Day2: Designing Data Access Layer Using ORM:

Layered Architecture in Java Based Business application:

A Java based business application will be divided into the logical partition depending upon the role played by each part.

Logical partition of a business application is known as layer.

Presentation Layer:

It is set of Java classes, which are responsible for generating user input screen and response page(output screen) is known as Presentation Layer.

This layer provides the interaction with the end-user.

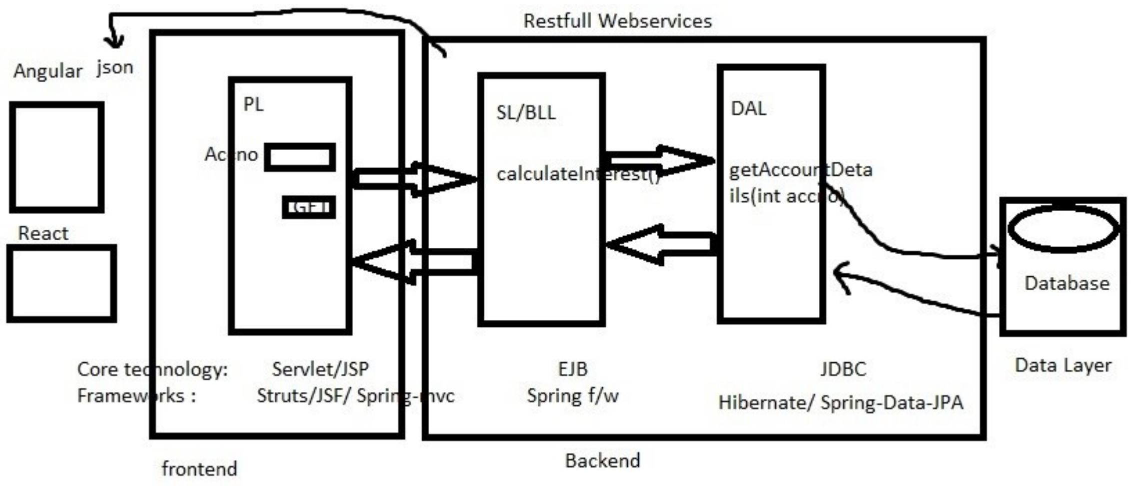
Business Logic Layer/Service Layer:

Programmatical 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 Service Layer/Business Logic Layer.

Data Access Layer:

A set of java classes whose methods r exclusively meant for performing CRUD operation with the Database server is known as Data Access Layer.

Layered Architecture of Java Based Business application:



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 the Java objects state into the database s/w is 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 the Java objects(which is holding business data) into the file using IO Streams is known as "**IO stream based persistence logic".**

  Example: the following method we can implement using Files and Serialization.

  public String saveStudentDetails(Student student)
- But in the real-time application, we store/save/persist the business data inside the database using **JDBC.**

- The logic that we write to store the Java objects data into the database using JDBC is known as **"Jdbc based persistence logic".**

limitation of JDBC based persistence logic:

1. Jdbc can't store the Java objects into the table directly, because the 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 database.
2. Jdbc code is the database dependent code because it uses database s/w dependent queries. so the JDBC based persistence logic is not 100% portable across various database s/w.

3. Jdbc code having boiler plate code problem (writing the same code except the SQL queries in multiple classes of our application again and again).
4. Jdbc code throws lots of checked exceptions, programmer need to handle them at compile time itself.
5. After the select operation, we get the ResultSet object. this ResutSet object we can not transfer from one layer to another layer and to get the data from the ResultSet object we need to know the structure of the ResultSet object also.
6. There is no any caching and transaction management support is available in the JDBC.

To overcome the above limitations we need to use ORM approach.

ORM (Object-relation mapping):

Java object < —— >relation(database table):

The process of mapping the Java classes with the database tables, Java class member variables with the database table columns and making the object of the Java class represents the DB table records having synchronization between them is called a **OR mapping.**

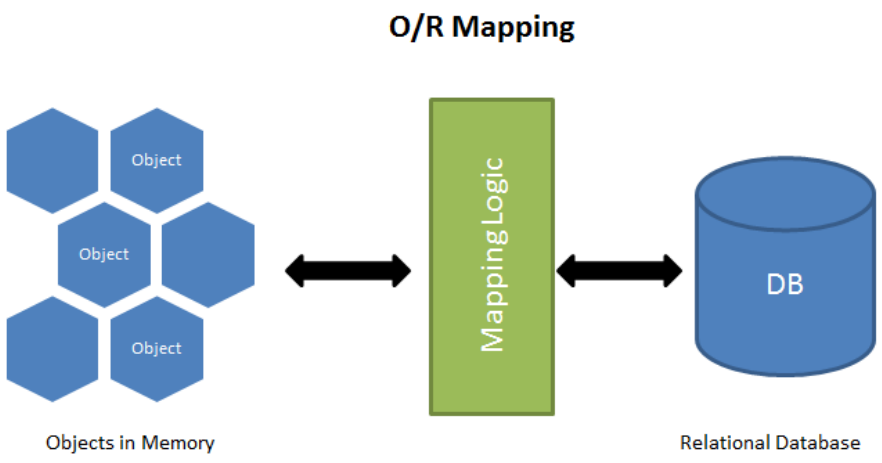
table:

student (roll, name, marks):

Java class:

```
class Student{
        int roll;
        String name;
         int marks
}
```

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



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

- The logic that we write to store the Java objects into the DB using ORM approach is called as **ORM based persistence logic.**
- There are various ORM s/w are available in the market, these s/w will act as f/w software to perform ORM based persistence logic.

Example:

Hibernate ***
Toplink
Ibatis
Eclipselink
etc...

Java Based 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 file)

In order to use/work on these f/w software we need to add those jar files in the class path of the application.

POJO class: (Plain old java object)

It is a normal Java class not bounded with any technology or framework software.
i.e a Java class that is not implementing or extending technology/framework API related classes or interfaces is known as POJO.

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 but every POJO is not a java bean.

Example:

```
//The following class comes under the category of POJO class
public class X implments Serializable, Runnable{

}


// this class does not comes under POJO
public class X implements Servlet{

}



// following class is a POJO class but it is not a java bean class.becoz of parameterized constructor.
public class X {

public X(int x){
}
}
```

ORM s/w features:

1. It can persist/store java obj to the DB directly.
2. It supports POJO and POJI model
3. It is a lightweight s/w becoz to excute 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/ws.
   (because here we deal with object, not with the SQL queries)
5. Prevent the developers from boiler plate code coding to perform CRUD operations.
6. It generates fine tuned SQL statements internally 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 exceptions.
10. it has a special Query language called JPQL (JPA query language) that totally depends upon the objects.

For example:

```
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 whereas in JPQL we write the Query in the term of classes and variables.
```

JPA with Hibernate:

Each ORM software 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 software.

All the ORM s/w implements the JPA API.

Hibernate API belongs to the **“org.hibernate”** package.

JPA API comes in the form of **"javax.persistence"** package.

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

Hibernate:- it is one of the ORM based framework software. other software are : Toplink, Ibatis, etc.

JPA:- (Java persistence API) :- It is an open specification given by Oracle corp, to develop any ORM based s/w .

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

Hibernate is one of the most frequently used JPA implementation

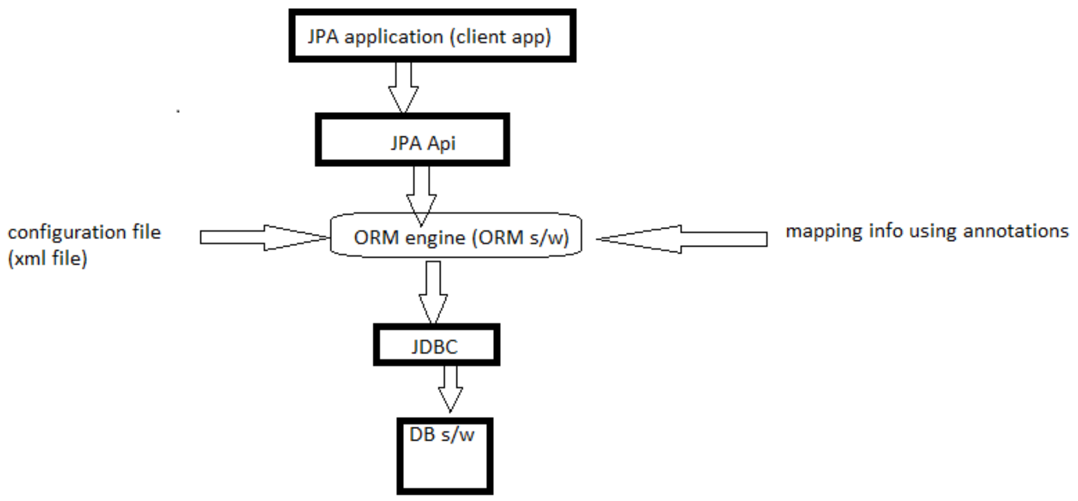
- -HB provides its own API to develop ORM based persistence logic, if we use those API then our application will become vendor lock, i.e we can not port our application across multiple ORM s/w.

Note:- We get the JPA API , along with any ORM s/w, because all the ORM s/w implements JPA specification.

JPA Application:

Any Java application, that uses JPA API to perform persistence operation (CRUD) operation with the DB s/w is called as JPA application.

JPA architecture:


JPA application (client app)
JPA Api
configuration file (xml file)
ORM engine (ORM s/w)
mapping info using annotations
JDBC
DB s/w

Entity class or persistence class:

It is a class using which we map our table.

- If we are using the annotaion, 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 annotaitons.

Note:- when we gives this persistence /Entity class obj to the ORM s/w, then ORM s/w will decide the destination DB s/w based on the configuration done in a xml file which is called as hibernate-configuration file.

JPA 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/resources/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 2 types of details we specify:-

1. **Database connection details**
2. **ORM specific details (some instruction to the ORM s/w like dialect information, show_sql, etc.)**

Note:- Generally we take this file 1 per Database 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.
example:

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

   /*
           <property name="hibernate.connection.driver_class" value="com.mysql.cj.jdbc.Driver"/>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.connection.password"  value="root"/>
            <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/ratandb"/>
     */



        </properties>
    </persistence-unit>
</persistence>
```

Here the root tag is:

<persistence> with some xml-namespace

The child tag of **<persistence>** tag is **<persistence-unit>**

- This **<persistence-unit>** has 1 child tag **<properties>** tag using this tag,we specify some configuration details to the ORM s/w

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 to the classpath of our project.
2. Maven approach:

   Add the **hibernate-core.jar** file inside the pom.xml.

persistence.xml: Take this file from the 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 sends 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:

1. create a maven project(change the java version) and add the hibernate-core dependency to the pom.xml.
2. Add the Jdbc driver-related dependency to the pom.xml
3. Create a folder called "**META-INF**" inside the **src/main/resources folder**, and create the "**persistence.xml**" file inside this folder by taking reference from Hibernate docs or from google.

Example:

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


        </properties>

    </persistence-unit>
</persistence>
```

Step 4:- Create as many Entity/Persistence classes as there are tables in the DB, and apply at least 2 annotations to these classes

@Entity : On the top of the class.
@Id : on the top of the Primary key mapped variable.

- If we apply the above 2 annotations then our Java bean class will become Entity or Persistence class.
- Inside these classes, we need to take variables corresponding to the columns of the tables.

Step 5: Create a client application and activate the 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

Configuration class

SessionFactory(I)

Session(I)

Example:

```
Student.java:  // Entity class
----------------
package com.masai;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Student {

  @Id
  private int roll;

  private int marks;
  private String name;

  public Student() {
    // TODO Auto-generated constructor stub
  }

  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;
  }

  @Override
  public String toString() {
    return "Student [roll=" + roll + ", name=" + name + ", marks=" + marks + "]";
  }

}
```

```
student table:

>create table student
(
roll int primary key,
name varchar(12),
marks int
);


>insert into student values(10,'Ram',500);
>insert into student values(20,'Ramesh',450);
```

```
Demo.java:
-------------

package com.masai;

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 student= em.find(Student.class, 20);

    if(student != null)
      System.out.println(student);
    else
      System.out.println("Student does not exist");

    em.close();


  }

}
```

To get the Object from the Database we need to call **find(--)** method on the **EntityManager** object this **find(--)** method takes 2 parameters:

1. The class name of the Entity Object which we want.
2. The ID value for which Entity we want the object.