Hibernate:

Why Hibernate:

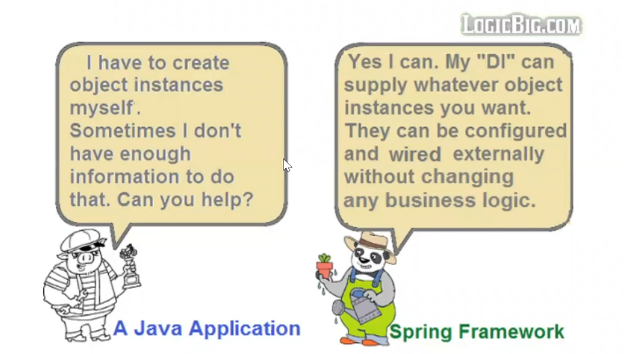
* Directly storing the Java Objects into the Database table.
* No need to write boilerplate code coding to perform CRUD operations.
* Managing Data Access Layer using object Oriented way.
* No need to deal with mapping ResultSet to our domain class object.
* Not required to handle too many exceptions.

| **JDBC** | **Hibernate** |
| --- | --- |
| In JDBC, one needs to write code to map the object model’s data representation to the schema of the relational model. | Hibernate maps the object model’s data to the schema of the database itself with the help of annotations. |
| JDBC enables developers to create queries and update data to a relational database using the Structured Query Language (SQL). | Hibernate uses HQL (Hibernate Query Language) which is similar to SQL but understands object-oriented concepts like inheritance, association etc. |
| JDBC code needs to be written in a try-catch databases block as it throws checked exceptions (SQLexception). | Whereas Hibernate manages the exceptions itself by marking them as unchecked. |
| JDBC is database dependent i.e. one needs to write different codes for different database. | Whereas Hibernate is database-independent and the same code can work for many databases with minor changes. |
| Creating associations between relations is quite hard in JDBC. | Associations like one-to-one, one-to-many, many-to-one, and many-to-many can be acquired easily with the help of annotations. |
| It is a database connectivity tool. | It is a Java framework. |
| Lazy Loading is not supported. | Lazy Loading is supported. |
| It has low performance than Hibernate. | It has high performance. |
| One needs to maintain explicitly database connections and transactions. | It itself manages its own transactions. |
| It has a dedicated customer support service system. | Waiting time is more for any answer to an issue. |

Why Spring Boot:

* Provides a radically faster and widely accessible ‘getting started’ experience for all spring development.
* No clumsy XML Configuration by developers.
* Provide opinionated ‘starter’ POMs to simplify your Maven configuration
* Uses project management tools such as MAVEN and GRADLE
* Helps fast development and production ready code
* Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
* In memory DB

What is Dependency Injection:



**MAVEN:**

--Maven is one type of build tool.

Build Process in Java based project/ application management:

----------------------------------------------------------------------------------

Developer duty/ responsibility to develop a Java based business application:

1.write some source code (bunch of classes, interfaces ..)

2.add some external jar files to the class path (dependencies of our application) in JDBC DAO project, driver jar file is the dependency of our project.

3.compile the code.

4.prepare some test cases (Unit test, Junit, Mockito, Sonar)

5.add Junit/ Mockito related jar files inside the class path.

6.compile and run the test cases.

7.arrange our code in a standard folder structure.

Java based web application:

\*.java

\*.class

\*.jar

\*.html

\*.css

\*.js

\*.xml files

\*.mp3, mp4, jpg, gif

8.Do the packing: build the jar, war file.

9.Deploy this jar file/ war file to the server.

--if any mistake identified then developer performs above task again and again.

Build process: keeping our project ready for execution/ release is called build process.

--manually doing this build process will increase the burden in developer

--to automate this build process of java application, apache foundation released a tool called ANT (Another Neet Tool).

The problem with ANT tool is:

1.ANT does not have the capabilities of downloading the required jar files from internet automatically.

2.It cannot prepare, compile and run the test cases automatically.

3.In ANT we need to write a build. xml file which is very lengthy xml file.

As a solution Apache foundation released another build tool called MAVEN.

--Gradle

--we can develop java application using Maven by 2 ways:

1.using console based (here we need to download the maven software and install it in our local computer, and then we need to set path and some environment variable to develop java application)

>mvn

--after that using some command in command prompt we can generate the template of our java application without any IDE.

2.using IDE: here we don’t need to install any kind of extra software.

STS has in built support of Maven and Gradle

Maven terminology:

---------------------------

1.artifact:

--An artifact is an outcome in maven, it can be a file, .class file or a jar file, war file, ear file, etc.

2. archetype:

--It is project template for creating similar type of project in maven.

3.groupid:

--It is an ID used to identify the artifacts of a particular organization (naming convention is similar to package name) com.masai

4.artifactid: It is the id for the final outcome (artifactid name will be the root folder of our application)

5.pom.xml: (project object model): all the information will be their in this file.

It defines following properties for a project:

1.Name

2.version

3.packing (jar, war, ear)

4.dependencies: required jar files.

5.plugins: will enhance the functionality of our project.

Docker

Jenkins (CI/ CD)

Maven Repository:

--------------------------

--a repository is a store where maven maintains plugins, archetypes, and lots of jar files used for building different kinds of java projects.

Maven repositories are of 3 types:

1.central repo: It is the maven’s own repo in which it maintains all kinds of project related plugins, archetypes and jar files for multiple projects with in organization. Ex: masairepo

<https://repo1.maven.org/maven2>

2.remote repo: this repo id maintained with in the organization for sharing plugins, archetypes and jar files for multiple projects with in organization. Ex: masairepo

3.local repo: this repo will be created inside the developer computer. (.m2) is the name for this repo.

Mysql-connector.jar:

Pom.xml:

Package.json:

.m2 ---

Maven build life cycle:

------------------------------

Maven build life cycle contains different phases:

1.validate: in this phase it will verify the project directory structure is valid or not. And it has pom.xml file is there or not.

2.compile: maven compiles all the source code of the project by downloading and adding required jar files in the class path.

3.test-compile: if we have written any unit test cases those code will be compiled.

4.test: maven will run all the test cases and it will show how many test cases are success and how many fails.

5.package: maven will bundle our java code into a jar file inside ‘target’ folder.

6.install: that jar file in step5 will be stored in the local repo.

7.deploy: maven stores the application jar file to the central repo.

8.clean: here maven will delete and remove all the files that are generated in previous build. this phase is an isolated phase.

NOTE: If we execute any phase to build the maven project then maven will execute all the phases till that given phase, except phase 8.clean

>mvn clean: remove and delete previous build

>mvn test: till the test phase.

>mvn deploy:

>mvn validate:

--package.json--- pom.xml

Src/main/java: --here we need to place our source code

Src/main/resources: any xml file, properties files, text files

Src/test/java: --here we need to place our source code to unit testing

Src/test/resources: any xml file, properties files, text files to unit testing

After creating a maven project we need to change the java version of the maven project from jdk5 to jdk8

1.create the maven by skipping architype selection

--How to create maven project:

New -> project -> maven project -> enter artifact Id, Group Id -> Finish

2.change the java version from 1.5 to 1.8

Pom.xml

For adding properties to xml file

\*\*maven compiler source – Google search

<https://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html>

properties:

1. <properties>
2. <maven.compiler.source>1.8</maven.compiler.source>
3. <maven.compiler.target>1.8</maven.compiler.target>
4. </properties>

After pasting the above tag inside the pom.xml file we need to update our maven project:

--right click on the project -> maven -> update maven project. (select force update)

--to build our maven application: -> right click on the project -> run as -> maven build.. -> inside Goal type: clean package -> run

3.take a <dependencies> tag and inside this tag paste any dependency jar file. If it is required.

Mysql Connector (maven):

Google Search – mysql connector mvn

--check mysql version 8.0.31

<https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.31>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->

<dependency>

<groupId>mysql</groupId>

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

<version>8.0.31</version>

</dependency>

Layered Architecture to develop Java based Business application:

--------------------------------------------------------------------------------------

1.maintaining the business data in secure and easily retrival manner.

--the logic that we write to implement this part of business application is known as Data Access Logic.

2.processing the data according to the business rule.

--the logic that we write to implement this part of business application is known as Business/ Service logic.

3.presenting the data to the user in user-understandable format.

--the logic that we write to implement this part of business application is known as presentation logic.

--the above 3 logics is required for almost every business application.

Class MyBusinessApplication {

//Data Access Logic //fetch the account information from the DB

//Business Logic //calculating the Interest to the balance

//Presentation Logic //display the information to the client

}

NOTE:

We can write all the 3 logics to develop a business application in one program/ class itself, if we do so, the following problem we will face:

1.all the logics to develop the application will be mixed up with each other (no clear code separation)

2.modification done in one logic may affect the other logic.

3.logics will depend upon each other so, parallel development will not be possible.

4.testing each logic is also will become complex.

--to solve this problem, a java based business application, we divide into 3 logical partition:

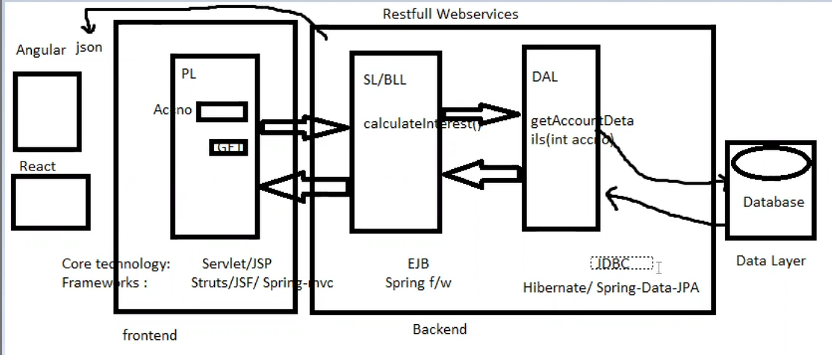
And each part we say as a layer:

1.presentatio layer (UI layer)

2.Business layer (service layer)

3.Data Access later

Layered Architecture:



Struts from: APACHE foundation

JSF from: Oracle Corporation

Spring – mvc:

--a business application will be divided into the logical partition depending upon the role played by each.

--logical partition of a business application is known as layer.

Presentation Layer: Dynamic web page (Front end)

--------------------------

Core Technology: Servlet/ JSP (Java Server Pages)

--It is set of java classes, which are responsible for generating user input screen and response page (output screen) is known as PL.

--this layer provides the intraction with the end – user.

Business Logic Layer/ Service Layer:

-----------------------------------------------

Core Technology: Enterprise Java Bean

--programatical implementation of business rule of a business organization is nothing but business logic.

--a collection of java classes whose methods have business logic to process the data according to the business rule is known as SL/ BLL.

Data Access Layer:

-------------------------

Core Technology: JDBC (Java Data Base Connectivity)

--a set of java classes whose methods are exclusively meant for performing CRUD operation with the DB server is known DAL.

Using JDBC and DAO pattern

Note: To communicate among these layers loose coupling should be promoted.

Developing Data Access Layer using ORM (Object relational mapping) approach:

-----------------------------------------------------------------------------------------------------------

Java persistence:

-----------------------

--the process of saving/ storing objects state into the DB software known as java persistence.

--for small application we can store business data (java object state) in the files using IO streams (serialization and deserialization approach).

--the logic that write to store java objects (which is holding business data) into the file using IO streams is known as “IO stream based persistence logic”.

--but in the real time application, we store/ save/ persist the business data inside the data base using JDBC.

Public String saveStudentDetails (Student student)

--the logic that we write to store java objects data into the DB using JDBC is known as “JDBC based persistence logic”

Limitations of JDBC based persistence logic:

----------------------------------------------------------

1.JDBC cannot store the java objects into the table directly, because sql queries does not allows the java objects as input, here we need to convert object data into the simple (atomic) value to store them in a DB.

2.JDBC code is the DB dependent code because it uses DB software dependent queries. So our JDBC based persistence logic is not 100% portable across various DB s/w.

3.JDBC code having boiler plate code problem (writing the same code except sql queries in multiple classes of our application again and again).

4.JDBC code throws lots of checked exceptions, programmer need to handle them.

5.After the select operation, we get the Resultset object. This ResultSet object we cannot transfer from one layer to another layer and to get the data from the ResultSet we need to now the structure of the ResultSet.

6.There is no any caching and transaction management support is available in JDBC.

Etc..

--To overcome the above limitation we need to use ORM approaches.

ORM (Object – Relation Mapping): Java--> relation

============================

--the process of mapping java classes with the DB tables, java class member variables with the DB table columns and

Making the object of java class represents the DB table records having synchronization between them is called a ORM mapping.

Student (roll, name, marks);

Class Student {

Roll;

Name;

Marks;

}

Student s1=new Student (10, “ram”, 500);

One student object ---------------> one row of the student table

--here synchronization between object and table row is nothing but, the modification done in the object will reflect the DB table row and vice – versa.

\*\*The logic that we write to store java objects into the DB using ORM approach is called ORM based persistence logic.

--there are various ORM s/w are available in the market, these s/w will act as framework software to perform ORM based persistence logic.

Eg:

Hibernate \*\*\*

Toplink

Ibatics

Eclipselink

Etc.,

Framework software:

-----------------------------

--It is a special type of s/w that provides abstraction layer on one or more existing core technology to simplify the process of application development.

--in java most of the f/w software comes in the form of jar files (one or more jar files).

--in order to use/ work on these f/w software we need to add those jar files in our class path.

--while working with the ORM based persistence logic we write all the logics in the form of objects without any sql query support. Due to which our logic will become DB s/w independent logic.

--In ORM based logic, the ORM s/w takes objects as an input and gives objects as an output so no need to convert object data to the primitive values.

--ORM s/w addresses the mismatches between object oriented representation of data and relational representation of data.

Object ---------> tables

Class

1.Inheritence mismatches/ IS-A mismatch

2.Granularity mismatch/ HAS-A mismatch

3.Assosiate mismatch/ table relation mismatch

i.e, for processing and presenting the data, we represent the data in form of object oriented fashion where as for storing the data we represent the data in the form of relational fashion (in the tables)

class student {

int roll,

String name,

Int marks,

Address addr; //HAS-A

}

Class Address {

(city, state, pin)

}

Student table:

Roll int

Name varchar

Marks int

Address table:

City varchar

State varchar

Pin int

--one object of student class will represent one row of student table

ORM s/w addresses these mismatches in very easy manner.

POJO Class: Java bean class

==========

Plain old java object

--It is a normal java class not bounded with any technology or f/w s/w s.

i.e, a java class that is not implementing or extending technology/ framework API related classes or interfaces.

--a java class that can be compiled without adding any extra jar files in the class path are known as a POJO class.

POJI (plain old java interface)

Note: every java bean class is a POJO class but every POJO is not a java bean.

Public class X {

Public X (int x) {

} //POJO class not a bean class (default constructor is missing)

}

--above class is a POJO class but it is not a java bean class. Because of parameterized constructor.

--the following class comes under the category of POJO class.

Class X implements Serializable, Runnable {

}

ORM software features:

====================

1.It can persist/ store java object to the DB directly.

2.It supports POJO and POJI model.

3.It is a light weight software because to execute the ORM based application we need not install any kind of servers.

4.ORM persistence logic is DB independent. It is portable across multiple DB s/w 9because here we deal with object, not with sql queries)

5.prevent the developers from boiler plate code coding to perform CRUD operations.

6.It generates fine tunel sql statements internally that that improves the performance.

7.It provides caching mechanism (maintaining one local copy to enhance the performance)

8.It provides implicit connection pooling.

9.exception handling is optional because it throws unchecked exception.

10.It has a special Query language called JPQL (JPA query language) that totally depends upon the objects. Select eid, ename from employee.

Sql> select roll, name, marks from student;

Jpql>select roll, name, marks from student;

--in sql we write the query in the term of tables and columns where as in JPQL we write the query in the term of classes and variables.

ORM

Hibernate “org.hibernate”

Toplink

Each ORM has their own API to perform ORM based persistence logic.

JPA (JAVA Persistence API):

------------------------------------

It is a standard api using which we can work with any kind of ORM s/w.

Hibernate has their own API also,

Hibernate: ORM ---> different classes, diff methods, diff interfaces

Ibatis: ORM ---> different classes, diff methods, diff interfaces

Sun-microsystems: JPA API (standard API)

--all the ORM s/w implements the JPA API.

Hibernate: JPA API

Ibatis: JPA API

Hibernate and JPA:

--------------------------

JPA is a specification and hibernate is its one of the famous implementation.

Hibernate: It is one of the ORM based framework s/w. other s/w are: toplink, ibatis etc.

JPA: (Java Persistence API): It is one open specification given by Oracle corporation, to develop any ORM based s/w.

JPA provides a standard api to work with any kind of ORM based s/w.

JPA API belongs from “javax. Persistence.” Package.

--Hibernate is one of the most frequently used JPA implementation.

--Hibernate provides its own API to develop ORM based persistence logic, if we use those API then our application will become vendor lock, i.e, we cannot port our application across multiple ORM s/w.

--HB api comes in the form of “org.hibernate” package.

Note: we get JPA api, along with any ORM s/w, because all the ORM s/w implements JPA specification.

Java. Sql

Javax. Sql – this JDBC API comes along with jdk installation

JPA with Hibernate:

---------------------------

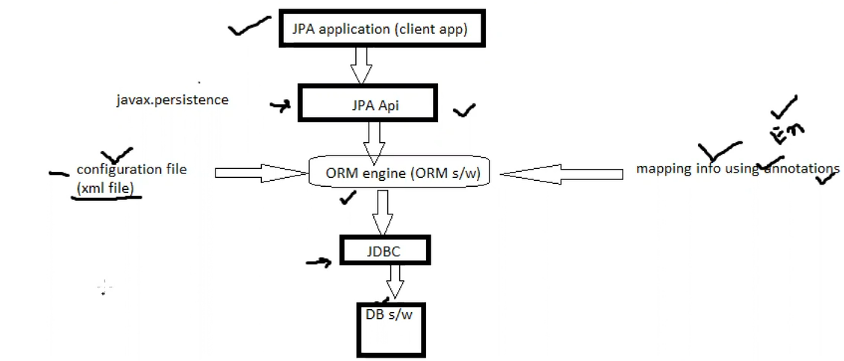
JPA Application:

---------------------

Any java application, that uses JPA api to perform persistence operation (CRUD) operation with the DB software is called as JPA application.

JPA Architecture:

-----------------------



Entity class or persistence class:

-------------------------------------------

--it is a class using which we map our table.

--if we are using the annotation, then we need not map this class with the table inside the xml mapping file.

--an entity class or persistence class is a java class that is developed corresponding to a table of DB.

--this class has many instance variables should be there as same as columns in the corresponding table.

--we should take entity class as a POJO class.

--we need to provide mapping information with the table in this class only using annotations.

NOTE: when we gives this persistence/ entity class object to the ORM software, then ORM software will decide the destination DB software based on the configuration done in a xml file which is called as hibernate-configuration file.

Configuration file:

---------------------------

--it is an xml file its name is “persistence.xml”.

--this file must be created under src/ META-INF folder in normal java application, where as in maven or gradle based application this file should be inside the src/ main/ resource/ META-INF folder.

--this file content will be used by ORM s/w (ORM engine) to locate the destination DB s/w.

--in this file generally 3 types of details we specify:

1.DB connection details

2.ORM specific details (some instruction to the ORM s/w like dialect info, show\_sql, etc)

3.annotation based entity/ persistence class name. (optional from latest hibernate version)

NOTE: Generally we take this file 1 per DB basis.

--We should always create this configuration file by taking support of example applications inside the project folder of hibernate download zip file or by taking the reference from the Google.

For persistence google search: hibernate doc (5.6 version)

<https://hibernate.org/orm/documentation/5.6/>

persistence.xml file hibernate (another way of google search)

Persistence.xml:

----------------------

<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"

version="2.0">

<persistence-unit name="studentUnit">

<properties>

<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/web19sb101db" />

<property name="javax.persistence.jdbc.user" value="root"/>

<property name="javax.persistence.jdbc.password" value="root" />

</properties>

</persistence-unit>

</persistence>

The root tag is:

<persistence> with some xml – name space

--the child tag of <persistence> tag is <persistence-unit>

--this <persistence-unit> has 2 child tags:

1.<class> tag: using which we specify the entity class name (fully qualified name) that used annotations to map a table (optional from latest version of hibernate)

2.<properties> tag: using this tag, we specify some configuration details to the ORM software

Persistence-unit: It is a logical name of the configuration of our DB and some other details.

How to get the Hibernate software:

------------------------------------------------

1.download the hibernate s/w (zip file) and add the required jar file in the class path of our project.

2.maven approach:

Hibernate-core jar files

Persistence.xml: take this file from sample application or from hibernate docs. And modify it accordingly.

ORM Engine:

-----------------

--It is a specialized s/w written in java that performs translation of JPA calls into the sql call by using mapping annotation and configuration file details and send the mapped sql to the DB s/w using JDBC.

--ORM engine is provided by any ORM s/w.

Steps to develop the JPA application:

-------------------------------------------------

Step1: create a maven project (change the java version) and add the hibernate-core dependency to the pom.xml.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.masai</groupId>

<artifactId>hibernateapp1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<!--Google Search for properties: maven compiler source-->

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<!--hibernate dependecies-->

<dependencies>

<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.6.14.Final</version>

</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->

<dependency>

<groupId>mysql</groupId>

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

<version>8.0.31</version>

</dependency>

</dependencies>

</project>

Step2: add JDBC driver related dependency to the pom.xml.

Step3: create a folder called “META-INF” inside src/main/resource folder, and create the “persistence.xml” file inside this folder by taking reference from Hibernate docs or from google.

hibernate dependencies

Google search: hibernate core mvn

<https://mvnrepository.com/artifact/org.hibernate/hibernate-core>

Example:

<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"

version="2.0">

<persistence-unit name="org.hibernate.tutorial.jpa">

<properties>

<property name="javax.presistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />

<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/web19sb201db" />

<property name="javax.persistence.jdbc.user" value="root" />

<property name="javax.persistence.jdbc.password" value="root"/>

</properties>

</persistence-unit>

</persistence>

Step4: create as many Entity/ persistence classes as there are tables in the DB, apply the at least 2 annotations to these classes.

@Entity: on the top of the class

@Id: on the top of Primary key mapped variable

--if we apply above 2 annotations then our java bean class will become entity or persistence class.

--inside these classes, we need to take variable corresponding to the columns of the tables.

Step5: create a client application and activate ORM engine by using JPA API related following classes and interface and perform the DB operations.

1.persistence class

2.EntityManagerFactory

3.EntityManager

--if we use Hibernate core API then we need to use

1.Configuration class

2.SessionFactory(I)

3.Session(I)

EXAMPLE: Student.java

**package** com.masai.entities;

**import** javax.persistence.Entity;

**import** javax.persistence.Id;

@Entity

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

@Id

**private** **int** roll;

**private** String name;

**private** **int** marks;

**public** Student(){

}

**public** Student(**int** roll, String name, **int** marks) {

**super**();

**this**.roll = roll;

**this**.name = name;

**this**.marks = marks;

}

**public** **int** getRoll() {

**return** roll;

}

**public** **void** setRoll(**int** roll) {

**this**.roll = roll;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getMarks() {

**return** marks;

}

**public** **void** setMarks(**int** marks) {

**this**.marks = marks;

}

}

Demo.java

--------------

//retrieving information

**package** com.masai.entities;

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**public** **class** Demo {

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

EntityManagerFactory emf=Persistence.*createEntityManagerFactory*("studentUnit");

EntityManager em=emf.createEntityManager();

Student stu=em.find(Student.**class**, 1);

**if**(stu!= **null**)

System.***out***.println(stu);

**else**

System.***out***.println("Record not found");

em.close();

}

}

Output:

Student [roll=1, name=Arjun, marks=850]

--to get the object from the DB we need to call: find (--) method of EntityManager object.

This find (--) method takes 2 parameter

1.the Classname of the object which we want.

2.the Id value for which we want the object.

Note: When we call createEntityManagerFactory(-) method by suppling persistence-unit name on the Persistence class, we will get the EntityManagerFactory object.

--this method loads the “persistence.xml” file into the memory.

--\*\*EntityManagerFactory object should be only one per application per DB.

This EMF object contains:

Connection pool (readily available some JDBC connection object)

Some meta information

--by using this EMF class only we create the EntityManager object.

--EMF is a heavy weight object, it should be one per application

EntityManager em=emf.createEntityManager();

Note:

Inside every DAO (Data Access Object) method (for every use case) we need to get the EntityManager object

--after performing the DB operation for that use-case we should close the EM object.

EntityManager should be one per use-case (one per DAO method)

JPA application ----------- >EntityManager(I) ----------> ORM Engine ----------> JDBC ------>DB s/w

Inserting a record:

------------------------

--In order to perform any DML (insert update delete) the method calls should be in a transactional area.

EntityTransaction et=em.getTransaction();

et.begin();

em.persist(student);

et.commit();

em.getTransaction(); --> method returns “javax.persistence.EntityTransaction(I)” object.

--this EntityTransaction object is a singleton object, i.e per EntityManager object, only one Transaction object is created.

--to store the bject we need to call persist(-) method on the EM object.

Example:

**package** com.masai.entities;

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.EntityTransaction;

**import** javax.persistence.Persistence;

**public** **class** Demo {

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

EntityManagerFactory emf=Persistence.*createEntityManagerFactory*("studentUnit");

EntityManager em=emf.createEntityManager();

Student student=**new** Student(4,"Ram",700);

// EntityTransaction et=em.getTransaction();

//

// et.begin();

// em.persist(student);

// et.commit();

em.getTransaction().begin();

em.persist(student);

em.getTransaction().commit();

em.close();

System.***out***.println("Record saved...");

}

}

Output:

Record saved…

Delete records:

---------------------

**package** com.masai.entities;

**import** java.util.Scanner;

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**public** **class** Demo {

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

EntityManagerFactory emf=Persistence.*createEntityManagerFactory*("studentUnit");

EntityManager em=emf.createEntityManager();

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter roll no delete");

**int** roll=sc.nextInt();

Student stu=em.find(Student.**class**, roll);

**if**(stu!=**null**) {

em.getTransaction().begin();

em.remove(stu);

em.getTransaction().commit();

System.***out***.println("Student removed..");

}

**else**

System.***out***.println("Student not found");

em.close();

System.***out***.println("Done");

sc.close();

}

}

Output:

Enter roll no delete 4

Student removed…

Done

Update record:

---------------------

**package** com.masai.entities;

**import** java.util.Scanner;

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**public** **class** Demo {

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

EntityManagerFactory emf=Persistence.*createEntityManagerFactory*("studentUnit");

EntityManager em=emf.createEntityManager();

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter roll no delete");

**int** roll=sc.nextInt();

Student stu=em.find(Student.**class**, roll);

**if**(stu==**null**) {

System.***out***.println("record not found..");

}

**else** {

System.***out***.println("Enter grace marks:");

**int** marks=sc.nextInt();

em.getTransaction().begin();

stu.setMarks(stu.getMarks()+marks);

em.getTransaction().commit();

System.***out***.println("Marks is graced..");

}

em.close();

System.***out***.println("Done");

sc.close();

}

}

Output:

Enter roll no delete

3

Enter grace marks:

150

Marks is graced..

Done

Explanation:

--In this above application we didn’t call any update method, we just change the state of the persistence/ entity object.

Inside the transactional area, at the end of the transaction, ORM engine will generate the update sql.

--this is known as the ORM s/w maintaining synchronization between entity object and the db table records.

--we have a method called merge() inside the EntityManager object to update a record also.

Life-cycle of persistence/ entity object:

---------------------------------------------------

An entity object has the 3 life-cycle state:

1. New state/ transient state
2. Persistence state/ managed state
3. Detached state

1.new state/ transient state:

--------------------------------------

--If we create a object of persistence class and this class is not attached with the EntityManager object, then this stage is known as new state/ transient state.

Student s=new Student (10, ” Ram”, 500);

2.persistence state:

--------------------------

--if a persistence class object or Entity object is associated with EM object, then this object, then this object will be in persistence state.

Eg:

When we call a persist (-) method by supplying student entity object then at time student object will be in persistence state.

Or

When we call find() method and this method returns the student object, then that object will also be in persistence state.

Note: when an entity class object is in persistence state, it is in-synchronization with the DB table, i.e any change made on that object inside the transaction area will reflect to the table automatically.

Eg:

Student s=new Student (150,”manoj”,500); //here student object is in transient object

Em.getTransaction().begin();

Em.persist(s); //here it is in the persistence state

s.setMarks(900);

em.getTransaction().commit();

3.DetachedState:  
-----------------------

--when we call close() method or call clear() method on the EntityManager object, then all the associated entity object will be in detached state.

--in this stage the entity objects will not be in- sync with the table.

Note: we have a merge() method in EM object, when we call this method by supplying any detached object then that detached object will bring back in the persistence state.

Example:

**package** com.masai.entities;

**import** javax.persistence.EntityManager;

**import** javax.persistence.EntityManagerFactory;

**import** javax.persistence.Persistence;

**public** **class** Demo {

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

EntityManagerFactory emf=Persistence.*createEntityManagerFactory*("studentUnit");

EntityManager em=emf.createEntityManager();

Student stu=em.find(Student.**class**, 1); //persistence state

em.clear();//detached state

em.getTransaction().begin();

stu.setMarks(800);

//persist will create the new record with existing roll -> it will give exception

//em.persist(stu); //it will throw duplicate ID related exception

em.merge(stu); // persistence state

em.getTransaction().commit();

em.close();

/

System.***out***.println("Done");

}

}

Output:

Done

Em.persist()

Em.find() --->persistence state ----->em.close(), em.clear()----->detached state ---->em.merge() --->reflect in the table

--after merge() method, we can not do modification on that object (it will not be reflected).

Hibernate Table creation:

=====================

Note: To see the ORM tool (HB) generated sql queries on the console add the following property inside the persistence.xml

//my sql connection to hibernate

<property name="hibernate.show\_sql" value="true" />

To create or update the table according to the entity class mapping information:

//value=”create” every time it will create the new table and drop the previous table.

<property name="hibernate.hbm2ddl.auto" value="create" />

//update will create new table & if the table is created then it will update the table

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

Create: Drop the existing table then create a fresh new table and insert the record.

Update: If table is not there then create a new table, and if table is already there it will perform insert operation only in the existing table.

Some of the annotation of JPA:

------------------------------------------

@Entity: To make a java bean class as entity, i.e to map with a table

@Id: To make a field as the ID field (to map with PK of a table)

@Table (name=”mystudent”): if the table name and the class name is different

@Column (name=”sname”): if the column name of table and corresponding variable of the class is different.

@Transient: it will ignore the field value

Eg:

@Entity

@Table(name="MyStudents")//It will create new table

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

@Id

@Column(name="sroll")

**private** **int** roll;

@Column(name="sname")

**private** String name;

@Transient

**private** **int** marks;

Generators in JPA:

-------------------------

--Generators are used to generate the ID field value automatically.

@Id

@GeneratedValue (strategy = GenerationType.AUTO)

Private int roll;

--here roll will be generated automatically for each row.

\*\*Note: If we use this @GeneratedValue annotation then we are not allowed to give the roll explicitly while inserting a record.

--so we should create a object by using zero argument constructor and set the each value by calling setter method.

AUTO: internally underlaying ORM s/w creates a table called “hibernate\_sequence” to maintain the Id value.

IDENTITY: It is used for auto\_increment feature to auto generate the id value.

SEQUENCE: It is used for sequence feature to auto generate the id value.

Eg:

@Entity

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

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** **int** roll;

**private** String name;

**private** **int** marks;

Limitation of EntityManager in performing CRUD operations:

---------------------------------------------------------------------------------

persist ()

find ()

merge ()

remove ()

1.Retriving Entity object based on only ID field (Primary Key field) @Id

2.multiple Entity object retrieval is not possible (multiple record)

3.bulk update and bulk delete is also not possible

4.To access Entity object we cannot specify some extra condition.

--To remove the above limitation JPA has provided JPQL (Java Persistence Query Language)

JPQL (Java Persistence Query Language):

=================================

Similarities between JPQL and SQL:

----------------------------------------------

--keywords in the both the languages are case insensitive

--GROUP BY, ORDER BY, WHERE clauses are similar

--aggregate function is also similar

--the way we express the condition to perform the CRUD operation is almost similar

Different between JPQL and SQL:

--------------------------------------------

--SQL queries are expressed in the term of table and columns, where as JPQL query is expressed in the term of Entity class names and its variables.

--the name of the class and its variables are case sensitive.

--SQL is not portable across multiple DBMS, where JPQL is portable.

Sql> select name, marks, from student; (name and marks are the column name and student is the table name)

Jpql> select name, marks from student; (here name and marks are the variables defined inside the student class)

NOTE: we should not use \* in jpql.

Example:

Sql> select \* from student;

Jpql> from student; //projecting all the column

Or

Jpql> select s from student s;

Jpql> select s.name, s.marks from student s;

Steps to use the JPQL in JPA application:

------------------------------------------------------

Step1: develop the JPQL query as string.

Step2: create javax.persistence.Query (I) object by calling “createQuery(-)” method on the EM object.

Query q=em.createQuery(“JPQL query”);

Query object the Object Oriented representation of JPQL.

Step3: bind the values if any placeholders are used. (here we have 2 types of place holders 1. Positional 2. Named placeholder)

Step4: submit the jpql query by calling either one of the following methods.

For **select** statements:

List getResultList (); (if more than one record)

Object getSingleResult (); (if at most one record)

For **insert/ update/ delete**:

Int executeQuery(); //this method should be called inside the transaction area.

Eg:

Getting all the records from Data base:

**package** com.mic.usecase;

**import** java.util.List;

**import** javax.persistence.EntityManager;

**import** com.mic.model.Account;

**import** com.mic.utility.EMUtil;

**import** javax.persistence.Query;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="from Account";

Query q=em.createQuery(jpql);

List<Account> accounts=q.getResultList();

**for**(Account acc:accounts) {

System.***out***.println(acc);

}

//accounts.forEach(acc -> System.out.println(acc));

}

}

Output:

Account [accno=12, name=Siri, balance=1050]

Account [accno=13, name=Arjun, balance=1200]

Search on non – primary key:

---------------------------------------

**package** com.mic.usecase;

**import** java.util.List;

**import** javax.persistence.EntityManager;

**import** com.mic.model.Account;

**import** com.mic.utility.EMUtil;

**import** javax.persistence.Query;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="from Account where name='Siri'";

Query q=em.createQuery(jpql);

List<Account> accounts=q.getResultList();

**for**(Account acc:accounts) {

System.***out***.println(acc);

}

//accounts.forEach(acc -> System.out.println(acc));

}

}

Output:

Account [accno=12, name=Siri, balance=1050]

If we conform that only one row will come then:

-----------------------------------------------------------------

**package** com.mic.usecase;

**import** javax.persistence.EntityManager;

**import** com.mic.model.Account;

**import** com.mic.utility.EMUtil;

**import** javax.persistence.Query;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="from Account where name='arjun'";

Query q=em.createQuery(jpql);

Object obj=q.getSingleResult();

Account acc=(Account)obj;

System.***out***.println(acc);

}

}

Output:

Account [accno=13, name=Arjun, balance=1200]

--If the above query will return more than one result then it will throw a runtime exception.

Int executeUpdate() – transactional area

List getResultSet()

Object getSingleResult()

Account acc=(Account)obj;

--In order to avoid the down casting problem we should use TypedQuery instead of Query object.

--TypedQuery is the child interface of Query interface.

Eg: TypedQuery

**package** com.mic.usecase;

**import** javax.persistence.EntityManager;

**import** com.mic.model.Account;

**import** com.mic.utility.EMUtil;

**import** javax.persistence.TypedQuery;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="from Account where name='arjun'";

TypedQuery<Account> q=em.createQuery(jpql, Account.**class**);

Account acc=q.getSingleResult();

System.***out***.println(acc);

}

}

Output:

Account [accno=13, name=Arjun, balance=1200]

Bulk update:

==========

**package** com.mic.usecase;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="update Account set balance=balance+500";

Query q=em.createQuery(jpql);

em.getTransaction().begin();

**int** n=q.executeUpdate();

em.getTransaction().commit();

System.***out***.println(n+" rows are updated");

}

}

Output:

2 rows are updated

Using Positional parameters:

---------------------------------------

After the placeholder numbers are consecutive ((0,1) (1,2) ….)

--index value can start with any number.

**package** com.mic.usecase;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="update Account set balance=balance-?1 where name=?2";

Query q=em.createQuery(jpql);

q.setParameter(1, 100);

q.setParameter(2, "siri");

em.getTransaction().begin();

**int** n=q.executeUpdate();

em.getTransaction().commit();

System.***out***.println(n+" rows updated");

}

}

Output:  
1 rows updated

Using Named parameters: (:, colon)

------------------------------------

**package** com.mic.usecase;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="update Account set balance=balance-:bal where name=:nm";

Query q=em.createQuery(jpql);

q.setParameter("bal", 150);

q.setParameter("nm", "arjun");

em.getTransaction().begin();

**int** n=q.executeUpdate();

em.getTransaction().commit();

System.***out***.println(n+" rows updated");

}

}

Output:

1 row updated

Note: For Insert operation we don’t use JPQL, we always use persist method of EntityManager.

--whenever we try to project all the columns then the return type of the TypedQuery will be the entire Entity object.

--TypedQuery is used with only one record type (Integer, String) of data, if we project List (multiple data) then Query object is enough.

1.if we try to access only one column then the return type will be:

Either String object

Or

Any wrapper class object (Integer, Float)

Or

LocalDate

2.If all column then the return type will be the entity class (internally it will be mapped)

3.If few columns then the return type will be object []. In this array each index will represent each column

Name: String

Balance: Integer

All columns: Account object

Name, balance: Object []

Eg: 1 row and 1 column

---------------------------------

**package** com.mic.usecase;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** javax.persistence.TypedQuery;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="select accno from Account where name=:nm";

String jpql1="select name from Account where accno=:acc";

//account number

Query q=em.createQuery(jpql);

q.setParameter("nm", "Siri");

//name

TypedQuery<String> q1=em.createQuery(jpql1,String.**class**);

q1.setParameter("acc", 13);

Integer accno=(Integer)q.getSingleResult();

System.***out***.println(accno);

String name=q1.getSingleResult();

System.***out***.println(name);

em.close();

}

}

Output:

12

Arjun

Eg: multiple row and 1 column

------------------------------------------

**package** com.mic.usecase;

**import** java.util.List;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** javax.persistence.TypedQuery;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="select balance from Account";

//TypedQuery<Integer> q=em.createQuery(jpql,Integer.class);

Query q=em.createQuery(jpql);

List<Integer> balances=q.getResultList();

System.***out***.println(balances);

em.close();

}

}

Output:

[1950, 2050]

Eg: few columns and all rows

**package** com.mic.usecase;

**import** java.util.List;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** javax.persistence.TypedQuery;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="select name,balance from Account";

Query q=em.createQuery(jpql);

List<Object[]> list=q.getResultList();

**for**(Object[] obj:list) {

String name=(String)obj[0];

Integer balance=(Integer)obj[1];

System.***out***.println(name);

System.***out***.println(balance);

System.***out***.println("===================");

}

em.close();

}

}

Output:

Siri

1950

===================

Arjun

2050

===================

Single row few columns:

--------------------------------

**package** com.mic.usecase;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** javax.persistence.TypedQuery;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="select name,balance from Account where accno=:acc";

//Query

Query q=em.createQuery(jpql);

q.setParameter("acc", 12);

//Object[] obj=(Object[]) q.getSingleResult();

Object obj=q.getSingleResult();

Object[] result=(Object[])obj;

System.***out***.println("Name is:"+result[0]);

System.***out***.println("Balance is:"+result[1]);

//TypedQuery

TypedQuery<Object[]> tq=em.createQuery(jpql,Object[].**class**);

tq.setParameter("acc", 13);

Object[] or1=tq.getSingleResult();

String name=(String) or1[0];

Integer balance=(Integer)or1[1];

System.***out***.println("Name is:"+name);

System.***out***.println("Balance is:"+balance);

em.close();

}

}

Output:

Name is:Siri

Balance is:1950

Name is:Arjun

Balance is:2050

Aggregate Functions:

-----------------------------

--any aggregate function will return:

Min, max, count: Integer

Average: Double

Sum: Long

Eg:

**package** com.mic.usecase;

**import** javax.persistence.EntityManager;

**import** javax.persistence.TypedQuery;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String jpql="select sum(balance) from Account";

TypedQuery<Long> tq=em.createQuery(jpql,Long.**class**);

Long sum=tq.getSingleResult();

System.***out***.println(sum);

em.close();

}

}

Output:

4000

NamedQuery:

----------------------

--If we require to write same query again and again in multiple data access layer classes, it is recommended to use NamedQuery.

--In which we centralize the query with a unique name inside the Entity class

And refer that name in all the data access layer classes.

EG:

@Entity

@NamedQuery(name="account.getBalance",query="from Account where balance<:bal")

**public** **class** Account {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** **int** accno;

**private** String name;

**private** **int** balance;

JPQLDemo.java

**package** com.mic.usecase;

**import** java.util.List;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** com.mic.model.Account;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

Query q=em.createNamedQuery("account.getBalance");

q.setParameter("bal", 4000);

List<Account> accounts=q.getResultList();

**for**(Account obj:accounts) {

System.***out***.println(obj);

}

em.close();

}

}

Output:  
Account [accno=12, name=Siri, balance=1950]

Account [accno=13, name=Arjun, balance=2050]

NativeQuery:

---------------------

--Here we write the Query in the term of tables and their columns. (normal sql)

Ex:

**package** com.mic.usecase;

**import** java.util.List;

**import** javax.persistence.EntityManager;

**import** javax.persistence.Query;

**import** com.mic.model.Account;

**import** com.mic.utility.EMUtil;

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

String nq="select \* from account";// here account name is table name

Query q=em.createNativeQuery(nq, Account.**class**);

List<Account> list=q.getResultList();

list.forEach(a-> System.***out***.println(a));

em.close();

}

}

Output:

Hibernate: select \* from account

Account [accno=12, name=Siri, balance=1950]

Account [accno=13, name=Arjun, balance=2050]

NamedNativeQuery:

---------------------------

Account.java

@Entity

@NamedNativeQuery(name="allAccount",query="select \* from account",resultClass=Account.**class**)

**public** **class** Account {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

**private** **int** accno;

**private** String name;

**private** **int** balance;

JPQLDemo.java

**public** **class** JPQLDemo {

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

EntityManager em=EMUtil.*provideEntityManager*();

Query q=em.createNamedQuery("allAccount");

List<Account> list=q.getResultList();

list.forEach(a-> System.***out***.println(a));

em.close();

}

}

Output:

Account [accno=12, name=Siri, balance=1950]

Account [accno=13, name=Arjun, balance=2050]

--Native queries are not recommended to use in real time application development.

**\*\*\*\*heart of hibernate\*\*\*\***

Mismatches between Object Oriented Representation and Relational Representation of Data:

=============================================================================

1.Granularity mismatch: HAS-A relationship problem

2.Inheritence mismatch: IS-A relationship problem

3.Association mismatch: table relationship problem

Granularity Mismatch: HAS-A relationship problem

-------------------------------------------------------------------

@Entity

Class Employee {--corse grain

@Id

Int eid;

String ename;

Int salary;

Address addr;

}

//This type of class is known as value class or normal class, it is not an Entity class.

Class Address {--fine grain

String city;

String country;

String pincode;

}

\*An Entity can exist independently.

--at table level we don’t have HAS-A relationship. (It is HAS-A relationship mismatch)

Solution for the above HAS-A relation problem:

---------------------------------------------------------------

Approach1:

--we need to create a single table with all column (all for corse gain + all for fine classes)

Apply @Embeddable at the top of (fine grain class) Address class or @Embedded at the top of Address addr variable inside the Employee Entity.

Example: hibernatemismatch1 – maven project

(Both Employee and Address are POJO classes/ Bean)

Employee.java

----------

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

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

**private** String name;

**private** **int** salary;

@Embedded

**private** Address addr;

}

Address.java

------------------

**public** **class** Address {

**private** String state;

**private** String city;

**private** String pincode;

}

Demo.java

----------------

**package** com.mic.model;

**import** javax.persistence.EntityManager;

**import** com.mic.utility.EMUtil;

**public** **class** Demo {

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

EntityManager em=EMUtil.*provideEntityManager*();

Employee e=**new** Employee();

e.setName("Siri");

e.setSalary(50000);

e.setAddr(**new** Address("AP","Vij","234567"));

em.getTransaction().begin();

em.persist(e);

em.getTransaction().commit();

System.***out***.println("Record inserted..");

em.close();

}

}

Output:

Record inserted..

--if we try to take 2 address (one for home and another for office) and then try to persist the employee object we will get exception “repeated column”

--we can solve this problem by overriding the column names of Embedded object by using “@AttributeOverrides” annotations.

Eg:

Employee.java

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

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

**private** String name;

**private** **int** salary;

@Embedded

@AttributeOverrides({

@AttributeOverride(name="state",column=@Column(name="HOME\_STATE")),

@AttributeOverride(name="city", column=@Column(name="HOME\_CITY")),

@AttributeOverride(name="pincode", column=@Column(name="HOME\_PINCODE"))

})

**private** Address homeAddr;

@Embedded

@AttributeOverrides({

@AttributeOverride(name="state", column=@Column(name="OFFICE\_STATE")),

@AttributeOverride(name="city", column=@Column(name="OFFICE\_CITY")),

@AttributeOverride(name="pincode",column=@Column(name="OFFICE\_PINCODE"))

})

**private** Address officeAddr;

}

Demo.java

**public** **class** Demo {

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

EntityManager em=EMUtil.*provideEntityManager*();

Employee e=**new** Employee();

e.setName("Siri");

e.setSalary(50000);

e.setHomeAddr(**new** Address("AP","Vij","234567"));

e.setOfficeAddr(**new** Address("Ts","Hyd","456789"));

em.getTransaction().begin();

em.persist(e);

em.getTransaction().commit();

System.***out***.println("Record inserted..");

em.close();

}

}

Approach2:

If any has more than two address then taking too many columns inside a table will violate the rules of normalization.

--To solve this problem we need to use @ElementCollection annotation, and let the user add the multiple addresses using List or Set.

--In this case ORM s/w will generate a separate table to maintain all the addresses details with a Foreign key that refers the PK of Employee table.

Example:

Employee.java

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

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

**private** String name;

**private** **int** salary;

@ElementCollection

@Embedded

**private** List<Address> addresses=**new** ArrayList<>();

}

Note: For set

It is recommended to override equals () and hashCode () method. If we want to put any user-defined objects inside the HashSet or a key of a HashSet.

Demo.java

---------------

**public** **class** Demo {

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

EntityManager em=EMUtil.*provideEntityManager*();

Employee e=**new** Employee();

e.setName("Siri");

e.setSalary(50000);

e.getAddresses().add(**new** Address("St1","Ci1","pin1"));

e.getAddresses().add(**new** Address("st2","ci2","pin2"));

em.getTransaction().begin();

em.persist(e);

em.getTransaction().commit();

System.***out***.println("Record inserted..");

em.close();

}

}

--When we execute the above application 2 tables will be created:

1.Employee: which will contain only Employee details (it will not contain any details of any address)

2.Employee\_addresses: This table will contain the details of all the addresses with a FK column employee\_eid which refers the id column of employee table.

Note: If we want to change the second table ‘employee\_addresses’ and the FK column with our choice name then we need to use @JoinTable and @JoinColumn

Example:  
Employee.java

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

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

**private** String name;

**private** **int** salary;

@ElementCollection

@Embedded

@JoinTable(name="empaddress",joinColumns=@JoinColumn(name="emp\_id"))

**private** List<Address> addresses=**new** ArrayList<>();

--with the above example the 2nd table will be created by the name “empaddress” and the FK column will be by the name “emp\_id”.

Address.java

-----------------

**public** **class** Address {

**private** String state;

**private** String city;

**private** String pincode;

**private** String addrType;

}

Demo.java

---------------

**public** **class** Demo {

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

EntityManager em=EMUtil.*provideEntityManager*();

Employee e=**new** Employee();

e.setName("Siri");

e.setSalary(50000);

e.getAddresses().add(**new** Address("St1","Ci1","pin1","HOME"));

e.getAddresses().add(**new** Address("st2","ci2","pin2","OFFICE"));

em.getTransaction().begin();

em.persist(e);

em.getTransaction().commit();

System.***out***.println("Record inserted..");

em.close();

}

}

Eager and lazy loading:

===================

--By default ORM software (Hibernate) perform lazy loading while fetching the objects, when we fetch the parent object (first level object), then only the first level object related data will be loaded into the memory, but the 2nd level object related data will be loaded at time of calling the 2nd level object related method.

Lazy Loading:

------------------

EntityManager em=EMUtil.*provideEntityManager*();

Employee e=**new** Employee();

em.close();

----

----

//Whenever we close the Entity manager em.close() like the above it will give the exception.

--To solve the above problem we need to use Eager loading:

Eager Loading

-------------------

Eg:

@Entity

**public** **class** Employee {

@Id

@GeneratedValue(strategy=GenerationType.***AUTO***)

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

**private** String name;

**private** **int** salary;

@ElementCollection(fetch= FetchType.Eager)

@Embedded

@JoinTable(name="empaddress",joinColumns=@JoinColumn(name="emp\_id"))

**private** List<Address> addresses=**new** ArrayList<>();

Association Mismatch: Table relation problem

=================== ===================

--At the table level different types of tables will participate in different kids of relationships.

Eg:

1.one to one (person ---- Driving licence): Primary key and Foreign key (unique)

2.one to many (Department ----- Employee): PK and FK (i.e PK of Department will be inside the Emp as FK)

3.many to many (student ---course): we need to take the help of 3rd table (linking table)

--To access the meaningful information from the multiple table we need to establish the relationship.

--these relationship enable us to navigate from one table record to another table records.

--to navigate from one table to another table, our tables must be in a relationship.

--when tables in the relationship then the Entity classes which represents the tables should also be in the relationships accordingly. So that objects of these classes should also be in a relationship.

--so we can navigate from one object details to another object details.

--JPA supports the relationship between the Entity classes not only with the cardinality but also with the direction.

--uni-directional and bi-directional is the another classification of relationship.

--In uni-direction, we can define child Entity object inside the parent Entity, or parent entity refers inside the child Entity, but both are not possible.

Parent class: child class

Parent Entity: Child Entity

Eg: Department class: Employee calss

--with this relation, we can access the child class object from parent object or parent class object from the child class object, both not possible at a time.

--In bi-directional: we define child entity object inside the parent Entity object inside the child Entity (navigation is possible from the either one of the any object)

So JPA supports 4 types of relationships:

1. One to one
2. One to many
3. Many to many
4. Many to many (It is by default bi-directional only)

One to many unidirectional: (from Dept to Emp)

----------------------------------------------------------------

One Department can have multiple employees

Step1: here we need to develop child entity class first as individual. (Employee Entity)

Step2: Develop a parent Entity class with its own properties and declare one extra Collection type of Child Entity class property (either List of child entity class or Set of child entity class).

And apply @OneToMany annotation to this property:

@OneToMany

Private List<Employee> emps=new ArrayList<Employee>();