

The Problem:

Map the member variable to the column

Mapping Relation ships

--> User object has reference to the address object

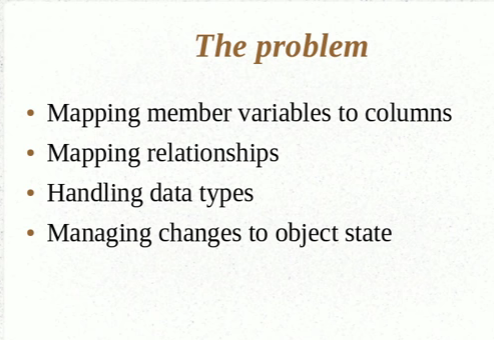
--> User object mapped to the User Table

--> Address object to the Address

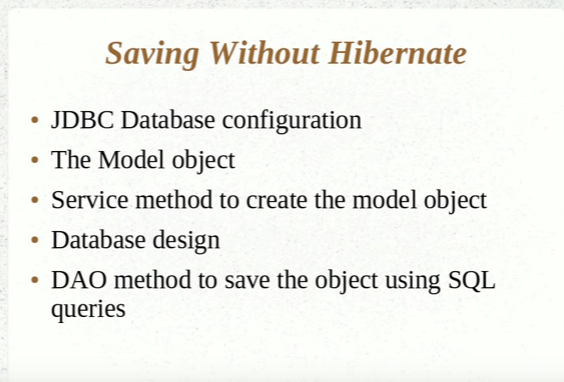
--> Referential Integrity between User and Address

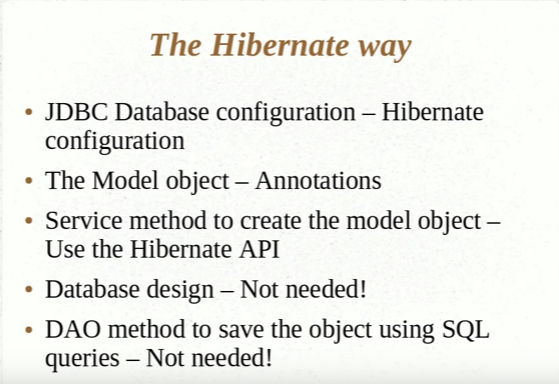
Handling Data Types :

Managing Changes to the Object State



Saving with out Hibernate:





Setting up the Hibernate Development Environment:

Create an Maven Project.

Add the following dependencies in the pom.xml file.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- 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>

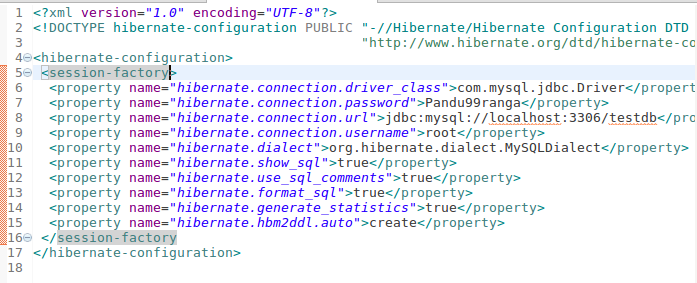
<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

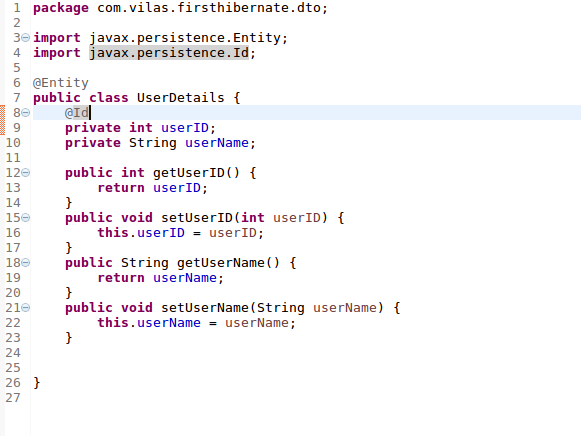
<version>5.1.38</version>

</dependency>

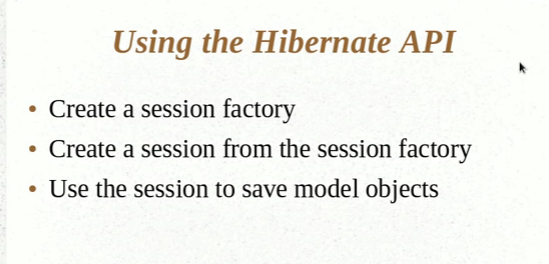
Create the Hibernate Configuration file

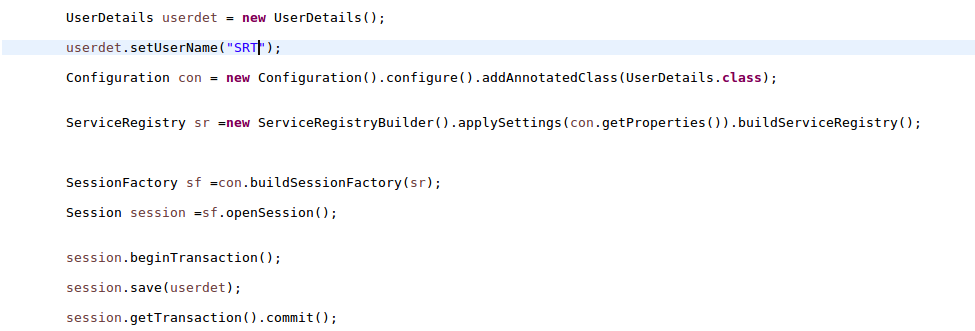


Creating the Model.



Using Hibernate API





**HBM2DDL and Name Annotations**

@Entity

@Table(name=”Custom\_Table\_Name”)

@Column(name=”Custom\_Column\_Name”)

This is used to create the table, update or create – drop

<property name=*"hibernate.hbm2ddl.auto"*>update</property>

**More Annotations:**

**@Transient:** If you would want some part of the attributes in the model to be skipped and not added we use the annotation that is @Transient.

**@Temporal:** This annotation is used to ensure that only the date is updated into the table and not the timestamp ( which includes the time / milliseconds etc)

**@Lob:**By marking the field as @Lob/@Clob or @Blob hibernate will over-ride the 255 char property.

# **Retrieving the objects from the session.get() object**

# Session Factory is created only once in an application.

# We have to use this session to being transaction and get all the details from the database.Below is the example.



**Primary Key – Hibernate**

**Natural Vs Surrogate Key.**

Every user has to provide and disctinct email id.

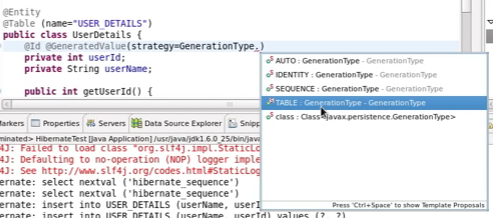
Such columns which are for business reason are called Natural Keys.

You cannot determine to mark as unique and have seperate column to mark as a key such is the surrogate key.

For Surrogate Key Hibernate can help us on the same.

For the automatic generated value we use the following annotation

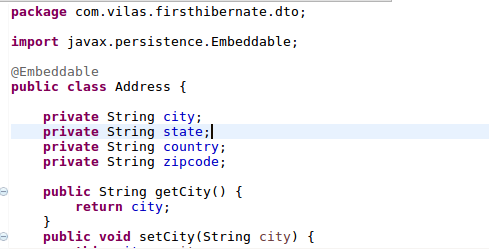
@GeneratedValue

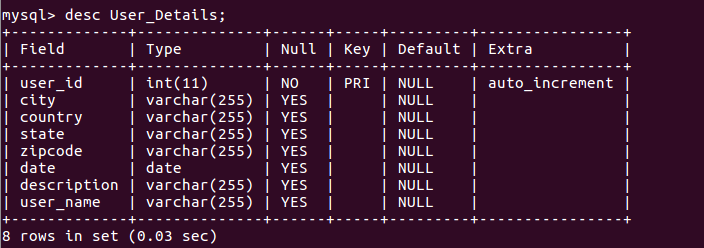


**Value Types and Embedding objects**



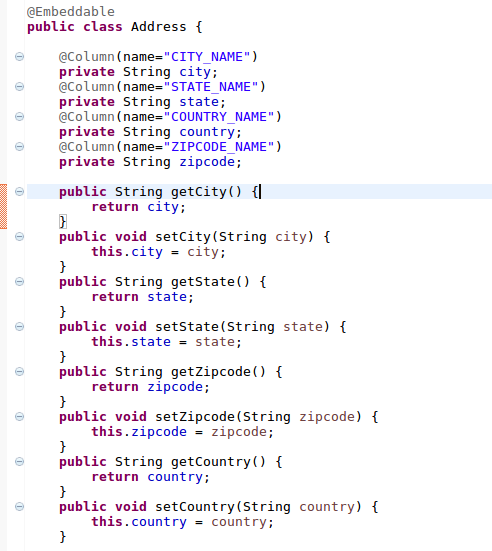
For embedding the Address class as an object in the User class we use the annotation @Embeddeble





**Attribute Overrides and Embedded Object Keys**

The column names of the Address attributes can be changed in the class



If the User\_Details has two elements

1) Home Address & 2) Office Address

When Hibernate is creating the tables it will not commit the transaction of creating the table as the fields will already be created.

In order to do that we need to use the annotation that is called as @AttributeOverrides and @AttributeOverride



# Saving Collections

# We have the USER\_DETAILS entity class.

# We would want to update the entity class with multiple Address objects

# 

# The Embeddeble class details

# 

# In the Main App

# 

# Creates that many records as tables

# 

# **Configuring Collections and Adding Keys**

# @Join\_Table(name=”User\_Address)

# private List<Address> listOfAddress = new ArrayList<Address>();

# will generate a new table called User\_Address.

# When the tables are created it shows as below:

# mysql> show tables;

# +---------------------+

# | Tables\_in\_testdb |

# +---------------------+

# | Address |

# | Alien |

# | Cart |

# | Employee |

# | Items |

# | Laptop |

# | Student |

# | UserDetails\_address |

# | User\_Details |

# | customer |

# +---------------------+

# 10 rows in set (0.04 sec)

# When the above code in the previous section is run it created the table ie., UserDetails\_address

# With @Join\_Table(name=”User\_Address”) it creates the table as User\_Address.

# For the UserID in the User\_Address it should be the following

# 

# Th

# The code for generating the above information is as follows.

# 

# When we do an desc on User\_Address table it gives the following output

# 

# It does not have an primary key . We should have the primary key defined for the the collection.

# The following annotation specific to Hibernate will solve the problem

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

@CollectionId(columns = { @Column(name="Address\_ID") }, generator = "hilo", type = @Type(type="long"))

**private** Collection<Address> address = **new** ArrayList<Address>();

# The snapshot of the code below.

# 

# The above code extract will do the following

# 1)Create the Join Table with the name “User\_Address”

# 2) The join column in the User\_Address table will be USER\_ID

# 3) The @CollectionId annotation will create and primary key based on the following annotation values.

# The Primary Key column is “Address\_ID”

# The Generator is hilo that hibernate produces.

# and the type of the key is “long”

# 4) The following annotation @GenericGenerator supports the generator.

# We are saying that we user Hilo-gen as the generator for the primary key for the Address\_ID

# the above annotations are specific to Hibernate only.

# 

# The new primary key object for the table USER\_ADDRESS is created in the form of “Address\_ID”

# Proxy Objects, Eager and Lazy.

# Lazy Initilization is you do not initilize the entire object you only linitilize the first level member variables of the object

# Then you initilize the list only when you access it.

# The opposite it of the Lazy Initilization is the Eager.

# Eager initilization will get all the values of the list from the objects.

# How this is done:

# Hibernate provides an proxy and returns the instance of the proxy.

# 

# One to One Mapping

# We have two classes Vehicle and User\_Details with an one to one mapping.

# package com.vilas.firsthibernate.dto;

# import javax.persistence.Entity;

# import javax.persistence.GeneratedValue;

# import javax.persistence.GenerationType;

# import javax.persistence.Id;

# import javax.persistence.JoinColumn;

# @Entity

# public class Vehicle {

# @Id

# @GeneratedValue(strategy=GenerationType.AUTO)

# @JoinColumn(name="Vehicle\_ID")

# private int vehicleId;

# private String vehicleName;

# public Vehicle(){}

# 

# public Vehicle(int vehicleId, String vehicleName){

# this.vehicleId = vehicleId;

# this.vehicleName = vehicleName;

# }

# public int getVehicleId() {

# return vehicleId;

# }

# public void setVehicleId(int vehicleId) {

# this.vehicleId = vehicleId;

# }

# public String getVehicleName() {

# return vehicleName;

# }

# public void setVehicleName(String vehicleName) {

# this.vehicleName = vehicleName;

# }

# }

# User\_Details Class that has the One-to-One Mapping

# package com.vilas.firsthibernate.dto;

# import java.util.ArrayList;

# import java.util.Collection;

# import java.util.Date;

# import java.util.List;

# ;

# import javax.persistence.Column;

# import javax.persistence.ElementCollection;

# import javax.persistence.Embedded;

# import javax.persistence.Entity;

# import javax.persistence.GeneratedValue;

# import javax.persistence.GenerationType;

# import javax.persistence.Id;

# import javax.persistence.JoinTable;

# import javax.persistence.OneToOne;

# import javax.persistence.Table;

# import javax.persistence.Temporal;

# import org.hibernate.annotations.CollectionId;

# import org.hibernate.annotations.GenericGenerator;

# import org.hibernate.annotations.Type;

# import javax.persistence.JoinColumn;

# @Entity

# @Table(name="User\_Details")

# public class UserDetails {

# @Id

# @Column(name="user\_id")

# @GeneratedValue(strategy=GenerationType.AUTO)

# private int userID;

# @Column(name="user\_name")

# private String userName;

# **@OneToOne**

# **private Vehicle vehicle;**

# public int getUserID() {

# return userID;

# }

# public void setUserID(int userID) {

# this.userID = userID;

# }

# public String getUserName() {

# return userName;

# }

# public void setUserName(String userName) {

# this.userName = userName;

# }

# public Vehicle getVehicle() {

# return vehicle;

# }

# public void setVehicle(Vehicle vehicle) {

# this.vehicle = vehicle;

# }

# }

# The output of the DB is:

# mysql> desc Vehicle;

# +-------------+--------------+------+-----+---------+----------------+

# | Field | Type | Null | Key | Default | Extra |

# +-------------+--------------+------+-----+---------+----------------+

# | vehicleId | int(11) | NO | PRI | NULL | auto\_increment |

# | vehicleName | varchar(255) | YES | | NULL | |

# +-------------+--------------+------+-----+---------+----------------+

# 2 rows in set (0.15 sec)

# mysql> desc User\_Details;

# +-------------------+--------------+------+-----+---------+----------------+

# | Field | Type | Null | Key | Default | Extra |

# +-------------------+--------------+------+-----+---------+----------------+

# | user\_id | int(11) | NO | PRI | NULL | auto\_increment |

# | user\_name | varchar(255) | YES | | NULL | |

# | **vehicle\_vehicleId | int(11) | YES | MUL | NULL | |**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |

# **One to Many & Many to One**

# 

# **Many to One**

# 

# 

# **Many to Many:**

# Many to Many should have mapping tables

# For the UserDetails:

# 

# Vehicle:

# 

# Main:

# 

# mysql> show tables;

# +----------------------+

# | Tables\_in\_testdb |

# +----------------------+

# | Address |

# | Alien |

# | Cart |

# | Employee |

# | Items |

# | Laptop |

# | Student |

# | USER\_VEHICLE |

# | User\_Details |

# | Vehicle |

# | customer |

# | hibernate\_unique\_key |

# +----------------------+

# 12 rows in set (0.00 sec)

# mysql> select \* from User\_Details;

# +---------+-----------+

# | user\_id | user\_name |

# +---------+-----------+

# | 1 | Kadhran |

# +---------+-----------+

# 1 row in set (0.00 sec)

# mysql> select \* from USER\_VEHICLE;

# +---------+------------+

# | USER\_ID | VEHICLE\_ID |

# +---------+------------+

# | 1 | 1 |

# | 1 | 2 |

# +---------+------------+

# 2 rows in set (0.02 sec)

# mysql> select \* from vehicle;

# ERROR 1146 (42S02): Table 'testdb.vehicle' doesn't exist

# mysql> select \* from Vehicle;

# +-----------+-------------+

# | vehicleId | vehicleName |

# +-----------+-------------+

# | 1 | Jeep |

# | 2 | Car |

# +-----------+-------------+

# Cascade Types and Other Things

# Assume that we have an annotation mapping @ManyToOne ( where we have a mappings of several vehicles to an single user.

# In this scenario if the User is not found then Hibernate will throw an error that the data is not found.

# Inorder to supress the errors we use the following annotation

# @NotFound(action=NotFoundAction.Ignore)

# The example of the snippet.

# 

# Hibernate Collections:

# Hibernate has its own collection objects

# 

# Cascade:

# For example if we have an one to many relation between the user and vehicle.

# The user has many vehicles

# We create one user and create multiple vehicles.

# WE assign the vehicles to the user

# when saving we save the user and save all the vehicle objects

# it is okay if the user is assigned one or two vehicles but when there are many vehicle objects it becomes difficult to save all.

# IN this case we use an annotation like @Cascade as in below.

# 

# WE then only save the user only.

# 

# In the above snippet whenever a persist happens the cascade happens.

# **Implementing Inheritance**

# 

# **Stratergy As Single Table**

# When you create objects for Vehicle,Two-Wheeler and Four-Wheeler. The defualt way that hibernate treats is as follows as it creates only one table.

# 

# Dtype is called as a discriminator and tells which object it belongs to which class it is.

# 

# In the above snippet we have the stratergy as Single\_Table

# In the default table the column name is shown as “dtype) which is default name.

# If we need to change the name of the column we use the annotation that is

# @DiscriminatorColumn(

# name=”VEHICLE\_TYPE”,

# discriminatorType=DiscriminatorType.String )

# In order to change the name of the class for the twowheeler to bike we set the discriminator value to the class.

# When we run the above hibernate java programme we get the following output with the new disciminator values

# 

# **Implementing INHERITANCE with Table Per Class**

# To implement the Inheritance with table per class we do the following.

# 

# We change the Stratergy of the InheritanceType as Table\_PER\_CLASS

# so when other classes inherit from this master table vehicle

# it created seperate tables in the database with different class names.

# Third Way to Implement Inheritance ( Stratergy – Joined)

# WHAT EVER PROERPTIES THAT ARE INHERITED FROM THE PARENT WILL REMAIN IN THE PARENT TABLE.

# The additional fields after inheriting will be extrapolated into the new table.

# The following is the snippet of the Joined Stratergy.

# 

# In the database for the vehicle.

# 

# 

# Only the new fields are populated in the table ( fourWheeler) that is inherited from the vehicle table.

# In order to see the join we use the following SQL statement to check the same

# 

# **CRUD Operations**

**package** com.vilas.hibernatecrud;

**import** java.util.List;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

**import** org.hibernate.service.ServiceRegistry;

**import** org.hibernate.service.ServiceRegistryBuilder;

/\*\*

\* Hello world!

\*

\*/

**public** **class** App

{

**public** **static** **void** main( String[] args )

{

Configuration conf =**new** Configuration().configure().addAnnotatedClass(UserDetails.**class**);

ServiceRegistry sr =**new** ServiceRegistryBuilder().applySettings(conf.getProperties()).buildServiceRegistry();

SessionFactory sf = conf.buildSessionFactory(sr);

Session session =sf.openSession();

session.beginTransaction();

//Creating users

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

{

UserDetails userdet = **new** UserDetails();

userdet.setUserName("User"+i);

session.save(userdet);

}

session.getTransaction().commit();

session.close();

//Get the users

session =sf.openSession();

session.beginTransaction();

List<UserDetails> userdetails = session.createCriteria(UserDetails.**class**).list();

userdetails.forEach(user -> System.***out***.println(user));

session.close();

//Delete the user

session = sf.openSession();

session.beginTransaction();

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

session.delete(user);

UserDetails user2 =(UserDetails)session.get(UserDetails.**class**, 4);

user2.setUserName("Updated User");

session.update(user2);

session.getTransaction().commit();

session.close();

//Get the users after delete

System.***out***.println("/n");

System.***out***.println("/n");

//Get the users

session =sf.openSession();

session.beginTransaction();

List<UserDetails> userdetails1 = session.createCriteria(UserDetails.**class**).list();

userdetails1.forEach(user1 -> System.***out***.println(user1));

session.close();

}

}

# **Transient , Persistent and Detached Objects**

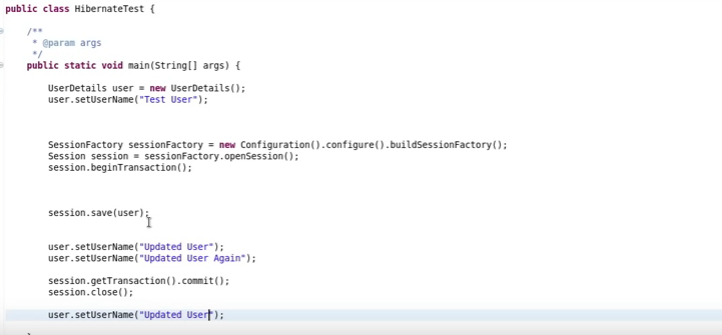
# Before the object is actually handed over to Hibernate it is an transient object.

# Once the Object is handled by hibernate it is an persistent object.

# And any updates to the object after that is handled by Hibernate.

# Once the session is closed it becomes a detached object.

# Detached object is it was tracked by hibernate before and closed by hibernate before and now it no longer taken care by it.



# In the above example the user created before the session object is created it is an “Transient” object

# When the object is created and is handed over to hibernate it becomes “Persistent” object

# Once the session is closed it becomes “detached” and is no longer handled by Hibernate.

# **Understanding State Changes**

# 

# 

# 

# For read we do not have a new() we only get the objects that are there in the database().

# 

# 

# **Persisting Detached Objects**

# In the below example we close the session after getting the user details based on the userid “1”.

# We are then setting the user.

# 

# **We** open the session and update the user.

# IF we are not updating anything to the user hibernate still runs the update query.

# To avoid this we need to tell hibernate that we should run the update query only when there is a change in the state of the object.

# To do this we need to update the Entity Class with the hibernate related entity annotation.

# Which is as in the below screen shot

# The annotation name is @org.hibernate.annotations.Entity(selectBeforeUpdate=**true**)

# 

# **HQL and Query Object**

# IT si very straight forward.

# In HQL we query on the class name rather than the name of the query and the property name rather than the column name in the table.

# The example as below:

# 

# **Select and Pagination in HQL**

# 

# **Select statements can be used in Hibernate in the following way.**

# 

# **Parameter Binding and SQL Injection**

# For Parameter binding can be done in two ways

# 1) in the Position holding with an “?” symbol

# The following is the example:

# 

# **2)** Another way is to have the field names directly.

# You put the userId and the userName directly into the substituion equation as below.

# 

# **Named Queries**

# Consolidate all teh queries to a particular place than they are scattered all over the place

# To ensure that that named query is maintained we use the annotation

# @NamedQuery(name=”userDetails.byID”, query=from UserDetails where userID =?”)

# The extact of the above code is as bleow screen shot.

# 

# The annotation @NamedQuery is used to get the UserDetails by ID and uses the HQL lang that is query=”from UserDetails where userId= ?”)

# It also has the the Native query that we use the annotation for the executing the SQL statemetns

# the annotation is

# @NamedNativeQuery(name=”UserDetails.byName”, query=”select \* from User\_Details where username =?”)

# we can also specify the result query class which is

# @NamedNativeQuery(name=”UserDetails.byName”, query=”select \* from User\_Details where username =?”, resultClass=UserDetails.class).

# We can then use the native query in the session.

# session.getNamedQuery(“UserDetails.byName”);

# 

# **Introduction to Cirteria API**

# 

# Restrictions are the ways we can get the data and it will return the data in the way we want.

# In the above example we are getting the details of the user with user 10

# **Understanding Restrictions:**

# With restrictions we can get all the users in the database whose userids are greater than 5

# Restrictions can be chained as well.

# The .add is an conjunction.

# eg:

List<UserDetails> userdetails = session.createCriteria(UserDetails.**class**).add(Restrictions.*gt*("userID", 5)).list();

userdetails.forEach(user -> System.***out***.println("All the users with user id after 5"+user));

# **.add is an conjunction** it will add to the query using an “and” keyword.

# Like select \* from User where user\_name like “%sp” and user\_id between ( 5, 10).

# **For the OR clause we can use the following example**

# List<UserDetails> userdetails = session.createCriteria(UserDetails.class).add(Restrictions.or(Restrictions.between("userID", 0, 3), Restrictions.between("userID", 7, 10))).list();

# 

# userdetails.forEach(user -> System.out.println("All the users with user id between 0 and 3 & UserIDs between 7 & 10"+user));

# **Projections and Query By Example**

# **Projects are used to prepare aggregations functions.**

**if we want that the object loads only two fields from database instead of all fields. Here we can make use of Hibernate Projections. It can be applied to Criteria Query and can result in an object loaded with only those fields which you need ie. can load partial objects.**

There are 2 main things to remember:

* Projection is an Interface given in “org.hibernate.criterion” package
* Projections is the class responsible for producing Projection objects.It has all static methods

Example of the Projection.

The below example produces only the list of userId’s that we are iterating and displaying

List<UserDetails> userdetails = session.createCriteria(UserDetails.**class**).setProjection(Projections.*property*("userID")).addOrder(Order.*desc*("userID")).list();

//userdetails.forEach(user -> System.out.println("All users in desc"+user.getUserID()));

**for**( Iterator it = userdetails.iterator(); it.hasNext();){

String userid =it.next().toString();

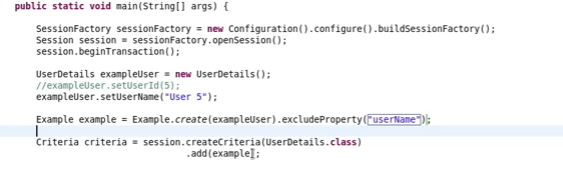
System.***out***.println(" The userID is " + userid);

}

**Query BY Example**

There will be a scenario where there will be multiple properties that we need to use say userName, userID, address, pincode and etc.

IN this case we can create an object with all these ids and pass the object as an example to the criteria.



# Here in the above exampe we created an Example object using the create() method.

# We then created the criteria and added the example to the session.

# **If you want you can use the exclude Property to exclude the properties that are not needed to be persisted.**

# For example we need to pull up all the users that are like 1% we do the following query.

# 

# We can use the command that is enableLike() method to pull all the users that are like 1%.

# **Cacheing In Hibernate**

# 

# First Level Cache.

# If we notice the below example we are running the same query multiple times after the session is closed. If we are going to the database and getting the same result instead of connecting to the cache.

# Configuring the Second Level Cache.

# To enable the second level cache We need to have the following configurations set in the hbmconfiguration.xml file.

# 

# In the actual class User\_Details we do the following

# 

# We have used the annotation @Cacheable

# and @Cache(usage=CacheConcurrencyStrratergy.Read\_only) <\*\* Please read the cache concurrency stratery about caching in hibernate)

# We have enabled the second level cache by opening the tags in hibernatecfg.xml file.

# Downloading the jars related to ehcache provider and adding them into the class path and then executing the above.

# **Using Query Cache**

# Hibernate differentiates queries seperately.

# We need to set the property to enable the query cache to **true.**

# 

# The query cache is different from the second level cache

# - Session cache is defacult

# - Second cache – is the second level cache

# - Third – is the query cache.

# In the second level cache we went to entity and made it cacheble.

# For the query also set the query as cacheble.

# IN the below example we have done a query cached by

# 1) Creating the query

# Query query = session.createquery( “ from User\_Details user where user.userId=1”);

# query.setCacheble(true);

# 2) Close the session.

# 

# 3) Create an new Session

# 4) Create a new query Query query2 = session2.createQuery(“from UserDetails user where user.userId = 1”);

# 5) query2.setCacheable(true);

# When we execute the java applicaiton it gives only one query.

# ----Vilas --- Hibernate --- Basics--- Done---- 21/08/2017-----------