**JPA :**

The Java Persistence Architecture API ([JPA](http://en.wikipedia.org/wiki/Java_Persistence_API)) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database. JPA was defined as part of the [EJB](http://en.wikipedia.org/wiki/EJB) 3.0 specification as a replacement for the EJB 2 CMP (Container manage Persistence )

JPA itself is just a specification, not a product; it cannot perform persistence or anything else by itself. JPA is just a set of interface. There are open-source JPA implementations to choose from application servers in which they provide support for its use. JPA also requires a database to persist to.

 JPA is an API, whereas Hibernate in an implementation. They’re actually different animals and can’t be compared one to one. But, if you simply compare the methods made available by Hibernate’s Session and JPA’s EntityManager, you’ll see a lot more functionality in the Session object:

**EntityManager**

Interface used to interact with the persistence context.

A connection to a database is represented by an [EntityManager](http://www.objectdb.com/api/java/jpa/EntityManager) instance, which also provides functionality for performing operations on a database.

The EntityManager invokes the hibernate session under the hood. And if you need some specific features that are not available in theEntityManager, you can obtain the session by calling:

Get the Hibernate Session out of entity Manager

JPA 2.0  
Session session = entityManager.unwrap(Session.class);

JPA 1.0  
org.hibernate.Session session = (Session) manager.getDelegate();

What is Persistence Context

Both the org.hibernateSession API and javax.persistence.EntityManager API represent a context for dealing with persistent data. This concept is called a persistence context.

Difference between JPA 1.0 and JPA 2.0

[Extended Map support](http://en.wikibooks.org/wiki/Java_Persistence/Relationships#Map_Key_Columns_.28JPA_2.0.29) - Support for maintaining a key column for a Basic, Embeddable, or Entity key value in any collection relationship using a Map.  
[Derived Identifiers](http://en.wikibooks.org/wiki/Java_Persistence/Identity_and_Sequencing#JPA_2.0)  
[Nested embedding](http://en.wikibooks.org/wiki/Java_Persistence/Embeddables#Nesting)  
[New collection mappings](http://en.wikibooks.org/wiki/Java_Persistence/ElementCollection) - Support for collections of Basic or Embeddable types.  
[Undirectional OneToMany](http://en.wikibooks.org/wiki/Java_Persistence/OneToMany#Undirectional_OneToMany.2C_No_Inverse_ManyToOne.2C_No_Join_Table_.28JPA_2.0.29)  
[Ordered List mappings](http://en.wikibooks.org/wiki/Java_Persistence/Relationships#Order_Column_.28JPA_2.0.29) - Support for maintaining an index column in any collection relationship using a List.

[Orphan removal](http://en.wikibooks.org/wiki/Java_Persistence/Relationships#Orphan_Removal_.28JPA_2.0.29) - Automatic deletion of objects removed from relationships.

* [Pessimistic Locking](http://en.wikibooks.org/wiki/Java_Persistence/Locking#Pessimistic_Locking)
* [EntityManager API updates](http://en.wikibooks.org/wiki/Java_Persistence/Persisting)
* [Cache APIs](http://en.wikibooks.org/wiki/Java_Persistence/Caching#JPA_2.0_Cache_APIs)
* [Standard Properties](http://en.wikibooks.org/wiki/Java_Persistence/Runtime#Java_Standard_Edition)
* Metadata
* [Criteria API](http://en.wikibooks.org/wiki/Java_Persistence/Querying#Criteria_API_.28JPA_2.0.29)
* [JPQL enhancements](http://en.wikibooks.org/wiki/Java_Persistence/JPQL_BNF#New_in_JPA_2.0)

Hibernate Annotations implement the JPA annotations, plus have additional annotations for Hibernate specific features.

Session has more stuff to it, the [EntityManager](http://docs.oracle.com/javaee/6/api/javax/persistence/EntityManager.html" \o "Java API" \t "_new) is a specification api, and the spec doesn't cover everything that Hibernate can do, so only certain methods of a Session object is called, not all of them.

org.hibernate.Session hibernateSession = (Session)entityManager.getDelegate();

Session session = entityManager.unwrap(org.hibernate.Session.class);

**Difference between Merge and Persist**

Merge creates a new instance of your entity, copies the state from the supplied entity, and makes the new copy managed. The instance you pass in will not be managed (any changes you make will not be part of the transaction - unless you call merge again).

Maybe a code example will help.

MyEntity e = new MyEntity();

// scenario 1

// tran starts

em.persist(e);

e.setSomeField(someValue);

// tran ends, and the row for someField is updated in the database

// scenario 2

// tran starts

e = new MyEntity();

em.merge(e);

e.setSomeField(anotherValue);

// tran ends but the row for someField is not updated in the database (you made the changes \*after\* merging

// scenario 3

// tran starts

e = new MyEntity();

MyEntity e2 = em.merge(e);

e2.setSomeField(anotherValue);

// tran ends and the row for someField is updated (the changes were made to e2, not e)

EntityManager implementation is taken from current transaction. So, if you have one transaction per thread - entitymanager can be treated thread-safe. I think it is most common case, and I had no problems yet with it.

2 ways to make EM thread safe

1. Create an EJB and let the container manage the lifecycle of EntityManager. if we are getting EntityManager like emf.getEntityManager() we are reponsible for manage its lifecycle. Or we can create using @PersistanceContext. PersistnceContext represents collections of Entities an EntityManager Manages. Create an @Local Stateless Ejb and create EM like @PersistanceContext.
2. Create a servlet and inject the EntitymanagerFactory using @Persistanceunit annotation, and in the service you can manage the EM lifecycle.
3. Manage EM using ThreadLocal and a request filter.

The difference between optional and nullable

It is the scope at which they are evaluated.

The definition of 'optional' talks about property and field values and suggests that this feature should be evaluated within the runtime.

'nullable' is only in reference to database columns.

If an implementation chooses to implement optional then those properties should be evaluated in memory by the Persistence Provider and an exception raised before SQL is sent to the database otherwise when using 'updatable=false' 'optional' violations would never be reported.

What is JTA

Java Transaction API allows the applications to perform distributed transactions between one or more systems connected to the network.

The JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system:

Optimitic Lock

NONE No lock.

OPTIMISTIC.Optimistic lock.

OPTIMISTIC\_FORCE\_INCREMENT Optimistic lock, with version update.

PESSIMISTIC\_FORCE\_INCREMENT Pessimistic write lock, with version update.

PESSIMISTIC\_READ Pessimistic read lock.

PESSIMISTIC\_WRITE Pessimistic write lock.

READ Synonymous with OPTIMISTIC.

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

JPA consists of 2 parts : APIS to support entity mapping class/table and APIs to support DML

🡪EntityManager

**JpaRepository** :

It is part of Spring-data that supports Hibernate 5 , it reduces the amount of code required to implement data access layer . it extends CrudRepository and provides DML methods out of the box.

**JpaRepository and CrudRepository**

jpaRepository extends PagingAndSortingRespositroy and PagingAndSorting extends Crud

Crud contains DML 🡪 find(), save, findBy and etc

PagingAndSortoing 🡪 pagination and sorting

JpaRepositroy

JPA inheritance  
1. Single table inheritance

Single table inheritance is the best performing and best solution. In single table inheritance a single table is used to store all of the instances of the entire inheritance hierarchy.

A discriminator column is used to determine which class the particular row belongs to, each class in the hierarchy defines its own unique discriminator value.

PROJECT (table)

|  |  |  |  |
| --- | --- | --- | --- |
| ID | PROJ\_TYPE | NAME | BUDGET |
| 1 | L | Accounting | 50000 |
| 2 | S | Legal | null |

@Entity  
@Inheritance

@DiscriminatorColumn(name="PROJ\_TYPE")

@ForceDiscriminator

@Table(name="PROJECT")

**public** **abstract** **class** **Project** {  
@Id  
**private** long id;

@DiscriminatorValue("L")

**public** **class** **LargeProject** **extends** Project {

**private** BigDecimal budget;

}

@DiscriminatorValue("S")

**public** **class** **SmallProject** **extends** Project {

}

2. Join multiple table inheritance strategy = InhertitanceType.JOINED  
It mirrors the object model in data model. Each model has its own table. To query any class the join classes are needed to be in query

PROJECT (table)

|  |  |  |
| --- | --- | --- |
| ID | PROJ\_TYPE | NAME |
| 1 | L | Accounting |
| 2 | S | Legal |

SMALLPROJECT (table)

|  |
| --- |
| ID |
| 2 |

LARGEPROJECT (table)

|  |  |
| --- | --- |
| ID | BUDGET |
| 1 | 50000 |

@Entity

@Inheritance(strategy=InheritanceType.JOINED)

@DiscriminatorColumn(name="PROJ\_TYPE")

@Table(name="PROJECT")

**public** **abstract** **class** **Project** {

@Id  
**private** long id   
**private** String name;

}

@Entity  
@DiscriminatorValue("L")

@Table(name="LARGEPROJECT")

**public** **class** **LargeProject** **extends** Project {

**private** BigDecimal budget;

}

@Entity

@DiscriminatorValue("S")

@Table(name="SMALLPROJECT")

**public** **class** **SmallProject** **extends** Project {

}

3. Table per class inheritance

Querying root or branch classes can be very difficult and inefficient. There is no table for superClass.   
 Queries to root or branch becomes very expensive as it requires multiple queries or unions

SMALLPROJECT (table)

|  |  |
| --- | --- |
| ID | NAME |
| 2 | Legal |

LARGEPROJECT (table)

|  |  |  |
| --- | --- | --- |
| ID | NAME | BUDGET |
| 1 | Accounting | 50000 |

SMALLPROJECT (table)

@Entity

@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)

**public** **abstract** **class** **Project** {

@Id

**private** long id;

...

}

@Entity

@Table(name="LARGEPROJECT")

**public** **class** **LargeProject** **extends** Project {

**private** BigDecimal budget;

}

@Entity

@Table(name="SMALLPROJECT")

**public** **class** **SmallProject** **extends** Project {

}

3. Mapped super class

Does not allow mapping, querying and relationship to the super class. it is main purpose is to allow mapping information to be inherited by subclasses. The subclasses are responsible for defining tables, id .   
This solution used when we want to define a common PersistentObject for your application.

SMALLPROJECT (table)

|  |  |
| --- | --- |
| ID | NAME |
| 2 | Legal |

LARGEPROJECT (table)

|  |  |  |
| --- | --- | --- |
| ID | PROJECT\_NAME | BUDGET |
| 1 | Accounting | 50000 |

@MappedSuperclass

**public** **abstract** **class** **Project** {

@Id

**private** long id;

@Column(name="NAME")

**private** String name;

...

}

@Entity

@Table(name="LARGEPROJECT")

@AttributeOverride(name="name", column=@Column(name="PROJECT\_NAME"))

**public** **class** **LargeProject** **extends** Project {

**private** BigDecimal budget;

}

@Entity

@Table("SMALLPROJECT")

**public** **class** **SmallProject** **extends** Project {

}

Just date to be saved --> as sql.date () --> no geographical information we use nothing

@Temporal : --> we use whn our date is calandaer or Java.util