**JPA (Java Persistence API )**

The Java Persistence API (JPA) was developed in order to establish a set of standards for the object-persistence programming world. It was successful in delivering guidelines for ORM persistence providers.

In the earlier days of the object-relational persistence world, there wasn’t any specification for ORM frameworks. We either had to write our own software or work with a nonstandard third-party product, which locked you into the vendor. Most of these frameworks were not generic and robust enough, in the sense that they polluted the architecture knitting the frameworks to the underlying databases. Whether you wrote your own software or went with one of these vendors, it was a big hassle—and in some cases a major headache—to port to another product or database without embarking on a major re-engineering or restructuring project.

Realizing the necessity of a persistence standard, the Java team created the Java Persistence API. In order for a framework to produce standardized and consistent Java persistence software modules, it must adhere to the JPA. Hibernate is one of the frameworks that adopted the specification to produce a good persistence ORM tool. In fact, Hibernate contributed a lot back to the standard with its proven designs.

**The Object-Relational Impedance Mismatch**

There are two different software worlds: one is the Java world, where none other than objects are known, while the other is the relational database world, where data is king.

Java developers always work with objects that represent state and behavior modeling real-world problems. Object *persistence* is a fundamental requirement of Java applications.

The state is modeled to be persisted in durable storage so it will be permanent. On the other hand, when it is time to store the data, we have to rely on relational databases, where the data is traditionally represented in a row-column format with relationships and associations. Bringing Java objects to the relational world is always a challenging and complex task for Java developers. This process is often referred to as *object-relational mapping* (ORM).

Enterprises employ object-oriented languages (such as Java) as their programming platforms and relational databases (such as Oracle, MySQL, Sybase, etc.) for data storage. The existence of these two software technologies is a must for most real-world applications in spite of the so-called “**object-relational impedance mismatch**.”

**Disadvantages of JDBC**

Java has a standard tool set for accessing databases. It is called the Java Database Connectivity (JDBC) application programming interface (API).

The API was very well used in Java applications until recently. While the API is well suited for small projects, it becomes quite cumbersome (and sometimes out of hand) as the domain model increases in complexity. The API also includes a lot of repetitious boilerplate code, requiring the developer to do a lot of manual coding.

**Meaning of ORM**

ORM allows you to use java objects as representation of a relational database. It maps the two concepts (object-oriented and relational)

Hibernate is an ORM framework - you describe how your objects are represented in your database, and hibernate handles the conversion.

JDBC is the API for database access, and it works "in a relational way" - you query tables and get rows and columns back. Hibernate uses JDBC under the hood to fetch the data and later convert it to objects.

**Hibernate and JPA**

While JPA is the specification, Hibernate is the implementation provider that follows the rules dictated in the specification.

**<persistencexmlns=*"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"***

**version=*"2.1"*>**

**<persistence-unitname=*"myPersistanceUnit"*>**

**<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>**

**<properties>**

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

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

**<propertyname=*"javax.persistence.jdbc.driver"*value=*"com.mysql.jdbc.Driver"*/>**

**<propertyname=*"javax.persistence.jdbc.url"*value=*"jdbc:mysql://localhost:3306/caps\_db"*/>**

**<propertyname=*"hibernate.dialect"*value=*"org.hibernate.dialect.MySQL57Dialect"*/>**

**<propertyname=*"hibernate.show\_sql"*value=*"true"*/>**

**<!-- <property name="hibernate.hbm2ddl.auto" value="update" /> -->**

**<propertyname=*"hibernate.format\_sql"*value=*"true"*/>**

**</properties>**

**</persistence-unit>**

**</persistence>**

* Here, we first define a persistence-unit with the name *myPersistanceUnit*.
* Then, we need to let the runtime know about our JPA implementation provider; in our case, of course, it’s Hibernate. We set the provider tag with the org.hibernate.jpa.HibernatePersistenceProvider Persistence value to enable Hibernate as our persistence provider.
* There are properties defined here, too, which encompass the database connection information that will be used by our provider to connect and access the database.
* **One important note is that you must place this configuration in a file named *persistence.xml*, which should be placed in the *META-INF* folder.**
* You may have to create this folder at the project level if it doesn’t already exist. Make sure to also add the *META-INF* directory to the classpath of your application. The Hibernate JPA runtime browses through the *META-INF* directory to find the *persistence.xml* file for loading and creating the persistence context.
* **We can have one or more persistence unit in a single persistence.xml file for different databases.**

**EntityManagerFactory**

* As the name suggests, EntityManagerFactory is a factory class for creating EntityManagers.
* It is a heavy-weight, thread-safe object, and hence for each persistence unit only one instance of the factory is created.
* Creating this factory is an expensive operation, so the recommended approach is to cache the instance.
* On the other hand, the EntityManagers created from this factory are safe to use and throw away whenever they’re not
* There will be only one factory created for one persistence unit, but we can have multiple factories, each against a single persistence unit in a single JVM. We obtain a factory from our Persistence instance by passing the persistence unit name, as shown here:

**EntityManagerFactory emf =**

* **Persistence.*createEntityManagerFactory*("myPersistenceUnit");**

**EntityManager**

* It manages the lifecycle of our entities. It encompasses a unit of work and interacts with the database to work with the entities.
* It is tied to the current thread of execution for a given transaction and maintains the first level of cache. Once the active transaction to which the EntityManager is bound completes, the cache will be recycled.
* There are two types of EntityManagers: one that runs in a container-managed environment, and another in a standalone JVM. The former is typically a Java Enterprise Edition (JEE) container, such as an application server or a web container. The latter is a Java Standard Edition (JSE) standalone program.

**EntityManager em = emf.createEntityManager();**

Saving and Querying Entities

We use session’s save and saveOrUpdate methods to persist the entities to the database. Parallel to this functionality, we use EntityManager’s persistEntity method to save the object in a database in JPA mode.

This is illustrated in the following example:

**public void** persistNewInstrument(){

*// Create the entity manager*

EntityManager manager = entityManagerFactory.createEntityManager();

*// Create and populate the instrument object*

Instrument instrument = **new** Instrument();

instrument.setIssue("IBM");

*// Save the domain object*

**manager.persist**(instrument);

We search for and retrieve the objects via the getReference method, as shown here:

**public void** findInstrument() {

Instrument instrument = **manager.getReference**(Instrument.class,1);

}

The preceding method takes two arguments: the domain class itself and a primary key.

There’s also a find method to retrieve the object, which is similar to the before mentioned

getReference method. This is shown in the following snippet:

**public void** findInstrument() {

Instrument instrument = **manager.find**(Instrument.class,1);

}

However, there are a few subtle differences between these two approaches. getReference fetches a **lazy-loaded entity**. This means that the attributes of the class, apart from its primary key, are not fetched (hence, they would be null) until we access them. The find method, on the other hand, does the opposite. Also, the getReference method throws an EntityNotFoundException if there’s no record in the database, while the find method simply returns a null value.

Deleting entities is simple too. Using the EntityManager’s remove method, we can delete

the entities from our durable storage:

**public void** deleteInstrument() {

manager.remove("IBM");

}

We use the flush and refresh methods to synchronize the state of the persistent entities

with the database. The flush method updates the database with the modified copies of

the objects, while refresh does the opposite: it updates the object model with the latest

copy of the records, reading from the database.

# Persistence LifeCycle States

Given an instance of an object that is mapped to Hibernate, it can be in any one of four different states: **transient, persistent, detached, or removed**.

## Transient Object

## Transient objects exist in heap memory. Hibernate does not manage transient objects or persist changes to transient objects.

To persist the changes to a transient object, you would have to ask the EntityManager to persist the transient object to the database, at which point Hibernate assigns the object an identifier and marks the object as being in persistent state.

## Persistent Object

## Persistent objects exist in the database, and Hibernate manages the persistence for persistent objects.

## If fields or properties change on a persistent object, Hibernate will keep the database representation up to date when the application marks the changes as to be committed.

## Detached Object

## Detached objects have a representation in the database, but changes to the object will not be reflected in the database, and vice-versa.

## A detached object can be created by closing the EntityManager that it was associated with or by evicting it from the session with a call to the EntityManager’s detach() method.

## Removed Object

## Removed objects are objects that are being managed by Hibernate (persistent objects, in other words) that have been passed to the EntityManger.remove() method. When the application marks the changes held in the EntityManager as to be committed, the entries in the database that correspond to removed objects are deleted.

## Note:

1. Newly created POJO object will be in the **transient state**. Transient object doesn’t represent any row of the database i.e. not associated with any session object. It’s plain simple java object.
2. Persistent object represent one row of the database and always associated with some unique EntityManger. Changes to persistent objects are tracked by hibernate and are saved into database when commit call happen.
3. Detached objects are those who were once persistent in past, and now they are no longer persistent. To persist changes done in detached objects, you must reattach them to EntityManger Session.
4. Removed objects are persistent objects that have been passed to the EntityManger remove() method and soon will be deleted as soon as changes held in the EntityManager session will be committed to database.

The Java Persistence API (JPA) sets the standards for the Java persistence world. In this

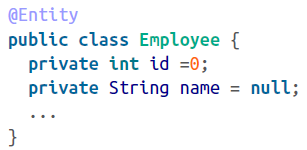
chapter, you learned the basics of JPA and skimmed through the APIs. You learned that

Hibernate supports the JPA fully, and we should try to plug into the JPA APIs whenever

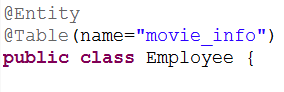
possible, or when we know we need to make our code portable.

**Annotations In Hibernate Explained**

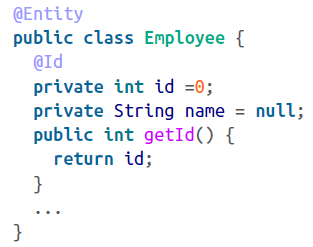
* To make this class persistent using annotations, we first define it as *Entity.* We do so by annotating the class with the @*Entity* annotation.



* We have to use the @Table annotation to tell hibernate which object to map with which Table:



* Now that we have a persistent entity, our next step is to define an identifier. Remember, all persistent entities must have their identifiers defined; otherwise, Hibernate will complain.



* If we only use @id annotation then the column name should match the variable name, We can also set a different column name, for example:

