a trabsaction reads the value before it is persisted by another transaction
Non Repeatable read : Transaction 1 reads x. Transaction 2 updates x while transaction 2 in still in progress. Now transaction 1 reads the same vale and gets a different value for x
Phantom Read: Transaction1 does a select query and gets n number of records. transaction2 inserts/deletes some record while transaction 1 is in progress. transaction1 again does a select and gets another set of results ‘m’ different than initial ‘n’

Isolation levels

read uncommited : solves none of the issues
read commited : solves dirty read. Lock on all the values changed while a transaction in progress
repeateble read: solves dirty read and non-repeateble read. lock on entire row
Serialiable : solves all the problems. But performance issue.Everything locked that matches with query condiition

CACHING

2 levels of cache. L1 and L2.
Level 1 is bounded by each transaction. Once a value is fetched within a transaction boundary this is not again queries but fetched from the Persistence Contecxt
Level 2 cache spans across multiple transactions and this needs ti be specified inc ode what alldata needs to be in level 2 cache.

@Cacheable to be declared on the entity which needs to be cached

spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.internal.EhcacheRegionFactory
spring.jpa.properties.javax.persistence.sharedCache.mode=ENABLE_SELECTIVE

The above 3 show the properties. EhCache is one of the L2 cache libraries. ENABLE_SELCTIVE signifies bydefault none of the entt=ities will be cached but we can select that all needs to be cached
The opposite is DISABLE_SELECTIVE