JPA

# Introduction

Any enterprise application performs database operations by storing and retrieving vast amounts of data. Despite all the available technologies for storage management, application developers normally struggle to perform database operations efficiently.

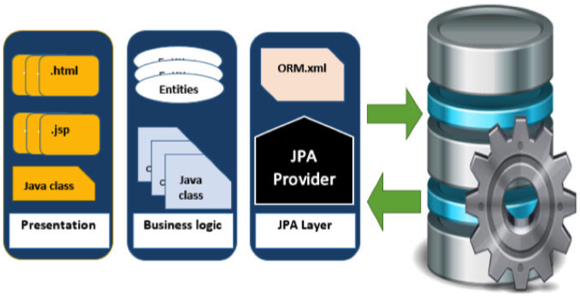
Generally, Java developers use lots of code, or use the proprietary framework to interact with the database, whereas using JPA, the burden of interacting with the database reduces significantly. It forms a bridge between object models (Java program) and relational models (database program).

## Mismatches between relational and object models

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

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

## What is JPA?

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

## Where to use JPA?

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

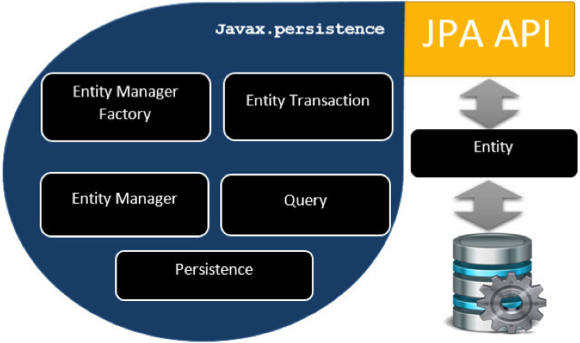
## JPA History

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.**

# Architecture

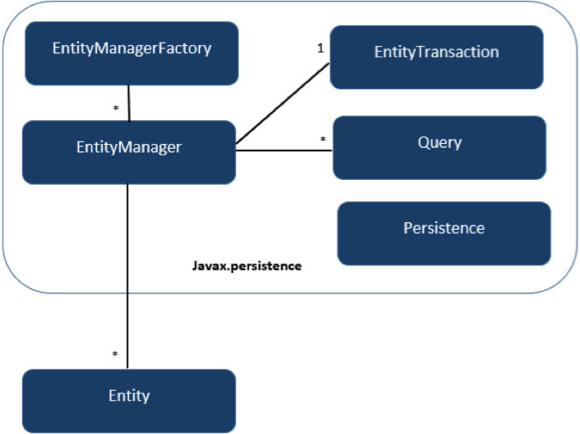
Java Persistence API is a source to store business entities as relational entities. It shows how to define a Plain Oriented Java Object (POJO) as an entity and how to manage entities with relations.

## 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.

# ORM Components

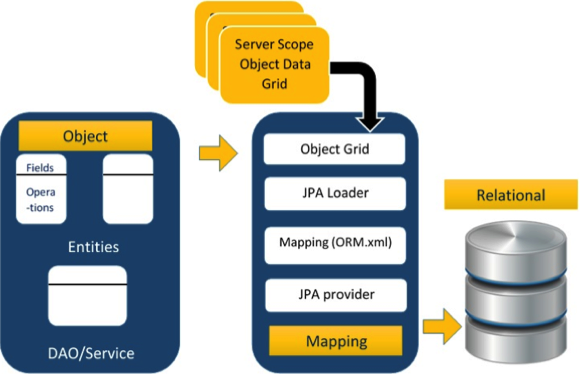
Most contemporary applications use relational database to store data. Recently, many vendors switched to object database to reduce their burden on data maintenance. It means object database or object relational technologies are taking care of storing, retrieving, updating, and maintenance. The core part 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.

The 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 ()’.

# Entity Managers

This chapter takes you through simple example with JPA. Let us consider employee management as example. It means the employee management is creating, updating, finding, and deleting an employee. As mentioned above we are using MySQL database for database operations.

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

## Creating Entities

Entities are nothing but beans or Models. It contains default constructor, setter and getter methods of those attributes.

@Entity

@Table

public class Employee { }

## 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.

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.

EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Eclipselink\_JPA" );

EntityManager entitymanager = emfactory.createEntityManager( );

entitymanager.getTransaction( ).begin( );

Employee employee = new Employee( );

entitymanager.persist( employee );

entitymanager.getTransaction( ).commit( );

entitymanager.close( );

emfactory.close( );

## Update Employee

To Update an employee, we need to get record form database, make changes, and finally committ it.

Employee employee = entitymanager.find( Employee.class, 1201 );

//before update

System.out.println( employee );

employee.setSalary( 46000 );

entitymanager.getTransaction( ).commit( );

//after update

System.out.println( employee );

entitymanager.close();

emfactory.close();

## Find Employee

To Find an employee we will get record from database and display it. In this operation, EntityTransaction is not involved any transaction is not applied while retrieving a record.

EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Eclipselink\_JPA" );

EntityManager entitymanager = emfactory.createEntityManager();

Employee employee = entitymanager.find( Employee.class, 1201 );

## Deleting Employee

To Delete an Employee, first we will find the record and then delete it. Here EntityTransaction plays an important role.

Employee employee = entitymanager.find( Employee.class, 1201 );

entitymanager.remove( employee );

The effected database named **employee** will have null records.

# JPQL

This chapter tells you about JPQL and how it works with persistence units. In this chapter, examples follow the same package hierarchy, which we used in the previous chapter as follows:

## Java Persistence Query language

JPQL is Java Persistence Query Language defined in JPA specification. It is used to create queries against entities to store in a relational database. JPQL is developed based on SQL syntax. But it won’t affect the database directly.

JPQL can retrieve information or data using SELECT clause, can do bulk updates using UPDATE clause and DELETE clause. EntityManager.createQuery() API will support for querying language.

## Query Structure

JPQL syntax is very similar to the syntax of SQL. Having SQL like syntax is an advantage because SQL is a simple structured query language and many developers are using it in applications. SQL works directly against relational database tables, records and fields, whereas JPQL works with Java classes and instances.

For example, a JPQL query can retrieve an entity object rather than field result set from database, as with SQL. The JPQL query structure as follows.

SELECT ... FROM ...

[WHERE ...]

[GROUP BY ... [HAVING ...]]

[ORDER BY ...]

The structure of JPQL DELETE and UPDATE queries is simpler as follows.

DELETE FROM ... [WHERE ...]

UPDATE ... SET ... [WHERE ...]

## Scalar and Aggregate Functions

Scalar functions returns resultant values based on input values. Aggregate functions returns the resultant values by calculating the input values.

Follow the same example employee management used in previous chapters. Here we will go through the service classes using scalar and aggregate functions of JPQL.

Let us assume the jpadb.employee table contains following records.

|  |  |  |  |
| --- | --- | --- | --- |
| **Eid** | **Ename** | **Salary** | **Deg** |
| 1201 | Gopal | 40000 | Technical Manager |
| 1202 | Manisha | 40000 | Proof Reader |
| 1203 | Masthanvali | 40000 | Technical Writer |
| 1204 | Satish | 30000 | Technical Writer |
| 1205 | Krishna | 30000 | Technical Writer |
| 1206 | Kiran | 35000 | Proof Reader |

Create a class named **ScalarandAggregateFunctions.java** under**com.tutorialspoint.eclipselink.service** package as follows.

//Scalar function

Query query = entitymanager.

createQuery("Select UPPER(e.ename) from Employee e");

//Aggregate function

Query query1 = entitymanager.createQuery("Select MAX(e.salary) from Employee e");

Double result = (Double) query1.getSingleResult();

## Between, And, Like Keywords

‘Between’, ‘And’, and ‘Like’ are the main keywords of JPQL. These keywords are used after Where clause in a query.

//Between

Query query = entitymanager.createQuery( "Select e " + "from Employee e " + "where e.salary " + "Between 30000 and 40000" );

//Like

Query query1 = entitymanager.createQuery("Select e " + "from Employee e " + "where e.ename LIKE 'M%'");

## Ordering

To Order the records in JPQL we use ORDER BY clause. The usage of this clause is same as the use in SQL, but it deals with entities.

//ORDER BY

Query query = entitymanager.createQuery( "Select e " + "from Employee e " + "ORDER BY e.ename ASC" );

## Named Queries

A @NamedQuery annotation is defined as a query with a predefined unchangeable query string. Instead of dynamic queries, usage of named queries may improve code organization by separating the JPQL query strings from POJO. It also passes the query parameters rather than embedding literals dynamically into the query string and results in more efficient queries.

First of all, add @NamedQuery annotation to the entity class.

@NamedQuery(query = "Select e from Employee e where e.eid = :id", name = "find employee by id")

Query query = entitymanager.createNamedQuery("find employee by id");

query.setParameter("id", 1204);

## Eager and Lazy Loading

The main concept of JPA is to make a duplicate copy of the database in cache memory. While transacting with the database, first it will effect on duplicate data and only when it is committed using entity manager, the changes are effected into the database.

There are two ways of fetching records from the database - eager fetch and lazy fetch.

### Eager fetch

Fetching the whole record while finding the record using Primary Key.

### Lazy fetch

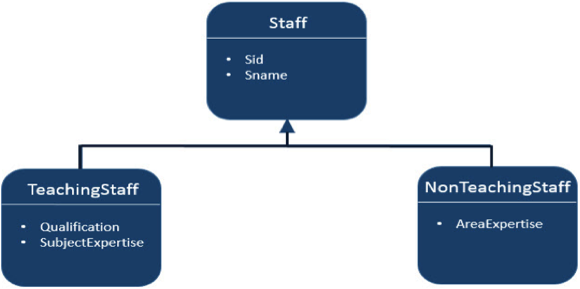
It checks for the availability of notifies it with primary key if it exists. Then later if you call any of the getter method of that entity then it fetches the whole.

But lazy fetch is possible when you try to fetch the record for the first time. That way, a copy of the whole record is already stored in cache memory. Performance wise, lazy fetch is preferable.

# Advanced Mappings

JPA is a library which is released with java specification. Therefore, it supports all object oriented concepts for entity persistence. Till now we are done with the basics of object relational mapping. This chapter takes you through the advanced mappings between objects and relational entities.

## Inheritance Strategies

Inheritance is the core concept of object oriented language, therefore we can use inheritance relationships or strategies between entities. JPA support three types of inheritance strategies such as SINGLE\_TABLE, JOINED\_TABLE, and TABLE\_PER\_CONCRETE\_CLASS.

Let us consider an example of Staff, TeachingStaff, NonTeachingStaff classes and their relationships as follows:

In the above shown diagram Staff is an entity and TeachingStaff and NonTeachingStaff are the sub entities of Staff. Here we will discuss the above example in all three strategies of inheritance.

## Single Table strategy

Single-Table strategy takes all classes fields (both super and sub classes) and map them down into a single table known as SINGLE\_TABLE strategy. Here discriminator value plays key role in differentiating the values of three entities in one table.

Let us consider the above example, TeachingStaff and NonTeachingStaff are the sub classes of class Staff. Remind the concept of inheritance (is a mechanism of inheriting the properties of super class by sub class) and therefore sid, sname are the fields which belongs to both TeachingStaff and NonTeachingStaff. Create a JPA project. All the modules of this project as follows:

### Creating Entities

Create a package under **‘src’**package. Create a new java class under given package.

@Entity

@Table

@Inheritance( strategy = InheritanceType.SINGLE\_TABLE )

@DiscriminatorColumn( name = "type" )

In the above code **@DescriminatorColumn** specifies the field name **(type)**and the values of it shows the remaining (Teaching and NonTeachingStaff) fields.

Create a subclass (class) to Staff class named **TeachingStaff.java** under the**com.tutorialspoint.eclipselink.entity** package. The TeachingStaff Entity class is shown as follows:

@Entity

@DiscriminatorValue( value="TS" )

Create a subclass (class) to Staff class named **NonTeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The NonTeachingStaff Entity class is shown as follows:

@Entity

@DiscriminatorValue( value = "NS" )

### Service class

Service classes are the implementation part of business component. Create a package under **‘src’** package. Create a class named SaveClient.java under the given package to store Staff, TeachingStaff, and NonTeachingStaff class fields. The SaveClient class is shown as follows:

//Teaching staff entity

TeachingStaff ts1=new TeachingStaff(1,"Gopal","MSc MEd","Maths");

//Non-Teaching Staff entity

NonTeachingStaff nts1=new NonTeachingStaff(2, "Satish", "Accounts");

//storing all entities

entitymanager.persist(ts1);

entitymanager.persist(nts1);

Finally you will get single table which contains all three class’s fields and differs with discriminator column named **‘Type’** (field).

## Joined table Strategy

Joined table strategy is to share the referenced column which contains unique values to join the table and make easy transactions. Let us consider the same example as above.

Create a JPA Project. All the project modules shown as follows:

### Creating Entities

@Entity

@Table

@Inheritance( strategy = InheritanceType.JOINED )

public class Staff implements Serializable { }

Create a subclass (class) to Staff class named **TeachingStaff.java** under the**com.tutorialspoint.eclipselink.entity** package. The TeachingStaff Entity class is shown as follows:

@Entity

@PrimaryKeyJoinColumn(referencedColumnName="sid")

public class TeachingStaff extends Staff { }

Create a subclass (class) to Staff class named **NonTeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The NonTeachingStaff Entity class is shown as follows:

@Entity

@PrimaryKeyJoinColumn(referencedColumnName="sid")

public class NonTeachingStaff extends Staff { }

### Service class

Service classes are the implementation part of business component. Create a package under **‘src’** package named.

Create a class named SaveClient.java under the given package to store Staff, TeachingStaff, and NonTeachingStaff class fields. Then SaveClient class as follows:

//Teaching staff entity

TeachingStaff ts1 = new TeachingStaff(1,"Gopal","MSc MEd","Maths");

//Non-Teaching Staff entity

NonTeachingStaff nts1 = new NonTeachingStaff(2, "Satish", "Accounts");

//storing all entities

entitymanager.persist(ts1);

entitymanager.persist(nts1);

Finally the three tables are created using their fields respectively and SID field is shared by all three tables. In staff table SID is primary key, in remaining (TeachingStaff and NonTeachingStaff) tables SID is foreign key.

## Table per class strategy

Table per class strategy is to create a table for each sub entity. The staff table will be created but it will contain null records. The field values of Staff table must be shared by TeachingStaff and NonTeachingStaff tables.

Let us consider the same example as above. All modules of this project are shown as follows:

### Creating Entities

Create a package under **‘src’**package. Create a new java class named **Staff.java** under given package. The Staff entity class is shown as follows:

@Entity

@Table

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

public class Staff implements Serializable {}

Create a subclass (class) to Staff class named **TeachingStaff.java** under the**com.tutorialspoint.eclipselink.entity** package. The TeachingStaff Entity class is shown as follows:

@Entity

public class TeachingStaff extends Staff {}

Create a subclass (class) to Staff class named **NonTeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The NonTeachingStaff Entity class is shown as follows:

@Entity

public class NonTeachingStaff extends Staff {}

### Service class

Service classes are the implementation part of business component. Create a package under **‘src’** package named.

Create a class named **SaveClient.java** under the given package to store Staff, TeachingStaff, and NonTeachingStaff class fields. The SaveClient class is shown as follows:

//Teaching staff entity

TeachingStaff ts1 = new TeachingStaff(1,"Gopal","MSc MEd","Maths");

//Non-Teaching Staff entity

NonTeachingStaff nts1 = new NonTeachingStaff(2, "Satish", "Accounts");

//storing all entities

entitymanager.persist(ts1);

entitymanager.persist(nts1);

Here the three tables are created and the **Staff** table contains null records.

The table TeachingStaff contains fields of both Staff and TeachingStaff Entities.

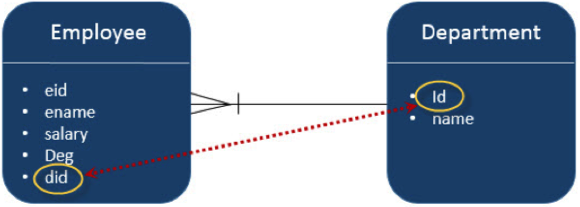
The table NonTeachingStaff contains fields of both Staff and NonTeachingStaff Entities.

# Entity Relationships

This chapter takes you through the relationships between Entities. Generally the relations are more effective between tables in the database. Here the entity classes are treated as relational tables (concept of JPA), therefore the relationships between Entity classes are as follows:

* @ManyToOne Relation
* @OneToMany Relation
* @OneToOne Relation
* @ManyToMany Relation

## @ManyToOne Relation

Many-To-One relation between entities: Where one entity (column or set of columns) is/are referenced with another entity (column or set of columns) which contain unique values. In relational databases these relations are applicable by using foreign key/primary key between tables.

Let us consider an example of relation between Employee and Department entities. In unidirectional manner, i.e.from Employee to Department, Many-To-One relation is applicable. That means each record of employee contains one department id, which should be a primary key in Department table. Here in the Employee table, Department id is foreign Key.

The diagram explains Many-To-One relation:

Create a JPA project in eclipse IDE named **JPA\_Eclipselink\_MTO**. All the modules of this project are shown as follows:

### Creating Entities

Follow the above given diagram for creating entities. Create a package named under **‘src’** package. Create a class named **Department.java** under given package. The class Department entity is shown as follows:

@Entity

public class Department { }

Create the second entity in this relation - Employee entity class named **Employee.java** package. The Employee entity class is shown as follows:

@Entity

public class Employee{

@ManyToOne

private Department department;

public Employee(int eid, String ename, double salary, String deg) {

super( );

this.eid = eid;

this.ename = ename;

this.salary = salary;

this.deg = deg;

}

### Service Classes

This module contains the service classes, which implements the relational part using the attribute initialization. Create a package under **‘src’** package named. The DAO class named **ManyToOne.java** is created under given package. The DAO class is shown as follows:

//Create Department Entity

Department department = new Department();

department.setName("Development");

//Store Department

entitymanager.persist(department);

//Create Employee1 Entity

Employee employee1 = new Employee();

employee1.setEname("Satish");

employee1.setDepartment(department);

//Create Employee2 Entity

Employee employee2 = new Employee();

employee2.setEname("Krishna");

employee2.setDepartment(department);

//Store Employees

entitymanager.persist(employee1);

entitymanager.persist(employee2);

## @OneToMany Relation

In this relationship each row of one entity is referenced to many child records in other entity. The important thing is that child records cannot have multiple parents. In a one-to-many relationship between Table A and Table B, each row in Table A is linked to 0, 1 or many rows in Table B.

Let us consider the above example. If **Employee** and **Department** is in a reverse unidirectional manner, relation is Many-To-One relation. Create a JPA project in eclipse IDE named **JPA\_Eclipselink\_OTM**. All the modules of this project are shown as follows:

### Creating Entities

Follow the above given diagram for creating entities. Create a package named under **‘src’** package. Create a class named **Department.java** under given package. The class Department entity is shown as follows:

@Entity

public class Department {

@OneToMany( targetEntity=Employee.class )

private List employeelist;}

Create the second entity in this relation -Employee entity class, named**Employee.java** under package. The Employee entity class is shown as follows:

@Entity

public class Employee {

public Employee(int eid, String ename, double salary, String deg) {

super( );

this.eid = eid;

this.ename = ename;

this.salary = salary;

this.deg = deg;}}

### Service Classes

This module contains the service classes, which implements the relational part using the attribute initialization. Create a package under **‘src’** package named. The DAO class named**OneToMany.java** is created under given package. The DAO class is shown as follows:

//Create Employee1 Entity

Employee employee1 = new Employee();

employee1.setEname("Satish");

//Create Employee2 Entity

Employee employee2 = new Employee();

employee2.setEname("Krishna");

//Store Employee

entitymanager.persist(employee1);

entitymanager.persist(employee2);

//Create Employeelist

List<Employee> emplist = new ArrayList();

emplist.add(employee1);

emplist.add(employee2);

//Create Department Entity

Department department = new Department();

department.setName("Development");

department.setEmployeelist(emplist);

//Store Department

entitymanager.persist(department);

## @OneToOne Relation

In One-To-One relationship, one item can belong to only one other item. It means each row of one entity is referred to one and only one row of another entity.

Let us consider the above example. **Employee** and **Department** in a reverse unidirectional manner, the relation is One-To-One relation. It means each employee belongs to only one department. Create a JPA project in eclipse IDE named **JPA\_Eclipselink\_OTO**. All the modules of this project are shown as follows:

### Creating Entities

Follow the above given diagram for creating entities. Create a package named under **‘src’** package. Create a class named **Department.java** under given package. The class Department entity is shown as follows:

@Entity

public class Department { }

Create the second entity in this relation -Employee entity class, named**Employee.java** under package. The Employee entity class is shown as follows:

@Entity

public class Employee {

@OneToOne

private Department department;}

### Service Classes

This module contains the service classes, which implements the relational part using the attribute initialization. Create a package under **‘src’** package named. The DAO class named**OneToOne.java** is created under the given package. The DAO class is shown as follows:

//Create Department Entity

Department department = new Department();

department.setName("Development");

//Store Department

entitymanager.persist(department);

//Create Employee Entity

Employee employee = new Employee();

employee.setEname("Satish");

employee.setDepartment(department);

//Store Employee

entitymanager.persist(employee);

## @ManyToOne Relation@ManyToMany Relation

Many-To-Many relationship is where one or more rows from one entity are associated with more than one row in other entity.

Let us consider an example of relation between Class and Teacher entities. In bidirectional manner, both Class and Teacher have Many-To-One relation. That means each record of Class is referred by Teacher set (teacher ids), which should be primary keys in Teacher table and stored in Teacher\_Class table and vice versa. Here, Teachers\_Class table contains both foreign Key fields. Create a JPA project in eclipse IDE named **JPA\_Eclipselink\_MTM**. All the modules of this project are shown as follows:

### Creating Entities

Follow the above given diagram for creating entities. Create a package named under **‘src’** package. Create a class named **Clas.java** under given package. The class Department entity is shown as follows:

@Entity

public class Clas {

@ManyToMany(targetEntity=Teacher.class)

private Set teacherSet;

}

Create the second entity in this relation -Employee entity class, named**Teacher.java** under package. The Employee entity class is shown as follows:

@Entity

public class Teacher {

@ManyToMany(targetEntity = Clas.class)

private Set clasSet;

}

### Service Classes

This module contains the service classes, which implements the relational part using the attribute initialization. Create a package under **‘src’** package named. The DAO class named**ManyToMany.java** is created under given package. The DAO class is shown as follows:

//Create Clas Entity

Clas clas1 = new Clas(0, "1st", null);

Clas clas2 = new Clas(0, "2nd", null);

//Store Clas

entitymanager.persist(clas1);

entitymanager.persist(clas2);

//Create Clas Set1

Set<Clas> classSet1 = new HashSet();

classSet1.add(clas1);

classSet1.add(clas2);

//Create Clas Set2

Set<Clas> classSet2 = new HashSet();

classSet2.add(clas1);

classSet2.add(clas2);

//Create Teacher Entity

Teacher teacher1 = new Teacher(0, "Satish","Java",classSet1);

Teacher teacher2 = new Teacher(0, "Krishna","Adv Java",classSet2);

//Store Teacher

entitymanager.persist(teacher1);

entitymanager.persist(teacher2);

# Criteria API

The Criteria API is a predefined API used to define queries for entities. It is the alternative way of defining a JPQL query. These queries are type-safe, and portable and easy to modify by changing the syntax. Similar to JPQL it follows abstract schema (easy to edit schema) and embedded objects. The metadata API is mingled with criteria API to model persistent entity for criteria queries.

The major advantage of the criteria API is that errors can be detected earlier during compile time. String based JPQL queries and JPA criteria based queries are same in performance and efficiency.

## History of criteria API

The criteria API is included into all versions of JPA therefore each step of criteria API is notified in the specifications of JPA.

* In JPA 2.0, the criteria query API, standardization of queries are developed.
* In JPA 2.1, Criteria update and delete (bulk update and delete) are included.

## Criteria Query Structure

The Criteria API and the JPQL are closely related and are allowed to design using similar operators in their queries. It follows javax.persistence.criteria package to design a query. The query structure means the syntax criteria query.

The following simple criteria query returns all instances of the entity class in the data source.

EntityManager em = ...;

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Entity class> cq = cb.createQuery(Entity.class);

Root<Entity> from = cq.from(Entity.class);

cq.select(Entity);

TypedQuery<Entity> q = em.createQuery(cq);

List<Entity> allitems = q.getResultList();

The query demonstrates the basic steps to create a criteria.

* EntityManager instance is used to create a CriteriaBuilder object.
* CriteriaQuery instance is used to create a query object. This query object’s attributes will be modified with the details of the query.
* CriteriaQuery.from method is called to set the query root.
* CriteriaQuery.select is called to set the result list type.
* TypedQuery<T> instance is used to prepare a query for execution and specifying the type of the query result.
* getResultList method on the TypedQuery<T> object to execute a query. This query returns a collection of entities, the result is stored in a List.

Create a JPA Project in the eclipse IDE named **JPA\_Eclipselink\_Criteria**. All the modules of this project are shown as follows:

### Creating Entities

Create a package named under **‘src’**package.

Create a class named **Employee.java** under given package. The class Employee entity is shown as follows:

@Entity

public class Employee { }

### Service classes

This module contains the service classes, which implements the Criteria query part using the MetaData API initialization. Create a package named. The class named **CriteriaAPI.java**is created under given package. The DAO class is shown as follows:

CriteriaBuilder criteriaBuilder = entitymanager.getCriteriaBuilder();

CriteriaQuery<Object> criteriaQuery = criteriaBuilder.createQuery();

Root<Employee> from = criteriaQuery.from(Employee.class);

//select all records

System.out.println(“Select all records”);

CriteriaQuery<Object> select = c riteriaQuery.select(from);

TypedQuery<Object> typedQuery = entitymanager.createQuery(select);

List<Object> resultlist = typedQuery.getResultList();

//Ordering the records

System.out.println(“Select all records by follow ordering”);

CriteriaQuery<Object> select1 = criteriaQuery.select(from);

select1.orderBy(criteriaBuilder.asc(from.get("ename")));

TypedQuery<Object> typedQuery1 = entitymanager.createQuery(select);

List<Object> resultlist1 = typedQuery1.getResultList();

}