

HIBERNATE

What is hibernate?

- Hibernate is a java framework that simplifies the development of java applications to interact with the database
- Hibernate is an **ORM**(Object Relational Mapping) tool.
- Hibernate is open source and lightweight.
- Hibernate is non-invasive framework, means it forces the programmer to extends/implement any class/interface
- Any kind of application can be built with Hibernate Framework.
- It was invented by GavinKing in 2001.

Hibernate Configuration

1. To configure hibernate
2. Create Maven Project
3. Add **hibernate core** and **mysql connector** dependency in pom.xml
4. Create **hibernate.cfg.xml** file at class level and do following configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost:3306/test</property>
    <property name="connection.username">fduser</property>
    <property name="connection.password">fdms!</property>
    <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
    <property name="hbm2ddl.auto">update</property>
    <property name="show_sql">true</property>
  </session-factory>
</hibernate-configuration>
```

5. Add doctype which can be found on internate by searching hibernate dtd.
6. If you get error like "**downloading from external resources is disabled**"
7. Go to the top bar : Window -> Preference -> Maven -> tick the option ('download artifact javadoc').

8. To get the sessionFactory in java class we use the **SessionFactory** interface of org.hibernate.SessionFactory package.
9. Since it is an Interface following is the code to get the object of SessionFactory

```

2
3 import org.hibernate.SessionFactory;
4 import org.hibernate.cfg.Configuration;
5
6 /**
7  * Hello world!
8  *
9  */
10 public class App
11 {
12     public static void main( String[] args )
13     {
14         System.out.println( "Hello World!" );
15         Configuration cfg = new Configuration();
16         cfg.configure("hibernate.cfg.xml");
17         SessionFactory factory = cfg.buildSessionFactory();
18
19         System.out.println(factory);
20     }
21 }

```

10.

First Hibernate Program

1. Create an java class/object for the database table using hibernate annotations

```

2
3 import javax.persistence.Entity;
4 import javax.persistence.Id;
5
6 @Entity
7 public class Student {
8
9     @Id
10     private int id;
11     private String name;
12     private String standerd;
13
14
15     public Student() {
16         super();
17         // TODO Auto-generated constructor stub
18     }
19 }

```

2. **Entity** : To define the class as database entity
3. **Id** : to define the primary key of the table

```

<property name="hbm2ddl.auto">update</property>
<property name="show_sql">true</property>

<mapping class="com.hibernate._FirstHibernateProject.Student"/>
</session-factory>

```

4. Map that class in hibernate.cfg.xml file using it's fully qualified name.
5. Then follow, Following steps

```

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");
        SessionFactory factory = cfg.buildSessionFactory();

        Student std = new Student();
        std.setId(2);
        std.setName("Devang");
        std.setStanderd("10nth");

        // Open the new session if not already opened
        // begin the transaction
        // Save object in session
        // Commit the transaction
        // Close the session
        Session session = factory.openSession();
        Transaction tr = session.beginTransaction();
        session.save(std);
        tr.commit();
        session.close();
    }
}

```

6. Property "*hbm2ddl.auto*" will create the table if already not exists, else will just insert the data.

Basic Hibernate Annotations

1. **@Entity** : Use to mark any class as entity
2. **@Table** : use to change the table details
3. **@Id** : Use to mark column as Id (primary key)
4. **@GeneratedValue** : It is same as we set column as auto generated in database hibernate will automatically generate values for that using internal sequence
5. **@Column** : Can be used to specify column mappings, To change the column name in the associate table in the database and other properties as well

6. **@Transient** : This tells hibernate not to save the field in the database
7. **@Temporal** : @Temporal over the date field tells
8. **@Lob** : @Lob tells hibernate that this is large object and not a simple object
9. **@OneToOne, @OneToMany, @ManyToMany, @JoinColumn**

```
5 import javax.persistence.Column;
5 import javax.persistence.Entity;
7 import javax.persistence.GeneratedValue;
8 import javax.persistence.GenerationType;
9 import javax.persistence.Id;
9 import javax.persistence.Lob;
1 import javax.persistence.Table;
2 import javax.persistence.Temporal;
3 import javax.persistence.TemporalType;
4 import javax.persistence.Transient;
5
5 @Entity
7 @Table(name = "teacher_info")
8 public class Teacher {
9
9     @Id
1     @GeneratedValue(strategy = GenerationType.IDENTITY)
2     private int id;
3
4     @Column(name = "teacher_name")
5     private String name;
6
7     @Column(name = "teacher_age", length = 10)
8     private int age;
9
9     @Column(name = "teacher_DOJ")
1     @Temporal(TemporalType.DATE)
2     private Date dateOfJoining;
3
4     @Transient
5     private String ignoreMe;
6
7     @Column(name = "teacher_img")
8     @Lob
9     private byte[] image;
```

10.

```

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

    Teacher teacher = new Teacher();
    teacher.setAge(34);
    teacher.setIgnoreMe("ABC");
    teacher.setName("Neha");
    teacher.setDateOfJoining(new Date());

    try {
        FileInputStream fileInputStream = new FileInputStream("C:\\Users\\gbhandari\\Downloads\\"
            + "pexels-rasikraj-1416900.jpg");
        byte[] data = new byte[fileInputStream.available()];
        fileInputStream.read(data);
        teacher.setImage(data);
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    Session session = factory.openSession();
    Transaction tr = session.beginTransaction();
    session.save(teacher);
    tr.commit();
    session.close();
}

```

11.

Fetching Object (get() & load())

Difference between get() and fetch()

get()	load()
Returns <code>null</code> if the object does not exist	Throws <code>ObjectNotFoundException</code> if the object does not exist
Executes immediately and hits the database	Returns a proxy without hitting the database immediately
Checks the first-level (session) cache first, and if not found, checks the second-level (SessionFactory) cache before hitting the database	Checks the first-level (session) cache before returning a proxy, does not check the second-level cache immediately
Use When you need the actual object immediately	Use When you want to delay the database call until the object is used

```

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

Session session = factory.openSession();

// Get method to fetch object from DB
System.out.println("----- Using get() -----");
Student student = (Student) session.get(Student.class, 1);
System.out.println(student.getAge());
System.out.println(student.getName());

// load method to fetch object from DB
System.out.println("----- Using load() -----");
Student student2 = (Student) session.load(Student.class, 1);
System.out.println(student2.getAge());
System.out.println(student2.getName());
session.close();

```

For get()

First-level cache: Hibernate checks if the `Student` with `id = 1` is in the session cache.

Second-level cache: If not found in the first-level cache, it checks the second-level cache.

Database: If not found in either cache, it queries the database.

The result is stored in the first-level cache for the session and also in the second-level cache for future use.

For load()

Initial Call: When `load()` is called, Hibernate returns a proxy object without hitting the database or caches immediately.

First-level cache: When you access `student.getAge()`, Hibernate first checks the session cache.

Second-level cache: If not found in the session cache, it checks the second-level cache.

Database: If not found in either cache, it queries the database.

If the `Student` with `id = 1` does not exist, an `ObjectNotFoundException` is thrown when accessing the proxy.

@Embeddable

The **@Embeddable** annotation in Hibernate is used to mark a class as a component that can be embedded in an entity. It signifies that the class's properties can be included in an entity's table rather than being a separate table.

```
@Entity
public class Student {

    @Id
    private int id;
    private String name;
    private Certificate cery;

    public int getId() {
```

```
@Embeddable
public class Certificate {

    private int ceryId;
    private String ceryName;

    public int getCeryId() {
```

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

Student student = new Student();
Certificate certificate = new Certificate();

certificate.setCeryId(100);
certificate.setCeryName("Java Certificate");

student.setId(1);
student.setName("Gaurav");
student.setCery(certificate);

Session session = factory.openSession();
Transaction tr = session.beginTransaction();
session.save(student);

tr.commit();
```

	id	ceryId	ceryName	name
	1	100	Java Certificate	Gaurav
*	(NULL)	(NULL)	(NULL)	(NULL)

@OneToOne

The **@OneToOne** annotation in Hibernate is used to define a one-to-one relationship between two entities. This relationship signifies that one instance of an entity is associated with one and only one instance of another entity.

```
@Entity
public class Question {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int qId;
    private String question;

    @OneToOne
    @JoinColumn(name = "a_id")
    private Answer answer;
```

```
@Entity
public class Answer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int aId;
    private String answer;

    @OneToOne(mappedBy = "answer")
    private Question question;

    public int getaId() {
        return aId;
    }
```

@JoinColumn

@JoinColumn: An annotation in Hibernate used to specify the name of foreign key column in a database table that joins two related entities.

@OneToOne(mappedBy = "answer")

mappedBy: An attribute that tells Hibernate which field in the other entity owns the relationship. It is used to define the inverse side of a relationship.

```
Question question = new Question();
Answer answer = new Answer();
question.setQuestion("what is java");

answer.setAnswer("java is a programing language");
question.setAnswer(answer);
answer.setQuestion(question);

Transaction tr = session.beginTransaction();
session.save(question);
session.save(answer);
tr.commit();

Question question2 = (Question) session.get(Question.class, 1);
System.out.println(question2.getQuestion());
System.out.println(question2.getAnswer().getAnswer());
session.close();
}
```


@OneToMany & @ManyToOne

@OneToMany: An annotation used in Hibernate to define a one-to-many relationship between two entities. This means one entity (the parent) is related to multiple instances of another entity (the child).

@ManyToOne: An annotation used in Hibernate to define a many-to-one relationship between two entities. This means many instances of one entity (the child) are associated with a single instance of another entity (the parent).

```
1 @Entity
2 public class Child {
3
4     @Id
5     @GeneratedValue(strategy = GenerationType.IDENTITY)
6     private int id;
7     private String name;
8     @ManyToOne
9     private Father father;
10
11     public int getId() {
12         return id;
13     }
14 }
```

```
10
11 @Entity
12 public class Father {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private int id;
17     private String name;
18     @OneToMany(mappedBy = "father")
19     private List<Child> childs;
20
21     public int getId() {
22         return id;
23     }
24 }
```

```
SessionFactory factory = org.hibernate.cfg.buildSessionFactory();
Session session = factory.openSession();

Father father = new Father();
father.setName("Vilas");

List<Child> childs = new ArrayList<>();

Child child = new Child();
child.setName("Vaibhav");
child.setFather(father);

Child child1 = new Child();
child1.setName("Dipali");
child1.setFather(father);

Child child2 = new Child();
child2.setName("Gaurav");
child2.setFather(father);

father.setChilds(childs);

Transaction tr = session.beginTransaction();
session.save(father);
session.save(child);
session.save(child1);
session.save(child2);
tr.commit();
session.close();
```

@ManyToMany

The `@ManyToMany` annotation in Hibernate is used to define a relationship where multiple records in one entity are associated with multiple records in another entity. For example, many students can enroll in many courses, and many courses can have many students enrolled. This annotation helps Hibernate manage and map this complex relationship in the database.

```
4
5 @Entity
6 public class Emp {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private int id;
11    private String eName;
12
13    @ManyToMany
14    @JoinTable(
15        name = "lk_emp_project",
16        joinColumns = {@JoinColumn(name = "emp_no")},
17        inverseJoinColumns = {@JoinColumn(name = "project_no")}
18    )
19    private List<Project> projects;
20
21    public int getId() {
22        return id;
23    }
24 }
```

```
10
11 @Entity
12 public class Project {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private int pId;
17     private String pName;
18
19     @ManyToMany(mappedBy = "projects")
20     private List<Emp> emps;
21
22     public int getpId() {
23         return pId;
24     }
25 }
```

@ManyToMany(mappedBy = "projects") :

Declaring that Project entity is an **inverse side** (non-owning side) of the mapping

@JoinTable(

```
    name = "lk_emp_project",
    joinColumns = {@JoinColumn(name = "emp_no")},
    inverseJoinColumns = {@JoinColumn(name = "project_no")}
```

)

1. **name** : to declare the join table name
2. **joinColumns** : to declare the name of the owning side of the mapping column.
3. **inverseJoinColumns** : to declare the name of the inverse side of the mapping column.

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

Emp emp1 = new Emp();
Emp emp2 = new Emp();

Project p1 = new Project();
Project p2 = new Project();

emp1.seteName("Gaurav");
emp2.seteName("Devang");

p1.setpName("FDMS");
p2.setpName("Wolkar");

List<Project> listProjectE1 = new ArrayList<>();
listProjectE1.add(p1);

List<Project> listProjectE2 = new ArrayList<>();
listProjectE2.add(p1);
listProjectE2.add(p2);

emp1.setProjects(listProjectE1);
emp2.setProjects(listProjectE2);

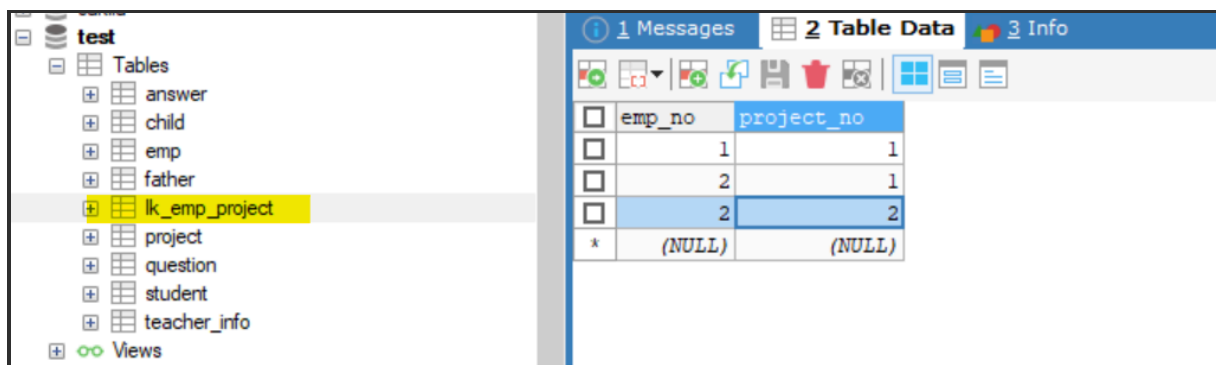
Session session = factory.openSession();
Transaction tr = session.beginTransaction();

session.save(emp1);
session.save(emp2);

session.save(p1);
session.save(p2);

tr.commit();
session.close();
```

Note : Here since the Emp entity is the Owning side of mapping, we are setting a list of projects to the emp entity.



emp_no	project_no
1	1
2	1
2	2
*	(NULL)

Fetch Technique

The most commonly used fetch techniques in Hibernate are **Lazy Loading** and **Eager Loading**.

Lazy Loading (Default type in hibernate)

Lazy loading is a design pattern in which data or resources are loaded only when they are specifically requested or needed, rather than at the initial loading time. In the context of Hibernate, lazy loading is a fetching strategy used for entity associations. When an entity is loaded, its associated entities or collections are not immediately fetched from the database. Instead, they are loaded on demand when their getters are accessed for the first time.

Common Use Case: This is the default fetch type for collections (e.g., **@OneToMany**, **@ManyToMany**) because it prevents unnecessary loading of large datasets when the parent entity is loaded.

```
17 Configuration cfg = new Configuration();
18 cfg.configure("hibernate.cfg.xml");
19 SessionFactory factory = cfg.buildSessionFactory();
20 Session session = factory.openSession();
21 Father father = session.get(Father.class, 1);
22 System.out.println(father.getName());
23 System.out.println("-----");
24 System.out.println(father.getChilds().size());
25
26 session.close();
```

Console × Problems Debug Shell

<terminated> FetchTechnique [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (09-Jul-2

Hibernate:

```
select
  father0_.id as id1_3_0_,
  father0_.name as name2_3_0_
from
  Father father0_
where
  father0_.id=?
Vilas
-----
```

Hibernate:

```
select
  childs0_.father_id as father_i3_1_0_,
  childs0_.id as id1_1_0_,
  childs0_.id as id1_1_1_,
  childs0_.father_id as father_i3_1_1_,
  childs0_.name as name2_1_1_
from
  Child childs0_
where
  childs0_.father_id=?
3
```

For Father and child entity refer the @OneToOne anno section

When we fetch the father info using get() only data from father table got fetched(as we can see in the sql)

And the data of the child entity or collection is fetched after the size() method is called that means when it's needed/requested.

Eager Loading

Eager loading is a design pattern in which related data or resources are loaded immediately at the time the parent entity is loaded, rather than on-demand. In the context of Hibernate, eager loading is a fetching strategy where all related entities or collections are fetched in a single query along with the main entity. This ensures that the associated entities are available right away without the need for additional database queries.

Common Use Case: Often used for @ManyToOne or @OneToOne relationships where the associated entity is small and almost always needed with the parent entity.

```
1
2 @Entity
3 public class Father {
4
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private int id;
8     private String name;
9     @OneToMany(mappedBy = "father", fetch = FetchType.EAGER)
10    private List<Child> childs;
11
12    public int getId() {
13        return id;
14    }
15
16    public static void main(String[] args) {
17        Configuration cfg = new Configuration();
18        cfg.configure("hibernate.cfg.xml");
19        SessionFactory factory = cfg.buildSessionFactory();
20        Session session = factory.openSession();
21        Father father = session.get(Father.class, 1);
22        System.out.println(father.getName());
23        System.out.println("-----");
24        System.out.println(father.getChilds().size());
25    }
26}

<
Console x Problems Debug Shell
<terminated> FetchTechnique [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (09-Jul-2024, 2:32:41 p
Hibernate:
select
    father0_.id as id1_3_0_,
    father0_.name as name2_3_0_,
    childs1_.father_id as father_i3_1_1_,
    childs1_.id as id1_1_1_,
    childs1_.id as id1_1_2_,
    childs1_.father_id as father_i3_1_2_,
    childs1_.name as name2_1_2_
from
    Father father0_
left outer join
    Child childs1_
        on father0_.id=childs1_.father_id
where
    father0_.id=?
Vilas
-----
3
```

Here we can see the Sql generated is join which will fetch the data from both the tables in a single query upfront.

Hibernate States (Hibernate Lifecycle)

The **Hibernate lifecycle** explains the states of an object in Hibernate from the time it is created until it is removed.

There are four main states in the life cycle:

1. Transient State
2. Persistence State
3. Detached State
4. Removed State

Transient State

The object is newly created and not yet saved in the database nor associated with open session.

```
java Copy code

// Creating a new entity
Employee employee = new Employee();
employee.setName("John Doe");
// At this point, the employee object is in a transient state
```

Persistence State

The object is saved in the database and linked to an active/open session. Changes to this object are synchronized with the database automatically.

```
Session session = sessionFactory.openSession();
session.beginTransaction();
```

```
session.save(emp); // Now emp is in the persistent state
emp.setSalary(6000); // This change will be automatically updated in the database
```

```
session.getTransaction().commit();
session.close();
```

Detached State

The object was once associated with a Hibernate **Session** but the session is now closed. Changes to the object are no longer synchronized with the database.

```
session.close(); // Now emp is in the detached state
emp.setSalary(7000); // This change will NOT be reflected in the database
```

Removed State

The object is marked for deletion from the database and will be deleted after the transaction is committed.

```
Session session = sessionFactory.openSession();  
session.beginTransaction();
```

```
session.delete(emp); // Now emp is in the removed state
```

```
session.getTransaction().commit();  
session.close();
```

Note : After committing the transaction using `session.getTransaction().commit();` following a call to `session.delete(emp);`, the object `emp` transitions to the **transient state**.

HQL (Hibernate Query Language)

An object-oriented query language designed for Hibernate that allows developers to write queries using Java class and property names instead of database tables and columns.

HQL	SQL
Hql is a database independent language	Sql is a database dependent language
Hql uses java objects and its properties to write queries.	Sql uses Database tables and columns to write queries.
Manages relationships with annotations like <code>@OneToOne</code> and <code>@OneToMany</code> , using Java objects instead of direct database tables.	Manages relationships using joins and subqueries
Type-safe: Queries are based on Java objects, ensuring errors are caught at compile time.	Less type-safe: Operates directly on raw table data, with errors detected only at runtime.

Read

1] get all rows from the table

```
Configuration cfg = new Configuration();
cfg.configure("hibernate.cfg.xml");
SessionFactory factory = cfg.buildSessionFactory();
String strQuery = "from Student";

Session session = factory.openSession();
Query query = session.createQuery(strQuery);
List<Student> list = query.list();
for(Student stuent : list) {
    System.out.println(stuent.getName());
    System.out.println(stuent.getCery());
}

session.close();
```

Here in **strQuery** variable Student is a **java entity** name and not a database table name

2] rows based on condition(s)

```
Configuration cfg = new Configuration();
cfg.configure("hibernate.cfg.xml");
SessionFactory factory = cfg.buildSessionFactory();
String strQuery = "from Student where name = 'Gaurav'";

Session session = factory.openSession();
Query query = session.createQuery(strQuery);
List<Student> list = query.list();
for(Student stuent : list) {
    System.out.println(stuent.getName());
    System.out.println(stuent.getCery().getCeryName());
}

session.close();
```

Here **name** is the **property/variable of Student object** and not the database column

3] rows by passing dynamic data for condition(s)

```
Configuration cfg = new Configuration();
cfg.configure("hibernate.cfg.xml");
SessionFactory factory = cfg.buildSessionFactory();
Session session = factory.openSession();

String strQuery = "from Student where name=: x and id=: y";
Query query = session.createQuery(strQuery);
query.setParameter("x", "Devang");
query.setParameter("y", 2);

List<Student> list = query.list();
for(Student student : list) {
    System.out.println(student.getName());
}
```

1. to use the placeholders for dynamic condition use `=:` and then **placeholder** name after the variable name.
2. And use the same placeholder name while setting the data for that placeholder by using **setParameter()** function of Query object.

4] We can use alias as well

```
Session session = factory.openSession();

String strQuery = "from Student as s where s.name=: x and s.id=: y";
Query query = session.createQuery(strQuery);
query.setParameter("x", "Devang");
query.setParameter("y", 2);

List<Student> list = query.list();
for(Student student : list) {
    System.out.println(student.getName());
}
```

Here you can see s is the alias for Object Student in the from clause.

Delete

```
public static void main(String[] args) {
    Configuration cfg = new Configuration();
    cfg.configure("hibernate.cfg.xml");
    SessionFactory factory = cfg.buildSessionFactory();
    Session session = factory.openSession();
    Transaction tr = session.beginTransaction();

    String strQuery = "delete from Student where id=: x";
    Query query = session.createQuery(strQuery);
    query.setParameter("x", 2);
    query.executeUpdate();

    tr.commit();
    session.close();
}
```

1. Use **executeUpdate()** for delete
2. And since we are manipulating data we have to use transaction

Update

```
SessionFactory factory = cfg.buildSessionFactory();
Session session = factory.openSession();
Transaction tr = session.beginTransaction();

String strQuery = "update Student set name=: x where id=: y";
Query query = session.createQuery(strQuery);
query.setParameter("x", "Devang");
query.setParameter("y", 1);
query.executeUpdate();

tr.commit();
session.close();
```

Join Query

```
1 @Entity
2 public class Child {
3
4     @Id
5     @GeneratedValue(strategy = GenerationType.IDENTITY)
6     private int id;
7     private String name;
8     @ManyToOne
9     private Father father;
10
11     public int getId() {
```

```
10 import java.util.List;
11
12 @Entity
13 public class Father {
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private int id;
18     private String name;
19     @OneToMany(mappedBy = "father", fetch = FetchType.EAGER)
20     private List<Child> childs;
21
22     public int getId() {
23         return id;
```

```
public class HQLJoinQuery {

    public static void main(String[] args) {
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");
        SessionFactory factory = cfg.buildSessionFactory();
        Session session = factory.openSession();

        String strQuery = "SELECT c.id, c.name, f.name FROM Child as c INNER JOIN c.father as f";
        Query query = session.createQuery(strQuery);
        List<Object[]> list = query.getResultList();
        for (Object[] columns : list) {
            System.out.println(Arrays.toString(columns));
        }

        session.close();
        factory.close();
    }
}
```

query returns `List<Object[]>`, each column is treated as a separate object within the `Object[]` array.

Note : The “`org.hibernate.Query`” interface is deprecated since hibernate 5.2 instead uses “`org.hibernate.query.Query`” interface.

```
5
6 // import org.hibernate.Query;
7 import org.hibernate.query.Query;
```

Pagination Using HQL

```
1 public class HQLPagination {
2
3     public static void main(String[] args) {
4         SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").buildSessionFactory();
5         Session session = factory.openSession();
6         String strQuery = "from Student";
7         Query query = session.createQuery(strQuery);
8         query.setFirstResult(1);
9         query.setMaxResults(5);
10        List<Student> list = query.list();
11        for(Student student : list) {
12            System.out.println(student.getId() + " " + student.getName() + " " + student.getCery());
13        }
14        factory.close();
15    }
16 }
```

Native Query

A native query in Hibernate is a SQL query that is directly executed against the database using Hibernate's Session object. Unlike HQL (Hibernate Query Language), which is an object-oriented query language specific to Hibernate, native queries allow you to write SQL in its native syntax, providing greater flexibility and control over database operations.

Key Points:

Direct SQL Execution: Native queries bypass Hibernate's query translation and directly execute the SQL as written.

Flexibility: They can be used to perform operations that are not easily achievable with HQL, such as database-specific functions, complex joins, or when using database features that Hibernate does not support directly.

Syntax: Written in the native SQL dialect of the underlying database, such as MySQL, PostgreSQL, etc.

Usage: Typically used via `createNativeQuery()` method provided by the Session interface in Hibernate.

```

public class NativeQuery {
    public static void main(String[] args) {
        SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").buildSessionFactory();
        Session session = factory.openSession();

        String sql = "SELECT * FROM student";
        org.hibernate.query.NativeQuery nq = session.createSQLQuery(sql);
        List<Object[]> list = nq.list();
        for(Object [] student : list) {
            String std = Arrays.toString(student);
            System.out.println(std);
        }

        session.close();
        factory.close();
    }
}

```

Cascading

Cascading in Hibernate means when you perform an action (like save or delete) on a parent object, the same action automatically happens to its related child objects. This helps keep related data in sync without extra coding.

Key Cascade Types:

- **CascadeType.PERSIST**: Automatically saves child entities when the parent is saved.
- **CascadeType.MERGE**: Automatically updates child entities when the parent is updated.
- **CascadeType.REMOVE**: Automatically deletes child entities when the parent is deleted.
- **CascadeType.ALL**: Applies all of the above cascade operations.

<pre> 1 @Entity 2 public class Father { 3 4 @Id 5 @GeneratedValue(strategy = GenerationType.IDENTITY) 6 private int id; 7 private String name; 8 @OneToMany(mappedBy = "father", fetch = FetchType.EAGER, cascade = CascadeType.ALL) 9 private List<Child> childs = new ArrayList<>(); 10 11 public int getId() { </pre>	<pre> 10 11 @Entity 12 public class Child { 13 14 @Id 15 @GeneratedValue(strategy = GenerationType.IDENTITY) 16 private int id; 17 private String name; 18 @ManyToOne 19 private Father father; 20 21 </pre>
---	--

```

public static void main(String[] args) {

    SessionFactory factory = new Configuration().configure("hibernate.cfg.xml").buildSessionFactory();
    Session session = factory.openSession();
    Father f = new Father();
    f.setName("Sanjay");

    Child child1 = new Child();
    child1.setName("Manoj");
    child1.setFather(f);

    Child child2 = new Child();
    child2.setName("Nishita");
    child2.setFather(f);

    f.getChildren().add(child1);
    f.getChildren().add(child2);
    Transaction tr = session.beginTransaction();
    session.save(f);
    tr.commit();
    session.close();
    factory.close();
}

```

Hibernate Caching

What is Caching?

Caching is the process of storing a copy of frequently accessed data in a temporary storage area (cache) to speed up data retrieval and reduce the need for repeated processing or access to slower storage systems, like databases or remote servers.

Hibernate caching is a technique used to temporarily store frequently accessed data in memory, so the application can retrieve it quickly without repeatedly querying the database, improving performance.

There are two types of hibernate caching :

1. First Level Cache
2. Second Level Cache

First Level Cache :

In Hibernate, the **First-Level Cache** is a default mechanism that stores data temporarily in the session. When we fetch an object in a session, Hibernate keeps it in the cache. If we request the same object again in the same session, Hibernate retrieves it from the cache instead of querying the database again.

```

9 public class FirstLevelCache {
10
11     public static void main(String[] args) {
12
13         Configuration cfg = new Configuration();
14         SessionFactory sessionFactory = cfg.configure("hibernate.cfg.xml").buildSessionFactory();
15         Session session = sessionFactory.openSession();
16
17         Student student = session.get(Student.class, 2);
18         System.out.println(student);
19
20         System.out.println("Performed some work.....");
21
22         Student student2 = session.get(Student.class, 2);
23         System.out.println(student2);
24
25         System.out.println(session.contains(student));
26
27         session.close();
28     }
}

```

```

references Answer (aId)
Dec 15, 2024 6:12:48 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator initiateService
INFO: HH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
Hibernate:
select
  student0_.id as id1_7_0_,
  student0_.ceryId as ceryid2_7_0_,
  student0_.ceryName as ceryname3_7_0_,
  student0_.name as name4_7_0_
from
  Student student0_
where
  student0_.id=?
Student [id=2, name=Gaurav, cery=Certificate [ceryId=200, ceryName=Python Certificate]]
Performed some work.....
Student [id=2, name=Gaurav, cery=Certificate [ceryId=200, ceryName=Python Certificate]]
true

```

Second Level Cache

The **Second-Level Cache** in Hibernate is an optional caching mechanism that stores data across multiple sessions. It allows cached data to be shared between sessions, reducing the number of database queries for commonly accessed data.

Steps to implement

1. Search “ehcache maven” and add the dependency in pom.xml

```

<!-- https://mvnrepository.com/artifact/org.ehcache/ehcache -->
<dependency>
    <groupId>org.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>3.8.1</version>
</dependency>

```

2. Search “hibernate cache maven” and add the dependency in pom.xml

```
<!-- https://mvnrepository.com/artifact/org.hibernate
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.4.25.Final</version>
</dependency>
```

3. Add following properties in “hibernate.cfg.xml”

```
<property name="format_sql">true</property>
<property name="cache.use_second_level_cache">true</property>
<property name="cache.region.factory_class">net.sf.ehcache.hibernate.EhCacheRegionFactory</property>
```

4. Make the entity cacheable and define the strategy using anno

```
1 @Cacheable
2 @Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
3 public class Student {
4
5     @Id
```

Example :

```
public static void main(String[] args) {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    System.out.println("Session-1 Begin");
    Student student = session.get(Student.class, 2);
    System.out.println(student);
    System.out.println("Session-1 End");

    session.close();

    Session session2 = sessionFactory.openSession();
    System.out.println("Session-2 Begin");
    Student student2 = session2.get(Student.class, 2);
    System.out.println(student2);
    System.out.println("Session-2 End");
    session2.close();
}
```

Output :

```
Session-1 Begin
Hibernate:
    select
        student0_.id as id1_7_0_,
        student0_.ceryId as ceryid2_7_0_,
        student0_.ceryName as ceryname3_7_0_,
        student0_.name as name4_7_0_
    from
        Student student0_
    where
        student0_.id=?
Student [id=2, name=Gaurav, cery=Certiicate [ceryId=200, ceryName=Python Certificate]]
Session-1 End
Session-2 Begin
Student [id=2, name=Gaurav, cery=Certiicate [ceryId=200, ceryName=Python Certificate]]
Session-2 End
```

As we can see the student object is fetched only once from DB and after that it is fetched from cache even in the different sessions due to second level cache is enabled.

Difference Between First Level Cache and Second Level Cache

First-Level Cache	Second-Level Cache
Works within a single Hibernate session.	Shared across multiple sessions.
It is a session-level cache .	It is a session-factory-level cache .
Enabled by default, no configuration needed.	Must be explicitly enabled and configured.
Data is stored in the session memory.	Data is stored in a shared external cache (e.g., Ehcache, Redis).
Improves performance within a single session.	Boosts performance across multiple sessions.

Entity Configuration Using XML

1. Define the entity which we want to map with DB table

```
2
3 public class Person {
4
5     private int id;
6     private String name;
7     private String add;
8
9     public int getId() {
```

2. Create one xml file for configuration with following naming convention
“<class_name>.hbm.xml” and do the following configurations add the dtd ass well

```
hibernate.cfg.xml × Person.java person.hbm.xml × XmlMapping.java
1 <!DOCTYPE hibernate-mapping SYSTEM "/Gaurav/Personal/Learning%20Teach/Hibernate/Hibernate/src/main/resources/dtd/hibernate-mapping-3.0.dtd">
2
3 <hibernate-mapping>
4   <class name="n.mapping.using.xml.Person">
5
6     <id name="id" column="p_id">
7       <generator class="native"></generator>
8     </id>
9     <property name="name" column="p_name" type="string"></property>
10    <property name="add" column="p_add" type="string"></property>
11  </class>
12 </hibernate-mapping>
```


3. Map the file in "hibernate.cfg.xml" file

```
<mapping class="f.manytomany.mapping.Project" />

<mapping resource="n/mapping/using/xml/person.hbm.xml" />
```

4. Save the entity in DB

```
public class XmlMapping {

    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration().configure("hibernate.cfg.xml").buildSessionFactory();
        Session session = sessionFactory.openSession();

        Transaction tr = session.beginTransaction();
        Person p = new Person();
        p.setId(1);
        p.setName("Gaurav");
        p.setAdd("Test Add");
        session.save(p);

        tr.commit();
        session.close();
    }
}
```

Hibernate Criteria API

Hibernate Criteria is a feature in Hibernate that allows us to create database queries dynamically using Java code instead of writing SQL or HQL. It uses an object-oriented approach to make querying more flexible and easier to manage.

For example, instead of writing a query like "SELECT * FROM Employee WHERE department = 'IT'", you can use Java methods to achieve the same result programmatically.

Example:

java

Copy code

```
Session session = sessionFactory.openSession();
Criteria criteria = session.createCriteria(Employee.class);
criteria.add(Restrictions.eq("department", "IT"));
List<Employee> employees = criteria.list();
```

This makes it useful for dynamic and complex queries but has been replaced by the JPA Criteria API in modern versions of Hibernate.

```
public static void main(String[] args) {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    Criteria c = session.createCriteria(Student.class);

    // get student with certificate is more than 200
    c.add(Restrictions.gt("cery.ceryId", 200));

    //get student which have "gau" in there name
    c.add(Restrictions.like("name", "%gau%"));

    List<Student> list = c.list();
    for(Student st : list) {
        System.out.println(st);
    }
}
```