What is Persistence

From a programming perspective, the ORM layer is an *adapter layer*: it adapts the language of object graphs to the language of SQL and relational tables. The ORM layer allows object-oriented developers to build software that persists data without ever leaving the object-oriented paradigm.

When you use JPA, you create a *map* from the datastore to your application's data model objects. Instead of defining how objects are saved and retrieved, you define the mapping between objects and your database, then invoke JPA to persist them. If you're using a relational database, much of the actual connection between your application code and the database will then be handled by JDBC, the Java Database Connectivity API.

As a spec, JPA provides *metadata annotations*, which you use to define the mapping between objects and the database. Each JPA implementation provides its own engine for JPA annotations. The JPA spec also provides the PersistanceManager or EntityManager, which are the key points of contact with the JPA system (wherein your business logic code tells the system what to do with the mapped objects).

What is Java ORM?

While they differ in execution, every JPA implementation provides some kind of ORM layer. In order to understand JPA and JPA-compatible tools, you need to have a good grasp on ORM.

Object-relational mapping is a *task*–one that developers have good reason to avoid doing manually. A framework like Hibernate ORM or EclipseLink codifies that task into a library or framework, an *ORM layer*. As part of the application architecture, the ORM layer is responsible for managing the conversion of software objects to interact with the tables and columns in a relational database. In Java, the ORM layer converts Java classes and objects so that they can be stored and managed in a relational database.

JPA

As a specification, the Java Persistence API is concerned with *persistence*, which loosely means any mechanism by which Java objects outlive the application process that created them. Not all Java objects need to be persisted, but most applications persist key business objects. The JPA specification lets you define *which* objects should be persisted, and *how* those objects should be persisted in your Java applications.

By itself, JPA is not a tool or framework; rather, it defines a set of concepts that can be implemented by any tool or framework. While JPA's object-relational mapping (ORM) model was originally based on Hibernate, it has since evolved. Likewise, while JPA was originally intended for use with relational/SQL databases, some JPA implementations have been extended for use with NoSQL datastores. A popular framework that supports JPA with NoSQL is EclipseLink, the reference implementation for JPA 2.2.

The Java Persistence API (JPA) is a specification of Java. It is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.

As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.

JPA Versions

The first version of Java Persistenece API, JPA 1.0 was released in 2006 as a part of EJB 3.0 specification.

Following are the other development versions released under JPA specification: -

JPA 2.0 - This version was released in the last of 2009. Following are the important features of this version: -

It supports validation.

It expands the functionality of object-relational mapping.

It shares the object of cache support.

JPA 2.1 - The JPA 2.1 was released in 2013 with the following features: -

It allows fetching of objects.

It provides support for criteria update/delete.

It generates schema.

JPA 2.2 - The JPA 2.2 was released as a development of maintainenece in 2017. Some of its important feature are: -

It supports Java 8 Date and Time.

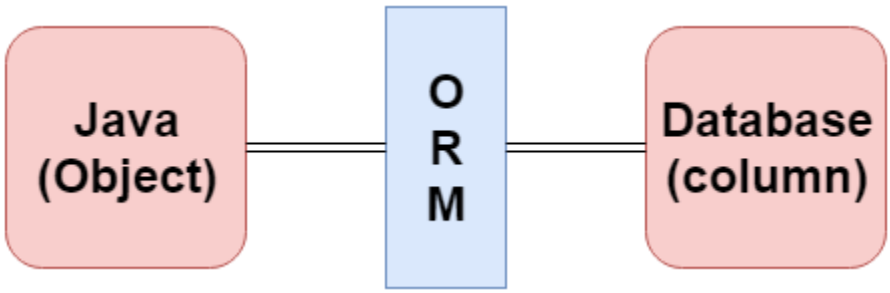
It provides @Repeatable annotation that can be used when we want to apply the same annotations to a declaration or type use.

It allows JPA annotation to be used in meta-annotations.

It provides an ability to stream a query result.

JPA Object Relational Mapping

Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column. It is capable to handle various database operations easily such as inserting, updating, deleting etc.



Following are the various frameworks that function on ORM mechanism: -

Hibernate

TopLink

ORMLite

iBATIS

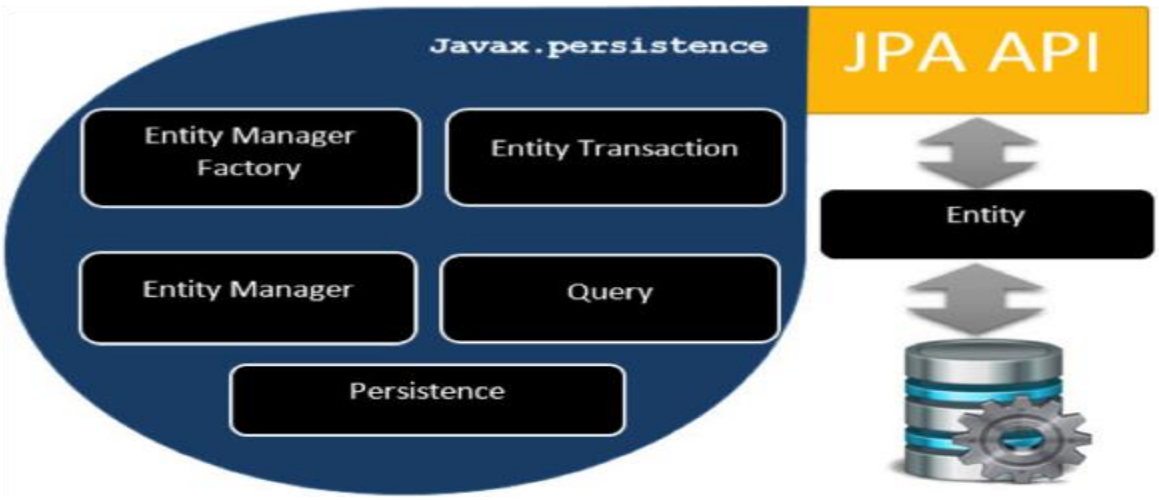
JPOX

JPA Architecture

Java Persistence API is a source to store business entities as relational entities. It shows how to define a PLAIN OLD JAVA OBJECT (POJO) as an entity and how to manage entities with relations.

Class Level Architecture

The following image shows the class level architecture of JPA. It shows the core classes and interfaces of JPA.



The following table describes each of the units shown in the above architecture.

Units	Description
EntityManagerFactory	This is a factory class of EntityManager. It creates and manages multiple EntityManager instances.
EntityManager	It is an Interface, it manages the persistence operations on objects. It works like factory for Query instance.
Entity	Entities are the persistence objects, stores as records in the database.
EntityTransaction	It has one-to-one relationship with EntityManager. For each EntityManager, operations are maintained by EntityTransaction class.
Persistence	This class contain static methods to obtain EntityManagerFactory instance.
Query	This interface is implemented by each JPA vendor to obtain relational objects that meet the criteria.

The above classes and interfaces are used for storing entities into a database as a record. They help programmers by reducing their efforts to write codes for storing data into a database so that

they can concentrate on more important activities such as writing codes for mapping the classes with database tables.

How JPA Works

JPA is basically an abstraction, using ORM techniques. If you map various model classes to the database, then JPA can a) generate an appropriate SQL query/update, b) convert the resultsets to the model classes. JPA also includes caching, and abstracts transaction handling.

In the end it doesn't really do any thing magical - everything ends up going through your JDBC driver, becoming raw SQL and returning JDBC resultsets and such. It merely allows you to hide a lot of that code away and just work with your model classes as Plain Old Java Objects (POJOs) where setting a property triggers a UPDATE and getting a property triggers a SELECT (the caching of everything and organization into transactions allows far better performance than you would get through a simple one-to-one implementation.

Example:-

Input.html

```
<html>
   <head>
      <title>TODO supply a title</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
   </head>
   <body>
      <form action="JPAServer.jsp" method="POST">
         Enter Roll no <input type="number" name="t1"><br>
         Enter Name <input type="text" name="t2"><br>
         Enter Address <input type="text" name="t3"><br>
         <input type="submit" name="s" value="SAVE">
      </form>
   </body>
</html>
```

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="JavaApplication11PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>p1.Student</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/studentdata"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Student.java (POJO File / Entity Class)

```
package p1;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class Student implements Serializable {

    @Id
    private int rollno;
    private String studname,studadd;

    public int getRollno() {
        return rollno;
    }

    public void setRollno(int rollno) {
        this.rollno = rollno;
    }

    public String getStudname() {
```

```
        return studname;
    }

    public void setStudname(String studname) {
        this.studname = studname;
    }

    public String getStudadd() {
        return studadd;
    }

    public void setStudadd(String studadd) {
        this.studadd = studadd;
    }


}
```

JPAServer.jsp

```
<%@page import="p1.Student"%>

<%@page contentType="text/html" pageEncoding="UTF-8" import="javax.persistence.*"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <%

            int id=Integer.parseInt(request.getParameter("t1"));
            String name=request.getParameter("t2");
            String add=request.getParameter("t3");
        EntityManagerFactory
emf=Persistence.createEntityManagerFactory("JavaApplication11PU");
        EntityManager em=emf.createEntityManager();
```

```
        em.getTransaction().begin();

        Student s1=new Student();
        s1.setRollno(id);
        s1.setStudname(name);
        s1.setStudadd(add);



        em.persist(s1);


        em.getTransaction().commit();

        em.close();
        emf.close();
        out.println("Data Saved");
        %>
    </body>
</html>
```

Hibernate

Hibernate is an Object-Relational Mapping (ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.

Hibernate Advantages

Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.

Provides simple APIs for storing and retrieving Java objects directly to and from the database.

If there is change in the database or in any table, then you need to change the XML file properties only.

Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.

Hibernate does not require an application server to operate.

Manipulates Complex associations of objects of your database.

Minimizes database access with smart fetching strategies.

Provides simple querying of data.

Hibernate Architecture

The Hibernate architecture includes many objects such as persistent object, session factory, transaction factory, connection factory, session, transaction etc.

The Hibernate architecture is categorized in four layers.

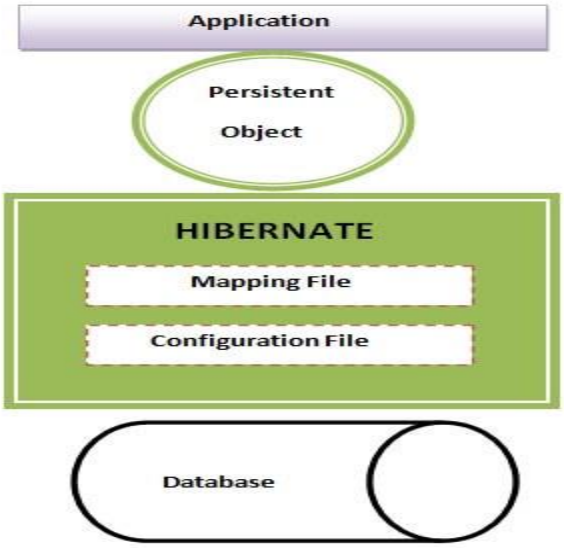
Java application layer

Hibernate framework layer

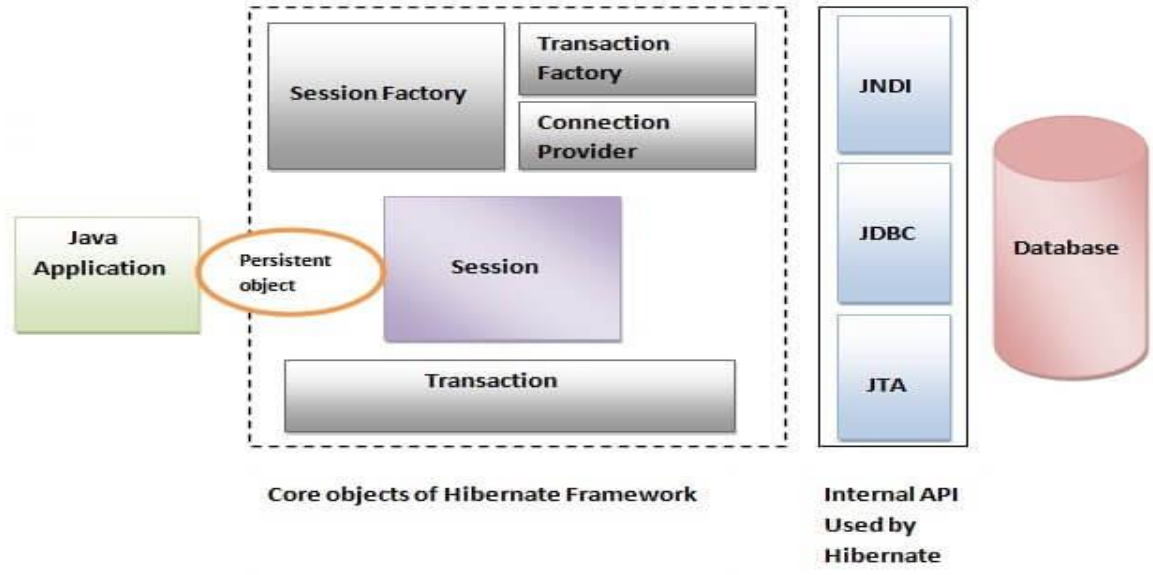
Backhand api layer

Database layer

Let's see the diagram of hibernate architecture:



This is the high level architecture of Hibernate with mapping file and configuration file.



Hibernate framework uses many objects such as session factory, session, transaction etc. alongwith existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

Components of Hibernate

Configuration Object

The Configuration object is the first Hibernate object you create in any Hibernate application. It is usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate.

The Configuration object provides two keys components −

Database Connection − This is handled through one or more configuration files supported by Hibernate. These files are hibernate.properties and hibernate.cfg.xml.

Class Mapping Setup − This component creates the connection between the Java classes and database tables.

SessionFactory Object

Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

Session Object

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

Transaction Object

A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

Query Object

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Criteria Object

Criteria objects are used to create and execute object oriented criteria queries to retrieve objects.

ConnectionProvider

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

TransactionFactory

It is a factory of Transaction. It is optional.

How does Hibernate works?

Hibernate is an ORM(Object-Relational-Mapping) framework in which like all frameworks we have some layers which coordinate with each other.

We have java classes and database tables and our aim is to build a connection between these two. For this Hibernate uses ORM technique in which it maps each java class entity to table entity present in database. Thus Hibernate is the architechture which is present between java objects and database.

The main part of Hibernate is its configuration. There are two parts of that configuration one that is for making the connection and other which is for mapping. This first part of configuration is present in ..cfg.xml file. And other part is present in ...hbm.xml file. There is only one cfg file but if we have three java classes then we have three mapping or hbm file.

Using these property file Hibernate creates a sessionFactory which is used to create sessions. This session is the entry path into database and it is shared among all classes of application. Hibernate provides us many methods like save(), persist(), saveUpdate() which are used to make

changes in the database table. These operations are done inside a transaction which maintains databse consistency. In transaction either our action is committed or rolledback if unsuccessful.

In Hibernate there are three phases of a Hibernate mapped object. When we make object of a POJO or java class then it is in its transient stage which means object is created but it is not in any database. Then we call methods of Hibernate(save(),persist() etc.) to change it to persistent state which means it is associated with the session now. A persistent object can be detached which means it is no longer present in session now. So these are three phases of an object in Hibernate.

Also there are many advantages related with Hibernate which make it quite popular with developers. Hibernate has a well layered architechture through which it becomes quite easier for any developer to operate. Also Hibernate can be used easily with any RDBMS.

Example

Input.html

```
<html>
   <head>
      <title>TODO supply a title</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
   </head>
   <body>
      <form action="Server.jsp" method="POST">
         Student Rollno <input type="number" name="t1"><br>
         Student Name <input type="text" name="t2"><br>
         Student Address <textarea rows="3" cols="20" name="t3"></textarea><br>
         <input type="submit" name="s1" value="SAVE">
      </form>
   </body>
</html>
```

Server.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"
import="org.hibernate.*,org.hibernate.cfg.*"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

```
        <title>JSP Page</title>
    </head>
    <body>
      <%
      int sid=Integer.parseInt(request.getParameter("t1"));
      String sname=request.getParameter("t2");
      String sadd=request.getParameter("t3");

      SessionFactory sf=new
Configuration().configure("hibernate.cfg.xml").buildSessionFactory();
      Session s=sf.openSession();
      Transaction t=s.beginTransaction();

      p1.Student e=new p1.Student();
      e.setRollno(sid);
      e.setStudname(sname);
      e.setStudadd(sadd);

      s.save(e);
      t.commit();
      s.close();
      out.println("Data Saved.");
      %>
    </body>
</html>
```

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
      <property name="hbm2ddl.auto">create</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/mes</property>
    <property name="hibernate.connection.username">root</property>
    <mapping resource="student.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Student.java (POJO class)

```
package p1;

public class Student
{
   private int rollno;
   private String studname;
   private String studadd;

   public int getRollno() {
      return rollno;
   }

   public void setRollno(int rollno) {
      this.rollno = rollno;
   }

   public String getStudname() {
      return studname;
   }

   public void setStudname(String studname) {
      this.studname = studname;
   }

   public String getStudadd() {
      return studadd;
   }

   public void setStudadd(String studadd) {
      this.studadd = studadd;
   }
}
```

student.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>
   <class name="p1.Student" table="stud">
      <id name="rollno" >
         <generator class="assigned"></generator>
      </id>
      <property name="studname"></property>
      <property name="studadd"></property>
   </class>
</hibernate-mapping>
```