Hibernate - ORM Overview

What is JDBC?

JDBC stands for **Java Database Connectivity**. It provides a set of Java API for accessing the relational databases from Java program. These Java APIs enables Java programs to execute SQL statements and interact with any SQL compliant database.

JDBC provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification.

Pros and Cons of JDBC

Pros of JDBC	Cons of JDBC
Clean and simple SQL processing	Complex if it is used in large projects
Good performance with large data	Large programming overhead
Very good for small applications	No encapsulation
Simple syntax so easy to learn	Hard to implement MVC concept
	Query is DBMS specific

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.

Consider the following Java Class with proper constructors and associated public function −

```
public class Employee {
   private int id;
   private String first_name;
   private String last_name;
   private int salary;
   public Employee() {}
   public Employee(String fname, String lname, int salary)
{
      this.first_name = fname;
      this.last_name = lname;
      this.salary = salary;
   }

   public int getId() {
      return id;
   }

   public String getFirstName() {
      return first_name;
   }

   public String getLastName() {
      return last_name;
   }

   public int getSalary() {
      return salary;
   }
                                    }
```

Consider the above objects are to be stored and retrieved into the following RDBMS table −

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

First problem, what if we need to modify the design of our database after having developed a few pages or our application? Second, loading and storing objects in a relational database exposes us to the following five mismatch problems −

Sr.No.	Mismatch & Description

1	**Granularity** Sometimes you will have an object model, which has more classes than the number of corresponding tables in the database.
2	**Inheritance** RDBMSs do not define anything similar to Inheritance, which is a natural paradigm in object-oriented programming languages.
3	**Identity** An RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity (a==b) and object equality (a.equals(b)).
4	**Associations** Object-oriented languages represent associations using object references whereas an RDBMS represents an association as a foreign key column.
5	**Navigation** The ways you access objects in Java and in RDBMS are fundamentally different.

The **O**bject-**R**elational **M**apping (ORM) is the solution to handle all the above impedance mismatches.

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 system has the following advantages over plain JDBC −

Sr.No.	Advantages
1	Let’s business code access objects rather than DB tables.
2	Hides details of SQL queries from OO logic.
3	Based on JDBC 'under the hood.'
4	No need to deal with the database implementation.
5	Entities based on business concepts rather than database structure.
6	Transaction management and automatic key generation.

7	Fast development of application.

An ORM solution consists of the following four entities −

Sr.No.	Solutions
1	An API to perform basic CRUD operations on objects of persistent classes.
2	A language or API to specify queries that refer to classes and properties of classes.
3	A configurable facility for specifying mapping metadata.
4	A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.

Java ORM Frameworks

There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database.

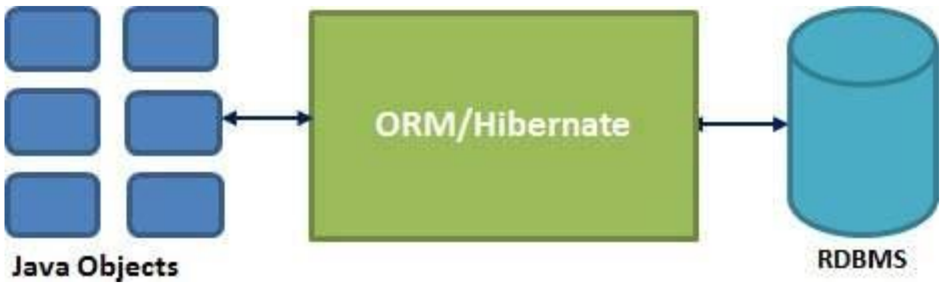
- Enterprise JavaBeans Entity Beans
- Java Data Objects
- Castor
- TopLink
- Spring DAO
- Hibernate
- And many more

Hibernate - Overview

ibernate is an **O**bject-**R**elational **M**apping (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.

Supported Databases

Hibernate supports almost all the major RDBMS. Following is a list of few of the database engines supported by Hibernate −

- HSQL Database Engine
- DB2/NT
- MySQL
- PostgreSQL
- FrontBase
- Oracle
- Microsoft SQL Server Database
- Sybase SQL Server
- Informix Dynamic Server

Supported Technologies

Hibernate supports a variety of other technologies, including −

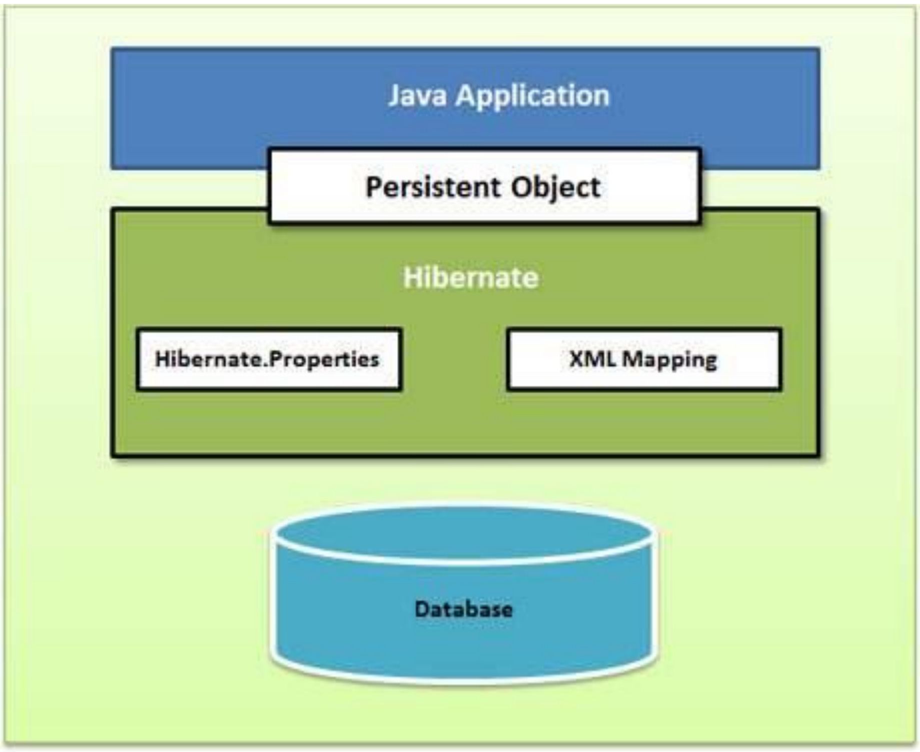
- XDoclet Spring

- J2EE
- Eclipse plug-ins
- Maven

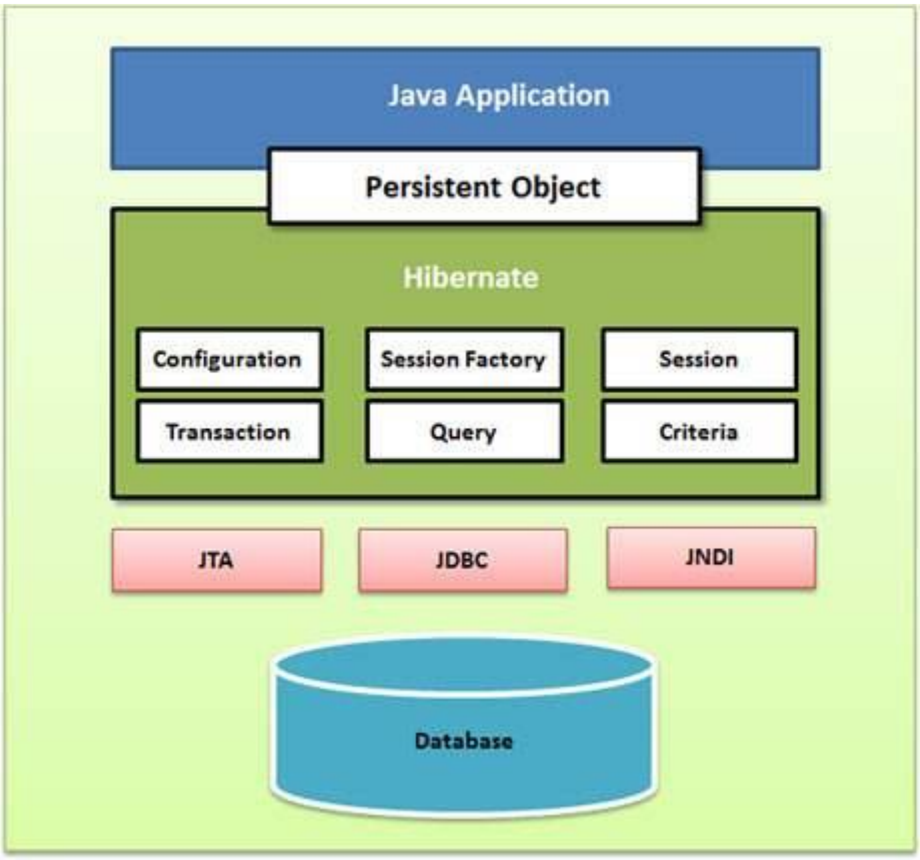
Hibernate - Architecture

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

Following is a very high level view of the Hibernate Application Architecture.



Following is a detailed view of the Hibernate Application Architecture with its important core classes.



Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JTA), and Java Naming and Directory Interface (JNDI). JDBC provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow Hibernate to be integrated with J2EE application servers.

Following section gives brief description of each of the class objects involved in Hibernate Application Architecture.

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.

Hibernate - Configuration

Hibernate requires to know in advance — where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called **hibernate.properties**, or as an XML file named **hibernate.cfg.xml**.

I will consider XML formatted file **hibernate.cfg.xml** to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application's classpath.

Hibernate Properties

Following is the list of important properties, you will be required to configure for a databases in a standalone situation −

Sr.No.	Properties & Description
1	**hibernate.dialect** This property makes Hibernate generate the appropriate SQL for the chosen database.
2	**hibernate.connection.driver_class** The JDBC driver class.
3	**hibernate.connection.url** The JDBC URL to the database instance.
4	**hibernate.connection.username** The database username.
5	**hibernate.connection.password** The database password.
6	**hibernate.connection.pool_size** Limits the number of connections waiting in the Hibernate database connection pool.
7	**hibernate.connection.autocommit** Allows autocommit mode to be used for the JDBC connection.

If you are using a database along with an application server and JNDI, then you would have to configure the following properties −

Sr.No.	Properties & Description
1	**hibernate.connection.datasource** The JNDI name defined in the application server context, which you are using for the

	application.
2	**hibernate.jndi.class** The InitialContext class for JNDI.
3	**hibernate.jndi.<JNDIpropertyname>** Passes any JNDI property you like to the JNDI *InitialContext*.
4	**hibernate.jndi.url** Provides the URL for JNDI.
5	**hibernate.connection.username** The database username.
6	**hibernate.connection.password** The database password.

Hibernate with MySQL Database

MySQL is one of the most popular open-source database systems available today. Let us create **hibernate.cfg.xml** configuration file and place it in the root of your application's classpath. You will have to make sure that you have **testdb** database available in your MySQL database and you have a user **test** available to access the database.

The XML configuration file must conform to the Hibernate 3 Configuration DTD, which is available at http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>

      <property name = "hibernate.dialect">
         org.hibernate.dialect.MySQLDialect
      </property>

      <property name = "hibernate.connection.driver_class">
         com.mysql.jdbc.Driver
      </property>

      <!-- Assume test is the database name -->
      <property name = "hibernate.connection.url">
         jdbc:mysql://localhost/test
      </property>

      <property name = "hibernate.connection.username">
         root
      </property>

      <property name = "hibernate.connection.password">
         root123
      </property>

      <!-- List of XML mapping files -->
      <mapping resource = "Employee.hbm.xml"/>

   </session-factory>
</hibernate-configuration>
```

The above configuration file includes **<mapping>** tags, which are related to hibernatemapping file and we will see in next chapter what exactly a hibernate mapping file is and how and why do we use it?

Following is the list of various important databases dialect property type −

Sr.No.	Database & Dialect Property
1	**DB2** org.hibernate.dialect.DB2Dialect
2	**HSQLDB** org.hibernate.dialect.HSQLDialect
3	**HypersonicSQL** org.hibernate.dialect.HSQLDialect
4	**Informix** org.hibernate.dialect.InformixDialect
5	**Ingres** org.hibernate.dialect.IngresDialect
6	**Interbase** org.hibernate.dialect.InterbaseDialect
7	**Microsoft SQL Server 2000**

	org.hibernate.dialect.SQLServerDialect
8	**Microsoft SQL Server 2005** org.hibernate.dialect.SQLServer2005Dialect
9	**Microsoft SQL Server 2008** org.hibernate.dialect.SQLServer2008Dialect
10	**MySQL** org.hibernate.dialect.MySQLDialect
11	**Oracle (any version)** org.hibernate.dialect.OracleDialect
12	**Oracle 11g** org.hibernate.dialect.Oracle10gDialect
13	**Oracle 10g** org.hibernate.dialect.Oracle10gDialect
14	**Oracle 9i** org.hibernate.dialect.Oracle9iDialect
15	**PostgreSQL** org.hibernate.dialect.PostgreSQLDialect
16	**Progress** org.hibernate.dialect.ProgressDialect
17	**SAP DB** org.hibernate.dialect.SAPDBDialect
18	**Sybase** org.hibernate.dialect.SybaseDialect
19	**Sybase Anywhere** org.hibernate.dialect.SybaseAnywhereDialect

Hibernate - Sessions

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. The main function of the Session is to offer, create, read, and delete operations for instances of mapped entity classes.

Instances may exist in one of the following three states at a given point in time −

- **transient** − A new instance of a persistent class, which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.
- **persistent** − You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
- **detached** − Once we close the Hibernate Session, the persistent instance will become a detached instance.

- ## Session Interface Methods
- There are number of methods provided by the **Session** interface, but I'm going to list down a few important methods only, which we will use in this tutorial. You can check Hibernate documentation for a complete list of methods associated with **Session** and **SessionFactory**.

Sr.No.	Session Methods & Description
1	**Transaction beginTransaction()** Begin a unit of work and return the associated Transaction object.
2	**void cancelQuery()** Cancel the execution of the current query.
3	**void clear()** Completely clear the session.
4	**Connection close()** End the session by releasing the JDBC connection and cleaning up.
5	**Criteria createCriteria(Class persistentClass)** Create a new Criteria instance, for the given entity class, or a superclass of an entity class.
6	**Criteria createCriteria(String entityName)** Create a new Criteria instance, for the given entity name.

7	**Serializable getIdentifier(Object object)** Return the identifier value of the given entity as associated with this session.
8	**Query createFilter(Object collection, String queryString)** Create a new instance of Query for the given collection and filter string.
9	**Query createQuery(String queryString)** Create a new instance of Query for the given HQL query string.
10	**SQLQuery createSQLQuery(String queryString)** Create a new instance of SQLQuery for the given SQL query string.
11	**void delete(Object object)** Remove a persistent instance from the datastore.
12	**void delete(String entityName, Object object)** Remove a persistent instance from the datastore.
13	**Session get(String entityName, Serializable id)** Return the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance.
14	**SessionFactory getSessionFactory()** Get the session factory which created this session.
15	**void refresh(Object object)** Re-read the state of the given instance from the underlying database.
16	**Transaction getTransaction()** Get the Transaction instance associated with this session.
17	**boolean isConnected()** Check if the session is currently connected.
18	**boolean isDirty()** Does this session contain any changes which must be synchronized with the database?
19	**boolean isOpen()** Check if the session is still open.
20	**Serializable save(Object object)**

	Persist the given transient instance, first assigning a generated identifier.
21	**void saveOrUpdate(Object object)** Either save(Object) or update(Object) the given instance.
22	**void update(Object object)** Update the persistent instance with the identifier of the given detached instance.
23	**void update(String entityName, Object object)** Update the persistent instance with the identifier of the given detached instance.

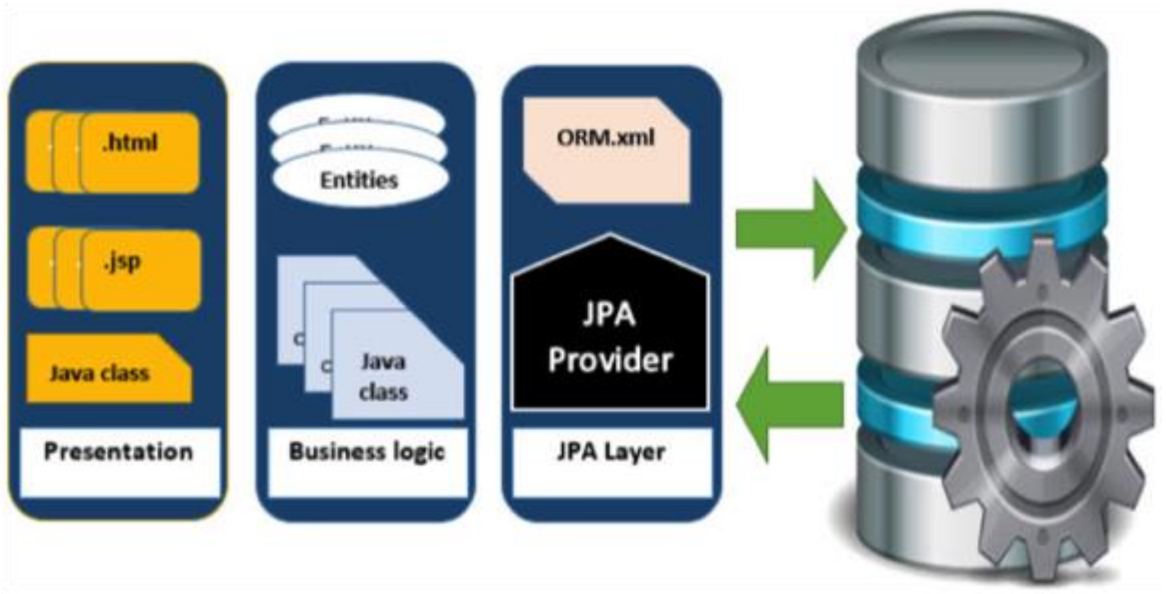
JPA

JPA is just a specification that facilitates object-relational mapping to manage relational data in Java applications. It provides a platform to work directly with objects instead of using SQL statements.

JPA Introduction

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

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 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.

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: -

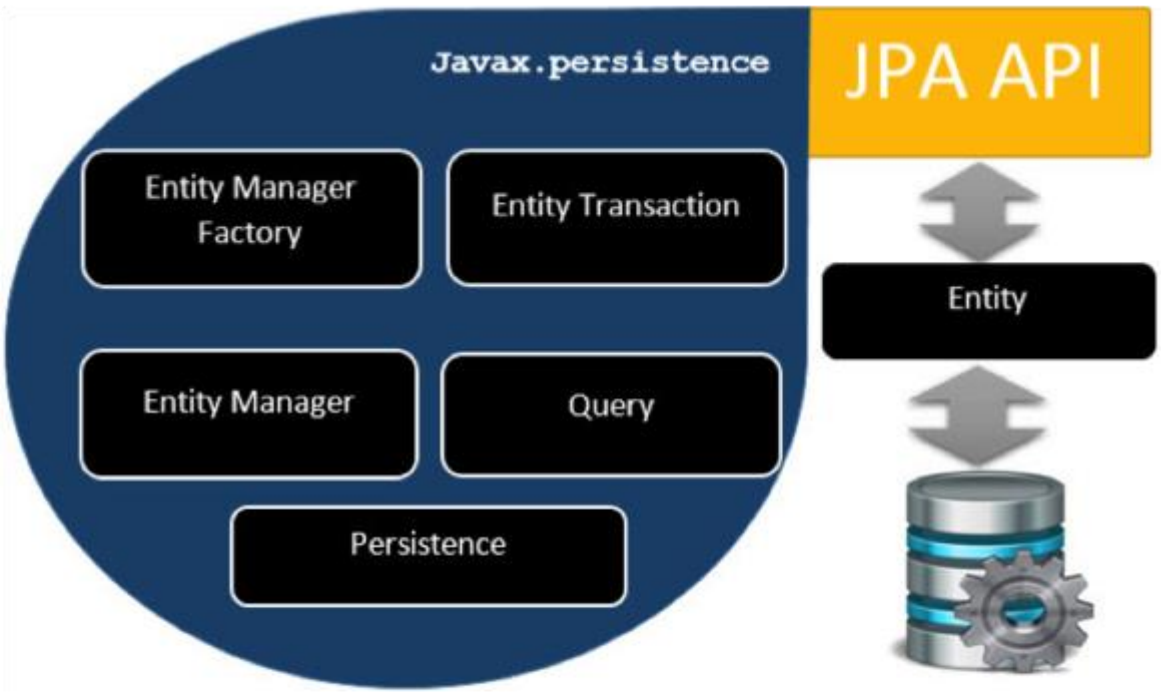
- JPA 2.0 - This version was released in the last of 2009. Following are the important features of this version: -
    - It supports validation.
    - It expands the functionality of object-relational mapping.
    - It shares the object of cache support.
- JPA 2.1 - The JPA 2.1 was released in 2013 with 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 – 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.



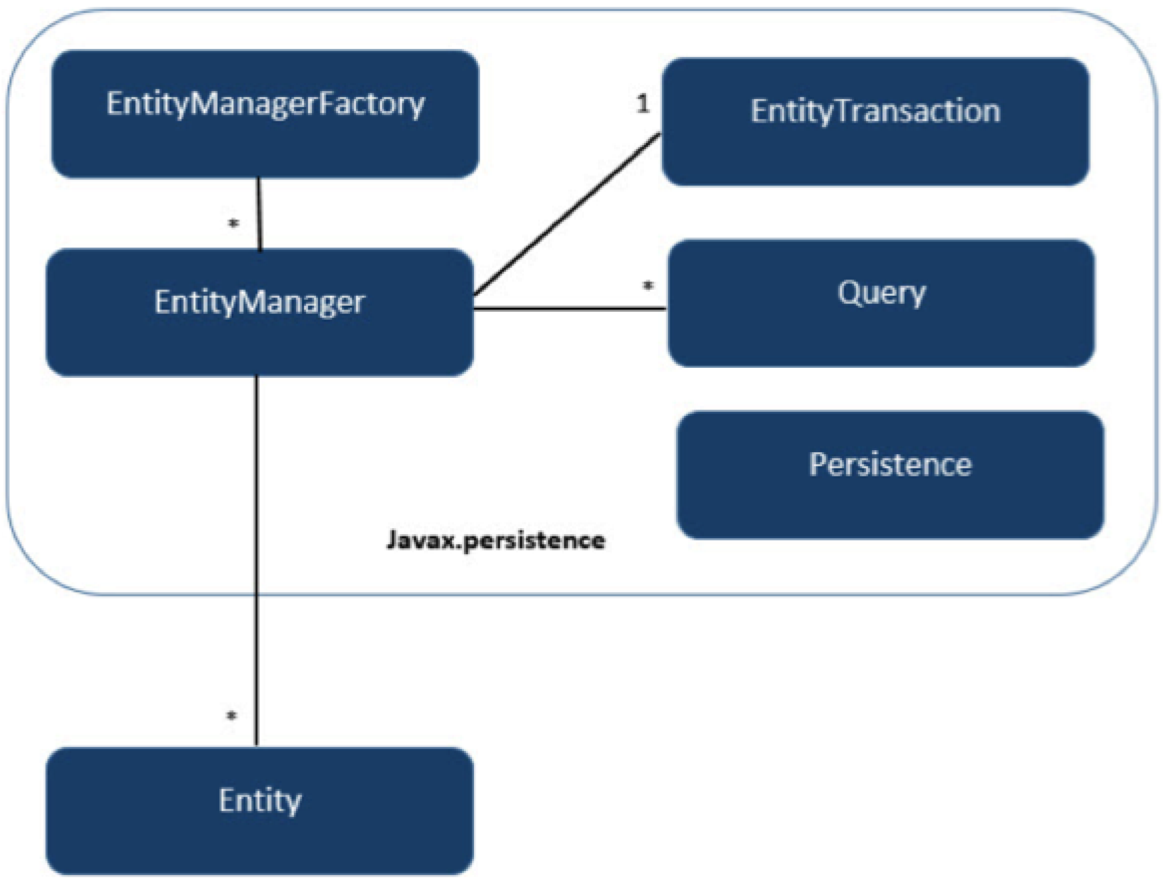
The following table describes each of the units shown in the above architecture.

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.

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 number 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.

JPA - ORM Components

Most contemporary applications use relational database to store data. Recently, many vendors switched to object database to reduce their burden on data maintenance. It means object database or object relational technologies are taking care of storing, retrieving, updating, and maintenance. The core part of this object relational technologies are mapping orm.xml file. As xml does not require compilation, we can easily make changes to multiple data sources with less administration.

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.

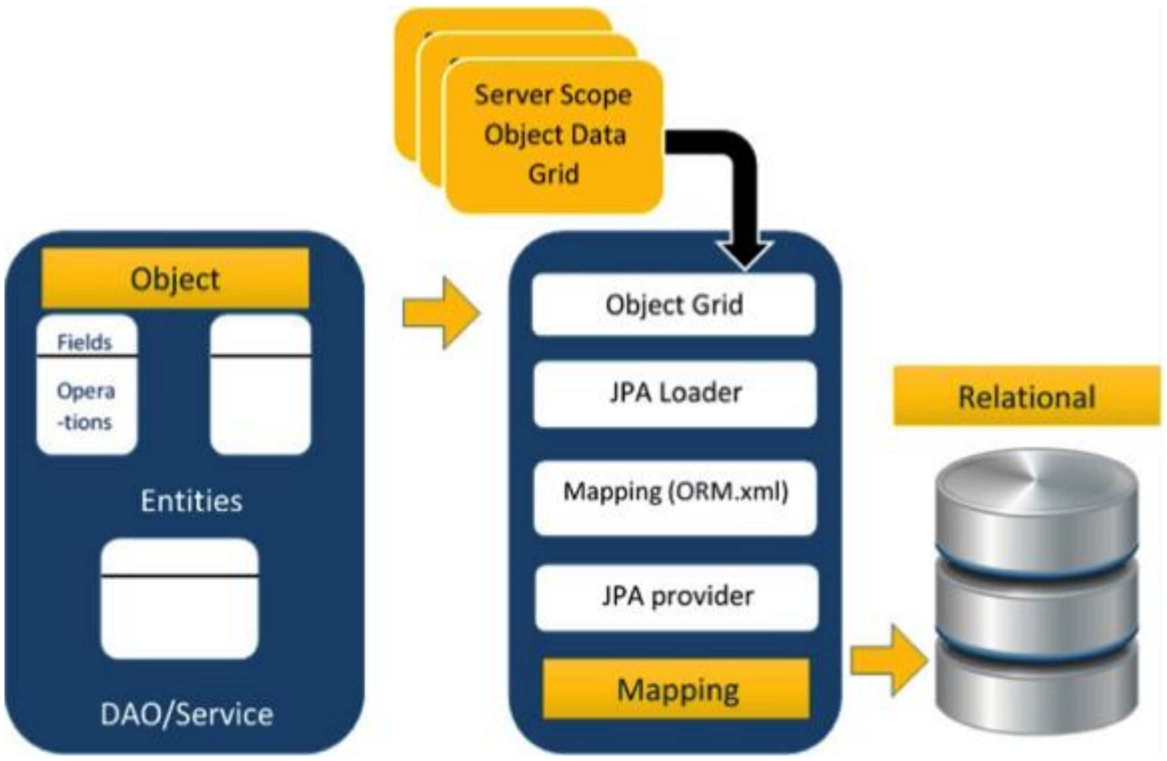
The main feature of ORM is mapping or binding an object to its data in the database. While mapping we have to consider the data, type of data and its relations with its self-entity or entity in any other table.

Advanced Features

- Idiomatic persistence : It enables you to write the persistence classes using object oriented classes.
- High Performance : It has many fetching techniques and hopeful locking techniques.
- Reliable : It is highly stable and eminent. Used by many industrial programmers.

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 effected 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 commit 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 mapping.xml 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:

```java
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 contain 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="EMPLOYEETABLE"/>
      <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 in 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 assess 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 callled 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 ()'.

JPA - Entity Managers

The main modules for this example are as follows:

- **Model or POJO**

  Employee.java
- **Persistence**

  Persistence.xml
- **Service**

  CreatingEmployee.java

  UpdatingEmployee.java

FindingEmployee.java

DeletingEmployee.java

Let us take the package hierarchy which we have used in the JPA installation with Eclipselink. Follow the hierarchy for this example as below:

Creating Entities

Entities are nothing but beans or Models, in this example we will use **Employee** as entity. **eid, ename, salary,** and **deg** are the attributes of this entity. It contains default constructor, setter and getter methods of those attributes.

In the above shown hierarchy, create a package named **'com.tutorialspoint.eclipselink.entity'**, under **'src'** (Source) package. Create a class named **Employee.java** under given package as follows:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table
public class Employee {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)

   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;
    }

    @Override
    public String toString() {
       return "Employee [eid=" + eid + ", ename=" + ename +
 ", salary=" + salary + ", deg=" + deg + "]";
    }
                          }
```

In the above code, we have used @Entity annotation to make this POJO class as entity.

Before going to next module we need to create database for relational entity, which will register the database in **persistence.xml** file. Open MySQL workbench and type query as follows:

```
create database jpadb
use jpadb
```

Persistence.xml

This module plays a crucial role in the concept of JPA. In this xml file we will register the database and specify the entity class.

In the above shown package hierarchy, persistence.xml under JPA Content package is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd
">

   <persistence-unit name="Eclipselink_JPA" transaction-
type="RESOURCE_LOCAL">

      <class>com.cts.Employee</class>

      <properties>
         <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/jpadb"/>
         <property name="javax.persistence.jdbc.user"
value="root"/>
         <property name="javax.persistence.jdbc.password"
value="root"/>
         <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>
         <property name="eclipselink.logging.level"
value="FINE"/>
         <property name="eclipselink.ddl-generation"
value="create-tables"/>
```

```
      </properties>

    </persistence-unit>
</persistence>
```

In the above xml, <persistence-unit> tag is defined with specific name for JPA persistence. The <class> tag defines entity class with package name. The <properties> tag defines all the properties, and <property> tag defines each property such as database registration, URL specification, username, and password. These are the Eclipselink properties. This file will configure the database.

Persistence Operations

Persistence operations are used against database and they are **load** and **store** operations. In a business component all the persistence operations fall under service classes.

In the above shown package hierarchy, create a package named **'com.cts.service'**, under **'src'** (source) package. All the service classes named as CreateEmloyee.java, UpdateEmployee.java, FindEmployee.java, and DeleteEmployee.java. comes under the given package as follows:

Create Employee

Creating an Employee class named as **CreateEmployee.java** as follows:

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class CreateEmployee {

   public static void main( String[ ] args ) {

      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );

      EntityManager entitymanager = emfactory.createEntityManager(
);
      entitymanager.getTransaction( ).begin( );

      Employee employee = new Employee( );
      employee.setEid( 1201 );
      employee.setEname( "Gopal" );
      employee.setSalary( 40000 );
      employee.setDeg( "Technical Manager" );

      entitymanager.persist( employee );
      entitymanager.getTransaction( ).commit( );
      entitymanager.close( );
      emfactory.close( );
   }
}
```

In the above code the **createEntityManagerFactory ()** creates a persistence unit by providing the same unique name which we provide for persistence-unit in persistent.xml file. The entitymanagerfactory object will create the entitymanger instance by using **createEntityManager ()** method. The entitymanager object creates entitytransaction instance for transaction management. By using entitymanager object, we can persist entities into database.

After compilation and execution of the above program you will get notifications from eclipselink library on the console panel of eclipse IDE.

For result, open the MySQL workbench and type the following queries.

```
use jpadb
select * from employee
```

The effected database table named **employee** will be shown in a tabular format as follows:

Eid	Ename	Salary	Deg
1201	Gopal	40000	Technical Manager

Spring Data JPA

Spring Data JPA API provides JpaTemplate class to integrate spring application with JPA.

JPA (Java Persistent API) is the sun specification for persisting objects in the enterprise application. It is currently used as the replacement for complex entity beans.

The implementation of JPA specification are provided by many vendors such as:

- Hibernate
- Toplink
- iBatis

- OpenJPA etc.

Advantage of Spring JpaTemplate

You don't need to write the before and after code for persisting, updating, deleting or searching object such as creating Persistence instance, creating EntityManagerFactory instance, creating EntityTransaction instance, creating EntityManager instance, commiting EntityTransaction instance and closing EntityManager.

So, it **save a lot of code**.

In this example, we are going to use hibernate for the implementation of JPA.
