AJAX -Asynchronous JavaScript and XML.

AJAX allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.



Creating XMLHttp Request object - To handle all browsers, including IE5 and IE6, check if the browser supports the XMLHttpRequest object. If it does, create an XMLHttpRequest object, if not, create an ActiveXObject:

var xhttp;  
if (window.XMLHttpRequest) {  
    xhttp = new XMLHttpRequest();  
    } else {  
    // code for IE6, IE5  
    xhttp = new ActiveXObject("Microsoft.XMLHTTP");  
}

Send the request to server

Syntax-

|  |  |
| --- | --- |
| open(*method, url, async*) | *method*: the type of request: GET or POST *url*: the server (file) location *async*: true (asynchronous) or false (synchronous) |

GET is simpler and faster than POST, and can be used in most cases.

However, always use POST requests when:

* A cached file is not an option (update a file or database on the server).
* Sending a large amount of data to the server (POST has no size limitations).
* Sending user input (which can contain unknown characters), POST is more robust and secure than GET.

*xhttp.open("GET", "demo\_get.asp", true);  
xhttp.send();*

you may get a cached result. To avoid this, add a unique ID to the URL:

*xhttp.open("GET", "demo\_get.asp?t=" + Math.random(), true);  
xhttp.send();*

Sending data in a GET request

xhttp.open("GET", "demo\_get2.asp?fname=Henry&lname=Ford", true);  
xhttp.send();

Sending data in POST Request

To POST data like an HTML form, add an HTTP header with setRequestHeader(). Specify the data you want to send in the send() method:

xhttp.open("POST", "ajax\_test.asp", true);  
xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");  
xhttp.send("fname=Henry&lname=Ford");

Using Async as true (recommended)- When using async=true, specify a function to execute when the response is ready in the onreadystatechange event:

xhttp.onreadystatechange = function() {  
  if (xhttp.readyState == 4 && xhttp.status == 200) {  
    document.getElementById("demo").innerHTML = xhttp.responseText;  
  }  
};  
xhttp.open("GET", "ajax\_info.txt", true);  
xhttp.send();

Using Async as false (donot use onreadystatechange event)- Using async=false is not recommended, but for a few small requests this can be ok.

when async is false,remember that the JavaScript will NOT continue to execute, until the server response is ready. If the server is busy or slow, the application will hang or stop.

xhttp.open("GET", "ajax\_info.txt", false);  
xhttp.send();  
document.getElementById("demo").innerHTML = xhttp.responseText;

RESPONSE FROM SERVER

To get the response from a server, use the responseText or responseXML property of the XMLHttpRequest object.

|  |  |
| --- | --- |
| **Property** | **Description** |
| responseText | get the response data as a string |
| responseXML | get the response data as XML data |

ResponseText Example-

document.getElementById("demo").innerHTML = xhttp.responseText;

ResponseXML Example-

xmlDoc = xhttp.responseXML;  
txt = "";  
x = xmlDoc.getElementsByTagName("ARTIST");  
for (i = 0; i < x.length; i++) {  
  txt += x[i].childNodes[0].nodeValue + "<br>";  
  }  
document.getElementById("demo").innerHTML = txt;

AJAX Events

Three important properties of the XMLHttpRequest object:

|  |  |
| --- | --- |
| **Property** | **Description** |
| onreadystatechange | Stores a function (or the name of a function) to be called automatically each time the readyState property changes |
| readyState | Holds the status of the XMLHttpRequest. Changes from 0 to 4:  0: request not initialized  1: server connection established 2: request received  3: processing request  4: request finished and response is ready |
| status | 200: "OK" 404: Page not found |

When readyState is 4 and status is 200, the response is ready

**Using a Callback Function**

A callback function is a function passed as a parameter to another function.

If you have more than one AJAX task on your website, you should create ONE standard function for creating the XMLHttpRequest object, and call this for each AJAX task.

The function call should contain the URL and what to do on onreadystatechange (which is probably different for each call):

*<!DOCTYPE html>*

*<html>*

*<body>*

*<p id="demo">Let AJAX change this text.</p>*

*<button type="button"*

*onclick="loadDoc('ajax\_info.txt', myFunction)">Change Content*

*</button>*

*<script>*

*function loadDoc(url, cfunc) {*

*var xhttp;*

*xhttp=new XMLHttpRequest();*

*xhttp.onreadystatechange = function() {*

*if (xhttp.readyState == 4 && xhttp.status == 200) {*

*cfunc(xhttp);*

*}*

*};*

*xhttp.open("GET", url, true);*

*xhttp.send();*

*}*

*function myFunction(xhttp) {*

*document.getElementById("demo").innerHTML = xhttp.responseText;*

*}*

*</script>*

*</body>*

*</html>*

**GETTING RESPONSE HEADER:**

The getResponseHeader() function is used to return specific header information from a resource, like length, server-type, content-type, last-modified, etc.

*<!DOCTYPE html>*

*<html>*

*<body>*

*<p>The getResponseHeader() function is used to return specific header information from a resource, like length, server-type, content-type, last-modified, etc.</p>*

*<p>Last modified: <span id="demo"></span></p>*

*<button onclick="loadDoc('ajax\_info.txt')">Get "Last-Modified" information</button>*

*<script>*

*function loadDoc(url) {*

*var xhttp=new XMLHttpRequest();*

*xhttp.onreadystatechange = function() {*

*if (xhttp.readyState == 4 && xhttp.status == 200) {*

*document.getElementById("demo").innerHTML = xhttp.getResponseHeader('Last-Modified');*

*}*

*};*

*xhttp.open("GET", url, true);*

*xhttp.send();*

*}*

*</script>*

*</body>*

*</html>*

HIBERNATE

Collection Mapping from Java brains- Kaushik’s video

**3.1 Saving Collections**

1)UserDetails class is marked as @Entity .Also @Table(name=”User\_details”) annotation is there

@Entity

@Table(name=”User\_details”)

public Class UserDetails{

@id

@GeneratedValue(strategy=GenerationType.AUTO)

private int userId;

private String username;

@ElementCollection

private Set<Address> listOfAddresses =new HashSet<Address>();

//getters and setters

}

2)Address class

@Embeddable

public class Address{

@Column(name=”STREET\_NAME”)

private String streetName;

//similarly city state and pincode

}

Hibernate will do following upon saving User

Insert into user\_details(username,userId) values(?,?)

Insert into user\_details\_listofAddresses(userDetails\_userid,city\_name,pin\_code,State\_name,street\_name)values(?,?,?,?)

Insert into user\_details\_listofAddresses(userDetails\_userid,city\_name,pin\_code,State\_name,street\_name) values(?,?,?,?)

Child table name is created by hibernate by appending the name of entity with name of member variable which is a collection.

The number of rows inserted in this child table will be same as the size of collection which you tried to insert. e.g if we add 2 addresses for a user and store them in the Set –listOfAddresses then this table will have 2 rows. Each of them will have same user id which is a foreign key in child table and maps to primary key in user\_details table

**3.2 Configuring Collections and adding keys**

1)In order to give a different name (different from the one which is auto generated) we need to do below in UserDetails Class

@ElementCollection

@JoinTable(name=”USER\_ADDRESS”)

Private Set<Address> listOfAddresses=new HashSet<Address>();

Now the result will be

**Insert into user\_details(username,userId) values(?,?)**

**Insert into USER\_ADDRESS (userDetails\_userid,city\_name,pin\_code,State\_name,street\_name)values(?,?,?,?)**

**Insert into USER\_ADDRESS (userDetails\_userid,city\_name,pin\_code,State\_name,street\_name) values(?,?,?,?)**

2)In order to give a different column name(see in red above) ,we can do below in UserDetails class

@JoinTable(name=”USER\_ADDRESS”,

joinColumns=@JoinColumn(name=”user\_id”))

Private Set<Address> listOfAddresses=new HashSet<Address>();

**Insert into user\_details(username,userId) values(?,?)**

**Insert into USER\_ADDRESS (user\_id,city\_name,pin\_code,State\_name,street\_name)values(?,?,?,?)**

**Insert into USER\_ADDRESS (user\_id,city\_name,pin\_code,State\_name,street\_name) values(?,?,?,?)**

3) Defining a primary id on child table

Since HashSet does not support indexes we cannot use HashSet. we should use ArrayList

Specification is JPA but implementer is hibernate for all annotations till now, but @CollectionId is not a JPA annotation it is specific to hibernate.

@ElementCollection

@JoinTable(name=”USER\_ADDRESS”,

joinColumns=@JoinColumn(name=”user\_id”))

@GenericGenerator(name=”hilo-gen”,strategy=”hilo”)

@CollectionId(columns={@Column(name=”ADDRESS\_ID”)},generator=”hileo-gen”,type=@Type(type=”long”))

private Collection<Address> listOfAddresses=new ArrayList<Address>();

### 3.3 Proxy Objects and Eager and Lazy Fetch Types

The default behaviour of hibernate is to not to fetch the inner objects(e.g list of addresses) when the parent (user) object is fetched.

Session=sessionFactory.openSession();

User=(UserDetails)session.get(UserDetails.Class,1);

But the moment I write below line, Hibernate actually then queries the database to get the list of addresses associated with the user object.

SOP(user.getListOfAddresses().size()); // will print 2

This is called **Lazy Initialisation.** This means that we initialise only first level member variables and not the entire object. You initialise the list only when you access it.

How the inner objects are retrieved by Hibernate?- This is done using Proxy class. Proxy class is a dynamic subclass of real class. So when you do getName or getId of your parent class , it actually accesses getId and getName method of proxy class. Proxy user class fills in all the first level member variables from DB. Hibernate will hand you the proxy object and not the actual object. You wouldn’t know because it implements the same methods. The proxy class getName invokes parent class getName and similarly getID.The proxy class’s getListOfAddresses has the code to fetch data from DB and then call the actual class getListOfAddresses method

CASE

Session=sessionFactory.openSession();

User=(UserDetails)session.get(UserDetails.Class,1);

Session.close();

SOP(user.getListOfAddresses().size());// will throw lazyIitialisationException. Since the session was closed, proxy object was unable to get the value from DB. In order to address this problem we need to do eager initialisation. This should be used when there are chances to fetch inner objects even when the session is closed.

EAGER INITIALISATION – This will take lot of resources, lot of time, so we should do this only when we actually need it

@ElementCollection(fetch=FetchType.EAGER)

@JoinTable(name=”USER\_ADDRESS”,joinColumns=@joinColumn(name=”userId”))

Private Collection<Address> listOfAddresses=new ArrayList<Address>();

**3.4 OnetoOne mapping**

Till now address was not an entity, it was a value type. Now we will see one entity inside another entity.

Suppose Vehicle is another class which is marked with @Entity( and not @Embeddable)

1)Let UserDetails class contain a member variable of Vehicle class.

@OneToOne

private Vehicle vehicle;

2)Inside main method –

PSVM(){

UserDetails user=new UserDetails();

User.setUseName(“First User”);

Vehicle vehicle=new Vehicle();

vehicle.setName(“Car”);

user.setVehicle(vehicle);

SessionFactory sessionFactory =new Configuration().configure().buildSessionfactory();

Session session= sessionFactory.openSession();

session.beginTransaction();

session.save(user);

session.save(vehicle);

session.getTransaction().commit();

session.close();

}

Result

Insert into user\_details (username,vehicle \_vehicleId,userid) values(?,?,?);

Insert into vehicle (name,vehicleid)values(?,?)

Update user\_details set username=?,vehicle\_vehicleId=? Where userId=?

If I want to change the name of the column from vehicle\_vehicleid to just VehicleId then i need to do below in UserDetails class

@OneToOne

@JoinColumn(name=”vehicle\_id”)

Result

Insert into user\_details values(username, vehicle\_Id,userid) values(?,?,?);

Insert into vehicle (name,vehicleid)values(?,?)

Update user\_details set username=?, vehicle\_Id=? Where userId=?

**3.5 OneToMany mapping**

1)Inside UserDetails class dobelow :-

@OneToMany

private Collection<Vehicle> vehicle=new Arraylist<Vehicle> ();

2)Inside main method

UserDetails user=new UserDetails();

user.setUsername(“First User”)l;

Vehicle vehicle=new vehicle();

vehicle.setName(“Car”);

Vehicle vehicle2=new vehicle();

Vehicle2.setName(“Jeep”);

user.getVehicle().add(vehicle);

user.getVehicle().add(vehicle2);

SessionFactory sessionFactory =new Configuration().configure().buildSessionfactory();

Session session= sessionFactory.openSession();

session.beginTransaction();

session.save(user);

session.save(vehicle);

session.save(vehicle2);

session.getTransaction().commit();

session.close();

The Result is

Insert into user\_details(username,userId) values(?,?)

Insert into vehicle(vehicle\_id,name)values(?,?);

Insert into vehicle(vehicle\_id,name)values(?,?);

Insert into user\_details\_vehicle(user\_details\_userId,vehicle\_vehicle\_id)values(?,?);

Insert into user\_details\_vehicle(user\_details\_userId,vehicle\_vehicle\_id)values(?,?);

In order to change the default names of newly created table and column we need to do below

@JoinTable(name=”USER\_VEHICLE”,

joinColumns=@joinColumn(name=”user\_Id”),

inverseJoinColumns=@joinColumn(name=”VEHICLE\_ID”))

Now the Result is

Insert into user\_details(username,userId) values(?,?)

Insert into vehicle(vehicle\_id,name)values(?,?);

Insert into vehicle(vehicle\_id,name)values(?,?);

Insert into user\_ vehicle(user\_Id, vehicle\_id)values(?,?);

Insert into user \_vehicle(user\_Id,vehicle\_id)values(?,?);

Instead of having OneToMany relation in user\_details class we can also have reverse relationship in the Vehicle class i.e ManytoOne

In the Vehicle class we will do –

@ManyToOne

Private UserDetails user;

In the vehicle class we will have getter and setter of User Object.

We can also have a bi directional relationship i.e. both the annotations at the same time. By doing this, we can get User object from vehicle object and vice versa.

In the main method we will do

UserDetails user=new UserDetails();

user.setUsername(“First User”)l;

Vehicle vehicle=new vehicle();

vehicle.setName(“Car”);

Vehicle vehicle2=new vehicle();

Vehicle2.setName(“Jeep”);

user.getVehicle().add(vehicle);

user.getVehicle().add(vehicle2);

vehicle.setUser(user);

vehicle2.setUser(user);

SessionFactory sessionFactory =new Configuration().configure().buildSessionfactory();

Session session= sessionFactory.openSession();

session.beginTransaction();

session.save(user);

session.save(vehicle);

session.save(vehicle2);

session.getTransaction().commit();

session.close();

**3.6 Mapped by and ManyToMany mapping**

For one to Many and Many to One relationship hibernate created a separate third table which stored primary keys of both the tables-User and Vehicle. Let’s say we want to have a reference relationship (foreign key relation).Now the User table cannot have Vehicle id since one user can have multiple vehicles, but vehicle table can have user id, since one vehicle can have only one user.. For doing that we need to do below

1)In UserDetails class

@OneToMany(mappedBy=”user”)// here user is the name of member variable of UserDetails in Vehicle class

private Collection<Vehicle> vehicle= new Arraylist<Vehicle>();

2)Inside Vehicle class

@ManyToOne

@JoinColumn(name=”USER\_ID”)

private UserDetails user;

The result will be

Insert into user\_details(userid,username)values(?,?);

Insert into Vehcile(user\_id,Vehicle\_id,name)values(?,?,?);

Insert into Vehcile(user\_id,Vehicle\_id,name)values(?,?,?);

ManyToMany : Suppose Vehicle is rented vehicle. It can be shared among multiple users. In this case none of the table can have reference to the other table. So we must have a separate table which maps primary key of both the tables.

1)Now inside Vehicle class also we have collection of UserDetails object type

@ManyToMany

private Collection<UserDetails> userList=new Arraylist<UserDetails>();

2)Inside UserDetails class we have

@ManyToMany

private Collection<Vehicle> vehicle=new Arraylist< Vehicle >();

3) In the main method we will do

UserDetails user=new UserDetails();

user.setUsername(“First User”)l;

Vehicle vehicle=new vehicle();

vehicle.setName(“Car”);

Vehicle vehicle2=new vehicle();

Vehicle2.setName(“Jeep”);

user.getVehicle().add(vehicle);

user.getVehicle().add(vehicle2);

vehicle.getUserList().add(user);

vehicle2.getUserList().add(user);

SessionFactory sessionFactory =new Configuration().configure().buildSessionfactory();

Session session= sessionFactory.openSession();

session.beginTransaction();

session.save(user);

session.save(vehicle);

session.save(vehicle2);

session.getTransaction().commit();

session.close();

The result will be

Insert into user\_details(userid,username)values(?,?);

Insert into Vehcile(Vehicle\_id,name)values(?,?,?);

Insert into Vehcile(Vehicle\_id,name)values(?,?,?);

Insert into user\_details\_vehicle(user\_details\_userid,vehicle\_vehicle\_id) values(?,?);

Insert into user\_details\_vehicle(user\_details\_userid,vehicle\_vehicle\_id) values(?,?);

Insert into vehicle\_ user\_details (vehicle\_vehicle\_id ,userList\_userid) values(?,?);

Insert into vehicle\_ user\_details (vehicle\_vehicle\_id ,userList\_userid) values(?,?);

Hence hibernate has created 2 new tables by seeing 2 annotations @ManyToMany in both the tables. Hibernate does not know that it has to create only one mapping table.

In order to have only one mapping table do below in Vehicle class

@ManyToMany(mappedBy=”vehicle”)

private Collection<User\_details> userList=new Arraylist< User\_details > ();

The result will be

Insert into user\_details(userid,username)values(?,?);

Insert into Vehcile(Vehicle\_id,name)values(?,?,?);

Insert into Vehcile(Vehicle\_id,name)values(?,?,?);

Insert into user\_details\_vehicle(user\_list\_userid,vehicle\_vehicle\_id) values(?,?);

Insert into user\_details\_vehicle(user\_list\_userid,vehicle\_vehicle\_id) values(?,?);

Hence the annotation ManyToMany in UserDetails class is doing the mapping and the annotation ManyToMany in Vehicle class is not doing the mapping.

In order to change the default names we can use the annotations JoinTable and JoinColumns. Please note that these should be used in UserDetails class which is actually doing the mapping and not in Vehicle Class where we used mappedBy property.

**3.7 Cascade Types and Other Things**

Consider One to Many relationship

1)In UserDetails class

@OneToMany(mappedBy=”user”

private Collection<Vehicle> vehicle= new Arraylist<Vehcile>();

2)Inside Vehicle class

@ManyToOne

@JoinColumn(name=”USER\_ID”)

private UserDetails user;

Now Assume that Vehicle is currently not associated with any user, so vehicle.getUser() will give an exception. In order to suppress this exception, in Vehicle class you need to do the following

@ManyToOne

@NotFound(action=NotFoundAction.IGNORE)

@JoinColumn(name=”USER\_ID”)

private UserDetails user;

Hibernate Collections –Hibernate has its own data types which it maps to our java data types

1)Bag Semantic – List/ArrayList. Here Bag means it does not matter how you put things into a bag, you can retrieve them in any order

2)Bag Semantic with ID –List/ArrayList

3)List Semantic –List/ArrayList

The difference between list and a bag is list can be ordered e.g sorted whereas bag cannot be ordered

4)Set semantic – Set

5)Map semantic –Map

In order to maintain the collections, hibernate uses these semantics behind the scenes.

CASCADE relations – Suppose there is one to many relationship between user and vehicle. A user is associated with 10 vehicles. While saving user object, we need to save 10 vehicle objects as well. Similarly while deleting the user object, we need to delete 10 vehicle objects associated with it. There is a way to avoid this –

UserDetails class :

@OneToMany

private Collection<Vehicle> vehicle= new Arraylist<Vehicle>();

Vehcile Class : There is no backward reference to UserDetails class. Only user.getVehicle() is possible

Inside main method

UserDetails user=new UserDetails();

user.setUsername(“First User”)l;

Vehicle vehicle=new vehicle();

vehicle.setName(“Car”);

Vehicle vehicle2=new vehicle();

Vehicle2.setName(“Jeep”);

user.getVehicle().add(vehicle);

user.getVehicle().add(vehicle2);

SessionFactory sessionFactory =new Configuration().configure().buildSessionfactory();

Session session= sessionFactory.openSession();

session.beginTransaction();

session.save(user);

//session.save(vehicle);

//session.save(vehicle2);

session.getTransaction().commit();

session.close();

See the commented lines above. Suppose I do not save vehicle objects and directly try to save user object, I will get an exception saying – object references an unsaved transient instance before flushing...since the user object references vehicle objects which are not saved. Since Vehicle itself is an entity Hibernate is not automatically saving them. Vehicle objects have life of their own. They are not bound by the user object.

Use below if you want to indicate hibernate that whenever I am saving parent object, the inner object which it references to should also be saved.

In UserDetails class

@OneToMany(cascade= CascadeType.PERSIST)

private Collection<Vehicle> vehicle=new Arraylist<Vehicle>();

In main method : Use session.persist(user) instead of session.save(user)

The Result is

Insert into user\_details(username,userId) values(?,?)

Insert into vehicle(vehicle\_id,name)values(?,?);

Insert into vehicle(vehicle\_id,name)values(?,?);

Insert into user\_details\_vehicle(user\_details\_userId,vehicle\_vehicle\_id)values(?,?);

Insert into user\_details\_vehicle(user\_details\_userId,vehicle\_vehicle\_id)values(?,?);

Also we have other cascade types e.g REMOVE which is used for session.delete(). Also we have MERGE ,DETACH. If you want cascade to happen every time irrespective of the operation being performed then use CascadeType.ALL

**3.8 Implementing Inheritance**

Mapping inheritance in OOP to relational tables -This is the challenge which every ORM tool faces.

Why inheritance is required? Lets say I have a Vehicle class and there are two subclasses of Vehicle class – Two Wheeler class and a Four Wheeler class. Let’s say UserDetails class holds a reference of Vehicle class which is an abstract class. The exact object is decided at run time i.e TwoWheeler or a FourWheeler. Now in relational model let’s say I have two different tables – twoWheeler and FourWheeler, in that case how would a user object know the Vehicle is of which type. Which table it refers to?Hence in order to implement polymorphism we need to have some way to communicate this to hibernate

Second reason is let’s say i have another field added in Vehicle class, then in OOP we have the field automatically added in the derived classes, but when it comes to relational tables the tables – twoWheeler and FourWheeler still he the original columns, they do not change dynamically.

@Entity

Public class TwoWheeler extends Vehicle{

Private String steeringHandle;

//getters and setters

}

@Entity

Public class FourWheeler extends Vehicle{

Private String steeringWheel;

//getters and setters

}

Main method

Vehicle vehicle=new Vehicle();

Vehicle.setName(“Car”);

TwoWheeler bike= new TwoWheeler();

bike.setName(“bike”);

bike.setSteeringHandle(“Bike steering handle”);

FourWheeler car= new FourWheeler ();

car.setName(“porsche”);

car.setSteeringWheel(“porsche steering wheel”);

session.save(“vehicle”);

session.save(“bike”);

session.save(“car”);

Here all three are entities

Result is :

Hibernate just created one table – Vehicle even though all the classes were marked with annotation @Entity.

Insert into Vehicle(name,DTYPE,vehicleId) values(?,’Vehicle’,?);

Insert into Vehicle(name,steeringHandle, DTYPE,vehicleId) values(?,?,’TwoWheeler’,?);

Insert into Vehicle(name,steeringWheel, DTYPE,vehicleId) values(?,?,’FourWheeler’,?);

Hibernate by default has single inheritance mapping strategy which means that no matter how many classes you have in a relationship, hibernate creates one table and will create different columns in that table depending on the fields defined in child classes.

Let’s say we have another class called Porsche which implements FourWheeler class, even in that case we will have a single table called vehicle with extra columns which Porsche class has.

Vehicle table has columns- DTYPE, vehicleid, name, steeringHanlde, streeringWheel

**3.8 Single Table Strategy**

Dtype column created by hibernate is actually the discriminator type which distinguishes the child record from other child and parent record. The previous tutorial explains the default inheritance strategy which hibernate implements. So, if @inheritance is missing then hibernate goes with single table strategy

In Vehicle class you can have below

@Entity

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

@ DiscriminatorColumn{

name=”VEHICLE\_TYPE”

discriminatorType=DiscriminatorType.STRING //Integer and char are other possible values

}

Public class Vehicle{

}

In TwoWheeler class you can have below

@Entity

@DiscriminatorValue(“Bike”) // now instead of taking class name hibernate will use this name to store TwoWheeler type records

Public class TwoWheeler{  
}

Same way In FourWheeler class you can have below

@Entity

@DiscriminatorValue(“Car”) // now instead of taking class name hibernate will use this name to store FourWheeler type records

Public class FourWheeler {  
}

So all the records of FourWheeler class will be persisted with Dtype column value as “Car” and name of the column will be Vehicle\_Type

Result wil be :

Insert into Vehicle(name,VEHICLE\_TYPE,vehicleId) values(?,’Vehicle’,?);

Insert into Vehicle(name,steeringHandle, VEHICLE\_TYPE,vehicleId) values(?,?,’Bike’,?);

Insert into Vehicle(name,steeringWheel, VEHICLE\_TYPE,vehicleId) values(?,?,’Car’,?);

**3.8 Implementing Inheritance with Table per Class Strategy**

When we used single table strategy, we needed some way to distinguish between records of each class type, that is the reason we used @DiscriminatorColumn annotation, but in case of table per class strategy, we no longer need this.

In Vehicle class

@Entity

@Inheritance{strategy=InheritanceType.Table\_PER\_CLASS}

Public class Vehicle{

}

Result will be

Insert into Vehicle(VehicleId,name)values(?,?);

Insert into twowheeler(VehicleId,name,steeringHandle)values(?,?);

Insert into FourWheeler(VehicleId,name,steeringWheel)values(?,?);

Note that the TwoWheeler and FourWheeler classes have only one property i.e SteeringHandle and SteeringWheel,however the corresponding tables have vehicleId and name as columns as well.

1) There will be one table per class in the hierarchy

2) The base class’s properties are also inherited as columns in child class’s tables.

3)The @GeneratedValue of the parent class is also inherited into the child class ‘s tables

Advantage:

1) We don’t need a discriminator to identify what type of record it is, since we have separate tables

2) It is in normalised form, we don’t have extra columns which do not have data.

e.g In single table strategy, we can have columns steeringWheel and steeringHandle which can have null values for some records.

**3.8 Implementing Inheritance with Joined Strategy**

The first strategy i.e single\_table strategy was least normalised. The second strategy was Table\_per\_class where columns of the base class were repeated in child class tables, this can be improved further.

In Vehicle class do below

@Entity

@Inheritance(strategy=InheritanceType.JOINED)

Public class Vehicle{

}

Main method

Vehicle vehicle=new Vehicle();

Vehicle.setName(“Car”);

TwoWheeler bike= new TwoWheeler();

bike.setName(“bike”);

bike.setSteeringHandle(“Bike steering handle”);

FourWheeler car= new FourWheeler ();

car.setName(“porsche”);

car.setSteeringWheel(“porsche steering wheel”);

session.save(“vehicle”);

session.save(“bike”);

session.save(“car”);

Result will be

Insert into Vehicle(VehicleId,name)values(?,?);

Insert into Vehicle(VehicleId,name)values(?,?);

Insert into twowheeler(VehicleId ,steeringHandle)values(?,?);

Insert into Vehicle(VehicleId,name)values(?,?);

Insert into FourWheeler(VehicleId ,steeringWheel)values(?,?);

**Note:** The Vehicle class now has 3 records. It holds data for two wheeler and four wheeler as well. TwoWheeler and FourWheeler tables have Vehicle Id and property which is specific to them.

So, in order to pull the data for either four wheeler or two wheeler , I need to use join

**Select \* from Vehicle join FourWheeler on Vehicle.VehicleId=FourWheeler.VehicleId**

Hence all the fields which are inherited from base class will remain as columns in the parent table only.

**HIBERNATE – APIs and Runtime**

**1.1 CRUD OPERATIONS**

Create, retrieve , update and delete operations.

We have seen create and retrieve, now we will see update and delete operations.

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

for(int i=0;i<10;i++){

UserDetails user = new UserDetails();

user.setname(“user”+i);

session.save(user);

}

session.getTransaction().commit();

session.close();

For Retrieval

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

UserDetails user=(userDetails)session.get(UserDetails.class,6); // here 6 is the primary key to fetch the particular record.

SOP(“user name is “+user.getUsername());

session.getTransaction().commit();

session.close();

change hbm2ddl.auto to update if you don’t want to create schema from scratch

if the sop line above is placed after session.close() even then we will get the data since we are only accessing first level object i.e . Username

Hence this is eager fetch and not a lazy one.

Deletion of records

Step 1- pull up the object

UserDetails user=(userDetails)session.get(UserDetails.class,6);

Step 2- delete it

session.delete(user);

Updating records

Step 1- pull up the object

UserDetails user=(userDetails)session.get(UserDetails.class,6);

Step 2- update it

User.setUsername(“Updated User”);

session. update(user);

**1.2 Transient Persistent and detatched objects**

Main method

UserDetais user = new UserDetails();

User.setUserName(“Test User”); // till this point user object is transient object since it is not yet saved by hibernate

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

//user.setUserName(“Updated user”);

session.save(user);// at this point it becomes persistent object. Hibernate tracks this object and saves it

user.setUserName(“Updated user”);

session.getTransaction().commit();

session.close();

If i place user.setUserName(“UpdatedUser”) before session.save(user) then user will be saved with name – UpdatedUser

However , if I place it after session.save(user) , then result is : -

Insert into UserDetails(userId,Username) values(?,?)

Update userDetails set Username=? where userId=?

Hence, the name changed to UpdatedUser even in this case. We did not ask hibernate to save user after changing the property Username but still hibernate took care of that. How does hibernate know what change to reflect in DB? The answer is once we ask session.save(user) to be done by hibernate, after that any change done to the object will go as an update statement to the DB. Hence, the last update done before transaction.commit() will be the final state.

For example if I do the below after session.save() –

user.setUserName(“UpdatedUser”)

user.setUserName(“UpdatedUser Again”)

Then the result will be (single update statement only)

Insert into UserDetails(userId,Username) values(?,?)

Update userDetails set Username=? where userId=?

In the database last value will be reflected – UpdatedUser Again.

Not every change triggers a update statement. Hibernate intelligently detects what is the change that needs to go into the DB.’ Hence once you hand over the object to hibernate(session.save(user)), hibernate keeps on watching the object for any change

Hence before an object is handed over to hibernate , it is transient object. Once we give session.save(), then I am giving the responsibility to hibernate to make sure that object’s state matches the database state

Any changes made to persistent object get reflected in the database. Once I do session.close(), it becomes a detached object. It is similar to transient object in the sense that hibernate is not going to track the changes on it. This just means that detached object was once tracked and persisted by hibernate

Lets do below line after session.close()

User.setUsername(“Updated UserName after session close”);

This will not get reflected since the session is already closed. Hibernate tracks the object as long as the session is open.

**1.3 Understanding state changes**

Create operation flow

new () Session.save() session.close()

Transient ---------------------🡪 Persistent----------------------------🡪 Detached

Read Operation flow

get()

Persistent ---------------🡪 detached

session.close()

When we do session.get(), the object by default goes to persistent state. Hence if changes are done hibernate will reflect those in Database.

Update operation

session.update is not required when we use session.get(0 ,it automatically comes to persistent state.

Delete operation flow

Two flows- one is retrieve the object second is delete it. When you retrieve an object from DB, it goes to persistent state. When you delete the object , it goes to transient state. Once it goes to transient state, I can do session.save() which will take it back to persistent state.

Session.delete() session.close()

Transient 🡨------------------------Persistent -------------------🡪Detached

The change from Transient to persistent and persistent to detached state is happening only because of session involved. The transient becomes persistent once it is attached to session and persistent becomes detached once it is removed from session

SESSION

Now how to make an object move from Detached state to Persistent object . i.e again associate it with the session.? We will see next

### 1.4 Persisting Detached Objects

Consider the below Scenario

Steps-

1. Getting data from DB
2. Getting some input from user
3. Updating database

User Input

Now the problem in this scenario is user might take some time it might be 5 sec, 20 sec or might be a minute. So it is not a good idea to perform all three steps in single transaction.

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

UserDetails user=(UserDetails)session.get(UserDetails.Class,1);// step 1 above

session.getTransaction().commit();

session.close();

user.setUsername(“Updated Username after session close”);// now since the session is closed this update is not going to happen automatically, we need to do something

session=sessionFactory.openSession();

session.beginTransaction();

session.update(user);// in this step I am telling hibernate that user object has to be persisted again.As long as I have handle to user object even though it is detached, hibernate can take primary id and other info from object

session.getTransaction().commit();

session.close();

Few points to be noted

Scenario 1)session=sessionFactory.openSession();

session.beginTransaction();

session.update(user);

user.setUsername(“update 1”); // even this change will be reflected in DB

session.getTransaction().commit();

session.close();

scenario 2) Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

UserDetails user=(UserDetails)session.get(UserDetails.Class,1);// step 1 above

session.getTransaction().commit();

session.close();

//user.setUsername(“Updated Username after session close”);// suppose skip this step

session=sessionFactory.openSession();

session.beginTransaction();

session.update(user);

session.getTransaction().commit();

session.close();

Result is :

Select userid,username from userDetails where userId-=?

Update userdetails setusername=? Where userid=?

Now even though there was no field update written in my code,update query was hit by hibernate. This is because hibernate does not know any change is done in the object or not. This is done by hibernate on safer side to avoid data inconsistency. There is a way to avoid this. By doing something like if there is any change in user object only then run the update query, otherwise not.

The way out is in UserDetails class

@Entity – this is JPA annotation(javax.persistence.Entity) in addition to this add hibernate entity annotation below

@org.hibernate.annotations.Entity(selectBeforeUpdate=true)- hibernate is asked to do a select before doing the update and make update query only if there is a change in user object

Hence a select query is fired and the details are compared with the details in the current user object. If it is same then update is not run and vice versa.

Public class Usedetails{}

Now if scenario 2 above(without update statement) will give below result

Result is :

Select userid, username from userDetails where userId-=?;- this came because of session.get

Select userid, username from userDetails where userId-=?- this came because of session.update and the new annotation. Since the details were same , the update was not fired.

Consider below case

User.setUserName(“Udate 2”);

session=sessionFactory.openSession();

session.beginTransaction();

session.update(user);

session.getTransaction().commit();

session.close();

So now because of the annotation, it will run a select and the compare the results with current user object. Since there is a change in username property, hibernate will run update statement too.

Result is :

Select userid, username from userDetails where userId-=?;- this came because of session.get

Select userid, username from userDetails where userId-=?- this came because of session.update and the new annotation. Since the details are different, the update will be fired.

Update userDetails set username=? where userId=?;

Hence SelectBefore update =true should be used when most of the times you aren’t changing anything, you just want to persist the object. However, if the change is frequently required , the it shouldn’t be used

**2.1 Introducing HQL and Query Object**

In addition to session.get() for retrieval of data, hibernate provides another way of fetching data which is by writing queries in code.

Hibernate has provided a way to write queries and retrieve the data (e.g including multiple tables, joins,etc). We donot use SQL, instead we use a separate query language i.e Hibernate Query language( HQL). HQL is similar to SQL . The only difference is that in HQL ,we think of classes and objects as compared to tables in SQL.

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

Query query=session.createQuery(“from UserDetails”);// query object is in org.hibernate

List users=query.list();//java.util package

SOP(“size of list is “+users.size());

session.getTransaction().commit();

session.close();

**Instead of “select \* from table\_name” we used “from class\_name”. There is no select \***

Using where Clause

Query query=session.createQuery(“from UserDetails where userId>5”);// **here userId is a property name and not column name**

List users=query.list();

HQL returns the list of entity objects as compared to recordset in SQL.

**2.2 Select and Pagination in HQL**

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

Query query=session.createQuery(“from UserDetails ”);

List<UserDetails> users=( List<UserDetails>)query.list();

//SOP(“size of list is “+users.size());

for(UserDetails u:users){

SOP(u.getUsername());

}

session.getTransaction().commit();

session.close();

First problem in this approach is that I may not require all columns .Second problem can be I may need all the users at once. I may be interested in pagination i.e. one set of users first followed by another set.

PAGINATION

Query query=session.createQuery(“from UserDetails ”);

query.setFirstResult(5);// **it will start from 5th record i.e. 5th record of select \* will be the first record.**

Query.setMaxResults(4);//**Hibernate will stop after fetching 4 records even though there are more records available.**

List<UserDetails> users=( List<UserDetails>)query.list();

for(UserDetails u:users){

SOP(u.getUsername());

}

Result

Select userId,userName from userdetails limit? offset?

Hibernate does lazy loading e.g there is a property of type list in UserDetails class, then it will not be retrieved from DB unless you do access getter of that list(e.g. getList()).However other properties(first level data) get retrieved from DB when you do session.CreateQuery()

If I want hibernate to pull up only one property

Query query=session.createQuery(“select userName from UserDetails”);

In this case query execution will not return list of UserDetails but it will return list of String

List<String> userNames=(List<String>)query.list();

for(String u:usernames){

SOP(u);

}

If I want hibernate to pull up more than one property , the query.list() will return list of list

If we write the query like – select new map(userId,userName) from userDetails. In that case , it will be list of map, in that case we will get list of map object. i.e. each member of the list is map. And the map is for userId and userName.

Aggregation functions : will still work .e.g we can write select max(userId) from userDetails. In that case it will return a list of single integer.

Hence most of the things which SQL select has will translate to HQlas well ,the only difference being table name should be replaced by class name and column name to be replaced by property name.

**2.3 Understanding Parameter Binding and SQL Injection**

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

Query query=session.createQuery(“from UserDetails where userId>5”);

List users=query.list();//java.util package

SOP(“size of list is “+users.size());

session.getTransaction().commit();

session.close();

Let’s say SQL injection is taking place and this 5 comes from user input

String minUserId=5;

Query query=session.createQuery(“from UserDetails where userId> ”+minUserId);

This type of queries pose a risk as attackers can easily get the unintended data. E.g minUserId is fetched from user and attacker may enter “ 5 or 1=1”. In that case, user will get all records of UserDetails table- THIS IS CALLED SQL INJECTION ATTACKS

So the way out of this problem is to do parameter substitution/binding.

String minUserId=5;

Query query=session.createQuery(“from UserDetails where userId> ?”);// this “?” is a placeholder

query.setInteger(0,Integer.parseInt(minUserId));//here 0 is the position indicator. i.e first ? should be replaced by this value.

EXAMPLE

Query query=session.createQuery(“from UserDetails where userId> ? and username=?”);

query.setInteger(0,Integer.parseInt(minUserId));

query.setString(1,”hiti”);

Another way to do parameter binding

Query query=session.createQuery(“from UserDetails where userId> :UserId and userName=:userName”);

Hence instead of putting random place holders, we are placing placeholders with name, so that they can be substituted with the names and not by position.

query.setInteger(“userId”, Integer.parseInt(minUserId));

query.setString(“username”,”hiti”);

**2.4 Named Queries**

For maintenance purpose, it is usually best practice to keep all the queries of application at one place. Named Queries concept allows you to consolidate the queries at entity level.

@Entity

@NamedQuery(name=”UserDetails.byID”, query=”from UserDetails where userId=?”)

@Table(name=”user\_details”)

@org.hibernate.annotations.Entity(selectBeforeUpdate=true)

Public class UserDetails

{

}

Main method

Query query=session.**getNamedQuery**(“UserDetails.byID”);

query.setInteger(0,5);

Note that in case query property of namedquery annotation is reqd to contain native SQL query not only HQL, in that case we need to use the annotation NamedNativeQuery

@NamedNativeQuery(name=”UserDetails.byName” query=”select \* from user\_details where user\_name=?”, **resultClass=UserDetails.class**)

**If I use HQL, hibernate knows the return type object, however if I use native SQL hibernate does not know the object type.**

Main method

Query query=session.**getNamedQuery**(“UserDetails.byName”);

query.setString(0,”hiti”);

List<UserDetails> users=( List<UserDetails>)query.list();

session.getTransaction.commit();

session.close();

for(UserDetails user:users){

SOP(user.getUserName());

}

One advantage of using native annotation is we can run a stored procedure as well.

**2.5 Introduction to Criteria API**

For fetching data

1)session.get()

2)use queries HQL or SQL

Problem with second approach is that if query is bigger (e.g. 15 to 20 lines), then it becomes difficult to maintain.

We can use Criteria API approach to address this problem. Criteria is more of like where clause where you say these are the restrictions I need to impose.

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

Criteria criteria=session.createCriteria(UserDetails.class);// mention which object you want to add restriction

criteria.add(Restrictions.eq(“username”,”User 10”));// if you do not need any restriction then skip this step. In this example I am adding a restriction of equals

List<UserDetails> users=( List<UserDetails>)criteria.list();

session.getTransaction().commit();

session.close();

for(UserDetails user:users){

SOP(user.getUserName());

}

**2.6 Understanding Restrictions**

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

Criteria criteria=session.createCriteria(UserDetails.class

criteria.add(Restrictions.eq(“userName”,”User 10”));// we can have less than ,greater than apart from equal restriction

List<UserDetails> users=( List<UserDetails>)criteria.list();

session.getTransaction().commit();

session.close();

for(UserDetails user:users){

SOP(user.getUserName()); }

If there are many restrictions to be added, then they can be added as below called as chaining

Criteria criteria=session.createCriteria(UserDetails.class);

criteria.add(Restrictions.eq(“userName”,”User 10”))

.add(Restrictions.gt(“userId”,5));

Also we can use like operation Restrictions.like(“username”,”%User 0%”));

Example

criteria.add(Restriction.like(“username”,”%User 1%”))

.add(Restrictions.between(“userId”,5,50));

How to implement OR operation

Criteria criteria= session.createCriteria(UserDetails.class);

Criteria.add(Restrictions.or(Restrictions.between(“userId”,0,3), Restrcitions.between(“userId”,7,10)));

**2.7 Projections and Query By Example**

Criteria API features: Projections and query by example

Criteria criteria=session.createCriteria(UserDetails.class)

.setProjection(Projections.property(“userId”));// in this way it will return only one property and not the entire object.

Example to select max userId

Criteria criteria=session.createCriteria(UserDetails.class)

.setProjection(Projections.max (“userId”));

Example to select count of userIds

Criteria criteria=session.createCriteria(UserDetails.class)

.setProjection(Projections.count (“userId”));

Example to select userIds in particular order

Criteria criteria=session.createCriteria(UserDetails.class)

.addOrder(Order.desc(“userId”));

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

Criteria criteria=session.createCriteria(UserDetails.class)

.setProjection(Projections.max (“userId”));

//if projection is used with property or with any aggregation function , then criteria.list() will not return list of type userDetails, it will return only list. So you should change below code

List<UserDetails> users=( List<UserDetails>)criteria.list();

session.getTransaction().commit();

session.close();

for(UserDetails user:users){

SOP(user.getUserName()); }

Querying by example

Let’s say I have 10 fields and I know value for 5 fields to fetch a record. It is very tedious to use criteria in this case..addRestriction() will be repeated many times

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

UserDetails exampleUser=new userDetails();

exampleUser.setUserId(5);

exampleUser.seUserName(“User 5”);

Example example=Example.create(exampleUser);// by doing this I am telling hibernate that I am interested in creating an example out of that object and may pass this to criteria to pull the objects which look like this.

Criteria criteria=session.createCriteria(UserDetails.class)

.add(example);

List<UserDetails> users=( List<UserDetails>)criteria.list();

session.getTransaction().commit();

session.close();

for(UserDetails user:users){

SOP(user.getUserName()); }

Case 1 -In the above code if you comment this line-

exampleUser.setUserId(5);

//exampleUser.seUserName(“User 5”);

Output will be all records in userDetails table since we did not specify username

Case 2 – In the above code if you comment the other line

//exampleUser.setUserId(5);

exampleUser.seUserName(“User 5”);

Output will be just User 5

Concept: - if there is any property having null value, then hibernate does not consider that. Also , if property is a primary key, then hibernate does not consider that. Apart from null property and primary key property, if any other property you have set(e.g username) that is what hibernate takes into consideration.

If I do not want hibernate to take into consideration some property, then I can mention this while creating example object.

Example example=Example.create(exampleUser).excludeProperty(“username”);

Another scenario: if i want all users whose names start with User 1

UserDetails exampleUser=new userDetails();

exampleUser.seUserName(“User 1 %”);

Example example=Example.create(exampleUser).enableLike();

Output will be User 1, User 10

**3.1 Caching in Hibernate**

**First level cache-** Implemented by Session object. You open a session object when you want to talk to database and the close it. This type of caching is only available in current session object and this data is not available across sessions. This is by default provided by hibernate.

**Second level cache-**

* Across sessions in an application – If we open sessions at different points in application, this type of caching is enabled, data will be available from cache. Also all these sessions will have first level cache too. So if data is not found in first level of cache, then it will be searched in second level
* Across applications- Multiple applications working on same set of data
* Across clusters-Different applications deployed on different servers and they are all talking to same DB.

This type of caching is not by default provided and needs to be configured.

Problem in case of multiple applications using same DB -Some application might not be using hibernate. This application would not know that there is data in cache and might directly go and talk to DB. It might update the DB and cache might be invalid in this case.

First level cache

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

**UserDetails user=( UserDetails)session.get(UserDetails.class,1);**

session.getTransaction().commit();

session.close();

Result

Select userId,username from userdetails where userId=1;

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

**UserDetails user=( UserDetails)session.get(UserDetails.class,1);**

**UserDetails user2=( UserDetails)session.get(UserDetails.class,1);**

session.getTransaction().commit();

session.close();

Result

Select userId,username from userdetails where userId=1;

Again the result is single select query. Hibernate intelligently knows that data is in cache. This is called first level of cache. In between those two lines hibernate sees that there is no update of data.

Example

**UserDetails user=( UserDetails)session.get(UserDetails.class,1);**

**User.setUsername(“Updated User”);**

**UserDetails user2=( UserDetails)session.get(UserDetails.class,1);**

Result

Select userId,username from userdetails where userId=1;

Update userdetails set username=? where userId=?;

It is not doing a select again. This is because the object is updated in the session and is already in cache. Hence hibernate has no need to pull it from DB. Hibernate intelligently detects all these things. It does not go the DB unless it is actually required. So, if you have made the change in the session, hibernate will know that same object, I already have.

Multiple session example

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

**UserDetails user=( UserDetails)session.get(UserDetails.class,1);**

session.getTransaction().commit();

session.close();

Session session2=sessionFactory.openSession();// opening a new session here

session2.beginTransaction();

**UserDetails user2=( UserDetails)session2.get(UserDetails.class,1);**

session2.getTransaction().commit();

session2.close();

Result

Select userId,username from userdetails where userId=1;

Select userId,username from userdetails where userId=1;

We can see there are 2 selects. This is because when session is closed, cache is closed as well. New session resulted in a new query. Here comes a need of second level cache.

**3.2 Configuring second level cache**

Make below change in hibernate.cfg.xml

From

<property name=”cache.provider\_class”>org.hibernate.cache.**NoCacheProvider**</property>- this means second level cache is disabled.

To

<property name=”cache.use\_second\_leve\_cache”>true</property>

<property name=”cache.provider\_class”>org.hibernate.cache.**EhCacheProvider** </property>

There are many classes in org.hbernate.cache package- EhCache, OSCache etc. Each cache type has a corresponding provider class too.

Download ehCache jar and add that in classpath : go to library(if already created any) and then add jar- ehcache/lib/ehcache-core-2.4.2.jar

Make below change in UserDetails class( entity class to be cached)

@Entity

@Cacheable

@Cache(usage=CacheConcurrencyStrategy.READ\_ONLY// then hibernate assumes that application only reads the UserDetails data, and never writes the UserDetails data back to DB. Other options are READ\_WRITE which means hibernate will keep track of updates being performed and will update the cache accordingly.

NONSTRICT\_READ\_WRITE- This means that it doesn’t really strictly enforce this kind of check. So, there is a chance that you bring something from cache that was updated somewhere else. READ\_WRITE option however is strict

TRANSACTIONAL – this option is strictest of all.

@NamedQuery(...and so on)

@Table(name=”User\_Details”)

Public class UserDetails{}

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

**UserDetails user=( UserDetails)session.get(UserDetails.class,1);**

session.getTransaction().commit();

session.close();

Session session2=sessionFactory.openSession();// opening a new session here

session2.beginTransaction();

**UserDetails user2=( UserDetails)session2.get(UserDetails.class,1);**

session2.getTransaction().commit();

session2.close();

Result

Select userId,username from userdetails where userId=1;

Reason- After first select , hibernate added the UserDetails object in to second level cache which is then retrieved in second session.

**3.3 using Query Cache**

Let’s say I am pulling data using query and not using session.get() and I want second level caching to be implemented in that case as well.

Main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

**Query query=session.createQuery(“from UserDetails user where user.userid=1”);**

**List<UserDetails> users=query.list();**

**for(UserDetails user:users){  
}**

session.getTransaction().commit();

session.close();

Session session2=sessionFactory.openSession();// opening a new session here

session2.beginTransaction();

**Query query2=session2.createQuery(“from UserDetails user where user.userid=1”);**

**users=query2.list();**

session2.getTransaction().commit();

session2.close();

Result

Select userId,username from userdetails where userId=1;

Select userId,username from userdetails where userId=1;

Two selects are hitting db.

Solution

In hibernate.cfg.xml make the following change

<property name=”cache.use\_second\_leve\_cache”>true</property>-- this is optional when only query cache is required

**<property name=”cache.use\_query\_cache”>true</property>**

<property name=”cache.provider\_class”>org.hibernate.cache.EhCacheProvider </property>

In order to make query cacheable make below changes in main method

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

**Query query=session.createQuery(“from UserDetails user where user.userid=1”);**

**query.setCacheable(true);**

**List<UserDetails> users=query.list();**

session.getTransaction().commit();

session.close();

Session session2=sessionFactory.openSession();// opening a new session here

session2.beginTransaction();

**Query query2=session2.createQuery(“from UserDetails user where user.userid=1”);**

**users=query2.list();**

session2.getTransaction().commit();

session2.close();

This will still result in 2 selects. Now make query2 also cacheable

SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();

Session session=sessionFactory.openSession();

session.beginTransaction();

**Query query=session.createQuery(“from UserDetails user where user.userid=1”);**

**query.setCacheable(true);**

**List<UserDetails> users=query.list();**

session.getTransaction().commit();

session.close();

Session session2=sessionFactory.openSession();// opening a new session here

session2.beginTransaction();

**Query query2=session2.createQuery(“from UserDetails user where user.userid=1”);**

**query2.setCacheable(true);**

**users=query2.list();**

session2.getTransaction().commit();

session2.close();

Hence setCacheable not only makes the caches the query results, but also tells the query to look at second level cache and see if it is already there. Hence setcacheable is performing 2 roles

1)if query cache is not having values go to DB and fetch the record.

2)if query cache is already having the value then pull up the data.

Ehcache.xml

<?xml version="1.0"?>

<ehcache>

<defaultCache maxElementsInMemory="100" eternal="false" timeToIdleSeconds="120" timeToLiveSeconds="200" />

<cache name="str.Product" maxElementsInMemory="100" eternal="false" timeToIdleSeconds="5" timeToLiveSeconds="200" />

</ehcache>