JPA – Java Persistence API is the official java specification to work with an ORM tool. Hibernate is the most popular ORM tool out there which implements the JPA standard.

The Java Persistence API (JPA) is a Java specification that bridges the gap between relational databases and object-oriented programming.

The JPA specification lets you define *which* objects should be persisted, and *how* those objects should be persisted in your Java applications.

JPA is not a tool or framework; rather, it defines a set of concepts that can be implemented by any tool or framework.

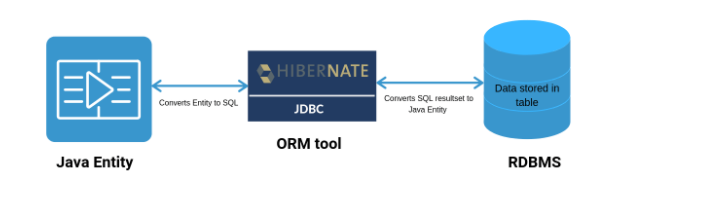
## What is Persistence?

Today’s application requires persistent data, meaning the data needs to be reliably stored in a database. Then the application should be able to retrieve the data from the database and convert it into java objects whenever needed.

**What is ORM?**

ORM stands for **O**bject-**R**elational **M**apping (ORM) is a programming technique for converting data between relational databases and object oriented programming languages such as Java, C#, etc.

An ORM (Object-relational mapping) is a tool that simplifies the Object creation, manipulation and object access to/from the database. Technically, ORM is a wrapper on top of the JDBC layer that automates a lot of boilerplate code. There are several ORM tools available in Java such as Jboss Hibernate, Oracle TopLink, EclipseLink, OpenJPA.



## Why Object Relational Mapping (ORM)?

When we work with an object-oriented system, there is a mismatch between the object model and the relational database. RDBMSs represent data in a tabular format whereas object-oriented languages, such as Java or C# represent it as an interconnected graph of objects.

**ORM Frameworks**. Hibernate. TopLink. ORMLite. iBATIS. JPOX.

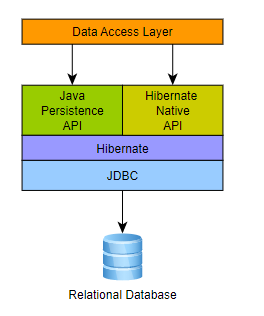
## What is Hibernate?

Hibernate is an ORM library for Java. Hibernate is an alternative to entity beans for persistence. Hibernate ORM is one of the most mature JPA implementations, and still a popular option for ORM in Java.

Hibernate is an ORM tool which is used in your DAO (data access layer) layer. It makes use of Hibernate API to effectively load, store, update, delete and query, etc its domain objects.

As you can see in the diagram, ORM seats between your database and DA Layer.

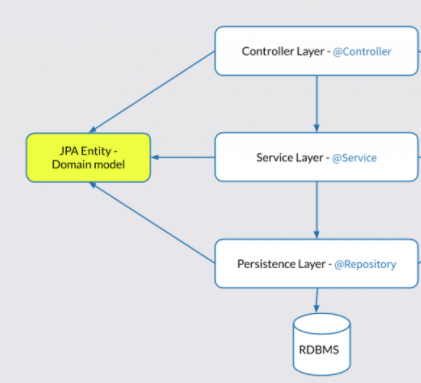
Remember, Hibernate is a JPA compliant ORM tool.



What is an Entity?

An entity is a lightweight persistent domain object. Entity classes are also known as domain models. In a School ERP System, Student is a domain model or Entity class that can be managed by JPA.

In a multilayer java architecture, a domain model is accessed from every component. DAO will use it to manage data, the presentation may use it to convert it to a JSON payload and so on.



**Rules for writing @Entity classes**

* Entity classes must be annotated with @Entity.
* Must have a public/protected no-arg constructor, can have other constructors.
* Entity classes must be a top-level class, enum or interface must not be designated as an entity.
* Both abstract and concrete classes can be entities.
* The entity class must not be final. No methods or persistent instance variables of the entity class may be final.
* I advise making entity class implement Serializable, though this is not mandatory.
* Every private/protected class properties should have Getter/Setter implemented.

Entity States:

An object that hibernate/JPA can persist in the database and retrieve it back whenever needed is called an Entity. JPA manages various states of these entities.

Basically, there are 4 states of an Object in JPA and these are New (Transient), Persistent (Managed), Detached (Unmanaged) and Removed (deleted).

## 

**1. New (Transient) state**

An object that is newly created and has never been associated with JPA Persistence Context (hibernate session) is considered to be in the New (Transient) state. The data of objects in this state is not stored in the database.

Example, student object is in New/Transient state.

Student student = new Student("email@dot.com");

**2. Persistent (JPA managed) state**

An Object that is associated with persistence context (hibernate session) are in Persistent state. Any changes made to objects in this state are automatically propagated to databases without manually invoking persist/merge/remove.

The example below will be clear to understand this.

Student student = new Student("test@email.com");

// Make changes and see if the data is updated automatically

EntityManager entityManager = entityManagerFactory.createEntityManager();

EntityTransaction tx = entityManager.getTransaction();

tx.begin();

entityManager.persist(student);

//purposely made changes - did not manually update

student.setEmail("updated\_email@gmail.com");

Long persistedId = student.getId();

tx.commit();

entityManager.close();

The object changes will be auto migrated at tx.commit(). To verify, use the below code. You will not need to use merge().

// Test if the email address was updated

entityManager = entityManagerFactory.createEntityManager();

tx = entityManager.getTransaction();

tx.begin();

student = entityManager.find(Student.class, persistedId);

tx.commit();

entityManager.close();

System.out.println("Persisted Student: " + student);

**3. Detached (unmanaged) state**

An Object becomes detached when the currently running Persistence Context is closed. Any changes made to detached objects are no longer automatically propagated to the database.

Once tx.commit() is executed, the student object becomes detached.

tx.begin();

student = entityManager.find(Student.class, persistedId);

tx.commit();

entityManager.close();

System.out.println("Persisted Student: " + student);

student.setEmail("updated\_again@gmail.com");

Now, the latest changes made to student object does not get updated in the database. You need to make use of merge to reattach the changes.

Use merge(T entity) from javax.persistence.EntityManager to synchronise the changes made to a detached object with the database.

This code explains the use of merge().

//To reattach a detached object

student.setEmail("updated\_again@gmail.com");

entityManager = entityManagerFactory.createEntityManager();

tx = entityManager.getTransaction();

tx.begin();

student = entityManager.merge(student);

tx.commit();

entityManager.close();

. System.out.println("Persisted Student: " + student);

**4. Removed object state**

As the name suggests, removed objects are deleted from the database. JPA provides entityManager.remove(object); method to remove an entity from the database.

NOTE: Only persistent objects can be removed in JPA, any attempt to delete a dettached object will cause java.lang.IllegalArgumentException: Removing a detached instance.

//Remove an Object from the database

entityManager = entityManagerFactory.createEntityManager();

tx = entityManager.getTransaction();

tx.begin();

student = entityManager.find(Student.class, persistedId);

entityManager.remove(student);

tx.commit();

entityManager.close();

## Entity relationships in JPA

Simply persisting an object with a primitive field is only half the equation. JPA also has the capability to manage entities in relation to one another. Four kinds of entity relationships are possible in both tables and objects:

One-to-many

Many-to-one

Many-to-many

One-to-one

**Entity Model:**

**EntityManagerFactory and EntityManager:**

EntityManager is used to interact with persistence context and EntityManagerFactory interacts with entity manager factory. Using EntityManager methods, we can interact with database. We can save, update and delete the data in database. The life cycle of entities are managed in persistence context.

Auto Generated Values

Marking a field with the [@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue) annotation specifies that a value will be automatically generated for that field. This is primarily intended for primary key fields

Several different value generation strategies can be used as explained below.

* The Auto Strategy: It is a special global number generator for every database. This number generator is used to generate automatic object IDs for entity objects.

[@Entity](https://www.objectdb.com/api/java/jpa/Entity)

public class EntityWithAutoId1 {

[@Id](https://www.objectdb.com/api/java/jpa/Id) [@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue)([strategy](https://www.objectdb.com/api/java/jpa/GeneratedValue/strategy)=[GenerationType](https://www.objectdb.com/api/java/jpa/GenerationType).[AUTO](https://www.objectdb.com/api/java/jpa/GenerationType/AUTO)) long id;

:

}

[@Entity](https://www.objectdb.com/api/java/jpa/Entity)

public class EntityWithAutoId2 {

[@Id](https://www.objectdb.com/api/java/jpa/Id) [@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue) long id;

:

}

During a commit the [AUTO](https://www.objectdb.com/api/java/jpa/GenerationType/AUTO) strategy uses the global number generator to generate a primary key for every new entity object. These generated values are unique at the database level and are never recycled

* The Identity Strategy:

The [IDENTITY](https://www.objectdb.com/api/java/jpa/GenerationType/IDENTITY) strategy is very similar to the [AUTO](https://www.objectdb.com/api/java/jpa/GenerationType/AUTO) strategy:

[@Entity](https://www.objectdb.com/api/java/jpa/Entity)

public class EntityWithIdentityId {

[@Id](https://www.objectdb.com/api/java/jpa/Id) [@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue)([strategy](https://www.objectdb.com/api/java/jpa/GeneratedValue/strategy)=[GenerationType](https://www.objectdb.com/api/java/jpa/GenerationType).[IDENTITY](https://www.objectdb.com/api/java/jpa/GenerationType/IDENTITY)) long id;

:

}

The [IDENTITY](https://www.objectdb.com/api/java/jpa/GenerationType/IDENTITY) strategy also generates an automatic value during commit for every new entity object. The difference is that a separate identity generator is managed per type hierarchy, so generated values are unique only per type hierarchy.

* The Sequence Strategy:

The sequence strategy consists of two parts - defining a named sequence and using the named sequence in one or more fields in one or more classes. The [@SequenceGenerator](https://www.objectdb.com/api/java/jpa/SequenceGenerator) annotation is used to define a sequence and accepts a name, an initial value (the default is 1) and an allocation size (the default is 50). A sequence is global to the application and can be used by one or more fields in one or more classes. The [SEQUENCE](https://www.objectdb.com/api/java/jpa/GenerationType/SEQUENCE) strategy is used in the [@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue) annotation to attach the given field to the previously defined named sequence:

[@Entity](https://www.objectdb.com/api/java/jpa/Entity)

// Define a sequence - might also be in another class:

[@SequenceGenerator](https://www.objectdb.com/api/java/jpa/SequenceGenerator)([name](https://www.objectdb.com/api/java/jpa/SequenceGenerator/name)="seq", [initialValue](https://www.objectdb.com/api/java/jpa/SequenceGenerator/initialValue)=1, [allocationSize](https://www.objectdb.com/api/java/jpa/SequenceGenerator/allocationSize)=100)

public class EntityWithSequenceId {

// Use the sequence that is defined above:

[@GeneratedValue](https://www.objectdb.com/api/java/jpa/GeneratedValue)([strategy](https://www.objectdb.com/api/java/jpa/GeneratedValue/strategy)=[GenerationType](https://www.objectdb.com/api/java/jpa/GenerationType).[SEQUENCE](https://www.objectdb.com/api/java/jpa/GenerationType/SEQUENCE), [generator](https://www.objectdb.com/api/java/jpa/GeneratedValue/generator)="seq")

[@Id](https://www.objectdb.com/api/java/jpa/Id) long id;

}

Unlike [AUTO](https://www.objectdb.com/api/java/jpa/GenerationType/AUTO) and [IDENTITY](https://www.objectdb.com/api/java/jpa/GenerationType/IDENTITY), the [SEQUENCE](https://www.objectdb.com/api/java/jpa/GenerationType/SEQUENCE) strategy generates an automatic value as soon as a new entity object is persisted (i.e. before commit). This may be useful when the primary key value is needed earlier. To minimize round trips to the database server, IDs are allocated in groups. The number of IDs in each allocation is specified by the [allocationSize](https://www.objectdb.com/api/java/jpa/SequenceGenerator/allocationSize) attribute. It is possible that some of the IDs in a given allocation will not be used. Therefore, this strategy does not guarantee there will be no gaps in sequence values.

CREATE TABLE CUSTOMER(

ID NUMBER(10) NOT NULL,

NAME VARCHAR2(100) NOT NULL,

EMAIL VARCHAR2(100) NOT NULL,

CREATED\_DATE DATE NOT NULL,

CONSTRAINT CUSTOMER\_PK PRIMARY KEY (ID)

);

CREATE SEQUENCE customer\_seq

MINVALUE 1

MAXVALUE 9999999999

START WITH 4

INCREMENT BY 1;

@Entity

public class Customer {

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "CUST\_SEQ")

@SequenceGenerator(sequenceName = "customer\_seq", allocationSize = 1, name = "CUST\_SEQ")

Long id;

String name;

String email;

@Column(name = "CREATED\_DATE")

Date date;

//...

}

@JoinColumn and mappedBy:

JPA Relationships can be either unidirectional or bidirectional. It simply means we can model them as an attribute on exactly one of the associated entities or both.

Defining the direction of the relationship between entities has no impact on the database mapping. It only defines the directions in which we use that relationship in our domain model.

For a bidirectional relationship, we usually define:

* the owning side
* inverse or the referencing side

The *@JoinColumn* annotation helps us specify the column we'll use for joining an entity association or element collection. On the other hand, the *mappedBy* attribute is used to define the referencing side (non-owning side) of the relationship.

**Fetch types in Hibernate and JPA:**

Defines strategies for fetching data from the database. The EAGER strategy is a requirement on the persistence provider runtime that data must be eagerly fetched. The LAZY strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first accessed.

The default depends on the cardinality of the relationship. All to-one relationships use *FetchType.EAGER* and all to-many relationships *FetchType.LAZY*.

***FetchType.EAGER* – Fetch it so you’ll have it when you need it**

The FetchType.EAGER tells Hibernate to get all elements of a relationship when selecting the root entity.

## FetchType.LAZY*– Fetch it when you need it*

The *FetchType.LAZY* tells Hibernate to only fetch the related entities from the database when you use the relationship. This is a good idea in general because there’s no reason to select entities you don’t need for your uses case.