**Hibernate**

1 What is hibernate

2 Benefits

3 Code snippet

**Hibernate**

A frame work for persisting/ saving java object in DB

Saving and retrieving from DB

Persistent object are instance of POJO class that represent rows in table in db

**Benefits**

1 handle all low level sql

2 minimize JDBC code

3 provide ORM ( store and retrieve object from DB)

**ORM**

Developer defines mapping between java class and DB table

Mapping need to be done through configuration either through XML or java annotation

**Relation Between Hibernate and JDBC**

- hibernate user JDBC for all DB communications

- hibernate is an another extraction layer on top of jdbc

- using hibernate API

- hibernate does all low level JDBC work in background but in background all goes through standard JDBC API

**Environment Setup**

Note: Hibernate 5.2 requires Java 8

Steps of Environment

1 Java JDK

2 IDE

3 database server

4 hibernate jar and JDBC driver

1 Install MYsql / postgress /oracle

2 setup up DB -> create user-> grant PRIVILEGES-> create DB and Schema -> create Table

CREATE USER 'hbstudent'@'localhost' IDENTIFIED BY 'hbstudent';

GRANT ALL PRIVILEGES ON \* . \* TO 'hbstudent'@'localhost';

CREATE DATABASE IF NOT EXISTS `hb\_student\_tracker`;

USE `hb\_student\_tracker`;

CREATE TABLE student (

id serial PRIMARY KEY,

first\_name varchar(45) DEFAULT NULL,

last\_name varchar(45) DEFAULT NULL,

email varchar(45) DEFAULT NULL

)

**Serial for auto increment**

**Setup hibernate in Eclipse**

1 create eclipse project

-new Java Project -HibernateLearning -- new folder lib add all hibernate n jdbc jars

2 download hibernate files

-https://hibernate.org/ download Hibernate ORM 5.4.9

- extract zip under lib copy all jars in required folder to project lib basic required jars

3 download driver as per DB

4 add jar files build path

**Write JDBC code to check working or not**

**public** **class** TestJdbc {

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

String jdbcUrl="jdbc:postgresql://localhost:5432/personal";

//String jdbcUrl="jdbc:postgresql://localhost:5432/personal?useSSL=false"; // to bypass SSL warings

String userName="postgres";

String passWord="sachin";

**try**

{

System.***err***.println("Connecting Data base: "+ jdbcUrl);

Connection con=DriverManager.*getConnection*(jdbcUrl, userName, passWord);

System.***err***.println("Connection Successful");

}

**catch** (Exception e) {System.***err***.println(e.getMessage());}

}

}

**Hibernate Development process**

1 add Hibernate Config file

- basically tell hibernate how to connect to the data base

-hibernate use jdbc in back end to communicate

Contains JDBCurl,userid,pwd,coonection pool etc

- create / paste hibernate.cfg.xml file in src because its actual class path off application

2 Annotate java Class

3 develop java code perform DB operations

**1 hibernate.cfg.xml**

<!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>

<!-- JDBC Database connection settings -->

<property name=*"connection.driver\_class"*>org.postgresql.Driver</property>

<property name=*"connection.url"*>jdbc:postgresql://localhost:6412/personal</property>

<property name=*"connection.username"*>postgres</property>

<property name=*"connection.password"*>postgress</property>

<!--<property name="connection.password">sachin</property>-->

<!-- JDBC connection pool settings ... using built-in test pool -->

<property name=*"connection.pool\_size"*>1</property>

<!-- Select our SQL dialect -->

<property name=*"dialect"*>org.hibernate.dialect.PostgreSQLDialect</property>

<!-- Echo the SQL to stdout -->

<property name=*"show\_sql"*>true</property>

<!-- Set the current session context -->

<property name=*"current\_session\_context\_class"*>thread</property>

</session-factory>

</hibernate-configuration>

**Session-Factory provides session object for connecting db**

Jdbc connection stpes

<property name=*"connection.driver\_class"*>org.postgresql.Driver</property>

<property name=*"connection.url"*>jdbc:postgresql://localhost:6412/personal</property>

<property name=*"connection.username"*>postgres</property>

<property name=*"connection.password"*>postgress</property>

Tells hibernate to connect jdbc

Connection pool will be set

<property name=*"connection.pool\_size"*>1</property>

Dialect

<property name=*"dialect"*>org.hibernate.dialect.PostgreSQLDialect</property>

Show sql

<property name=*"show\_sql"*>true</property>

Current session context class will make use thread model

<property name=*"current\_session\_context\_class"*>thread</property>

**This are basics for session can have more but these are key elements of session for hibernate connection and execution**

**2 Annotaion of java class**

Entity class: java class that is mapped to database table

Java class with fields and getter and setter that is mapped to db tables need to add annotations to map with DB tables

Two options for mapping Entity to DB table

1 XMl config file

2 java annotations

Mainly concentrate on Java Annotaion

**A Java annotation**

Mapping steps

1 map class to Db table

2 map fields to database column

**I Map class to DB**

Adding @Entity means table which need to be mapped

Which need to mapped with DB table

@Table actual name of db to map to class

**II Map fields to DB table**

@Id is primary key

@column maps the column and field

If both db name and entity name same no need to add 2column

Create a java class and annotate it

-**Entity pkg creation**

**-Create Student Class**

**-Imports from persistence package only because standard API for hibernate impl**

**-Define no-arg constructor/ default constructor its optional if we don’t JVM will create. we can create field constructor also for other kind of example, id is not required in constructor field since its auto increment**

**-Getter and setter method compulsory**

**-If debug required use tostring method in entity**

**package** com.java.demo.entity;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

**import** javax.persistence.Table;

@Entity

@Table(name="student",schema = "basicdb") // step 1 map class to DB table

//in case postgress if i dnt give schemane it will take public schema of that DB create tables

//and insert it so schema is important

**public** **class** Student {

//step 2 map feild to db columns

@Id // id to represnt it primary key

@GeneratedValue(strategy=GenerationType.***IDENTITY***) // if not generate seuqunec if will give duplicate and fail to save in db

@Column(name="id")

**private** **int** id;

@Column(name="first\_name")

**private** String firstName;

@Column(name="last\_name")

**private** String lastName;

@Column(name="email")

**private** String email;

**public** Student() {

// **TODO** Auto-generated constructor stub

}

**public** Student(String firstName, String lastName, String email) {

**super**();

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.email = email;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getFirstName() {

**return** firstName;

}

**public** **void** setFirstName(String firstName) {

**this**.firstName = firstName;

}

**public** String getLastName() {

**return** lastName;

}

**public** **void** setLastName(String lastName) {

**this**.lastName = lastName;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

@Override

**public** String toString() {

**return** "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + ", email=" + email + "]";

}

}

**Note :If you are using Java 9, 10 or 11, then you will encounter an error when you run your Hibernate program.**

**Error: Exception in thread "main" java.lang.NoClassDefFoundError: javax/xml/bind/JAXBException**

This happens because of Java 9 and higher.

Java 9 and higher has removed java.xml.bind from its default classpath. That's why we get the class not found exception.  We have to explicitly add JAR files to the build path.

***Solution***

For Java 9 and higher, you need to additional JAR files.

You need to download the following JAR files:

javax.activation-1.2.0.jar  
jaxb-api-2.3.0.jar  
jaxb-core-2.3.0.jar  
jaxb-impl-2.3.0.jar

The Generate toString() process fails in Eclipse with Java 9.

you will need to manually write the code for two string

**Why we are using JPA Annotation instead of Hibernate ?**

JPA is a standard specification. Hibernate is an implementation of the JPA specification.

Hibernate implements all of the JPA annotations.

The Hibernate team recommends the use of JPA annotations as a best practice.

**Hibernate Crud feature**

**1 Save java object using hibernate**

**A add hibernate config file**

**B Annotate java class**

**C develop java code to perform db operations**

**Fisrt we need to be aware of two key playes of hibernate**

1 **SessionFactory** : Reads the hibernate config file , create session Objects, heavy weight objects, only create once in your app

* This reads the config file get conection to DB and will create session obj for us session factory is heavy weight object only once created in our app we use it again n again . once sessionfactory created it creates session

2 **session** : wraps JDBC connection main obj for save/ retrieve obj, short lived obj, retrieve from session factory

* For a given method you will use it and throw it away so its short lived again we go back for another session use it throw it away

**Demo Code**

**package** com.java.hibernate.demo;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

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

**import** com.java.demo.entity.Student;

**public** **class** StudentDemo {

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

// create Session factory

// will be creatred once and "hibernate.cfg.xml" is not reuired if cfg is in src root

// bcz it will automatically search for this config file name in src

SessionFactory factory= **new** Configuration().configure("hibernate.cfg.xml")addAnnotatedClass(Student.**class**).buildSessionFactory();

// create session

// can get session from session factory

Session session=factory.getCurrentSession();

**try** {

// use session object to save / retive

//create student object

System.***out***.println("Creating new Student Object :");

Student student=**new** Student("sachin", "hs", "sachin19927@gmail.com");

//start a transaction

session.beginTransaction();

// save transation

System.***out***.println("saving Student Object");

session.save(student);

// commit tranaction

session.getTransaction().commit();

}

**catch** (Exception e) {

System.***err***.println(e.getMessage());

}

**finally** {

factory.close();

}

}

}

**Primary Key**

Uniquely identifies each row in a table, must be unique value and cant be null

**@Id in annotation tells that this field is primary key**

We can leave auto generation id to DB or we can explicitly do it we have to give generation strategy, if we don’t specify by default it will use appropriate strategy for DB impl

**By default it use Identity strategy if we dnt declare anything**

**@GeneratedValue(strategy=GenerationType.*IDENTITY*)**// postgres will handle auto increment

Id generation strategies

GenerationType.***AUTO :pick an appriate strategy for particular db***

GenerationType.***IDENTITY : assiagn primary key using db identity column***

GenerationType.***SEQUENCE : assign primary key using db seqeunce***

GenerationType.***TABLE : assign primary key using an underlaying db table to ensure uniqueness***

***We can write our own custom generation strategy***

***Create a subclass org.hibernate.id.SequenceGenerator***

***Override method public Serializable generate() add oue business logic***

***Make sure : 1 always generate unique value, 2 work on high –volume multi thread environment ,3 if its server cluster always use generate unique value***

How to change AUTO\_INCREMENT values

Alter AUTO\_increment value in DB so which has to max value if id in that table so it will continue form new AUTO increment value

To reset it one by deleting all records should go for truncate which dele data and rest sequence also

**package** com.java.hibernate.demo;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

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

**import** com.java.demo.entity.Student;

**public** **class** PrimaryKeyDemo {

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

SessionFactory factory=**new** Configuration().configure().addAnnotatedClass(Student.**class**).buildSessionFactory();

Session session=factory.getCurrentSession();

// add mulpile studnts

**try**

{

//create 3 student object

System.***out***.println("Creating new Student Object :");

Student student1=**new** Student("sachin", "gowda", "sachin19927@gmail.com");

Student student2=**new** Student("soumya", "indi", "sachin19927@gmail.com");

Student student3=**new** Student("darshan", "gowda", "sachin19927@gmail.com");

session.beginTransaction();

session.save(student1);

session.save(student2);

session.save(student3);

session.getTransaction().commit();

}

**catch** (Exception e) {

System.***err***.println(e.getMessage());

}

**finally** {

factory.close();

}

}

}

**READ Object from hibernate**

**CRUD**

**Create obj, read obj , update obj, delete obj**

try {

System.out.println("Creating new Student Object :");

Student student=new Student("sachin", "hs", "sachin19927@gmail.com");

session.beginTransaction();

session.save(student);

session.getTransaction().commit();

//fin out studnt ID : priamry key id is auto increment id cloum will be update once we save it

System.err.println("Save student generate ID :"+student.getId());

// now get new seesion and start trnaction

session = factory.getCurrentSession();

session.beginTransaction();

// retives studnt based on : primary key

System.err.println("getting student with id :"+ student.getId());

Student mystStudent=session.get(Student.class,student.getId());

// if primary key invalid result is null bcz not found

System.err.println("Get complete :"+mystStudent);

// commit transaction

session.getTransaction().commit();

}

catch (Exception e) {

System.err.println(e.getMessage());

}

finally {

factory.close();

}

**Querying Object with demo**

try {

//start a transaction

session.beginTransaction();

// query studnet

List<Student> myStudents=session.createQuery("from Student").list(); // .list()/ .getResultList

/\*

\* //display studnet

\* for (Student student : myStudents) {

\* System.err.println(student); }

\*/ // same extracketrd into method-> displayStudents

//display studnet

System.out.println("student whose last name is hs");

displayStudents(myStudents);

// query studnt with last name 'hs'

myStudents=session.createQuery("from Student s where s.lastName='hs'").getResultList();

displayStudents(myStudents);

// commit tranaction

session.getTransaction().commit();

}

catch (Exception e) {

System.err.println(e.getMessage());

}

finally {

factory.close();

}

Extract Method

private static void displayStudents(List<Student> myStudents) {

for (Student student : myStudents) {

System.err.println(student);

}

}

**To make a loop as method with refactor**

**Select code right click refactor -. Extract method**

If you are using Hibernate 5.2 or higher, then the Query list() method has been deprecated.

In your code you should make the following update:

**Replace**

session.createQuery("from Student").list()

**With**

session.createQuery("from Student").getResultList()

**Update Object**

try {

session.beginTransaction();

System.err.println("getting student with id :2");

Student mystStudent=session.get(Student.class,2);

System.err.println("updating student :"+mystStudent );

mystStudent.setFirstName("sachin");

mystStudent.setLastName("hs gowda");

// till now it is updated in memory

session.getTransaction().commit();

// once we commit it will update in DB no need to save explictly save or update by simply commit it will update

// since it is a persist object

//bulk update with condition

session=factory.getCurrentSession();

session.beginTransaction();

// update email for all students

session.createQuery("update Student set email='sachin.hs@bcits.in'").executeUpdate();

session.getTransaction().commit();

System.err.println("done") ;

}

catch (Exception e) {

System.err.println(e.getMessage());

}

finally {

factory.close();

}

**Delete Object**

try {

Session session=factory.getCurrentSession();

session.beginTransaction();

System.err.println("getting student with id :2");

Student mystStudent=session.get(Student.class,2);

System.err.println("deleting student :"+mystStudent );

// deleting student

session.delete(mystStudent);

session.getTransaction().commit();

// delete the object

session=factory.getCurrentSession();

session.beginTransaction();

session.createQuery("delete from Student where firstName='darshan'").executeUpdate();

session.getTransaction().commit();

System.err.println("done") ;

}

catch (Exception e) {

System.err.println(e.getMessage());

}

finally {

factory.close();

}

**How To View Hibernate SQL Parameter Values**

However, for debugging your application, you want to see the actual parameter values in the Hibernate logs. Basically, you want to get rid of the question marks in the Hibernate logs.

You can view the actual parameters by viewing the low-level trace of the Hibernate logs. This is not set up by default. However, we can add log4j to allow us to see these low-level logs.

**Here is an overview of the process:**

**1. Add log4j to your project classpath**

**2. Add log4j.properties to your “src” directory**

**1. Add log4j to your project classpath**

**1a. Download log4j v1.2.17 from this link: –**[**http://central.maven.org/maven2/log4j/log4j/1.2.17/log4j-1.2.17.jar**](http://central.maven.org/maven2/log4j/log4j/1.2.17/log4j-1.2.17.jar)

**1b. Copy this file to your project’s lib directory**

**2. Add log4j.properties to your “src” directory**

**2a. Copy the text from below**

# Root logger option

log4j.rootLogger=DEBUG, stdout

# Redirect log messages to console

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

log4j.logger.org.hibernate=TRACE

**2b. Save this file as "log4j.properties" in your “src” directory**

Note: This file has an important setting:

log4j.logger.org.hibernate=TRACE

This allows you see a low-level trace of Hibernate and this allows you see the real SQL parameter values.

This allows you see a low-level trace of Hibernate and this allows you see the real SQL parameter values.

Now run your application. You will see a lot of low-level TRACE logs in the Eclipse Console window.

Right-click in the Eclipse Console window and select **Find/Replace…**

Search for: **binding parameter**

or search for: **extracted value**

**Handling Dates with Hibernate**

You can make use of a combination of Java's date formatting class and Hibernate annotations.

**Development Process Overview**

1. Alter database table for student  
2. Add a date utils class for parsing and formatting dates  
3. Add date field to Student class  
4. Add toString method to Student class  
5. Update CreateStudentDemo

1 ALTER TABLE student

ADD COLUMN date\_of\_birth DATE NULL AFTER last\_name;\

2. Add a date utils class for parsing and formatting dates

We need to add a DateUtils class to handle parsing and formatting dates. The source code is here. The class should be placed in the package: com.luv2code.hibernate.demo.

The date formatter uses special symbols for formatting/parsing.

-  dd:  day in month (number)  
-  MM:  month in year (number)  
- yyyy: year

public class DateUtils {

// The date formatter

// - dd: day in month (number)

// - MM: month in year (number)

// - yyyy: year

//

// See this link for details: https://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html

//

//

private static SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");

// read a date string and parse/convert to a date

public static Date parseDate(String dateStr) throws ParseException {

Date theDate = formatter.parse(dateStr);

return theDate;

}

// read a date and format/convert to a string

public static String formatDate(Date theDate) {

String result = null;

if (theDate != null) {

result = formatter.format(theDate);

}

return result;

}

}

---

3. Add date field to Student class

We need to add a date field to the Student class. We map this field to the database column, "date\_of\_birth". Also, we make use of the @Temporal annotation. This is a Java annotation for storing dates.

@Column(name="date\_of\_birth")

@Temporal(TemporalType.DATE)

private Date dateOfBirth;

String theDateOfBirthStr = "31/12/1998";

Date theDateOfBirth = DateUtils.*parseDate*(theDateOfBirthStr);

Student student = **new** Student("Paully", "Doe", "paul@luv.com", theDateOfBirth);

//start a transaction

session.beginTransaction();

// save transation

System.***out***.println("saving Student Object");

session.save(student);

// commit tranaction

session.getTransaction().commit();