

Generally, Java developers use lots of code, or use the proprietary framework to interact with the database, whereas using JPA, the burden of interacting with the database reduces significantly. It forms a **bridge** between **object models** (Java program) and **relational models** (database program).

Mismatches between relational and object models

Relational objects are represented in a **tabular format**, while object models are represented in an interconnected graph of object format. [While storing and retrieving an object model from a relational database, some mismatch occurs](#) due to the following reasons:

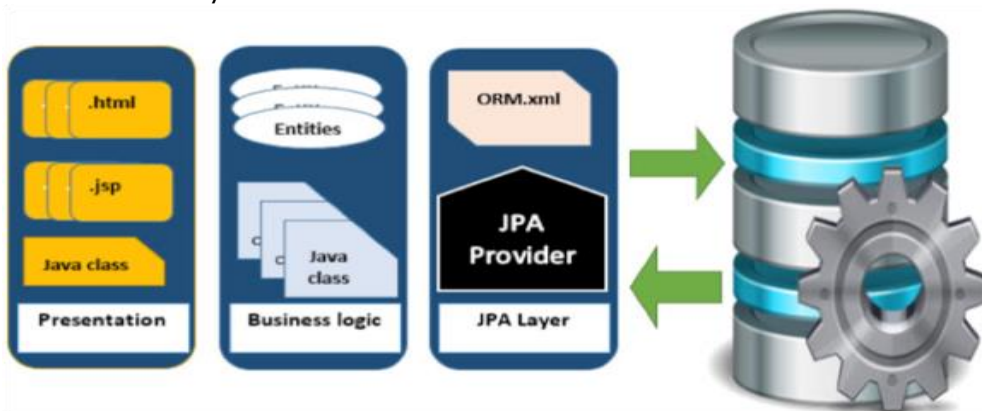
- **Granularity** : Object model has more granularity than relational model.
- **Subtypes** : Subtypes (means inheritance) are not supported by all types of relational databases.
- **Identity** : Like object model, relational model does not expose identity while writing equality.
- **Associations** : Relational models cannot determine multiple relationships while looking into an object domain model.
- **Data navigation** : Data navigation between objects in an object network is different in both models.

What is JPA?

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.

Where to use JPA?

To reduce the burden of writing codes for relational object management, a programmer follows the 'JPA Provider' framework, which allows easy interaction with database instance. Here the required framework is taken over by JPA.



JPA Providers

JPA is an open source API, therefore various enterprise vendors such as Oracle, Redhat, Eclipse, etc. provide new products by adding the JPA persistence flavor in them. Some of these products include: **Hibernate, Eclipselink, Toplink, Spring Data JPA, etc.**

JPA History

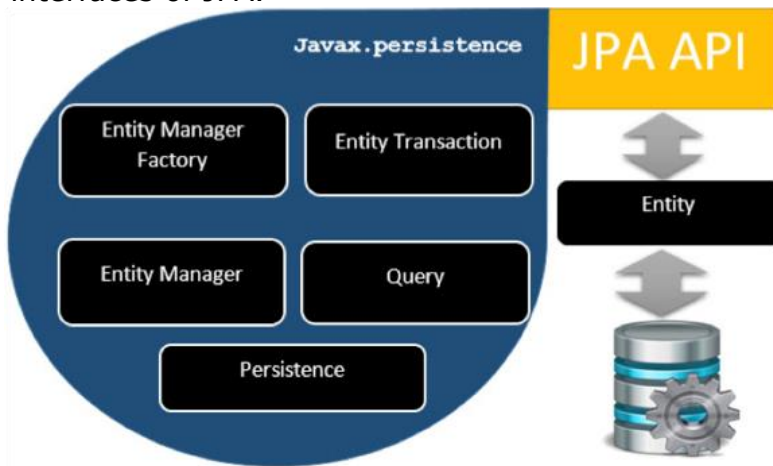
Earlier versions of EJB, defined **persistence layer** combined with business logic layer using javax.ejb.EntityBean Interface.

- While introducing **EJB 3.0**, the **persistence layer** was separated and specified as **JPA 1.0** (Java Persistence API). The specifications of this API were released along with the specifications of JAVA EE5 on May 11, 2006 using JSR 220.
- JPA 2.0 was released with the specifications of JAVA EE6 on December 10, 2009 as a part of Java Community Process JSR 317.
- JPA 2.1 was released with the specification of JAVA EE7 on April 22, 2013 using JSR 338.

JPA is a source to store **business entities** as **relational entities**. It shows **how to define** a POJO as an entity and **how to manage** entities with relations.

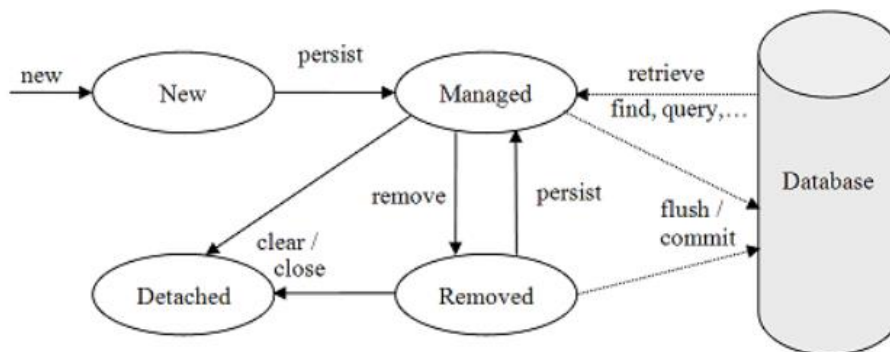
Class Level Architecture

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



Entity Object Life Cycle,

The life cycle of entity objects consists of four states: **New**, **Managed**, **Removed** and **Detached**.



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.

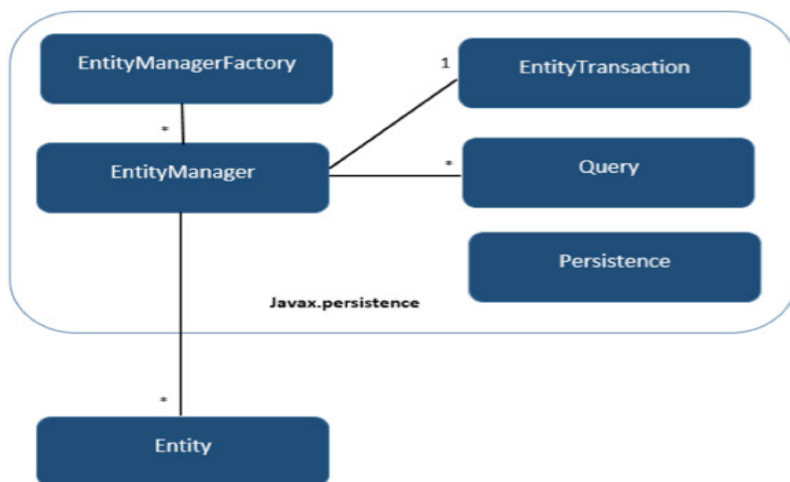
Persistence	This class contain static methods to obtain EntityManagerFactory instance.
Query	This interface is implemented by each JPA vendor to obtain relational objects that meet the criteria.

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.

JPA Class Relationships

In the above architecture, the relations between the classes and interfaces belong to the javax.persistence package. The following diagram shows the relationship between them.



- The relationship between EntityManagerFactory and EntityManager is **one-to-many**. It is a factory class to EntityManager instances.
- The relationship between EntityManager and EntityTransaction is **one-to-one**. For each EntityManager operation, there is an EntityTransaction instance.
- The relationship between EntityManager and Query is **one-to-many**. Many numbers of queries can execute using one EntityManager instance.
- The relationship between EntityManager and Entity is **one-to-many**. One EntityManager instance can manage multiple Entities.

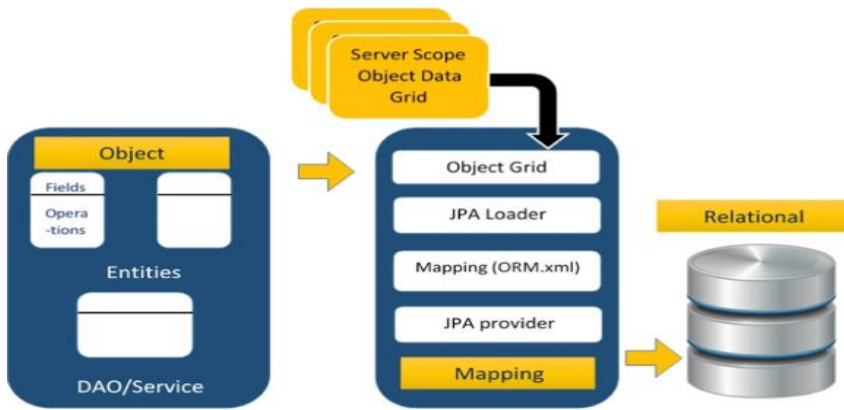
Object Relational Mapping

Object Relational Mapping (ORM) briefly tells you about what is ORM and how it works. ORM is a programming ability to covert data from object type to relational type and vice versa.

The main feature of ORM is mapping or binding an object to its data in the database. **While mapping we must consider the data, type of data and its relations with its self-entity or entity in any other table.**

ORM Architecture

Here follow the ORM architecture.



The above architecture explains how object data is stored into relational database in three phases.

Phase1

The first phase, named as the **Object data** phase contains POJO classes, service interfaces and classes. It is the main business component layer, which has business logic operations and attributes.

For example, let us take an employee database as schema-

- Employee POJO class contain attributes such as ID, name, salary, and designation. And methods like setter and getter methods of those attributes.
- Employee DAO/Service classes contains service methods such as create employee, find employee, and delete employee.

Phase 2

The second phase named as **mapping** or **persistence** phase which contains JPA provider, mapping file (ORM.xml), JPA Loader, and Object Grid.

- **JPA Provider** : The vendor product which contains JPA flavor (javax.persistence). For example, EclipseLink, Toplink, Hibernate, etc.
- **Mapping file** : The mapping file (ORM.xml) contains mapping configuration between the data in a POJO class and data in a relational database.
- **JPA Loader** : The JPA loader works like cache memory, which can load the relational grid data. It works like a copy of database to interact with service classes for POJO data (Attributes of POJO class).
- **Object Grid** : The Object grid is a temporary location which can store the copy of relational data, i.e. like a cache memory. All queries against the database is first affected on the data in the object grid. Only after it is committed, it effects the main database.

Phase 3

The third phase is the Relational data phase. It contains the relational data which is logically connected to the business component. As discussed above, only when the business component commits the data, it is stored into the database physically. Until then the modified data is [stored in a cache memory as a grid format](#). Same is the process for obtaining data.

The mechanism of the programmatic interaction of above three phases is called as object relational mapping.

Mapping.xml

The file is to instruct the JPA vendor for **mapping** the Entity classes **with** database tables. Let us take an example of Employee entity which contains four attributes. The POJO class of Employee entity named **Employee.java** is as follows:

```

public class Employee {
    private int eid;
    private String ename;
    private double salary;
    private String deg;

    public Employee(int eid, String ename, double salary, String deg) {
        super( );
        this.eid = eid;
        this.ename = ename;
        this.salary = salary;
        this.deg = deg;
    }

    public Employee( ) {
        super();
    }
    public int getEid( ) {
        return eid;
    }
    public void setEid(int eid) {
        this.eid = eid;
    }
    public String getEname( ) {
        return ename;
    }
    public void setEname(String ename) {
        this.ename = ename;
    }
    public double getSalary( ) {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    public String getDeg( ) {
        return deg;
    }
    public void setDeg(String deg) {
        this.deg = deg;
    }
}

```

The above code is the Employee entity POJO class. It contains four attributes eid, ename, salary, and deg. Consider these attributes are the table fields in the database and eid is the primary key of this table. Now we have to design hibernate mapping file for it.

The mapping file named **mapping.xml** is as follows:

```

<? xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
    http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
    version="1.0">

```

```

<description> XML Mapping file</description>

<entity class="Employee">
  <table name="EMPLOYEE" />
  <attributes>
    <id name="eid">
      <generated-value strategy="TABLE" />
    </id>

    <basic name="ename">
      <column name="EMP_NAME" length="100" />
    </basic>

    <basic name="salary">
    </basic>

    <basic name="deg">
    </basic>
  </attributes>
</entity>

</entity-mappings>

```

The above script for mapping the entity class with database table. In this file

- **<entity-mappings>** : tag defines the schema definition to allow entity tags into xml file.
- **<description>** : tag defines description about application.
- **<entity>** : tag defines the entity class which you want to convert into table in a database. Attribute class defines the POJO entity class name.
- **<table>** : tag defines the table name. If you want to keep class name as table name, then this tag is not necessary.
- **<attributes>** : tag defines the attributes (fields in a table).
- **<id>** : tag defines the primary key of the table. The **<generated-value>** tag defines how to assign the primary key value such as Automatic, Manual, or taken from Sequence.
- **<basic>** : tag is used for defining remaining attributes for table.
- **<column-name>** : tag is used to define user defined table field name.

Annotations

Generally, Xml files are used to configure specific component, or mapping two different specifications of components. In our case, we have to maintain xml separately in a framework. That means while writing a mapping xml file we need to compare the POJO class attributes with entity tags in mapping.xml file.

Here is the solution: In the class definition, we can write the configuration part using annotations.

The annotations are used for classes, properties, and methods.

Annotations starts with '@' symbol. Annotations are declared before the class, property or method is declared. All annotations of JPA are defined in [javax.persistence](#) package.

Here follows the list of annotations used in our examples

Annotation	Description
@Entity	This annotation specifies to declare the class as entity or a table.

@Table	This annotation specifies to declare table name.
@Basic	This annotation specifies non-constraint fields explicitly.
@Embedded	This annotation specifies the properties of class or an entity whose value instance of an embeddable class.
@Id	This annotation specifies the property, use for identity (primary key of a table) of the class.
@GeneratedValue	This annotation specifies, how the identity attribute can be initialized such as Automatic, manual, or value taken from sequence table.
@Transient	This annotation specifies the property which is not persistent i.e. the value is never stored into database.
@Column	This annotation is used to specify column or attribute for persistence property.
@SequenceGenerator	This annotation is used to define the value for the property which is specified in @GeneratedValue annotation. It creates a sequence.
@TableGenerator	This annotation is used to specify the value generator for property specified in @GeneratedValue annotation. It creates a table for value generation.
@AccessType	This type of annotation is used to set the access type. If you set @AccessType(FIELD) then Field wise access will occur. If you set @AccessType(PROPERTY) then Property wise access will occur.
@JoinColumn	This annotation is used to specify an entity association or entity collection. This is used in many- to-one and one-to-many associations.
@UniqueConstraint	This annotation is used to specify the field, unique constraint for primary or secondary table.
@ColumnResult	This annotation references the name of a column in the SQL query using select clause.
@ManyToMany	This annotation is used to define a many-to-many relationship between the join Tables.
@ManyToOne	This annotation is used to define a many-to-one relationship between the join Tables.
@OneToMany	This annotation is used to define a one-to-many relationship between the join Tables.
@OneToOne	This annotation is used to define a one-to-one relationship between the join Tables.
@NamedQueries	This annotation is used for specifying list of named queries.
@NamedQuery	This annotation is used for specifying a Query using static name.

Java Bean Standard

Java class, encapsulates the instance values and behaviors into a single unit called object. Java Bean is a temporary storage and reusable component or an object. It is a serializable class which has default constructor and getter & setter methods to initialize the instance attributes individually.

Bean Conventions

- Bean contains the default constructor or a file that contains serialized instance. Therefore, a bean can instantiate the bean.
- The properties of a bean can be segregated into Boolean properties and non-Boolean properties.
- Non-Boolean property contains **getter** and **setter** methods.
- Boolean property contain **setter** and **is** method.
- **Getter** method of any property should start with small lettered 'get' (java method convention) and continued with a field name that starts with capital letter. E.g. the field name is 'salary' therefore the getter method of this field is 'getSalary ()'.
- **Setter** method of any property should start with small lettered 'set' (java method convention), continued with a field name that starts with capital letter and the argument value to set to field. E.g. the field name is 'salary' therefore the setter method of this field is 'setSalary (double sal)'.
- For Boolean property, is method to check if it is true or false. E.g. the Boolean property 'empty', the is method of this field is 'isEmpty ()'.

Best practices for an efficient and maintainable persistence layer with JPA and Hibernate

Use **composition** instead of inheritance, the repository and Data-Transfer Object (DTO) patterns, and avoid the Open Session in View antipattern.

Composition over Inheritance Pattern

Composition models a “has a” association as foreign key associations between one or more database tables. **Well supported** by relational databases and **easy to map** in your domain model.

Inheritance models an “is a” association.

Can't be modeled in a relational table model.

Set of mapping strategies that enable you to map an inheritance hierarchy:

- A **mapped superclass** strategy
- A **table-per-class** strategy
- A **joined** strategy
- A **single-table** strategy

None of them provides an **ideal** solution.

Prefer composition over inheritance.

Repository Pattern

The goal of this pattern is to **separate** your **persistence**-related code **from** your **business** code.

Spring Data's CrudRepository interface defines a set of methods for **standard** write **operations**:

```
public interface EmployeeRepository
    extends CrudRepository <Employee, Long> {
    List<Employee> findByLastname(String lastname);
}
```

Apache's **DeltaSpike** project provides you with similar functionality.

Open Session in View Antipattern

Open and close Hibernate's **Session** in the **view** layer instead of in the business layer **so that** Hibernate can **load lazily** initialized **entity** associations **without** throwing a **LazyInitializationException**.

Very **easy** to use and it doesn't cause any problems **on small development**.

When in production, the initialization of these associations requires lots of additional queries known as the **n+1 select issue**.

Initialize associations with a JOIN FETCH clause.

Easiest option **for initializing** an association.

Join the tables of the two associated entities and initialize the association.

Initialize associations with a NamedEntityGraph.

Define a **graph** of entities that will be **fetches** from the **database**.

Data Transfer Object Pattern

Transfer and provide access to its **data** in the most **efficient** way.

Reasons to use DTOs.

- To **implement** and deploy your **presentation** and **business** layers **independently** of each other.
- **Enables** you to **exclude some** entity **attributes** from your API

Disadvantages of DTOs.

- Lot of **redundancy** when they are identical to your entities
- If you **change** your **entities**, you most often also **need** to **change** your **DTOs**

Conclusion

You should **prefer composition** over inheritance, **looked at** the **repository** and **DTO** patterns, and **compared** two **options for initializing lazily** fetched associations in the business layer to **avoid** the **Open Session in View** antipattern.