|  |  |
| --- | --- |
| 1. Why Hibernate: Do not need to interact directly with jdbc and with the help of Hibernate we can map class and its object to the database table and columns. 2. What is Hibernate: ORM Framework or toll for java for persisting the data into database? 3. Implementation 4. Advanced Concept   **Idea:** Let’s say we have one class Student having three primitive variables (values) [rollNo, name, marks]. So when we create the object of this class then this object will contain these entire three variables inside the object and this object having all variable values will be saved in the database with the help of Hibernate just using save() and get() method. Now if don’t use the hibernate then we will have to do several steps in case of JDBC.   1. Create connection to the database in java 2. Write the jdbc sql query for insert, update, select and delete data from the database 3. Write the prepared statement 4. Will have to set values into the   In case of Hibernate the class name represents the table name and the variable inside the class represents the column name and the object (having variables values) that we are saving using the save () method will be nothing the row data of the table.  So with the help of hibernate using the class object we can directly save the data into database using the save() method.  So the save the object with the help of hibernate we will have to follow few steps given below.   1. Create the SessionFactory and pass the hibernate configuration detail having information about the database like – username, password, url, driver name database dialect etc. 2. Create the Session object using the SessionFactory object 3. Now using the Session object save the object using the save () method and get() to fetch the data from the database. | |
| To Implement the hibernate:   1. Create a maven project 2. Add the hibernate and database (like MySQL) dependencies in the POM.xml 3. Create a hibernate configuration xml file or mention the database detail like driver\_name, username, password, dialect name and hbm2ddl (create or update) in the application.properties file 4. Create a Entity class and variable inside the class, annotate the class name with @Entity or @Table and @Id on variable which will be PK the table and @Column(name=”Col\_name”)   **POM.XML**  <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>4.1.6.Final</version>  </dependency>    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>8.0.15</version>  </dependency> | |
| **hibernate.cfg.xml:** We can get this configuration file automatically after installing plugin in the eclipse.  <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <property name=*"hibernate.connection.****driver****\_****class****"*>com.mysql.jdbc.Driver</property>  <property name=*"hibernate.connection.****password****"*>Admin@123</property>  <property name=*"hibernate.connection.****url****"*>jdbc:mysql://localhost:3306/neon</property>  <property name=*"hibernate.connection.****username****"*>root</property>  <property name=*"hibernate.****dialect****"*>org.hibernate.dialect.MySQLDialect</property>  <property name=*"****hbm2ddl.auto****"*>update</property>  </session-factory>  </hibernate-configuration>  **Note:** We use this configuration file to configure driver-class name, url, username, password,MySql dialect and hbm2ddl.auto = update>  **Note:** In case of update if table is present in the db then it will create the table and if table is present then it does not create new table and just update the record.  But in case of “create” every time it will create a new table with the new record that is why mostly we use update instead of create. | |
| **Entity class:**  **import** javax.persistence.Entity;  **import** javax.persistence.Id;  @Entity  **public** **class** Alean {    @Id  **private** **int** id;  **private** String aname;  **private** String color;  Note: Here table will be created with the name of Alean PK id and all other variable would be treated as columns.  Similarly instead of default name created with variable name we can specify our own column name as given below  @Column(name="alean\_color")  **private** String color;  Now column name will be created with the name annotated with **@Column** | Note: Let’s say if want to specify the particular name of table instead of the default table name created with the name of Class name then we have to specify the table as given below:  @Entity(name= "My\_Alean")  **public** **class** Alean {}  **Note: Another way:**  @Table(name="Alean\_Table")  @Entity(name= "My\_Alean")  **public** **class** Alean {…}  **Note:**  In this case table will be created annotated with @Table and the table name annotated with @Entity will not be created: |
| **package** com.telesco.demo.hibernate;  **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 Hibernate" );  // Step- 1 Creating object of entity class and setting its all three properties  Alean alean = **new** Alean();    alean.setId(103);  alean.setAname("Shilpi");  alean.setColor("Blue");  **// In case of JDBC we will have write all seven steps like database connection and insert query to insert the above object's data into DB But in case of hibernate we have to configure everything in hibernate.cfg.xml file**  **// Step2- Create the Configuration object by passing the hibernate config xml and Entity class**    Configuration con = **new** Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Alean.**class**);  **// Step3- Using the Configuration object create the SessionFactory(I) object**  SessionFactory sf = con.~~buildSessionFactory~~();  // Step4- Using SF object create the Session(I) object  Session session = sf.openSession();    **//Step5- Begin the transaction suing the Session object just to Follow the ACID property so that every operation should be the part of transaction and save() operation will be start after transaction start**  Transaction tx = session.beginTransaction();  **// Step6- save the object into database using save () method.**  session.save(alean);  // Step7- Finally to persist the data into database commit the transaction.  tx.commit();  }} | |

**Add Property to create the table and to show the sql query in the log:**

|  |  |
| --- | --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <property name=*"hibernate.connection.driver\_class"*>com.mysql.jdbc.Driver</property>  <property name=*"hibernate.connection.password"*>Admin@123</property>  <property name=*"hibernate.connection.url"*>jdbc:mysql://localhost:3306/neon</property>  <property name=*"hibernate.connection.username"*>root</property>  <property name=*"hibernate.dialect"*>org.hibernate.dialect.MySQLDialect</property>  <property name=*"hbm2ddl.auto"*>create</property> **// To create the table in db**  <property name=*"show\_sql"*>true</property> **// To show the sql query in log**  </session-factory>  </hibernate-configuration> | |
|  |  |

**Note:** So in case “create” every time it will drop the table and will create new table with the latest record. And if we use update instead of create then it will not drop the table and just update the record in the same table. And as we have added the property to show sql =true so in the log we will see the sql query generated by hibernates:

|  |
| --- |
| INFO: HHH000227: Running hbm2ddl schema export  Hibernate: drop table if exists Alean  Hibernate: create table Alean (id integer not null, aname varchar(255), color varchar(255), primary key (id))  Jun 23, 2021 9:56:43 PM org.hibernate.tool.hbm2ddl.SchemaExport execute  INFO: HHH000230: Schema export complete  Hibernate: insert into Alean (aname, color, id) values (?,?,?)  **Note: In case of update:**  INFO: HHH000232: Schema update complete  Hibernate: insert into Alean\_Table (aname, alean\_color, id) values (?, ?, ?) |

**@Transient:** If we don’t want any variable of the class to be the part of table then in that case just annotate that particular variable with @Transient and then in the table that particular variable will not be created as column in the table:

|  |  |
| --- | --- |
| @Table(name="Alean\_Table")  @Entity(name= "My\_Alean")  **public** **class** Alean {  @Id  **private** **int** id;    @Transient  **private** String aname;  @Column(name="alean\_color")  **private** String color; | So it the **aname** will be the part of table and the hibernate will drop the existing table and will create a new able with two columns name only as we can see the same in the log below. |
| Log:  INFO: HHH000227: Running hbm2ddl schema export  Hibernate: drop table if exists Alean\_Table  Hibernate: create table Alean\_Table (id integer not null, alean\_color varchar(255), primary key (id))  Jun 23, 2021 10:31:56 PM org.hibernate.tool.hbm2ddl.SchemaExport execute  INFO: HHH000230: Schema export complete  Hibernate: insert into Alean\_Table (alean\_color, id) values (?, ?) | |

So far we have talked about the below annotation

|  |  |
| --- | --- |
| 1. @Entity 2. @Table 3. @Id 4. @Column 5. @Transient | [org](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate).[cfg](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate.cfg).Configuration Class  [org](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate).SessionFactory – Interface  [org](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate).Session- Interface  [org](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate).Transaction – Interface  Need to define the above class and interfaces in simple language: |

In the above discussion we have seen that how we can create table and save the data in the same table. Now let’s see how to fetch data from the table.

|  |
| --- |
| **Final Conclusion with all the steps for hibernate project:**  To Implement the hibernate:   1. Create a maven project 2. Add the hibernate and database (like MySQL) dependencies in the POM.xml 3. Create a hibernate configuration xml file or mention the database detail like driver\_name, username, password, dialect name and hbm2ddl (create or update) in the application.properties file 4. Create a Entity class and variable inside the class, annotate the class name with @Entity or @Table and @Id on variable which will be PK the table and @Column(name=”Col\_name”)   Now in the main()  Step1- Creating object of entity class and setting its all three properties  Step2- Create the Configuration object by passing the hibernate config xml and Entity class  Step3- Using the Configuration object create the SessionFactory(I) object  Step4- Using SF object create or Open the Session(I) object for the save() operation  Step5- Begin the transaction using the Session object just to Follow the ACID property so that every operation should be the part of transaction and save () operation will be start after transaction start. Because whenever we save the data into database then we have to follow the ACID properties(Atomicity- > Consistency->Isolation -> Durability) and to maintain that we have to be assure that all these changes should be the part of transaction and for this we begin the transaction.  Step6- save the object into database using save () method during the transaction lifecycle  Step7- Finally to persist the data into database commit the transaction |
| **Note:** As we have seen above in the code that buildSessionFactory() seems to be deprecated method so to get the latest active method we have to add one line of code which will build a ServiceRegistery for all the configuration added in the Configuration con. And then pass this ServiceRegistery object into buildSessionFactory (registry) and then this deprecated method will become active. So the modified line of code is given below. |
| Configuration **con** = **new** Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Alean.**class**);  **ServiceRegistry regestry =new ServiceRegistryBuilder().applySettings(con.getProperties()).buildServiceRegistry();**  SessionFactory sf = con.buildSessionFactory(regestry);  Session session = sf.openSession();  Transaction tx = session.beginTransaction();  session.save(alean);  tx.commit(); |

**Fetching Data from the DB Using Hibernate:**

In the last section we have saved the data into database using hibernate. Now in this section we will fetch the data from the database.

|  |
| --- |
| **package** com.telesco.demo.hibernate;  **import** org.hibernate.Session;  **import** org.hibernate.SessionFactory;  **import** org.hibernate.Transaction;  **import** org.hibernate.cfg.Configuration;  **import** org.hibernate.service.ServiceRegistry;  **import** org.hibernate.service.ServiceRegistryBuilder;  **public** **class** FetchAleanData {  **public** **static** **void** main(String[] args) {    Alean alean = **new** Alean();  // Configure the Hibernate cfg xml file and Entry class  System.***out***.println("Step- Configure the Hibernate cfg xml file and Entry class ");  Configuration configure = **new** Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Alean.**class**);  System.***out***.println("Step2- Create ServiceRegistry to register the configuration");  ServiceRegistry registry = **new** ServiceRegistryBuilder().applySettings(configure.getProperties()).buildServiceRegistry();  System.***out***.println("Step3- Creating the SessionFactory using configure object and by passing SR Object");  SessionFactory sf = configure.buildSessionFactory(registry);  System.***out***.println("Step4- Now open or create the hibernate session using session factry object");  Session session = sf.openSession();  System.***out***.println("Step5- Now Begin the transaction to maintain ACID property ");  Transaction tx = session.beginTransaction();  System.***out***.println("Step6- Start operation of fetching data from DB ");  Object object = session.get(Alean.**class**, 101);  System.***out***.println(object);  alean = (Alean)session.get(Alean.**class**, 104);// it will fetch only one record based on PK 104  System.***out***.println(alean);  System.***out***.println("Step7- Commit the transaction");  }  } // Output: |
| Step1- Configure the Hibernate cfg xml file and Entry class  Step2- Create ServiceRegistry to register the configuration  Step3- Creating the SessionFactory using configure object and by passing SR Object  Step4- Now open or create the hibernate session using SessionFactory object  Step5- Now Begin the transaction to maintain ACID property using SessionFactory object  Step6- Start opearation of fetching data from DB  Hibernate: select alean0\_.id as id0\_0\_, alean0\_.aname as aname0\_0\_, alean0\_.alean\_color as alean3\_0\_0\_ from Alean\_Table alean0\_ where alean0\_.id=?  Alean [id=101, aname=Shilpi, color=Blue]  Hibernate: select alean0\_.id as id0\_0\_, alean0\_.aname as aname0\_0\_, alean0\_.alean\_color as alean3\_0\_0\_ from Alean\_Table alean0\_ where alean0\_.id=?  Alean [id=104, aname=Bunty, color=Blue]  Step7- Commit the transaction |

So with the help of get() we can fetch the record:

**How to Use Embeddable Object OR Feilds in Hibernate:**

Discussion:

As in the above Entity class we have only one variable or fields aname (alean name). Now let’s say instead of having one field for name if we have three fields [fName, mName, lName] to be embedded in the same Alean\_Table.

So in this case we have to follow the given steps given below.

1. Create a class with name having all three fields [fName, mName, lName]
2. Create the reference object of this class into main Entity class and set the [fName, mName, lName] here.

Ques: So the question comes now: Earlier we have all the variable of the primitive type(String) to be saved in DB but now we have created one more class and setting this class object into Entity class object and finally passing this object to save () method. So will it create these three fields in the same existing table or will it create two table and save other three fields also in the DB or not?

|  |  |
| --- | --- |
| **public** **class** AleanFullName {  **private** String fName;  **private** String mName;  **private** String lName;  }  **import** javax.persistence.Embeddable;  **@Embeddable**  **public** **class** AleanFullName {  **private** String fName;  **private** String mName;  **private** String lName;  } | @Entity  @Table(name="Alean\_Table")  **public** **class** Alean {  @Id  **private** **int** id;  **private** String aname;    @Column(name="alean\_color")  **private** String color;  **private** AleanFullName aleanFullName; |
| **public** **class** App  {  **public** **static** **void** main( String[] args )  {  System.***out***.println( "Hello Hibernate" );    // Step- 1 Creating object and setting its all three properties    AleanFullName fullName = **new** AleanFullName();  fullName.setfName("Arun");  fullName.setmName("Kumar");  fullName.setlName("Gupta");    Alean alean = **new** Alean();  alean.setId(105);  alean.setAname("Junty");  alean.setColor("Black");  alean.setAleanFullName(fullName);  // Hibernate remaining codes will be written below  } | Now if run this program then let’s see what will happen:  Exception in thread "main" org.hibernate.MappingException: Could not determine type for: com.telesco.demo.hibernate.AleanFullName, at table: Alean\_Table, for columns: [org.hibernate.mapping.Column(aleanFullName)]  So here hibernate got confused that from where it came and what to do with this object aleanFullName  Solution: In this case we will have to annotate the AleanFullName class with **@Embeddable** as shown in the second part above |

Now after mentioning new class as @Embeddable let’s run and see:

INFO: HHH000232: Schema update complete

Hibernate: insert into Alean\_Table (fName, **lName, mName, aname,** alean\_color, id) values (?,?,?, ?, ?, ?)

So now the three fields now are getting embedded into existing table as we can see the above log. Now let’s see table structure:

|  |  |
| --- | --- |
|  | So @Embeddable just embed the fields into existing table instead of creating new table. But if we put @Entity instead of @Embeddable then we will get a complete new table. |

|  |  |
| --- | --- |
| 1. @Entity 2. @Table 3. @Id 4. @Column 5. @Transient 6. **@Embeddable** | [org](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate).[cfg](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate.cfg).Configuration Class  [org](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate).SessionFactory – Interface  [org](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate).Session- Interface  [org](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=DemoHibernate/C:%5C/Users%5C/saraswati%5C/.m2%5C/repository%5C/org%5C/hibernate%5C/hibernate-core%5C/4.1.6.Final%5C/hibernate-core-4.1.6.Final.jar%3Corg.hibernate).Transaction – Interface  Need to define the above class and interfaces in simple language: |

**Mapping Relation- Theory:**

As in the database if we have several tables to save data or fetch data then in that case we will have several kind of mapping relations.

1. One-to-one
2. One-to-many
3. Many-to-one
4. Many-to-many

For example let’s say we have two table [ Employee\_Table and Department\_Table] then here we could have several relation

1. One department can have multiple employees (one-to-many)
2. Many employee works for same department ( many-to-one)
3. One Student could have one laptop ( one-to-one)
4. Let’s say Student goes to library then here in this case multiple books can be read by multiple employee or multiple employee can have multiple books (Many-to-many)

So now same thing let’s implement in the hibernate.

**One-To-One mapping Example:**

|  |  |
| --- | --- |
| @Entity  @Table(name="Laptop\_Table")  **public** **class** Laptop {  @Id  **private** **int** lid;    @Column(name="laptop\_name")  **private** String name;  } | @Entity  @Table(name="Student\_Table")  **public** **class** Student {  @Id  **private** **int** rollNo;  @Column(name="student\_name")  **private** String name;    @Column(name="student\_marks")  **private** **int** marks;    @OneToOne  **private** Laptop lap;  } |

1. Created one Laptop class
2. Created on Student class
3. Now let’s say we want each student should have one laptop then, we will create one Laptop class reference inside Student class and annotate that Laptop reference with @OneToOne as show above.
4. In that we will get two table and in the Student table the laptop id will become FK

So this about OneToOne mapping of two entity.

|  |  |
| --- | --- |
| **public** **class** OneToOneMappingAppTest {  **public** **static** **void** main(String[] args) {  // Create the Laptop object and set the values  Laptop lap = **new** Laptop();  lap.setLid(1002);  lap.setName("Lenevo");    // Create the Student class object and set the values  Student stud= **new** Student();  stud.setRollNo(2);  stud.setMarks(50);  stud.setName("Tarun");    stud.setLap(lap);    // Create the Configuration Interface and pass the hibernate cfg and both the entity class  Configuration config = **new** Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Laptop.**class**).addAnnotatedClass(Student.**class**);  ServiceRegistry registry = **new** ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();  SessionFactory sf= config.buildSessionFactory(registry);  Session session = sf.openSession();  session.beginTransaction();  session.save(lap);  session.save(stud);  session.getTransaction().commit();  }  } | |
| INFO: HHH000232: Schema update complete  Hibernate: insert into Laptop\_Table (laptop\_name, lid) values (?, ?)  Hibernate: insert into Student\_Table (lap\_lid, student\_marks, student\_name, rollNo) values (?, ?, ?, ?) | |
| Laptop Table | Student\_Table |

So we can see that here we have create one to one mapping in the Student class for Laptop and hence One student would have one laptop and that is why lap\_lid got created in student\_table.

**One-To-Many Mapping Example:**

Let’s say one student has many laptops so this will be the relational mapping – OneToMany. In this case will follow as:

1. Create a List type reference variable of Laptop class instead of creating simple class reference variable
2. Annotate this List of Laptop type reference variable with @OneToMany annotation.

So the above defined student class will look as given below.

|  |  |
| --- | --- |
| @Entity  @Table(name="Laptop\_Table")  **public** **class** Laptop {  @Id  **private** **int** lid;    @Column(name="laptop\_name")  **private** String name;  } | @Entity  @Table(name="Student\_Table")  **public** **class** Student {    @Id  **private** **int** rollNo;    @Column(name="student\_name")  **private** String name;    @Column(name="student\_marks")  **private** **int** marks;    @OneToMany  **private** List<Laptop> laptop;  } |

In this case three tables will be generated.

1. Laptop\_Table having lid as PK
2. Student\_Table having rollNo as PK
3. Third Table with the name Student\_Laptop having lid and rollNo columns where one student could be mapped to many laptops

But we can avoid creating the third table. So to avoid creating third table and to achieve the above OneToMany relational mapping we will have to follow the below steps.

1. Create one Student Class reference inside Laptop class
2. Annotate this class reference variable with @ManyToOne annotation.

So the above entity classes will look like as below.

|  |  |
| --- | --- |
| @Entity  @Table(name="Laptop\_Table")  **public** **class** Laptop {  @Id  **private** **int** lid;    @Column(name="laptop\_name")  **private** String name;    @ManyToOne  **private** Student student;  } | @Entity  @Table(name="Student\_Table")  **public** **class** Student {    @Id  **private** **int** rollNo;    @Column(name="student\_name")  **private** String name;    @Column(name="student\_marks")  **private** **int** marks;    @OneToMany  **private** List<Laptop> laptop;  } |

Now in this case we will have only two tables with one extra column (Student rollNo) in the Laptop Column. And now we meet the objective where one student could have many laptops because now rollNo will be treated as FK and lid will be treated as PK

**But please note here:**

As in the Student Class we have List of Laptop variables so this class will think that it is Student Entity class responsibility to create one table with extra laptop lid column.

Similarly in the Laptop class we have Student class reference variable, so this class will think it is Laptop Entity class responsibility to create table having extra column of roll number in the Laptop\_Table

So we have to mention in the Student class explicitly that it is the Laptop class responsibility to create the table and for that we have to mention like as below in the Student class

|  |  |
| --- | --- |
| @OneToMany(mappedBy="student")  **private** List<Laptop> laptop; | Now once we mention this (mappedBy="student") then now Student class will say I am doing OneToMany but mapping will be done by Laptop class (Laptop table) for each student. |

Now in this case will be having two tables only.

1. Student\_Table with no effect
2. Laptop\_Table with one extra column (rollNo) as FK

|  |  |
| --- | --- |
| **public** **class** OneToManyMappintTest {  **public** **static** **void** main(String[] args) {    // Set the laptop values  Laptop laptop = **new** Laptop();  Student student = **new** Student();    laptop.setLid(1004);  laptop.setName("HP");  laptop.setStudent(student);    student.setRollNo(4);  student.setName("Shilpi");  student.setMarks(100);  student.getLaptop().add(laptop);    Configuration config = **new** Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Laptop.**class**).addAnnotatedClass(Student.**class**);  ServiceRegistry registry= **new** ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();  SessionFactory sf = config.buildSessionFactory(registry);  Session session = sf.openSession();  session.beginTransaction();  session.save(laptop);  session.save(student);  session.getTransaction().commit();  }  } | |
| **Laptop Table** | **Student Table:** |

**Many-To-Many Mapping Example:**

|  |  |
| --- | --- |
| @Entity  @Table(name="Laptop\_Table")  **public** **class** Laptop {  @Id  **private** **int** lid;    @Column(name="laptop\_name")  **private** String name;    @ManyToMany  **private** List<Student> student;  } | @Entity  @Table(name="Student\_Table")  **public** **class** Student {    @Id  **private** **int** rollNo;    @Column(name="student\_name")  **private** String name;    @Column(name="student\_marks")  **private** **int** marks;    @ManyToMany (mappedBy="student")  **private** List<Laptop> laptop;  } |

In ManyToMany cases we just need to make one changes in both the classes.

1. Create the List of Student in the Laptop class and annotate it with @ManyToMany
2. Create the list of Laptop in the Student class and annotate it with @ManyToMany (mappedBy="student") and mention mapped by student.

|  |  |
| --- | --- |
| **public** **class** ManyToManyMappingTest {  **public** **static** **void** main(String[] args) {  Laptop laptop = **new** Laptop();  Student student = **new** Student();  laptop.setLid(1005);  laptop.setName("MAcbBook");  student.setRollNo(10);  student.setName("AAA");  student.setMarks(100);  **if**(laptop!=**null**)  student.getLaptop().add(laptop);  **if**(student!=**null**)  laptop.getStudent().add(student);  Configuration config = **new** Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Laptop.**class**).addAnnotatedClass(Student.**class**);  ServiceRegistry registry= **new** ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();  SessionFactory sf = config.buildSessionFactory(registry);  Session session = sf.openSession();  session.beginTransaction();  session.save(student);  session.save(laptop);  session.getTransaction().commit();  }  }  select \* from laptop\_table;  select \* from student\_table;  select \* from Laptop\_Table\_Student\_Table;  So here we will get one new table having lid and rollNo: | |
|  |  |
|  |  |

**Lazy and Eager fetch in hibernate:**

In this section will talk about different type of fetch of data from the database in the hibernate.

1. Lazy Fetch
2. Eager Fetch

|  |  |
| --- | --- |
| @Entity  @Table(name="Laptop")  **public** **class** Laptop {  @Id  @Column(name="laptop\_id")  **private** **int** lid;    @Column(name="laptop\_brand")  **private** String brand;  @Column(name="laptop\_price")  **private** **int** price;    // Multiple laptop will be mapped to one employee so will create one Employee Reference and will annotate it with ManyToOne    @ManyToOne  **private** Employee employee; | @Entity  @Table(name="Employee")  **public** **class** Employee {    @Id  @Column(name="emp\_id")  **private** **int** eid;    @Column(name="emp\_name")  **private** String eName;    // One employee can have multiple laptop so will create one relational mapping OneToMany on Laptop and will create // a List of laptop and mappling will be done by Laptop entity that is why mentioned mappedBy="employee"  @OneToMany( mappedBy="employee")  **private** List<Laptop> laptop = **new** ArrayList<Laptop>(); |

1. Created one Employee Entity where one employee can have multiple laptop so created OneToMany on the List of Laptop object
2. Created One Laptop entity where multiple laptop will be mapped to one employee so created ManyToOne on employee reference

|  |  |
| --- | --- |
| **// Saving the Entity in the DB**  **public** **class** LazyEagerFetchTest {  **public** **static** **void** main(String[] args) {    Laptop laptop1 = **new** Laptop();  laptop1.setLid(2001);  laptop1.setBrand("HP");  laptop1.setPrice(50000);    Laptop laptop2 = **new** Laptop();  laptop2.setLid(2002);  laptop2.setBrand("Lenevo");  laptop2.setPrice(55000);    Laptop laptop3 = **new** Laptop();  laptop3.setLid(2003);  laptop3.setBrand("Dell");  laptop3.setPrice(60000);    Employee employee = **new** Employee();  employee.setEid(1001);  employee.seteName("Arun");  employee.getLaptop().add(laptop1)  employee.getLaptop().add(laptop2);  employee.getLaptop().add(laptop3);        Configuration config = **new** Configuration().configure("hibernate\_office.cfg.xml").addAnnotatedClass(Laptop.**class**).addAnnotatedClass(Employee.**class**);  ServiceRegistry registry = **new** ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();  SessionFactory sf = config.buildSessionFactory(registry);  Session session = sf.openSession();    session.beginTransaction();    session.save(laptop1);  session.save(laptop2);  session.save(laptop3);  session.save(employee);    session.getTransaction().commit();  }  } | **// Lazy Fetch**  **public** **class** FetchDataFromDB {  **public** **static** **void** main(String[] args) {    **Configuration config = new Configuration().configure("hibernate\_office.cfg.xml").addAnnotatedClass(Laptop.class).addAnnotatedClass(Employee.class);**  **ServiceRegistry registry = new ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();**  SessionFactory sf = config.buildSessionFactory(registry);  Session session = sf.openSession();  session.beginTransaction();  Employee emp = (Employee)session.get(Employee.**class**, 1002);  System.***out***.println(emp.geteName());  // List<Laptop> laptops= emp.getLaptop();  //  // for(Laptop l : laptops) {  // System.out.println(l);  // }  }  }  INFO: HHH000232: Schema update complete  Hibernate: select employee0\_.emp\_id as emp1\_1\_0\_, employee0\_.emp\_name as emp2\_1\_0\_ from Employee employee0\_ where employee0\_.emp\_id=?  Tarun  Here we have fetched the employee detail which contains laptop detail also but it provides the information from employee table only because we have fetched the employee name ((emp.geteName()) only and it will fire query to fetch name only.  This is called Lazy fetch.  And if required we can fetch the laptop detail mapped to employee (1002) and in this case we will un-comment the commented line.  So in lazy fetch the object will have all the information but it will not fetch the data until we fetch it explicitly.  Note: Now let’s uncomment the above commented lines and we will observe that it will fire the query to fetch the laptop record also.  INFO: HHH000232: Schema update complete  Hibernate: select employee0\_.emp\_id as emp1\_1\_0\_, employee0\_.emp\_name as emp2\_1\_0\_ from Employee employee0\_ where employee0\_.emp\_id=?  Tarun  Hibernate: select laptop0\_.employee\_emp\_id as employee4\_1\_1\_, laptop0\_.laptop\_id as laptop1\_1\_, laptop0\_.laptop\_id as laptop1\_0\_0\_, laptop0\_.laptop\_brand as laptop2\_0\_0\_, laptop0\_.employee\_emp\_id as employee4\_0\_0\_, laptop0\_.laptop\_price as laptop3\_0\_0\_ from Laptop laptop0\_ where laptop0\_.employee\_emp\_id=?  Laptop [lid=2003, brand=Dell, price=60000]  So we can see that now we have got two queries one for fetching name and another for fetching laptop detail.  Conclusion: So in the lazy fetch it will not fire the query until we explicitly fetching the records from the object. |

**Eager Fetch:**

So to make eager fetch we will have to mention in the Employee entity class as given below.

@OneToMany (mappedBy="employee”, **fetch=FetchType**.***EAGER***)

**private** List<Laptop> laptop = **new** ArrayList<Laptop> ();

|  |
| --- |
| INFO: HHH000232: Schema update complete  Hibernate: select employee0\_.emp\_id as emp1\_1\_1\_, employee0\_.emp\_name as emp2\_1\_1\_, laptop1\_.employee\_emp\_id as employee4\_1\_3\_, laptop1\_.laptop\_id as laptop1\_3\_, laptop1\_.laptop\_id as laptop1\_0\_0\_, laptop1\_.laptop\_brand as laptop2\_0\_0\_, laptop1\_.employee\_emp\_id as employee4\_0\_0\_, laptop1\_.laptop\_price as laptop3\_0\_0\_ from Employee employee0\_ left outer join Laptop laptop1\_ on employee0\_.emp\_id=laptop1\_.employee\_emp\_id where employee0\_.emp\_id=?  Tarun |

Now in this case even if we have comment to the line which are fetching the laptop detail, it is firing the query to fetch laptop detail also using join as we can see above and we are getting only one query while in case of the Lazy fetch we are getting two query [one for fetching employee name and another for fetching laptop detail]

**Note: By default fetch type is lazy.**

**Hibernate Caching: First Level-**

In the previous section we have seen one of the biggest advantages which are nothing but Object Relational Mapping, and we don’t need to write boiler code like database connection and sql query to save and fetch the data from the database.

Caching is another most imp feature of the hibernate. So whenever we are hitting database multiple time to fetch the same, so instead of hitting database every time we can use cache and can fetch the data from cache.

Let’s say we have one client – Server and Database, So whenever client send the request to application server then the same request goes to database server. And in response application server fetch the data from the database server and further this data will be send to client application in that particular session and transaction created by hibernate.

Now let’s say the same user again in the same session send the request to server to fetch same data then again the same request will hit the database.

So don’t we think hitting the database every time for same record is time wasting? And to save time for this case very first time when application server fetch the record from database then it should be stored somewhere (cache memory) in the application server itself.

Now again when client send the request for same record to the application server then in this case application server instead of hitting the database it will provide the record from the cache. This is provided by hibernate and this caching mechanism is called first level of cache and this caching will be available for a particular session of the user and it by-default provided by hibernate.

Let’s say if the same user going for the second session or we have another user who is going to fetch same record from the database then we will have two different user session and for these two differ session we will have two different first level cache. Means for the second session or for the second user the data will not be available in the first level cache and then again it will hit the database.

So the advantage of the first level cache is only for one session and for the same request. But if we have two different sessions then hibernate says you cannot use the same cache for two different sessions even-though you are requesting for same record.

**To resolve this problem there we have second level cache.**

**Second Level Cache:**

In the second level cache once the records store in the cache it will be available for the entire user and for all different sessions.

This second level cache does not provided by hibernate by-default and to get second level cache we have do following configurations.

1. We have to use third party library like [EH-cache, OS-Cache, SWAM] but mostly EH-Cache is used. And for getting this library we have mention dependency in the POM.xml.
2. Second one- we have mention the dependency [hibernate-ehcache] to integrate EH-Cache with the hibernate.
3. The third one is we have to enable the second-level cache in the hibernate.cfg.xml file and to do that we have to add few properties. Here in this file we have to add one property for EH-cache and another property is to provide permission for EH-Cache
4. The next one is we have to change our entity, because by-default every entity is not cacheable. So we have to provide two annotation for that [1- @Cacheable – making the entity cacheable] [2- @Cache- Define the strategies like read-only or read-write]

Example:

|  |
| --- |
| Output of the below First Level cache :  INFO: HHH000232: Schema update complete  Hibernate: select employee0\_.emp\_id as emp1\_1\_0\_, employee0\_.emp\_name as emp2\_1\_0\_ from Employee employee0\_ where employee0\_.emp\_id=?  Employee [eid=1001, eName=Arun]  Employee [eid=1001, eName=Arun]  Requesting same request in session - 2  Hibernate: select employee0\_.emp\_id as emp1\_1\_0\_, employee0\_.emp\_name as emp2\_1\_0\_ from Employee employee0\_ where employee0\_.emp\_id=?  Employee [eid=1001, eName=Arun]  Note: As we can see that in the session1 same request was being requested two times and hence hibernate fired query only one time and we got two result for two request. The reason behind is that for the second request hibernate first checks whether the record is available in the First Level cache and that is why for the second request it did not send any request to database server and thus we did not get any query for second request. But when we fired the same request in the another session then hibernate fetch the record from the database server and does not look in the first level cache created by session1  So First level works only for same session and for same request. |
| **public** **class** FirstAndSecondLevelCache {  **public** **static** **void** main(String[] args) {  Employee emp=**null**;  Configuration config = **new** Configuration().configure("hibernate\_office.cfg.xml").addAnnotatedClass(Laptop.**class**)  .addAnnotatedClass(Employee.**class**);  ServiceRegistry registry = **new** ServiceRegistryBuilder().applySettings(config.getProperties())  .buildServiceRegistry();  SessionFactory sf = config.buildSessionFactory(registry);  // Session1  Session session1 = sf.openSession();  session1.beginTransaction();  emp = (Employee) session1.get(Employee.**class**, 1001);  System.***out***.println(emp);  emp = (Employee) session1.get(Employee.**class**, 1001);  System.***out***.println(emp);  session1.getTransaction().commit();    //Session2  Session session2 = sf.openSession();  session1.beginTransaction();  emp = (Employee)session2.get(Employee.**class**, 1001);  session1.getTransaction().commit();  }  } |

**Example of Second Level Cache:**

Second level cache does not provided by-default by hibernate; we will have to configure it. So we will follow the below steps to configure second level cache.

Step1- Add dependency in pom.xml

1. Add net.sf.echcache for getting second level caching feature.
2. Add hibernate-ehcache for integrating ehcache with the ehcache

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/net.sf.ehcache/ehcache -->  <dependency>  <groupId>net.sf.ehcache</groupId>  <artifactId>ehcache</artifactId>  <version>2.10.3</version>  </dependency>    <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-ehcache -->  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-ehcache</artifactId>  <version>4.1.6.Final</version>  </dependency> |

Step2- Add two properties in the hibernate.cfg.xml to enable second level cache.

|  |
| --- |
| <!-- properties for second level cache -->  <property name=*"hibernate.cache.use\_second\_level\_cache"*>true</property>  <!-- to add the third party library ehcache -->  <propertyName =*"hibernate.cache.region.factory\_class"*>org.hibernate.cache.ehcache.EhCacheRegionFactory </property> |

**Step3-** Since by-default no entity classes are cacheable so to make particular entity cacheable let’s add two dependencies at the entity class.

1. @Cacheable: To make the entity class cacheable
2. @Cache(usage=CacheConcurrencyStrategy.***READ\_ONLY)***

So now the entity class will look like:

|  |
| --- |
| @Entity  @Table(name="Employee")  @Cacheable  @Cache(usage=CacheConcurrencyStrategy.***READ\_ONLY***)  **public** **class** Employee {    @Id  @Column(name="emp\_id")  **private** **int** eid;    @Column(name="emp\_name")  **private** String eName;    // One employe can have multiple laptop so will create one relational mapping OneToMany on Laptop and will create  // a List of laptop and mappling will be done by Laptop entity that is why mentioned mappedBy="employee"  @OneToMany( mappedBy="employee",fetch=FetchType.***LAZY***)  **private** List<Laptop> laptop = **new** ArrayList<Laptop>(); |

Now if we run the same above program where we have create session1 and session2 having same request then it will fire the query only one time for two different session also because now we have created second level cache and now the if the same request comes from different user or different session it will hit the database second and in the first response only the response will be cached in the second level cache.

|  |
| --- |
| Output:  INFO: HHH000232: Schema update complete  Hibernate: select employee0\_.emp\_id as emp1\_1\_0\_, employee0\_.emp\_name as emp2\_1\_0\_ from Employee employee0\_ where employee0\_.emp\_id=?  Employee [eid=1001, eName=Arun]  Employee [eid=1001, eName=Arun]  Requesting same request in session - 2  Employee [eid=1001, eName=Arun] |

**Hibernate Caching Level 2 with Query:**

In the earlier section we have used get method to fetch the record and same record got cached in the first and second level cache. Now let’s try to fetch the data using HQL query and see whether records are getting cached or not.

|  |
| --- |
| **public** **class** CacheingQueriedRecord {  **public** **static** **void** main(String[] args) {  Employee emp=**null**;  Configuration config = **new** Configuration().configure("hibernate\_office.cfg.xml").addAnnotatedClass(Laptop.**class**)  .addAnnotatedClass(Employee.**class**);  ServiceRegistry registry = **new** ServiceRegistryBuilder().applySettings(config.getProperties())  .buildServiceRegistry();  SessionFactory sf = config.buildSessionFactory(registry);  // Session1  Session session1 = sf.openSession();  session1.beginTransaction();  Query q1 = session1.createQuery("from Employee where eid=1001");  emp = (Employee)q1.uniqueResult();  System.***out***.println(emp);    emp = (Employee) session1.get(Employee.**class**, 1001);  System.***out***.println(emp);  session1.getTransaction().commit();  session1.close();    System.***out***.println("Requesting same request in session - 2 ");    Session session2 = sf.openSession();  session2.beginTransaction();    Query q2 = session2.createQuery("from Employee where eid=1001");  emp = (Employee)q2.uniqueResult();  System.***out***.println(emp);    session2.getTransaction().commit();  session2.close();  }  } |

Output of above program for caching query

|  |
| --- |
| INFO: HHH000232: Schema update complete  Hibernate: select employee0\_.emp\_id as emp1\_1\_, employee0\_.emp\_name as emp2\_1\_ from Employee employee0\_ where employee0\_.emp\_id=1001  Employee [eid=1001, eName=Arun]  Employee [eid=1001, eName=Arun]  Requesting same request in session - 2  Hibernate: select employee0\_.emp\_id as emp1\_1\_, employee0\_.emp\_name as emp2\_1\_ from Employee employee0\_ where employee0\_.emp\_id=1001  Employee [eid=1001, eName=Arun] |

So we can see that if query to fetch data then the record is not getting cached in second level cache for two different session having same request, because by default second level cache is enabled of get() method only. And for record got from query, we have to add one property in the hibernate configuration file and we have set to true for the query result in the program.

Step1- set the below property in the hibernate.cfg.xml file

|  |
| --- |
| <!-- Query to be cached -->  <property name=*"hibernate.cache.use\_query\_cache"*>true</property> |

Step2- add the below line in the above program to cache the both query.

|  |
| --- |
| // Session1  Session session1 = sf.openSession();  session1.beginTransaction();  Query q1 = session1.createQuery("from Employee where eid=1001");  q1.setCacheable(**true**);  emp = (Employee)q1.uniqueResult();  System.***out***.println(emp);  emp = (Employee) session1.get(Employee.**class**, 1001);  System.***out***.println(emp);  session1.getTransaction().commit();  session1.close();  System.***out***.println("Requesting same request in session - 2 ");  Session session2 = sf.openSession();  session2.beginTransaction();  Query q2 = session2.createQuery("from Employee where eid=1001");  q2.setCacheable(**true**);  emp = (Employee)q2.uniqueResult();  System.***out***.println(emp);  session2.getTransaction().commit();  session2.close(); |

Now if run the above program after adding these configurations then:

|  |
| --- |
| INFO: HHH000232: Schema update complete  Hibernate: select employee0\_.emp\_id as emp1\_1\_, employee0\_.emp\_name as emp2\_1\_ from Employee employee0\_ where employee0\_.emp\_id=1001  Employee [eid=1001, eName=Arun]  Employee [eid=1001, eName=Arun]  Requesting same request in session - 2  Employee [eid=1001, eName=Arun] |

So now for all different session having same request the database query is being generated one time only and from the next time it is getting fetched from second level cache.

**Hibernate Query Language (HQL) - Theory:**

One question arises why to use HQL when we already have method like save, get, and update in the hibernate. When we use:

Session.save (obj) 🡺 it saves the object into database

Session.get(Employee.class, 1001) 🡺 Here we are passing class name and PK so here in this case can get the record based on passed PK, what if we want to get the data from other table or want to get one unique record using different table?

So to solve this problem we need HQL

HQL is exactly same as SQL with little different

|  |  |
| --- | --- |
|  |  |
| * In sql query we used to select column name from the table. * Select \* from student; | * While in HQL we select property from the entity class. * Select from Student [ no \* is being used] |
| Here when we execute query then we get result set and we fetch the data inside while loop | Here we get list of object when fetch the data and using enhanced for loop we can fetch data. |

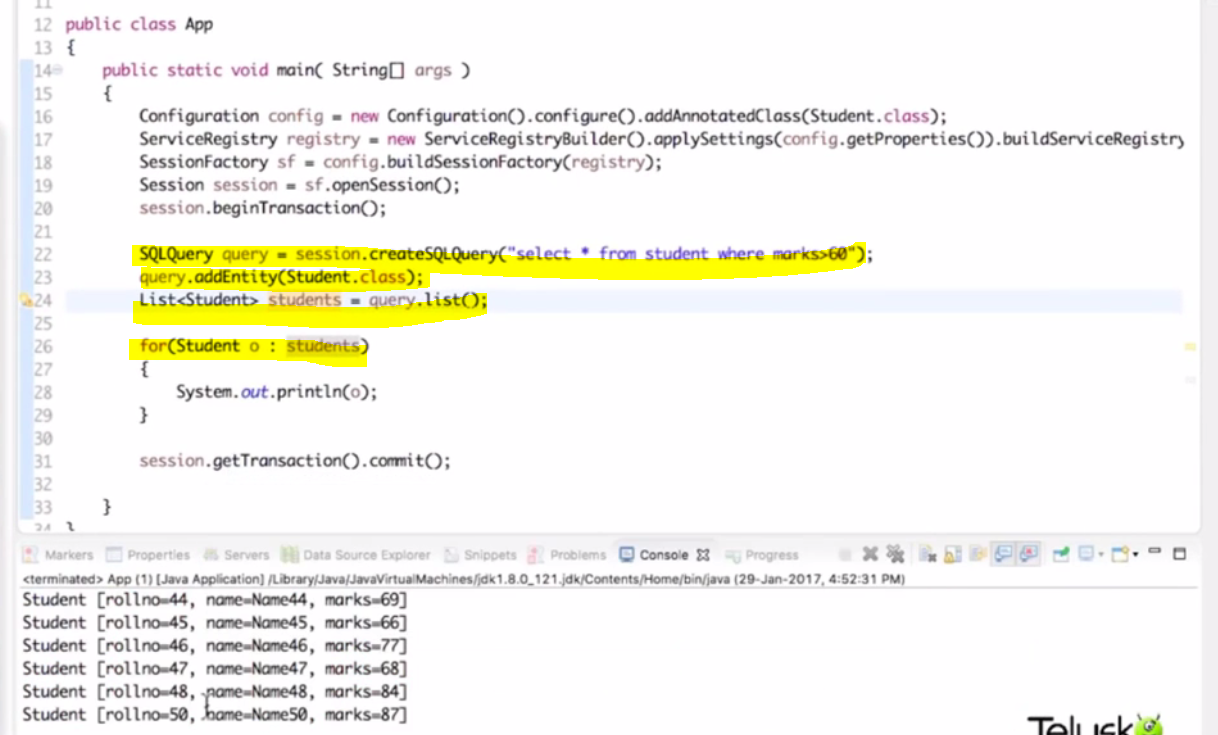
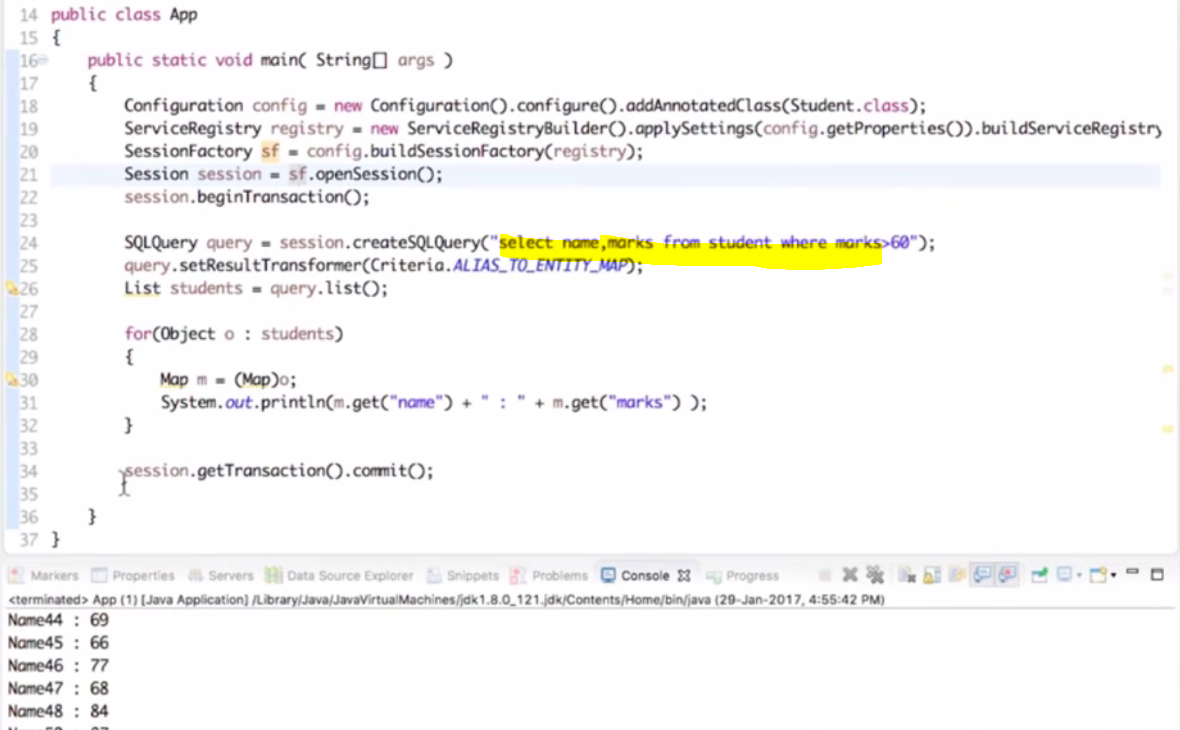
Apart from that we can do all in HQL what we used to do in SQL like join, groupBY, and order by.

Note: We can write SQL query in the hibernate and this called native query.

|  |  |
| --- | --- |
| @Entity  **public** **class** Student {  @Id  **private** **int** rollno;  **private** String name;  **private** **int** marks;    **public** **int** getRollno() {  **return** rollno;  }  **public** **void** setRollno(**int** rollno) {  **this**.rollno = rollno;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getMarks() {  **return** marks;  }  **public** **void** setMarks(**int** marks) {  **this**.marks = marks;  }  @Override  **public** String toString() {  **return** "Student [rollno=" + rollno + ", name=" + name + ", marks=" + marks + "]";  }  } | **public** **class** InsertStudentRecords {  **public** **static** **void** main(String[] args) {    Configuration config = **new** Configuration().configure("hibernate\_office.cfg.xml")  .addAnnotatedClass(Student.**class**);  ServiceRegistry registry = **new** ServiceRegistryBuilder().applySettings(config.getProperties())  .buildServiceRegistry();  SessionFactory sf = config.buildSessionFactory(registry);  Session session = sf.openSession();    session.beginTransaction();  Random r = **new** Random();  **for**(**int** i =1; i<=50; i++) {    Student s = **new** Student();  s.setRollno(i);  s.setName("Name"+i);  s.setMarks(r.nextInt(100));  session.save(s);  }  session.getTransaction().commit();  }  } |

|  |
| --- |
| **public** **class** GetStudentRecords {  **public** **static** **void** main(String[] args) {    Configuration config = **new** Configuration().configure("hibernate\_office.cfg.xml").addAnnotatedClass(Student.**class**);  ServiceRegistry registry = **new** ServiceRegistryBuilder().applySettings(config.getProperties()).buildServiceRegistry();  SessionFactory sf = config.buildSessionFactory(registry);  Session session = sf.openSession();    session.beginTransaction();  //Query allData = session.createQuery("from Student"); // It provide the list of student  Query allData = session.createQuery("from Student where rollno >49"); // it will retrun the student  List<Student> studentList = allData.list();    **for**(Student s : studentList ) {  System.***out***.println(s);  }  System.***out***.println("To fetch the particular one unique record using where clause");  // To fetch the particular one unique record using where clause  Query oneStudent = session.createQuery("from Student where rollno=7"); // it will retrun the student  Student one = (Student)oneStudent.uniqueResult(); // uniqueResult fetch only one record from the table  System.***out***.println(one);    // fetching more than one column using select: Since column or object would have different kind of datatyps so  // we can put into Student class object other wise will get issue. To solve this instead of using Student class object  // we will Object class object as given below  System.***out***.println("fetching more than one column using select:");    Query selectColumn = session.createQuery("select rollno, name , marks from Student s where s.rollno=7");  Object[] student = (Object[])selectColumn.uniqueResult();  **for**(Object obj : student) {  System.***out***.println(obj);  }    // Get all the record using mentioning different column name  System.***out***.println(":::::::Get all the record using mentioning different column name::::");  Query allrecords = session.createQuery("select rollno, name , marks from Student");  List<Object[]> students = (List<Object[]>)allrecords.list();  **for**(Object[] obj : students) {  System.***out***.println(obj[0] + ":"+ obj[1] + ":" + obj[2]);  }    // sum of total marks  **int** b =60;  System.***out***.println(":::::::Sum of all student marks::::");  //Query totalMarks = session.createQuery("select sum(marks) from Student");  // Query totalMarks = session.createQuery("select sum(marks) from Student s where s.marks>60");  Query totalMarks = session.createQuery("select sum(marks) from Student s where s.marks >"+b);  Long marks= (Long) totalMarks.uniqueResult();  System.***out***.println("Total marks of the student::"+marks);  session.getTransaction().commit();  }  } |
| **Output:**  INFO: HHH000232: Schema update complete  Hibernate: select student0\_.rollno as rollno0\_, student0\_.marks as marks0\_, student0\_.name as name0\_ from Student student0\_ where student0\_.rollno>49  Student [rollno=50, name=Name50, marks=37]  To fetch the particular one unique record using where clause  Hibernate: select student0\_.rollno as rollno0\_, student0\_.marks as marks0\_, student0\_.name as name0\_ from Student student0\_ where student0\_.rollno=7  Student [rollno=7, name=Name7, marks=35]  fetching more than one column using select:  Hibernate: select student0\_.rollno as col\_0\_0\_, student0\_.name as col\_1\_0\_, student0\_.marks as col\_2\_0\_ from Student student0\_ where student0\_.rollno=7  7  Name7  35  :::::::Get all the record using mentioning different column name::::  Hibernate: select student0\_.rollno as col\_0\_0\_, student0\_.name as col\_1\_0\_, student0\_.marks as col\_2\_0\_ from Student student0\_  1:Name1:31  2:Name2:3  3:Name3:65  4:Name4:55  5:Name5:19  6:Name6:37  7:Name7:35  8:Name8:3  9:Name9:66  10:Name10:93  11:Name11:63  12:Name12:43  13:Name13:11  14:Name14:8  15:Name15:10  16:Name16:35  17:Name17:94  18:Name18:33  19:Name19:85  20:Name20:28  21:Name21:96  22:Name22:48  23:Name23:43  24:Name24:24  25:Name25:34  26:Name26:61  27:Name27:68  28:Name28:56  29:Name29:25  30:Name30:79  31:Name31:75  32:Name32:58  33:Name33:67  34:Name34:12  35:Name35:10  36:Name36:14  37:Name37:39  38:Name38:81  39:Name39:43  40:Name40:65  41:Name41:7  42:Name42:62  43:Name43:1  44:Name44:78  45:Name45:99  46:Name46:69  47:Name47:13  48:Name48:81  49:Name49:15  50:Name50:37  :::::::Sum of all student marks::::  Hibernate: select sum(student0\_.marks) as col\_0\_0\_ from Student student0\_ where student0\_.marks>60  Total marks of the student::1447 |

**Using SQL query in Hibernate: (Native Query):**

**Hibernate Object States / Persistence Life Cycle:**

As we have seen earlier, when we work with hibernate then we use Session and it consist several method like save, get, update, delete. So when use these methods then there is change in state of objects.

Analogy: When we create an object in java using new operator then we use this object in some operation and when we have done with this object then either we nullify this object or made it available for garbage collector. So we have this two state for every object in java 1- Create 2- Destroy. And between this two state we can put hibernate, hibernate has its own state.

Let’s say we have got an object of class Student and this student object would have certain features or properties or variable like, rollno, name, and marks. Now let’s say we have set some values in these properties using object. So the moment we create the object of Student class then hibernate become in transient state. So now our newly created object now in transient state and if we make any changes in the object properties and close the project then this data will not be persist if we reopen the project i.e. we will not get the data back. So to get the data back we have to persist or save it somewhere like in database or file. So the moment we save or persist our object then the object goes into persistent state. Now in this state we make any changes in the object, it will be available in the database or file. So far we have to state of the object .

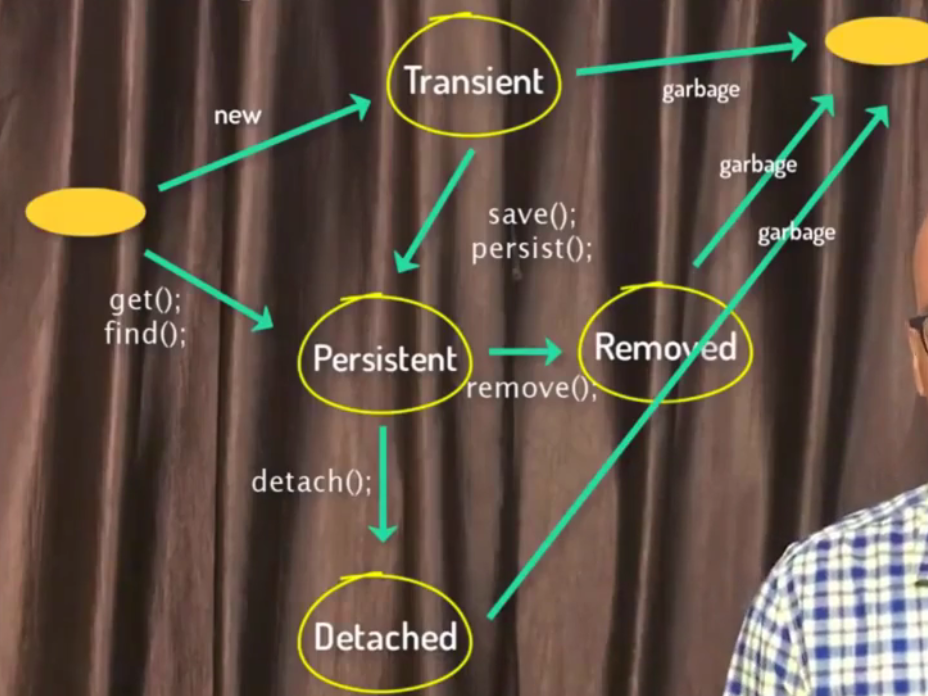
1. Transient state
2. Persistent state

Now the next that will come is called detached state. Let’s say if want to modify the persisted data then in that case first the object should be detached and the start any operation on the object. The transient and detached state is exactly same.

Now let’s say we have removed the data from the database so the moment we delete the data from the database then the persisted data goes into removed state. So the object would be available in the java but not in database.

Q: The above scenario for save data in the DB but what about fetching data from the database?

So when we use get or find method hibernate then the object does not go into transient state because we are trying to the fetch the data from the persisted state from the DB. In this case the object will go from create or new to persisted directly.



So finally we have several state in the hibernate or jpa

1. New or Create state
2. Transient state
3. Persistent State
4. Detached State
5. Remove State

Program to be mentioned from the eclipse:

**Hibernate Get Vs Load:**

|  |
| --- |
| System.***out***.println("-----Fetch using get method-------------- ");  Student useGet = (Student)session.get(Student.**class**, 5);  System.***out***.println(useGet);    System.***out***.println(":::::::::Fetch using load method::::::::::::");  Student useLoad= (Student)session.load(Student.**class**, 5);  System.***out***.println(useLoad); |
| Ouput :  -----Fetch using get method--------------  Hibernate: select student0\_.rollno as rollno0\_0\_, student0\_.marks as marks0\_0\_, student0\_.name as name0\_0\_ from Student student0\_ where student0\_.rollno=?  Student [rollno=5, name=Name5, marks=19]  :::::::::Fetch using load metod::::::::::::  Student [rollno=5, name=Name5, marks=19] |

It seems like there is no difference in get and load method because both the method fetching the same detail. Now let’s make one change in the above code. Let’s comment the code which is displaying the record.

|  |
| --- |
| System.***out***.println("-----Fetch using get method-------------- ");  Student useGet = (Student)session.get(Student.**class**, 5);  //System.out.println(useGet);    System.***out***.println(":::::::::Fetch using load metod::::::::::::");  Student useLoad= (Student)session.load(Student.**class**, 5);  //System.out.println(useLoad); |
| Ouput:  -----Fetch using get method--------------  Hibernate: select student0\_.rollno as rollno0\_0\_, student0\_.marks as marks0\_0\_, student0\_.name as name0\_0\_ from Student student0\_ where student0\_.rollno=?  :::::::::Fetch using load method::::::::::::  // No Query fired in case of load |

So here we can see that in case of get the query is getting fired whether we are using or not that result. While in case of load the query will not be fired until we use the result object further.

So In load method the performance will be better in compare to get method because it hit the data base when we are using the result object somewhere.

So load will give us the proxy object and will not be enabled and hit DB until we use it. That is why we don’t use load because instead of actual object it provides us a proxy object. So load we can use when we want to create a fake or proxy object.