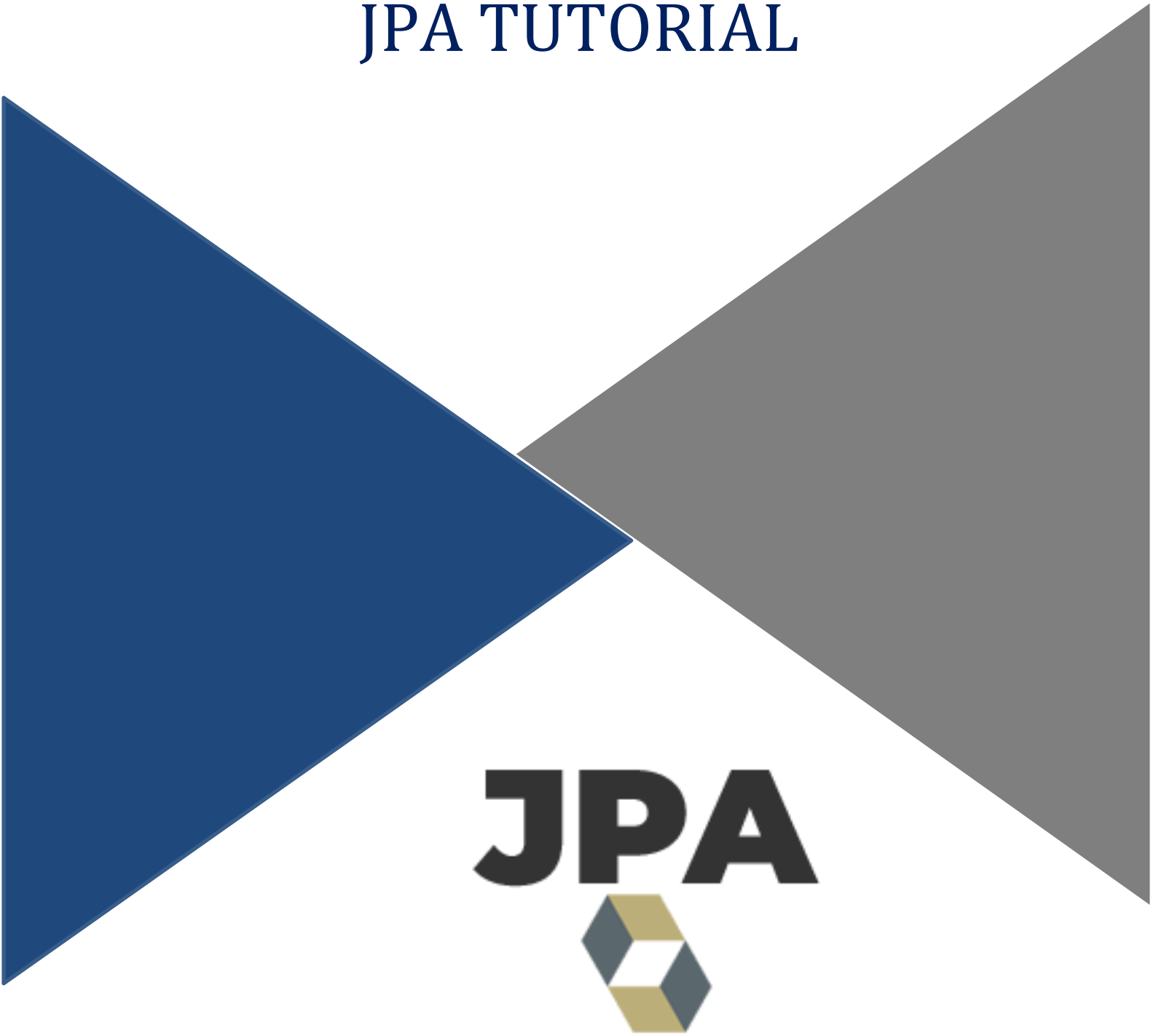
ACE Inspiration

Professional Development Center

JPA TUTORIAL



ACE Inspiration

Professional Development Center

About the Tutorial

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database. This tutorial provides you the basic understanding of Persistence (storing the copy of database object into temporary memory), and we will learn the understanding of JAVA Persistence API (JPA).

Audience

This tutorial is designed for the readers pursuing Java programming with Database, using Persistence API and for all the enthusiastic readers.

Prerequisites

Before proceeding with this tutorial, you should have a good understanding of Java programming language and JDBC knowledge. It is required that you have adequate exposure to SQL and Database concepts.

Table of Contents

1. JPA - OVERVIEW

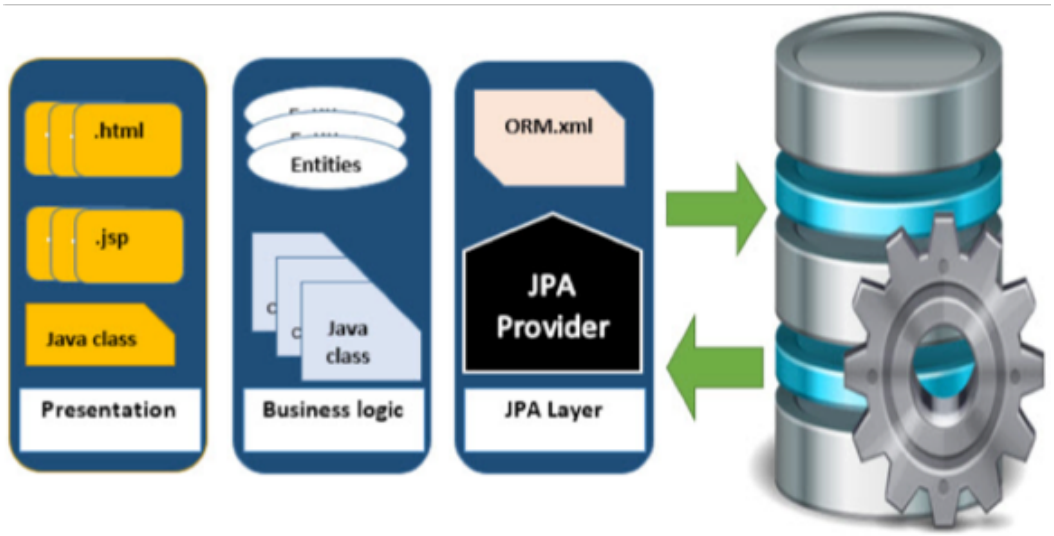
The Java Persistence API (JPA) is a standard API for accessing databases from within Java applications.
JPA acts as a bridge between object-oriented domain models and relational database systems.
The main advantage of JPA over JDBC (the older Java API for interacting with databases) is that in JPA data is represented by classes and objects rather than by tables and records as in JDBC.
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.
It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.

JPA Versions

- The first version of Java Persistence API, JPA 1.0 was released in 2006.
- JPA 2.0 - This version was released in the last of 2009.
- JPA 2.1 - The JPA 2.1 was released in 2013.
- JPA 2.2 - The JPA 2.2 was released as a development of maintainance in 2017.

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.

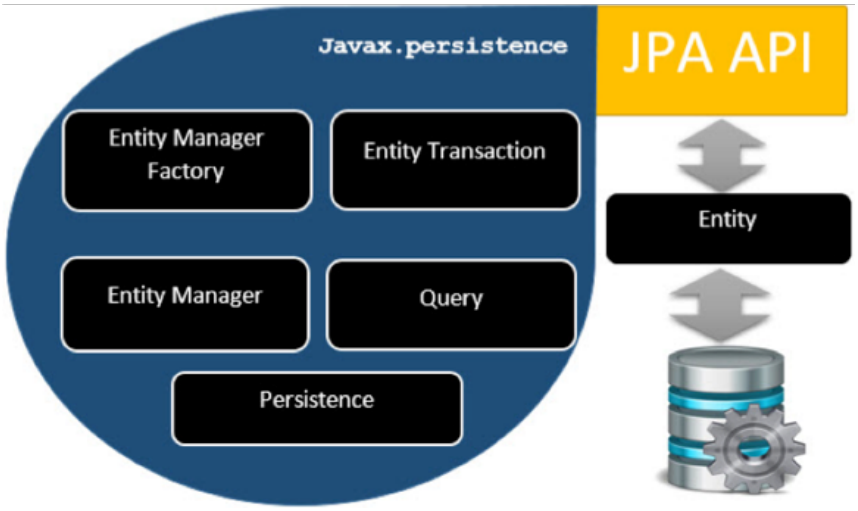


JPA Providers

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.

Class and Interface in JPA

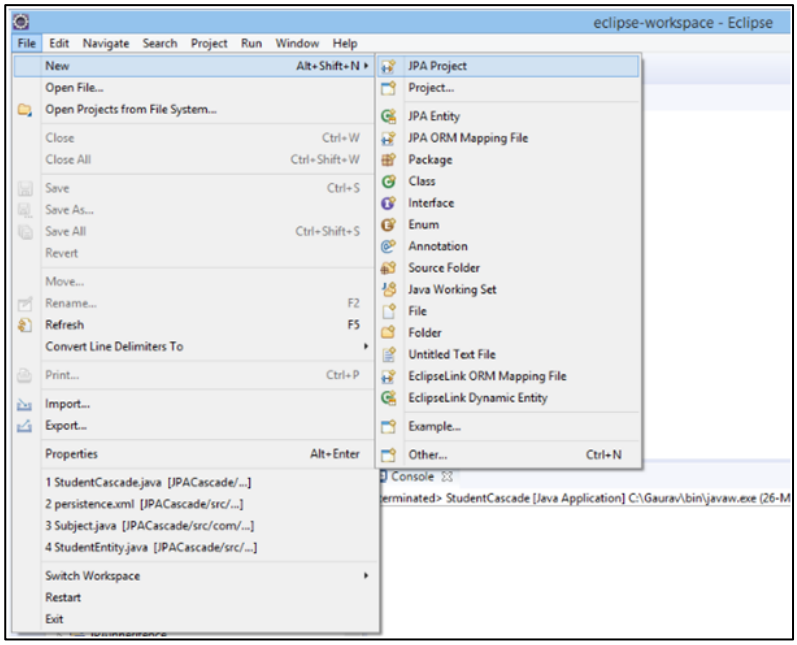


Units	Description
EntityManagerFactory	This is a factory class of EntityManager. It creates and manages multiple EntityManager instances.
EntityManager	It is an Interface, it manages the persistence operations on objects. It works like factory for Query instance.
Entity	Entities are the persistence objects, 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.

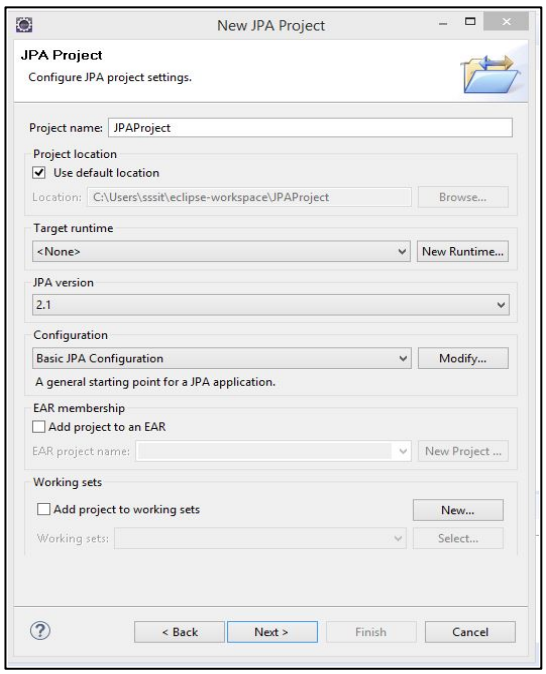
2. JPA - Installation

To install JPA on eclipse. See the following steps: -

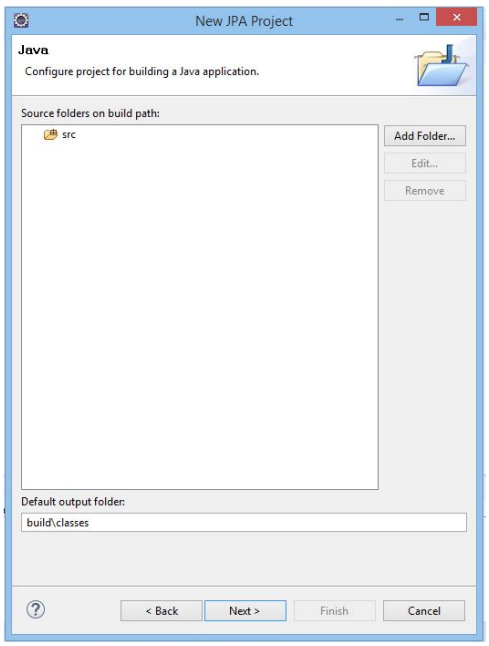
- Open eclipse and click on File>New>JPA Project.



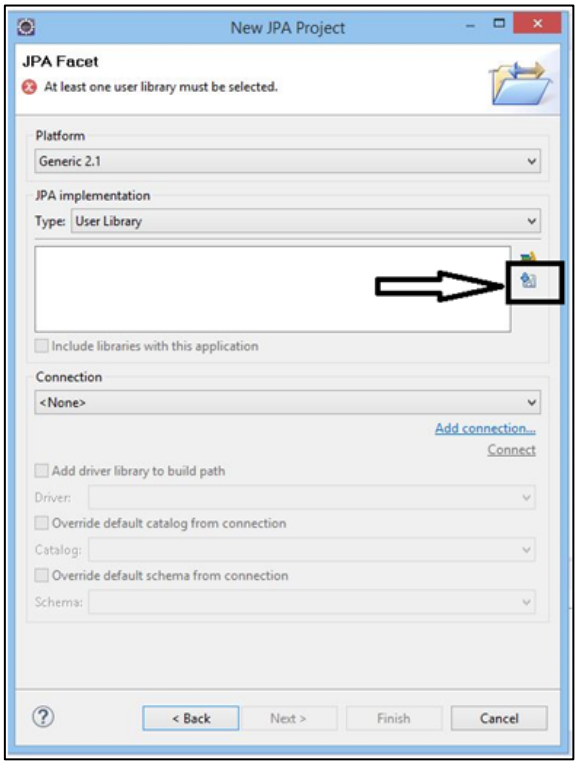
- Specify any particular project name (here, we named JPAProject) and click next.



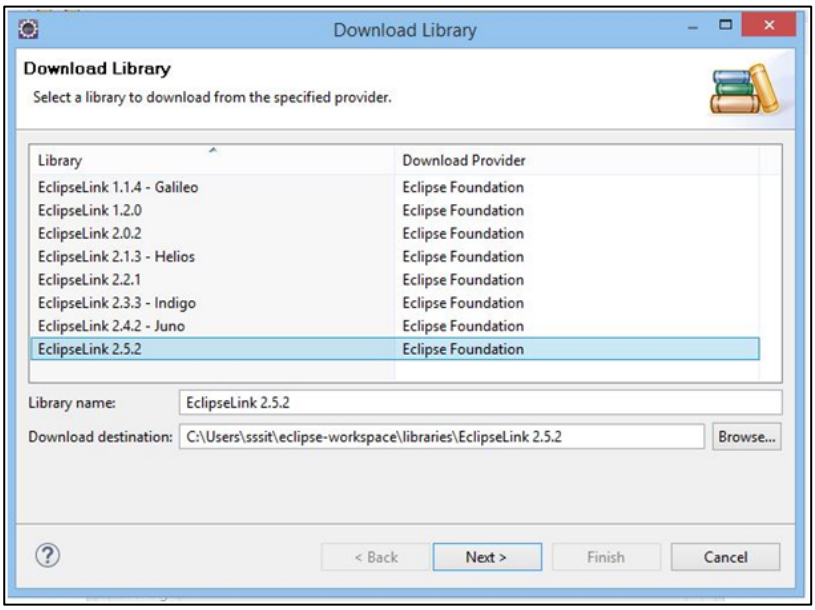
- Again, click next.



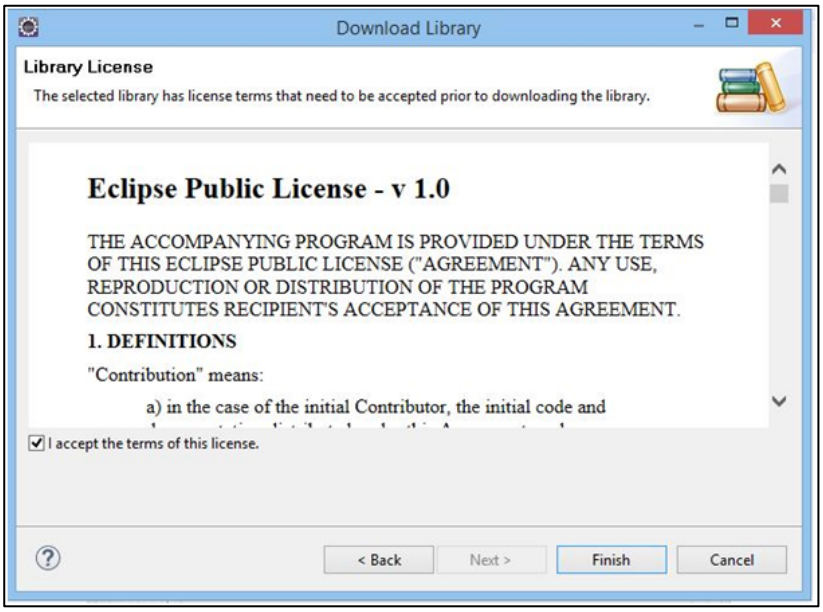
- Click on download library icon (here, enclosed within black box).



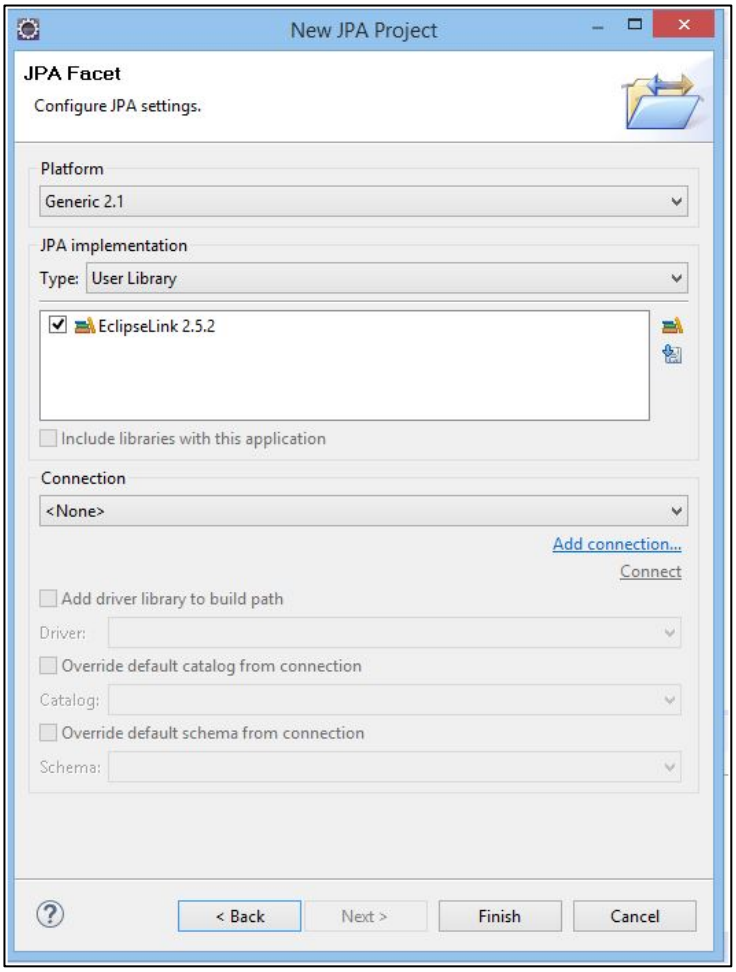
- Click on EclipseLink 2.5.2 and then next.



- Click on checkbox to accept the terms and then click finish. After that all the required jars will be downloaded.



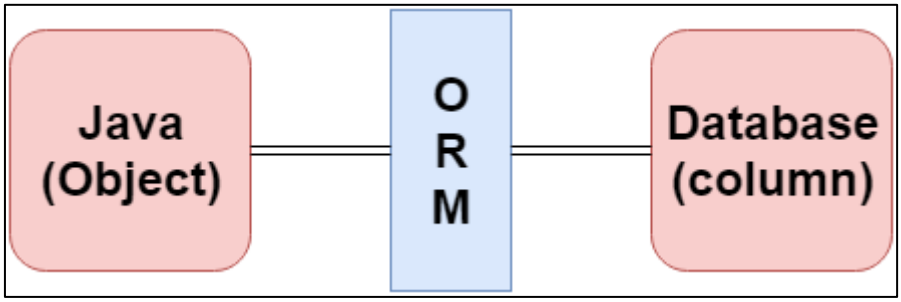
- Now,click finish.



3. JPA - Object Relational Mapping (ORM)

Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column.

It is capable to handle various database operations easily such as inserting, updating, deleting etc.



ORM Frameworks

Following are the various frameworks that function on ORM mechanism: -

- Hibernate
- TopLink
- ORMLite
- iBATIS
- JPOX

Mapping Directions

- **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.
- **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

Types of Mapping

- **One-to-one** - This association is represented by @OneToOne annotation. Here, instance of each entity is related to a single instance of another entity.

- **One-to-many** - This association is represented by @OneToMany annotation. In this relationship, an instance of one entity can be related to more than one instance of another entity.

- **Many-to-one** - This mapping is defined by @ManyToOne annotation. In this relationship, multiple instances of an entity can be related to single instance of another entity.

- **Many-to-many** - This association is represented by @ManyToMany annotation. Here, multiple instances of an entity can be related to multiple instances of another entity. In this mapping, any side can be the owing side.

4. JPA - Entity

In general, entity is a group of states associated together in a single unit. On adding behaviour, an entity behaves as an object.

Entity Properties

These are the properties of an entity that an object must have: -

Persistability - An object is called persistent if it is stored in the database and can be accessed anytime.

Persistent Identity - In Java, each entity is unique and represents as an object identity. Similarly, when the object identity is stored in a database then it is represented as persistence identity. This object identity is equivalent to primary key in database.

Transactionality - Entity can perform various operations such as create, delete, update. Each operation makes some changes in the database. It ensures that whatever changes made in the database either be succeed or failed atomically.

Granuality - Entities should not be primitives, primitive wrappers or built-in objects.

Entity Metadata

Each entity is associated with some metadata that represents the information of it. Instead of database, this metadata is exist either inside or outside the class. This metadata can be in following forms: -

- **Annotation -** In Java, annotations are the form of tags that represents metadata. This metadata persist inside the class.
- **XML -** In this form, metadata persist outside the class in XML file.

Creating an Entity

A Java class can be easily transformed into an entity. For transformation the basic requirements are: -

- No-argument Constructor
- Annotation

Student.java

```
public class Student {

   private int id;
   private String name;
   private long fees;

   public Student() {}

//getter and setter
}
```

Above class is a regular java class having three attributes id, name and fees. To transform this class into an entity add @Entity and @Id annotation in it.

- **@Entity -** This is a marker annotation which indicates that this class is an entity. This annotation must be placed on the class name.
- **@Id -** This annotation is placed on a specific field that holds the persistent identifying properties. This field is treated as a primary key in database.

Simple Entity Class

```
import javax.persistence.*;

@Entity
public class Student {

   @Id
   private int id;

   private String name;
   private long fees;

   public Student() {}
   public Student(int id){
      this.id = id;
   }
}
```

5. JPA - Entity Manager

Following are some of the important roles of an entity manager: -

- The entity manager implements the API and encapsulates all of them within a single interface.
- Entity manager is used to read, delete and write an entity.
- An object referenced by an entity is managed by entity manager.

Steps to persist an entity object

1) Creating an entity manager factory object

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student_details");
```

- **Persistence -** The Persistence is a bootstrap class which is used to obtain an EntityManagerFactory interface.
- **createEntityManagerFactory() method -** The role of this method is to create and return an EntityManagerFactory for the named persistence unit. Thus, this method contains the name of persistence unit passed in the Persistence.xml file.

2) Obtaining an entity manager from factory.

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

- **EntityManager -** An EntityManager is an interface
- **createEntityManager() method -** It creates new application-managed EntityManager

3) Intializing an entity manager.

```
em.getTransaction().begin();
```

- **getTransaction() method -** This method returns the resource-level EntityTransaction object.
- **begin() method -** This method is used to start the transaction.

4) Persisting a data into relational database.

```
em.persist(s1);
```

- **persist() -** This method is used to make an instance managed and persistent. An entity instance is passed within this method.

5) Closing the transaction

```
em.getTransaction().commit();
```

6) Releasing the factory resources.

```
emf.close();
em.close();
```

- **close() -** This method is used to releasing the factory resources.

Entity Operations

- Inserting an Entity
- Finding an Entity
- Updating an Entity
- Deleting an Entity

Inserting an Entity

The EntityManager provides persist() method to insert records.

StudentEntity.java

```
import javax.persistence.*;

@Entity
@Table(name="student")
public class StudentEntity {

   @Id
   private int s_id;
   private String s_name;
   private int s_age;

   public StudentEntity(int s_id, String s_name, int s_age) {
      super();
      this.s_id = s_id;
      this.s_name = s_name;
      this.s_age = s_age;
   }
}
```

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"          xmlns="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">

  <persistence-unit name="Student_details">
     <class>com.javatpoint.jpa.student.StudentEntity</class>
 <properties>
 <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
 <property name="javax.persistence.jdbc.url"
 value="jdbc:mysql://localhost:3306/studentdata"/>
 <property name="javax.persistence.jdbc.user" value="root"/>
 <property name="javax.persistence.jdbc.password" value=""/>
 <property name="eclipselink.logging.level" value="SEVERE"/>
 <property name="eclipselink.ddl-generation" value="create-or-extend-tables"/>
 </properties>
 </persistence-unit>  </persistence>
```

PersistStudent.java

```
import javax.persistence.*;
public class PersistStudent {

   public static void main(String args[])
    {

        EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student_details");
        EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

        StudentEntity s1=new StudentEntity();
        s1.setS_id(101);
        s1.setS_name("Allen");
        s1.setS_age(24);

        StudentEntity s2=new StudentEntity();
        s2.setS_id(102);
        s2.setS_name("Joe");
        s2.setS_age(22);

        StudentEntity s3=new StudentEntity();
        s3.setS_id(103);
        s3.setS_name("Smith");
        s3.setS_age(26);

        em.persist(s1);
        em.persist(s2);
        em.persist(s3);

em.getTransaction().commit();

        emf.close();
        em.close();

    }
}
```

Finding an entity

EntityManger interface provides find() method that searches an element on the basis of primary key.

FindStudent.java

```
import javax.persistence.*;

public class FindStudent {
public static void main(String args[])
{
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student_details");
EntityManager em=emf.createEntityManager();

StudentEntity s=em.find(StudentEntity.class,101);

System.out.println("Student id = "+s.getS_id());
System.out.println("Student Name = "+s.getS_name());
System.out.println("Student Age = "+s.getS_age());

}
}
```

Updating an entity

UpdateStudent.java

```
import javax.persistence.*;

public class UpdateStudent {

public static void main(String args[])
{
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student_details");
EntityManager em=emf.createEntityManager();


StudentEntity s=em.find(StudentEntity.class,102);
System.out.println("Before Updation");
System.out.println("Student id = "+s.getS_id());
System.out.println("Student Name = "+s.getS_name());
System.out.println("Student Age = "+s.getS_age());

s.setS_age(30);

System.out.println("After Updation");
System.out.println("Student id = "+s.getS_id());
System.out.println("Student Name = "+s.getS_name());
System.out.println("Student Age = "+s.getS_age());

}
}
```

Deleting an entity

EntityManager interface provides remove() method. The remove() method uses primary key to delete the particular record.

DeleteStudent.java

```
import javax.persistence.*;

public class DeleteStudent {

public static void main(String args[])
{
EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student_details");
EntityManager em=emf.createEntityManager();
em.getTransaction().begin();

StudentEntity s=em.find(StudentEntity.class,102);
em.remove(s);
em.getTransaction().commit();
emf.close();
em.close();

}
}
```

6. JPA - Collection Mapping

In JPA, we can persist the object of wrapper classes and String using collections.

JPA allows three kinds of objects to store in mapping collections - Basic Types, Entities and Embeddables.

Collection Types

On the basis of requirement, we can use different type of collections to persist the objects.

- List
- Set
- Map

JPA List Mapping

A List is an interface which is used to insert and delete elements on the basis of index. It can be used when there is a requirement of retrieving elements in a user-defined order.

Employee.java

```
import java.util.*;
import javax.persistence.*;

@Entity
public class Employee {

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int e_id;
private String e_name;

@ElementCollection
private List<Address> address=new ArrayList<Address>();

// getter and setter
}
```

Address.java

```
import javax.persistence.*;

@Embeddable
public class Address {

private int e_pincode;
private String e_city;
private String e_state;

//getter and setter

}
```

ListMapping.java

```
import javax.persistence.*;

public class ListMapping{

public static void main(String[] args) {

EntityManagerFactory emf=
Persistence.createEntityManagerFactory("Collection_Type");

EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

Address a1=new Address();
a1.setE_pincode(201301);
a1.setE_city("United States");
a1.setE_state("United States");

Address a2=new Address();
a2.setE_pincode(302001);
a2.setE_city("London");
a2.setE_state("London");

Employee e1=new Employee();
e1.setE_id(1);
e1.setE_name("Smith");
e1.getAddress().add(a1);
```

```
Employee e2=new Employee();
e2.setE_id(2);
e2.setE_name("John");
e2.getAddress().add(a2);

em.persist(e1);
em.persist(e2);

em.getTransaction().commit();

em.close();
emf.close();

}
}
```

JPA Set Mapping

A Set is an interface that contains unique elements. These elements don't maintain any order. A Set can be used when there is a requirement of retrieving unique elements in an unordered manner.

Employee.java

```
import java.util.*;
import javax.persistence.*;

@Entity
public class Employee {

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int e_id;
private String e_name;

@ElementCollection
private Set<Address> address=new HashSet<Address>();

// getter and setter
}
```

SetMapping.java

```
import javax.persistence.*;

public class SetMapping{

   public static void main(String[] args) {

   EntityManagerFactory emf=
   Persistence.createEntityManagerFactory("Collection_Type");

   EntityManager em=emf.createEntityManager();

      em.getTransaction().begin();

      Address a1=new Address();
      a1.setE_pincode(201301);
      a1.setE_city("United States");
      a1.setE_state("United States");


      Address a2=new Address();
      a2.setE_pincode(302001);
      a2.setE_city("London");
      a2.setE_state("London");

      Address a3=new Address();
      a3.setE_pincode(133301);
      a3.setE_city("California");
      a3.setE_state("California ");

      Address a4=new Address();
      a4.setE_pincode(80001);
      a4.setE_city("New York");
      a4.setE_state("New York");


   Employee e1=new Employee();
   e1.setE_id(1);
   e1.setE_name("Sparky");


   Employee e2=new Employee();
   e2.setE_id(2);
   e2.setE_name("Allen");
```

```
 Employee e3=new Employee();
  e3.setE_id(3);
  e3.setE_name("William");

  Employee e4=new Employee();
  e4.setE_id(4);
  e4.setE_name("John");

  e1.getAddress().add(a1);
  e2.getAddress().add(a2);
  e3.getAddress().add(a3);
  e4.getAddress().add(a4);

  em.persist(e1);
  em.persist(e2);
  em.persist(e3);
  em.persist(e4);

  em.getTransaction().commit();

  em.close();
  emf.close();
  }
}
```

JPA Map Mapping

A Map is an interface in which a unique key is associated with each value object. Thus, operations like search, update, delete are performed on the basis of key.

Employee.java

```
  import java.util.*;
  import javax.persistence.*;

  @Entity
  public class Employee {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int e_id;
  private String e_name;

  @ElementCollection
  private Map<Integer,Address> map=new HashMap<Integer,Address>();
  // getter and setter
  }
```

7. JPA - Type of Mapping

There are four types of mapping in JPA:

1. One-To-One Mapping
2. One-To-Many Mapping
3. Many-To-One Mapping
4. Many-To-Many Mapping

One-To-One Mapping

The One-To-One mapping represents a single-valued association where an instance of one entity is associated with an instance of another entity. In this type of association one instance of source entity can be mapped atmost one instance of target entity.

In this example,

we will create a One-To-One relationship between a Student and Library in such a way that one student can be issued only one type of book.

Student.java

```
import javax.persistence.*;
@Entity
public class Student {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int s_id;
    private String s_name;
    public int getS_id() {
        return s_id;
    }
    public void setS_id(int s_id) {
        this.s_id = s_id;
    }
    public String getS_name() {
        return s_name;
    }
    public void setS_name(String s_name) {
        this.s_name = s_name;
    }
    }
```

Library.java

```
import javax.persistence.*;

@Entity
public class Library {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int b_id;
	private String b_name;

	@OneToOne
	private Student stud;

	public Library(int b_id, String b_name, Student stud) {
		super();
		this.b_id = b_id;
		this.b_name = b_name;
		this.stud = stud;
	}

	public Library() {
		super();

	}

	public int getB_id() {
		return b_id;
	}

	public void setB_id(int b_id) {
		this.b_id = b_id;
	}

	public String getB_name() {
		return b_name;
	}

	public void setB_name(String b_name) {
		this.b_name = b_name;
	}

	public Student getStud() {
		return stud;
	}

	public void setStud(Student stud) {
		this.stud = stud;
	}

}
```

OneToOneExample.java

```
import javax.persistence.*;

public class OneToOneEg {
      public static void main(String[] args) {

       EntityManagerFactory emf =
          Persistence.createEntityManagerFactory("Book_issued");


         EntityManager em = emf.createEntityManager( );
         em.getTransaction( ).begin( );

         Student st1=new Student();
         st1.setS_id(1);
         st1.setS_name("Jon");

         Student st2=new Student();
         st2.setS_id(2);
         st2.setS_name("Smith");


         em.persist(st1);
         em.persist(st2);

         Library lib1=new Library();
         lib1.setB_id(101);
         lib1.setB_name("Data Structure");
         lib1.setStud(st1);

         Library lib2=new Library();
         lib2.setB_id(102);
         lib2.setB_name("DBMS");
         lib2.setStud(st2);

         em.persist(lib1);
         em.persist(lib2);

         em.getTransaction().commit();

         em.close();
         emf.close();
   }

}
```

One-To-Many Mapping

The One-To-Many mapping comes into the category of collection-valued association where an entity is associated with a collection of other entities. Hence, in this type of association the instance of one entity can be mapped with any number of instances of another entity.

In this example,

we will create a One-To-Many relationship between a Student and Library in such a way that one student can be issued more than one type of book.

Student.java

```
import java.util.List;
import javax.persistence.*;

@Entity
public class Student {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
   private int s_id;
   private String s_name

   @OneToMany(targetEntity=Library.class)
   private List books_issued;
// getter, setter
}
```

Library.java

```
import javax.persistence.*;

@Entity
public class Library {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
private int b_id;
private String b_name;
// getter, setter

}
```

OneToManyExample.java

```
import java.util.ArrayList;
 import javax.persistence.*;
public class OneToManyExample {
    public static void main(String[] args) {
        EntityManagerFactory emf=
        Persistence.createEntityManagerFactory("books_issued");
        EntityManager em=emf.createEntityManager();

        em.getTransaction().begin();

         Library lib1=new Library();
           lib1.setB_id(101);
           lib1.setB_name("Data Structure");

           Library lib2=new Library();
           lib2.setB_id(102);
           lib2.setB_name("DBMS");

           em.persist(lib1);
           em.persist(lib2);

           ArrayList<Library> list=new ArrayList<Library>();
           list.add(lib1);
           list.add(lib2);

        Student st1=new Student();
         st1.setS_id(1);
           st1.setS_name("Smith");
           st1.setBooks_issued(list);

           em.persist(st1);

        em.getTransaction().commit();
        em.close();
        emf.close();
          }
      }
```

Many-To-One Mapping

The Many-To-One mapping represents a single-valued association where a collection of entities can be associated with the similar entity. Hence, in relational database any more than one row of an entity can refer to the similar rows of another entity.

In this example,

we will create a Many-To-One relationship between a Student and Library in such a way that more than one student can issued the same book.

Student.java

```
import javax.persistence.*;

@Entity
public class Student {

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int s_id;
private String s_name;

@ManyToOne
private Library lib;

//getter , setter
}
```

Library.java

```
import javax.persistence.*;


@Entity
public class Library {
   @Id
   @GeneratedValue(strategy=GenerationType.AUTO)
private int b_id;
private String b_name;
// getter, setter


}
```

ManyToOneExample.java

```
import javax.persistence.*;
import javax.persistence.EntityManagerFactory;

public class ManyToOneExample {

public static void main(String[] args) {

EntityManagerFactory emf=
Persistence.createEntityManagerFactory("books_issued");

EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

Library lib=new Library();
lib.setB_id(101);
lib.setB_name("Data Structure");

em.persist(lib);

Student st1=new Student();
st1.setS_id(1);
st1.setS_name("Jon");
st1.setLib(lib);


Student st2=new Student();
st2.setS_id(2);
st2.setS_name("Smith");
st2.setLib(lib);

em.persist(st1);;
em.persist(st2);


em.getTransaction().commit();
em.close();
emf.close();

}

}
```

Many-To-Many Mapping

The Many-To-Many mapping represents a collection-valued association where any number of entities can be associated with a collection of other entities. In relational database any number of rows of one entity can be referred to any number of rows of another entity.

In this example,

we will create a Many-To-Many relationship between a Student and Library in such a way that any number of students can be issued any type of books.

Student.java

```
import javax.persistence.*;

@Entity
public class Student {

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int s_id;
private String s_name;

@ManyToMany(targetEntity=Library.class)
private List lib;

//getter , setter
}
```

Library.java

```
import java.util.List;

import javax.persistence.*;

@Entity
public class Library {
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private int b_id;
private String b_name;

@ManyToMany(targetEntity=Student.class)
private List stud;

 //getter, setter
 }
```

ManyToManyExample.java

```
import javax.persistence.*;

public class ManyToManyExample {

public static void main(String[] args) {

EntityManagerFactory emf=Persistence.createEntityManagerFactory("books_issued");
EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

Student st1=new Student(1,"Jon",null);
Student st2=new Student(2,"Smith",null);

em.persist(st1);
em.persist(st2);


ArrayList<Student> al1=new ArrayList<Student>();
ArrayList<Student> al2=new ArrayList<Student>();

al1.add(st1);
al1.add(st2);

al2.add(st1);
al2.add(st2);

Library lib1=new Library(101,"Data Structure",al1);
Library lib2=new Library(102,"DBMS",al2);


em.persist(lib1);
em.persist(lib2);

em.getTransaction().commit();
em.close();
mf.close();

}


}
```

8. JPA - Cascading

In JPA, if any operation is applied on an entity then it will perform on that particular entity only. These operations will not be applicable to the other entities that are related to it.

To establish a dependency between related entities, JPA provides javax.persistence.CascadeType enumerated types that define the cascade operations. These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many.

Operation	Description
PERSIST	In this cascade operation, if the parent entity is persisted then all its related entity will also be persisted.
MERGE	In this cascade operation, if the parent entity is merged then all its related entity will also be merged.
DETACH	In this cascade operation, if the parent entity is detached then all its related entity will also be detached.
REFRESH	In this cascade operation, if the parent entity is refreshed then all its related entity will also be refreshed.
REMOVE	In this cascade operation, if the parent entity is removed then all its related entity will also be removed.
ALL	In this case, all the above cascade operations can be applied to the entities related to parent entity.

Cascade Persist

The cascade persist is used to specify that if an entity is persisted then all its associated child entities will also be persisted.

StudentEntity.java

```
import javax.persistence.*;

@Entity
@Table(name="student")
public class StudentEntity {

@Id
private int s_id;
private String s_name;
private int s_age;

@OneToOne(cascade=CascadeType.PERSIST)
private Subject sub;


// getter, setter
}
```

Subject.java

```
import javax.persistence.*;

@Entity
@Table(name="subject")

public class Subject {

private String name;
private int marks;
@Id
private int s_id;

//getter, setter

}
```

StudentCascade.java

```
import javax.persistence.*;

public class StudentCascade {

public static void main( String[ ] args ) {

EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );

EntityManager em = emf.createEntityManager( );
em.getTransaction().begin();


StudentEntity s1=new StudentEntity();
s1.setS_id(101);
s1.setS_name("Jon");
s1.setS_age(20);

StudentEntity s2=new StudentEntity();
s2.setS_id(102);
s2.setS_name("Smith");
s2.setS_age(22);

Subject sb1=new Subject();
sb1.setName("ENGLISH");
sb1.setMarks(80);
sb1.setS_id(s1.getS_id());

Subject sb2=new Subject();
sb2.setName("Maths");
sb2.setMarks(75);
sb2.setS_id(s2.getS_id());

s1.setSub(sb1);
s2.setSub(sb2);


em.persist(s1);
em.persist(s2);

em.getTransaction().commit();

em.close( );
emf.close( );

}
}
```

Cascade Remove

The cascade remove is used to specify that if the parent entity is removed then all its related entities will also be removed.

StudentEntity.java

```
import javax.persistence.*;

@Entity
@Table(name="student")
public class StudentEntity {

    @Id
    private int s_id;
    private String s_name;
    private int s_age;

    @OneToOne(cascade={CascadeType.REMOVE})
    private Subject sub;

// getter, setter

}
```

Subject.java

```
import javax.persistence.*;

@Entity
@Table(name="subject")

public class Subject {

private String name;
private int marks;
@Id
private int s_id;

//getter, setter

}
```

StudentCascade.java

```
import javax.persistence.*;
import com.javatpoint.jpa.student.*;
public class StudentCascade {

public static void main( String[ ] args ) {

EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );

EntityManager em = emf.createEntityManager( );
em.getTransaction().begin();


StudentEntity s=em.find(StudentEntity.class, 101);
em.remove(s);


em.getTransaction().commit();

em.close( );
emf.close( );

}

}
```

9. JPA - JPQL

JPQL Introduction

The JPQL (Java Persistence Query Language) is an object-oriented query language which is used to perform database operations on persistent entities.

Instead of database table, JPQL uses entity object model to operate the SQL queries.

Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks.

JPQL is an extension of Entity JavaBeans Query Language (EJBQL), adding the following important features to it: -

- It can perform join operations.
- It can update and delete data in a bulk.
- It can perform aggregate function with sorting and grouping clauses.
- Single and multiple value result types.

JPQL Features

- It is a platform-independent query language.
- It is simple and robust.
- It can be used with any type of database such as MySQL, Oracle.
- JPQL queries can be declared statically into metadata or can also be dynamically built in code.

Creating Queries in JPQL

JPQL provides two methods that can be used to access database records. These methods are: -

Query createQuery(String name) - The createQuery() method of EntityManager interface is used to create an instance of Query interface for executing JPQL statement.

Query createNamedQuery(String name) - The createNamedQuery() method of EntityManager interface is used to create an instance of Query interface for executing named queries.

Query interface methods: -

- int executeUpdate() - This method executes the update and delete operation.
- int getFirstResult() - This method returns the first positioned result the query object was set to retrieve.
- int getMaxResults() - This method returns the maximum number of results the query object was set to retrieve.
- java.util.List getResultList() - This method returns the list of results as an untyped list.
- Query setFirstResult(int startPosition) - This method assigns the position of first result to retrieve.
- Query setMaxResults(int maxResult) - This method assigns the maximum numbers of result to retrieve.

JPQL Basic Operations

JPQL Dynamic Query Example

StudentEntity.java

```
import javax.persistence.*;

@Entity
@Table(name="student")
public class StudentEntity {

@Id
private int s_id;
private String s_name;
private int s_age;


// getter, setter


}
```

FetchColumn.java

```
import javax.persistence.*;
import java.util.*;
public class FetchColumn {

    public static void main( String args[]) {

        EntityManagerFactory emf =
        Persistence.createEntityManagerFactory( "Student_details" );

        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

        Query query = em.createQuery("Select s.s_name from StudentEntity s");
        @SuppressWarnings("unchecked")
      List<String> list =query.getResultList();
        System.out.println("Student Name :");
        for(String s:list) {

           System.out.println(s);

        }

        em.close();
        emf.close();
    }

}
```

JPQL Static Query Example

StudentEntity.java

```
import javax.persistence.*;

   @Entity
   @Table(name="student")
   @NamedQuery(name = "find name" , query = "Select s from StudentEntity s")
   public class StudentEntity {

      @Id
      private int s_id;
      private String s_name;
      private int s_age;

//getter, setter

}
```

FetchColumn.java

```
import javax.persistence.*;
import java.util.*;
public class FetchColumn {

    public static void main( String args[]) {

        EntityManagerFactory emf =
          Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

        Query query = em.createNamedQuery("find name");

       List<StudentEntity> list =query.getResultList();
        System.out.println("Student Name :");
        for(StudentEntity s:list) {

          System.out.println(s.getS_name());

        }

        em.close();
        emf.close();
    }

}
```

JPQL Bulk Data Operations

JPQL Fetch

FetchData.java

```
import javax.persistence.*;
import java.util.*;
public class FetchData {

    public static void main( String args[]) {

        EntityManagerFactory emf =
          Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );
        Query query = em.createQuery( "Select s from StudentEntity s ");

       List<StudentEntity> list=(List<StudentEntity>)query.getResultList( );

        System.out.print("s_id");
        System.out.print("\t s_name");
       System.out.println("\t s_age");

        for( StudentEntity s:list ){
           System.out.print( s.getS_id( ));
           System.out.print("\t" +  s.getS_name( ));
          System.out.print("\t" + s.getS_age( ));
          System.out.println();
        }
          em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

JPQL Update

UpdateData.java

```
import javax.persistence.*;
public class UpdataData {

    public static void main( String args[]) {

        EntityManagerFactory emf =
         Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

         Query query =
          em.createQuery( "update StudentEntity SET s_age=25 where s_id>103");
          query.executeUpdate();

        em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

JPQL Delete

DeleteData.java

```
import javax.persistence.*;
public class DeleteData {
    public static void main( String args[]) {
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

        Query query = em.createQuery( "delete from StudentEntity where s_id=102");
        query.executeUpdate();
        em.getTransaction().commit();
        em.close();
        emf.close();
    }
 }
```

JPQL Advanced Operations

JPQL Filter

Filter.java

```
import javax.persistence.*;
import java.util.*;
public class Filter {
    public static void main( String args[]) {

        EntityManagerFactory emf =
          Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

       Query q1 =
em.createQuery("Select s from StudentEntity s where s.s_age between 22 and 28");

       List<StudentEntity> l1 =  (List<StudentEntity>)q1.getResultList();
        System.out.println("Between Clause");
        System.out.print("s_id");
        System.out.print("\t s_name");
        System.out.println("\t s_age");

        for(StudentEntity s:l1)  {
        System.out.print(s.getS_id());
        System.out.print("\t"+s.getS_name());
        System.out.println("\t"+s.getS_age());
        }

        Query q2 =
em.createQuery("Select s from StudentEntity s where s.s_age IN(20,22,23)");

          List<StudentEntity> l2 =  (List<StudentEntity>)q2.getResultList();
           System.out.println("IN Clause");
           System.out.print("s_id");
           System.out.print("\t s_name");
            System.out.println("\t s_age");
```

```
 for(StudentEntity s:l2){
            System.out.print(s.getS_id());
            System.out.print("\t"+s.getS_name());
            System.out.println("\t"+s.getS_age());
            }

            Query q3 =
em.createQuery("Select s from StudentEntity s where s.s_name like '%a%'");

             List<StudentEntity> l3 =  (List<StudentEntity>)q3.getResultList();

              System.out.println("Like Clause");
              System.out.print("s_id");
              System.out.print("\t s_name");
              System.out.println("\t s_age");

               for(StudentEntity s:l3) {
               System.out.print(s.getS_id());
               System.out.print("\t"+s.getS_name());
               System.out.println("\t"+s.getS_age());
               }

        em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

JPQL Aggregate

Aggregate.java

```
import javax.persistence.*;
import java.util.*;
public class Aggregate {

    public static void main( String args[]) {

        EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );


        Query q1 = em.createQuery("Select count(s) from StudentEntity s");
        System.out.println("Number of Student : "+q1.getSingleResult());

        Query q2 = em.createQuery("Select MAX(s.s_age) from StudentEntity s");
        System.out.println("Maximum age : "+q2.getSingleResult());

        Query q3 = em.createQuery("Select MIN(s.s_age) from StudentEntity s");
        System.out.println("Minimum age : "+q3.getSingleResult());
em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

JPQL Sorting

Sorting.java

```
import javax.persistence.*;
import java.util.*;
public class Sorting {

    public static void main( String args[]) {

        EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin( );

        Query q1 =
em.createQuery("Select s from StudentEntity s order by s.s_age desc");

          List<StudentEntity> l1 =  (List<StudentEntity>)q1.getResultList();

           System.out.print("s_id");
           System.out.print("\t s_name");
           System.out.println("\t s_age");

            for(StudentEntity s:l1){
            System.out.print(s.getS_id());
            System.out.print("\t"+s.getS_name());
            System.out.println("\t"+s.getS_age());
            }

em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

10. JPA - Criteria API

The Criteria API is one of the most common ways of constructing queries for entities and their persistent state. It is just an alternative method for defining JPA queries.

Criteria API defines a platform-independent criteria queries, written in Java programming language. It was introduced in JPA 2.0. The main purpose behind this is to provide a type-safe way to express a query.

Steps to create Criteria Query

1. EntityManager em = emf.createEntityManager();
2. CriteriaBuilder cb=em.getCriteriaBuilder();
3. CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);
4. Root<StudentEntity> stud=cq.from(StudentEntity.class);
5. CriteriaQuery<StudentEntity> select = cq.select(stud);
6. Query q = em.createQuery(select);
7. List<StudentEntity> list = q.getResultList();

Methods of Criteria API Query Clauses

Clause	Criteria API Interface	Methods
SELECT	CriteriaQuery	select()
FROM	AbstractQuery	from()
WHERE	AbstractQuery	where()
ORDER BY	CriteriaQuery	orderBy()
GROUP BY	AbstractQuery	groupBy()
HAVING	AbstractQuery	having()

Criteria SELECT Clause

SingleFetch.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;
public class SingleFetch {
   public static void main( String args[]) {

        EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
         EntityManager em = emf.createEntityManager();
         em.getTransaction().begin( );
         CriteriaBuilder cb=em.getCriteriaBuilder();
         CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

        Root<StudentEntity> stud=cq.from(StudentEntity.class);

        cq.select(stud.get("s_name"));

         CriteriaQuery<StudentEntity> select = cq.select(stud);
         TypedQuery<StudentEntity> q = em.createQuery(select);
         List<StudentEntity> list = q.getResultList();

         System.out.println("s_id");

         for(StudentEntity s:list) {
         System.out.println(s.getS_id());

       }
          em.getTransaction().commit();
         em.close();
         emf.close();
    }
}
```

MultiFetch.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;
public class MultiFetch {

   public static void main( String args[]) {

        EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
         EntityManager em = emf.createEntityManager();
         em.getTransaction().begin( );
         CriteriaBuilder cb=em.getCriteriaBuilder();
         CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

        Root<StudentEntity> stud=cq.from(StudentEntity.class);

              cq.multiselect(stud.get("s_id"),stud.get("s_name"),stud.get("s_age") );
         CriteriaQuery<StudentEntity> select = cq.select(stud);
         TypedQuery<StudentEntity> q = em.createQuery(select);
         List<StudentEntity> list = q.getResultList();

         System.out.print("s_id");
           System.out.print("\t s_name");
           System.out.println("\t s_age");

         for(StudentEntity s:list)
         {
         System.out.print(s.getS_id());
         System.out.print("\t"+s.getS_name());
         System.out.println("\t"+s.getS_age());
         }

em.getTransaction().commit();
         em.close();
         emf.close();
    }
}
```

ORDER BY Clause

Asc.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;
public class Asc {

public static void main( String args[]) {

EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
EntityManager em = emf.createEntityManager();
em.getTransaction().begin( );
CriteriaBuilder cb=em.getCriteriaBuilder();
CriteriaQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

Root<StudentEntity> stud=cq.from(StudentEntity.class);

cq.orderBy(cb.asc(stud.get("s_age")));
CriteriaQuery<StudentEntity> select = cq.select(stud);
TypedQuery<StudentEntity> q = em.createQuery(select);
List<StudentEntity> list = q.getResultList();

System.out.print("s_id");
System.out.print("\t s_name");
System.out.println("\t s_age");

for(StudentEntity s:list){
System.out.print(s.getS_id());
System.out.print("\t"+s.getS_name());
System.out.println("\t"+s.getS_age());
}

em.getTransaction().commit();
em.close();
emf.close();
}
}
```

Criteria WHERE Clause

Comparison.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;
public class Comparison {

public static void main( String args[]) {

EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
EntityManager em = emf.createEntityManager();
em.getTransaction().begin( );

CriteriaBuilder cb=em.getCriteriaBuilder();

AbstractQuery<StudentEntity> cq1=cb.createQuery(StudentEntity.class);

Root<StudentEntity> stud1=cq1.from(StudentEntity.class);

cq1.where(cb.greaterThan(stud1.get("s_age"), 22));

CriteriaQuery<StudentEntity> select1 =
((CriteriaQuery<StudentEntity>) cq1).select(stud1);
TypedQuery<StudentEntity> tq1 = em.createQuery(select1);
List<StudentEntity> list1 = tq1.getResultList();

System.out.println("Students having age greater than 22");

System.out.print("s_id");
System.out.print("\t s_name");
System.out.println("\t s_age");

for(StudentEntity s:list1){
System.out.print(s.getS_id());
System.out.print("\t"+s.getS_name());
System.out.println("\t"+s.getS_age());
}

em.getTransaction().commit();
em.close();
emf.close();
}
}
```

JPQL Between

Between.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;
public class Between {

public static void main( String args[]) {

EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
EntityManager em = emf.createEntityManager();
em.getTransaction().begin( );

CriteriaBuilder cb=em.getCriteriaBuilder();

AbstractQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

Root<StudentEntity> stud=cq.from(StudentEntity.class);

cq.where(cb.between(stud.get("s_age"), 22, 26)) ;
CriteriaQuery<StudentEntity> select =
((CriteriaQuery<StudentEntity>) cq).select(stud);
TypedQuery<StudentEntity> tq = em.createQuery(select);
List<StudentEntity> list = tq.getResultList();

System.out.println("Students having age between 22 and 26");

System.out.print("s_id");
System.out.print("\t s_name");
System.out.println("\t s_age");

for(StudentEntity s:list){
System.out.print(s.getS_id());
System.out.print("\t"+s.getS_name());
System.out.println("\t"+s.getS_age());
}

em.getTransaction().commit();
em.close();
emf.close();
}
}
```

JPQL Like

Like.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;
public class Like {

public static void main( String args[]) {

EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
EntityManager em = emf.createEntityManager();
em.getTransaction().begin( );

CriteriaBuilder cb=em.getCriteriaBuilder();

AbstractQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);


Root<StudentEntity> stud=cq.from(StudentEntity.class);


cq.where(cb.like(stud.get("s_name"), "R%"));
CriteriaQuery<StudentEntity> select =
((CriteriaQuery<StudentEntity>) cq).select(stud);
TypedQuery<StudentEntity> tq = em.createQuery(select);
List<StudentEntity> list = tq.getResultList();

System.out.println("Students name starting with R");

System.out.print("s_id");
System.out.print("\t s_name");
System.out.println("\t s_age");

for(StudentEntity s:list){
System.out.print(s.getS_id());
System.out.print("\t"+s.getS_name());
System.out.println("\t"+s.getS_age());
}
em.getTransaction().commit();
em.close();
emf.close();
}
}
```

JPQL In

In.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;

public class In.java {

public static void main( String args[]) {


EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
EntityManager em = emf.createEntityManager();
em.getTransaction().begin( );

CriteriaBuilder cb=em.getCriteriaBuilder();

AbstractQuery<StudentEntity> cq=cb.createQuery(StudentEntity.class);

Root<StudentEntity> stud=cq.from(StudentEntity.class);

cq.where(cb.in(stud.get("s_age")).value(22).value(24));
CriteriaQuery<StudentEntity> select =
((CriteriaQuery<StudentEntity>) cq).select(stud);
TypedQuery<StudentEntity> tq = em.createQuery(select);
List<StudentEntity> list = tq.getResultList();

System.out.println("Students having age 22 and 24");

System.out.print("s_id");
System.out.print("\t s_name");
System.out.println("\t s_age");

for(StudentEntity s:list){
System.out.print(s.getS_id());
System.out.print("\t"+s.getS_name());
System.out.println("\t"+s.getS_age());
}

em.getTransaction().commit();
em.close();
emf.close();
}
}
```

GROUP BY Clause

StudentGroup.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;

public class StudentGroup {

public static void main( String args[]) {


EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
EntityManager em = emf.createEntityManager();
em.getTransaction().begin( );


CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);
Root<StudentEntity> stud = cq.from(StudentEntity.class);
cq.multiselect(stud.get("s_age"),cb.count(stud)).groupBy(stud.get("s_age"));


System.out.print("s_age");
System.out.println("\t Count");
List<Object[]> list = em.createQuery(cq).getResultList();
for(Object[] object : list){

     System.out.println(object[0] + "     " + object[1]);

}

em.getTransaction().commit();
em.close();
emf.close();
}
}
```

Having clause

StudentHaving.java

```
import javax.persistence.*;
import javax.persistence.criteria.*;
import java.util.*;

public class StudentHaving {

public static void main( String args[]) {


EntityManagerFactory emf =
Persistence.createEntityManagerFactory( "Student_details" );
EntityManager em = emf.createEntityManager();
em.getTransaction().begin( );


CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);
Root<StudentEntity> stud = cq.from(StudentEntity.class);

cq.multiselect(stud.get("s_age"),cb.count(stud)).groupBy(stud.get("s_age")).
having(cb.ge(stud.get("s_age"), 24));

System.out.print("s_age");
System.out.println("\t Count");
List<Object[]> list = em.createQuery(cq).getResultList();
for(Object[] object : list){
System.out.println(object[0] + " " + object[1]);

}

em.getTransaction().commit();
em.close();
emf.close();
}
}
```

11. JPA - Inheritance

Inheritence is a key feature of object-oriented programming language in which a child class can acquire the properties of its parent class. This feature enhances reusability of the code.

The relational database doesn't support the mechanism of inheritance. So, Java Persistence API (JPA) is used to map the key features of inheritance in relational database model.

JPA Inheritence Annotations

Following are the most frequently used JPA inheritance annotations: -

@Inheritence - This annotation is applied on the root entity class to define the inheritance strategy. If no strategy type is defined with this annotation then it follows single table strategy.

@MappedSuperclass - This annotation is applied to the classes that are inherited by their subclasses. The mapped superclass doesn't contain any separate table.

@DiscriminatorColumn - The discriminator attribute differentiates one entity from another. Thus, this annotation is used to provide the name of discriminator column. It is required to specify this annotation on the root entity class only.

@DiscriminatorValue - This annotation is used to specify the type of value that represents the particular entity. It is required to specify this annotation on the sub-entity classes.

JPA Inheritence Strategies

JPA provides three strategies through which we can easily persist inheritance in database.

- Single table strategy
- Joined strategy
- Table-per-class strategy

Single Table Strategy

The single table strategy is one of the most simplest and efficient way to define the implementation of inheritance. In this approach, instances of the multiple entity classes are stored as attributes in a single table only.

The following syntax represents the single table strategy: -

@Inheritance**(strategy=InheritanceType.SINGLE_TABLE)**

Employee.java

```
import java.io.Serializable;
import javax.persistence.*;

@Entity
@Table(name="employee_details")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Employee implements Serializable {

@Id
private int e_id;
private String e_name;
// getter, setter
}
```

ActiveEmployee.java

```
import javax.persistence.*;

@Entity
public class ActiveEmployee extends Employee {

private int e_salary;
private int e_experience;

  public ActiveEmployee(int e_id, String e_name, int e_salary, int e_experience) {
        super(e_id, e_name);
        this.e_salary = e_salary;
        this.e_experience = e_experience;
    }
//getter, setter
}
```

RetiredEmployee.java

```
import javax.persistence.*;
@Entity
public class RetiredEmployee extends Employee {

private int e_pension;

public RetiredEmployee(int e_id, String e_name, int e_pension) {
super(e_id, e_name);
this.e_pension = e_pension;
}

//getter, setter
}
```

EmployeePersistence.java

```
import javax.persistence.*;

public class EmployeePersistence {

public static void main(String[] args) {

EntityManagerFactory emf=
Persistence.createEntityManagerFactory("Employee_details");
EntityManager em=emf.createEntityManager();

em.getTransaction().begin();


ActiveEmployee ae1=new ActiveEmployee(101,"Keith",10000,5);
ActiveEmployee ae2=new ActiveEmployee(102,"Sparky",12000,7);

RetiredEmployee re1=new RetiredEmployee(103,"Allen",5000);
RetiredEmployee re2=new RetiredEmployee(104,"Fred",4000);

em.persist(ae1);
em.persist(ae2);

em.persist(re1);
em.persist(re2);

em.getTransaction().commit();

em.close();
emf.close();

  }
}
```

Joined strategy

In joined strategy, a separate table is generated for every entity class. The attribute of each table is joined with the primary key. It removes the possibility of duplicacy.

The following syntax represents the joined strategy: -

@Inheritance**(strategy=InheritanceType.JOINED)**

Employee.java

```
import java.io.Serializable;

import javax.persistence.*;

@Entity
@Table(name="employee_details")
@Inheritance(strategy=InheritanceType.JOINED)
public class Employee implements Serializable {

@Id
private int e_id;
private String e_name;
// getter, setter


}
```

ActiveEmployee.java

```
import javax.persistence.*;

@Entity
public class ActiveEmployee extends Employee {

private int e_salary;
private int e_experience;

public ActiveEmployee(int e_id, String e_name, int e_salary, int e_experience) {
super(e_id, e_name);
this.e_salary = e_salary;
this.e_experience = e_experience;
} // getter, setter
}
```

RetiredEmployee.java

```
import javax.persistence.*;

@Entity
public class RetiredEmployee extends Employee {

private int e_pension;

public RetiredEmployee(int e_id, String e_name, int e_pension) {
super(e_id, e_name);
this.e_pension = e_pension;
}
// getter, setter
}
```

EmployeePersistence.java

```
import javax.persistence.*;

public class EmployeePersistence {

public static void main(String[] args) {

EntityManagerFactory emf=
Persistence.createEntityManagerFactory("Employee_details");
EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

ActiveEmployee ae1=new ActiveEmployee(101,"Keith",10000,5);
ActiveEmployee ae2=new ActiveEmployee(102,"Sparky",12000,7);

RetiredEmployee re1=new RetiredEmployee(103,"Fred",5000);
RetiredEmployee re2=new RetiredEmployee(104,"Allen",4000);

em.persist(ae1);
em.persist(ae2);

em.persist(re1);
em.persist(re2);

em.getTransaction().commit();
em.close();
emf.close();
}
}
```

Table-per-class Strategy

In table-per-class strategy, for each sub entity class a separate table is generated. Unlike joined strategy, no separate table is generated for parent entity class in table-per-class strategy.

The following syntax represents the table-per-class strategy: -

@Inheritance**(strategy=InheritanceType.TABLE_PER_CLASS)**

Employee.java

```
import java.io.Serializable;

import javax.persistence.*;

@Entity
@Table(name="employee_details")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Employee implements Serializable {

@Id
private int e_id;
private String e_name;
// getter, setter
}
```

ActiveEmployee.java

```
import javax.persistence.*;

@Entity
public class ActiveEmployee extends Employee {

private int e_salary;
private int e_experience;
public ActiveEmployee(int e_id, String e_name, int e_salary, int e_experience) {
super(e_id, e_name);
this.e_salary = e_salary;
this.e_experience = e_experience;
}
//getter, setter
}
```

RetiredEmployee.java

```
import javax.persistence.*;
@Entity
public class RetiredEmployee extends Employee {

private int e_pension;

public RetiredEmployee(int e_id, String e_name, int e_pension) {
super(e_id, e_name);
this.e_pension = e_pension;
}
//getter, setter
}
```

EmployeePersistence.java

```
import javax.persistence.*;

public class EmployeePersistence {

public static void main(String[] args) {

EntityManagerFactory emf=
Persistence.createEntityManagerFactory("Employee_details");
EntityManager em=emf.createEntityManager();

em.getTransaction().begin();

ActiveEmployee ae1=new ActiveEmployee(101,"Keith",10000,5);
ActiveEmployee ae2=new ActiveEmployee(102,"Sparky",12000,7);

RetiredEmployee re1=new RetiredEmployee(103,"Fred",5000);
RetiredEmployee re2=new RetiredEmployee(104,"Allen",4000);

em.persist(ae1);
em.persist(ae2);

em.persist(re1);
em.persist(re2);

em.getTransaction().commit();
em.close();
emf.close();
}
}
```

12. JPA and Spring MVC Integration

To integrate JPA with Spring MVC, the following steps are required to do :

- Step -1 : Project – Dynamic Web Project
- Step -2 : Add jar files of (Spring MVC, JPA EclipseLink)

The following steps are the same in the Spring MVC project.

In this example, we show JPA with Dynamic Web Project (BookManagementSystem).

Let's start Spring MVC with JPA integration.

1. web.xml

```
<display-name>JPASpringBook_20200916</display-name>

    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

2. spring-servlet.xml

```
<bean id="ld"
		class=
"org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />

<context:load-time-weaver />


<bean id="jpaVendor"
		class="org.springframework.orm.jpa.vendor.EclipseLinkJpaVendorAdapter">
		<property name="showSql" value="true" />
		<property name="generateDdl" value="true" />
		<property name="database" value="MYSQL" />
</bean>


<bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="loadTimeWeaver" ref="ld" />
<property name="dataSource" ref="ds" />
<property name="jpaVendorAdapter" ref="jpaVendor"></property>
<property name="persistenceUnitName"
		value="JPASpringBook_20200916" />
<property name="packagesToScan">

<list>
    <value>com.book.model</value>
</list>

</property>

</bean>


<bean id="myTxManager"
		class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="emf" />
    <property name="dataSource" ref="ds" />
</bean>

<tx:annotation-driven transaction-manager="myTxManager" />
```

3. Book.java

```
@Entity
public class Book {

	@Id
	private String bookCode;
	private String bookTitle;
	private String bookAuthor;
	private String bookPrice;
//getter, setter
}
```

4. BookDao Interface

```
public interface BookDao {
	public int insertData(Book book);
	public int updateData(Book book);
	public int deleteData(String bookCode);
	public Book selectOne(String bookCode);
	public List<Book> selectAll();
}
```

5. BookDaoImpl.java

```
@Component
public class BookDaoImpl implements BookDao {

	@PersistenceContext
	private EntityManager em;

	@Transactional
	@Override
	public int insertData(Book book) {
		try {
			em.persist(book);
		} catch (Exception e) {
			System.out.println(e.getMessage());
			return 0;
		}
		return 1;
	}
```

```
    @Transactional
    @Override
    public int updateData(Book book) {

    Query query = em.createQuery(
"UPDATE Book b SET b.bookTitle = :bookTitle, b.bookAuthor =:bookAuthor ,
b.bookPrice =:bookPrice WHERE b.bookCode =:bookCode");

            query.setParameter("bookTitle", book.getBookTitle());
            query.setParameter("bookAuthor", book.getBookAuthor());
            query.setParameter("bookPrice", book.getBookPrice());
            query.setParameter("bookCode", book.getBookCode());
            int i = query.executeUpdate();
            return i;

    }


    @Transactional
    @Override
    public int deleteData(String bookCode) {
Query query = em.createQuery("DELETE FROM Book b WHERE b.bookCode
=:bookCode");
            query.setParameter("bookCode", bookCode);
            int i = query.executeUpdate();
            return i;
    }

    @Override
    public Book selectOne(String bookCode) {
Book book = em.createQuery("SELECT b FROM Book b WHERE b.bookCode
=:bookCode", Book.class).setParameter("bookCode", bookCode).getSingleResult();

            return book;
    }

    @Override
    public List<Book> selectAll() {

List<Book> lstBook = em.createQuery("SELECT b FROM Book b",
Book.class).getResultList();
            return lstBook;
    }

}
```

6. BookController.java

```
@Controller
public class BookController {

    @Autowired
    BookDao bookDao;

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String displayView(ModelMap model) {
            List<Book> list = bookDao.selectAll();
            model.addAttribute("list", list);
            return "displaybook";
    }

    @RequestMapping(value = "/setupaddbook", method = RequestMethod.GET)
    public ModelAndView setupaddBook() {
            return new ModelAndView("addbook", "bean", new Book());
    }

    @RequestMapping(value = "/addbook", method = RequestMethod.POST)
public String addBook(@ModelAttribute("bean") Book bean,
ModelMap model) {

            int res = bookDao.insertData(bean);
            if (res == 0) {
                    model.addAttribute("error", "Insert Failed");
                    return "addbook";
            }
            return "redirect:/";
    }

    @RequestMapping(value = "setupupdateBook/{bookCode}", method =
RequestMethod.GET)
    public ModelAndView setupUpdate(@PathVariable String bookCode) {

            return new ModelAndView("updatebook", "bean",
bookDao.selectOne(bookCode));
    }
```

```
    @RequestMapping(value = "/updatebook", method = RequestMethod.POST)
    public String updateBook(@ModelAttribute("bean") Book bean, ModelMap
model) {

            int res = bookDao.updateData(bean);
            if (res == 0) {
                    model.addAttribute("error", "Update Failed");
                    return "updatebook";
            }
            return "redirect:/";
    }

    @RequestMapping(value = "deleteBook/{bookCode}", method =
RequestMethod.GET)
    public String deleteBook(@PathVariable String bookCode, ModelMap model) {

            int res = bookDao.deleteData(bookCode);
            if (res == 0) {
                    model.addAttribute("error", "Delete Failed");
                    return "redirect:/";
            }
            return "redirect:/";
    }

}
```

header.jsp

```
<body>
<h3> Book Management</h3>
<a href="/JPASpringBook_20200916/setupaddbook">
Add New Book
</a>   
<a href="/JPASpringBook_20200916/">
Show All Book
</a>
</body>
```

displaybook.jsp

```
<style>
table, th, td {
	border: 1px solid blue;
	border-collapse: collapse;
}
</style>
</head>
<body>
	<center>
		<jsp:include page="header.jsp"></jsp:include>
		<br />
		<div style="color: blue;">${msg}</div>
		<div style="color: red;">${error}</div>
		<br />
		<table>
			<tr>
				<th>Book Code</th>
				<th>Book Title</th>
				<th>Book Author</th>
				<th>Book Price</th>
				<th>Action</th>
			</tr>
			<c:forEach items="${list}" var="data">
				<tr>
					<td>${data.bookCode}</td>
					<td>${data.bookTitle}</td>
					<td>${data.bookAuthor}</td>
					<td>${data.bookPrice}</td>
					<td><a
href="setupupdateBook/${data.bookCode}">Update</a> <a

	href="deleteBook/${data.bookCode}">Delete</a></td>
				</tr>
			</c:forEach>
		</table>
	</center>
</body>
```

addbook.jsp

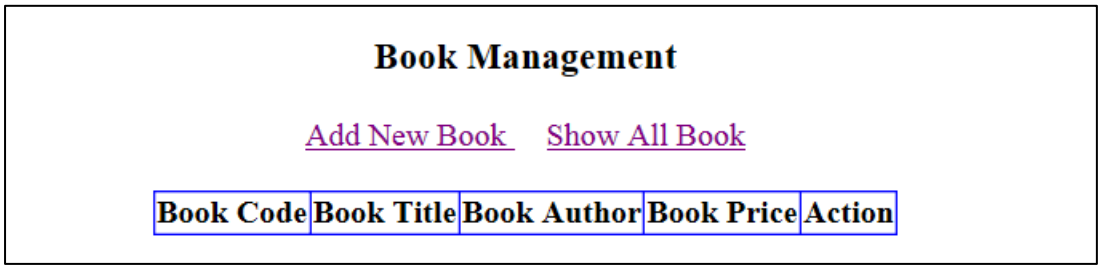
```
<body>
	<center>
		<jsp:include page="header.jsp"></jsp:include>
		<br />

		<form:form action="/JPASpringBook_20200916/addbook"
method="post"
			modelAttribute="bean">
			<table>
				<tr>
					<td>Book Code</td>
			<td><form:input type="text" path="bookCode" /></td>

				</tr>
				<tr>
					<td>Book Title</td>
			<td><form:input type="text" path="bookTitle" /></td>

				</tr>
				<tr>
					<td>Book Author</td>
			<td><form:input type="text" path="bookAuthor" /></td>

				</tr>
				<tr>
					<td>Book Price</td>
			<td><form:input type="text" path="bookPrice" /></td>

				</tr>
				<tr>
					<td></td>
			<td><input type="submit" value="Save" /></td>
				</tr>
			</table>
		</form:form>
	</center>
</body>
```

updatebook.jsp

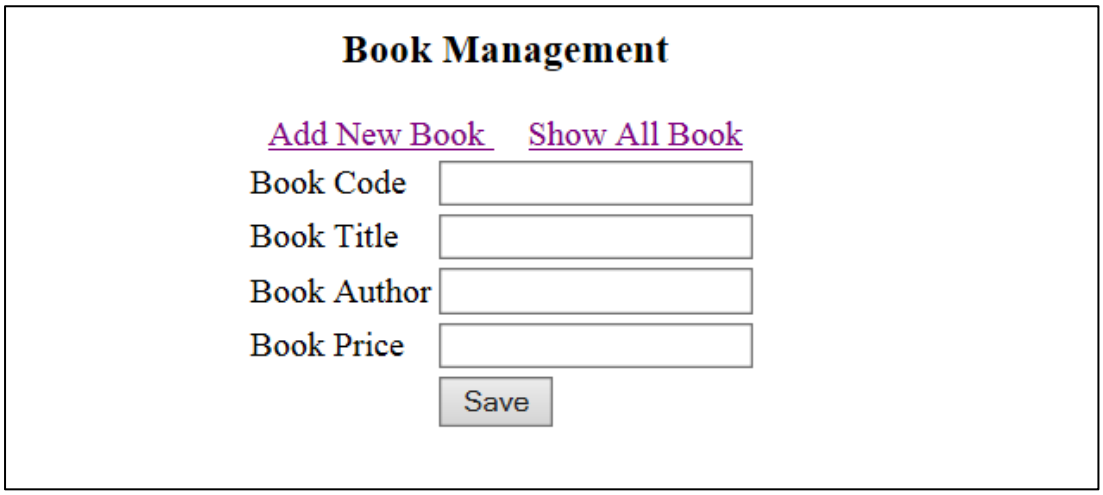
```
<body>
	<center>
		<jsp:include page="header.jsp"></jsp:include>
		<br />

		<form:form action="/JPASpringBook_20200916/updatebook"
method="post"
			modelAttribute="bean">
			<table>
				<tr>
					<td>Book Code</td>
			<td><form:input type="text" path="bookCode" /></td>

				</tr>
				<tr>
					<td>Book Title</td>
			<td><form:input type="text" path="bookTitle" /></td>

				</tr>
				<tr>
					<td>Book Author</td>
			<td><form:input type="text" path="bookAuthor" /></td>

				</tr>
				<tr>
					<td>Book Price</td>
			<td><form:input type="text" path="bookPrice" /></td>

				</tr>
				<tr>
					<td></td>
			<td><input type="submit" value="UpdateBook" /></td>

				</tr>
			</table>
		</form:form>
	</center>
</body>
```

Output:

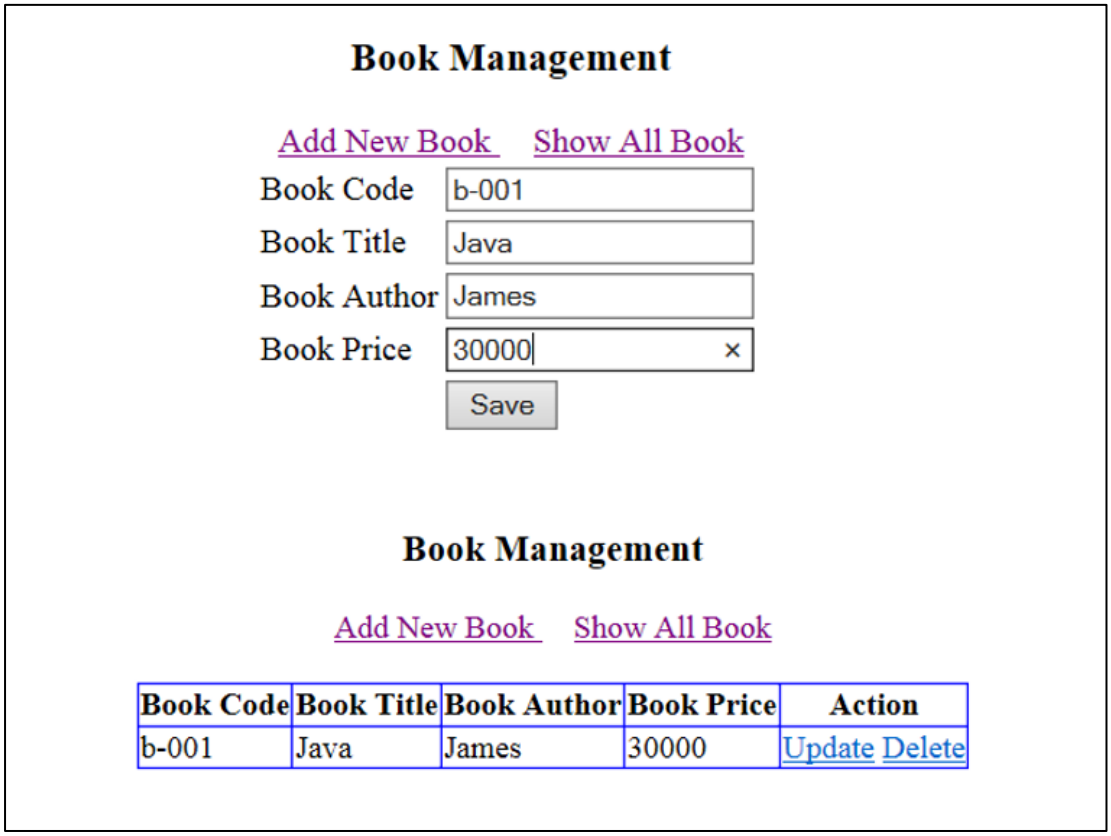
Project – R Click – Run on Server



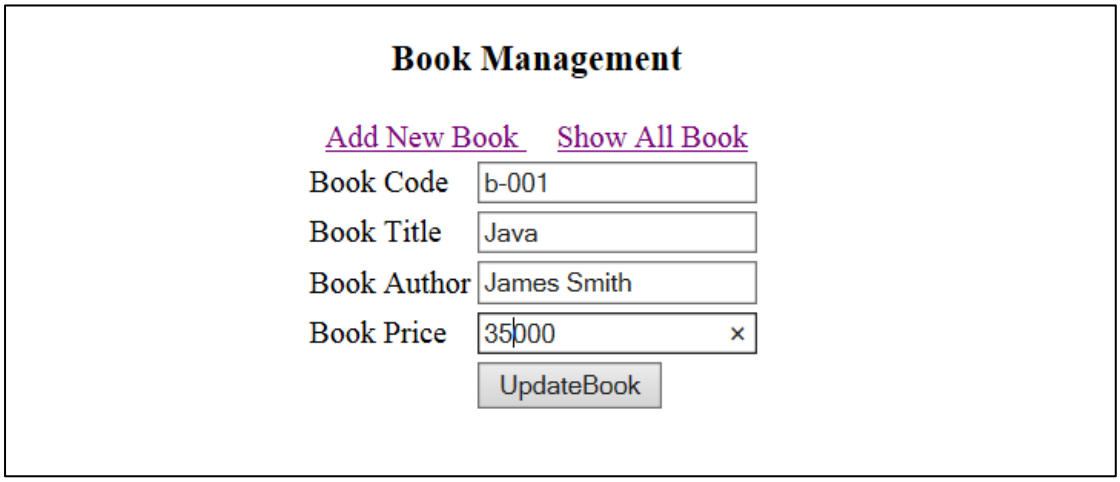
When click Add New Book:



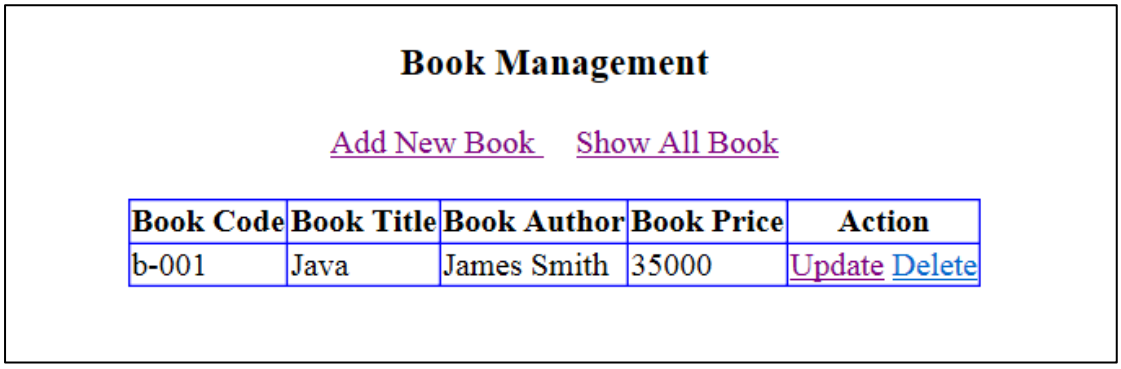
When click **Save** with data then show information:



When click **Update** then show Update form and change data:



When click **UpdateBook:**



When click **Delete:**