Table of Contents

ORM Object Relational Map

Es en el **desarrollo de aplicaciones con acceso a base de datos** donde la pregunta de qué es un ORM cobra más importancia. Un ORM simplifica el trabajo del desarrollo de este tipo de aplicaciones enormemente. Veamos en detalle, cómo lo consigue.

Qué es un ORM

Si eres un desarrollador de aplicaciones multiplataforma, seguramente, alguna vez has programado una aplicación que se conecta a una base de datos. Realizar el mapeo, que no es otra cosa que transformar la información de la base de datos en tablas a objetos de la aplicación y viceversa, es un arduo trabajo. Hasta no hace mucho se usaba el lenguaje de programación SQL. Pero esto ha cambiado. ¿Qué es un ORM? Pues bien, es el responsable del **mapeo automático**.

ORM procede de las siglas ORM, **Object Relational Mapping**. El trabajo deja de ser manual ya que el ORM lo realizará de forma independiente de la base de datos. Además, gracias al mapeo automático podrás cambiar de motor de base de datos fácilmente y cuando quieras.

El ORM es un **modelo de programación** que transforma las tablas de una base de datos en entidades para simplificar enormemente la tarea del programador.

¿Por qué es mejor un ORM que otro lenguaje de programación?

Un ORM tiene diferentes ventajas y alguna desventaja también. Para decidirte a usarlo en lugar de otro lenguaje de programación debes conocerlos. Sus ventajas son la **facilidad y velocidad de uso**, la seguridad contra ataques informáticos o la forma de abstracción de la base de datos que estemos utilizando.

Por otro lado, para programar con ORM es necesario aprender su lenguaje y hay entornos con gran volumen que pueden ver mermado el rendimiento.

Uno de los mapeos automáticos más utilizados es de JAVA y se llama Hibernate, pero también están iBatis, Ebean, para .NET nHibernate, Entity Framework, o para PHP Doctrine y Propel, entre otros.

JPA Java Persistence API

JPA - Introducción

Cualquier aplicación empresarial realiza operaciones con la base de almacenar y recuperar grandes cantidades de datos. A pesar de todas las tecnologías disponibles para la gestión de almacenamiento, los desarrolladores de aplicaciones normalmente lucha para realizar operaciones de base eficiente.

En general, los desarrolladores de Java utilizan gran cantidad de código, o bien utilice el marco propio para interactuar con la base de datos, mientras que con JPA, la carga de interactuar con la base de datos reduce considerablemente. Constituye un puente entre los modelos de objetos (programa Java) y modelos relacionales (programa de base de datos).

Las discordancias entre los modelos relacionales y objeto

Objetos relacionales están representados en un formato tabular, mientras que modelos de objetos son representados en un gráfico de interconexión formato de objeto. Al almacenar y recuperar un modelo de objetos a partir de una base de datos relacional, alguna incongruencia se produce debido a las siguientes razones:

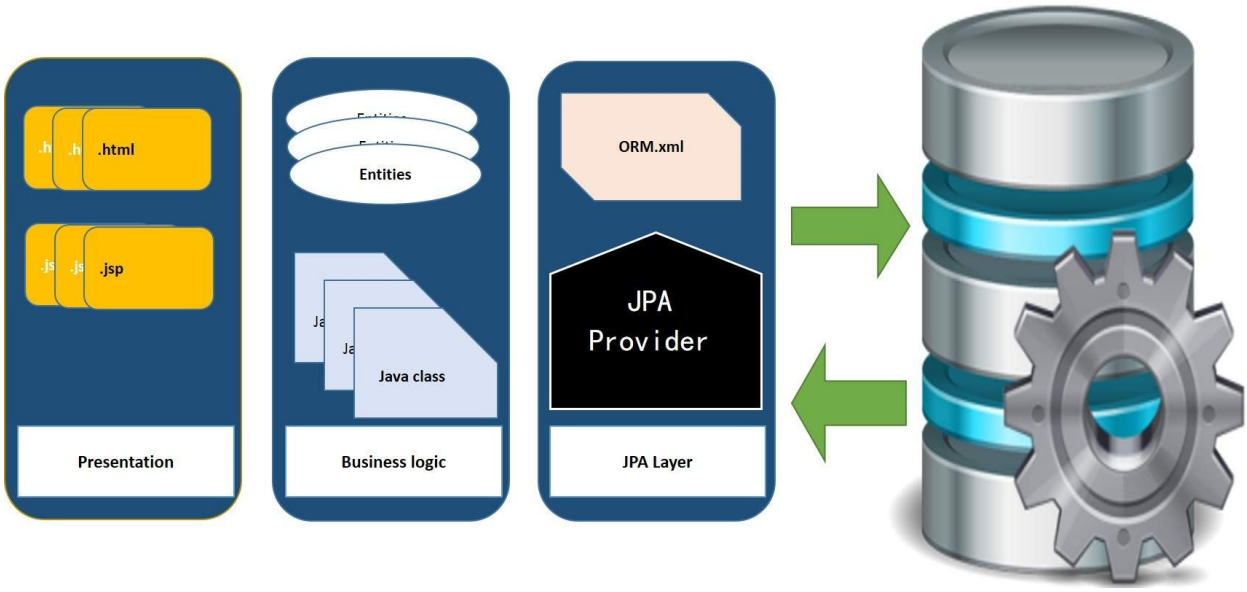
- **Granularidad**: modelo de objetos tiene mayor nivel de detalle del modelo relacional.
- **Los subtipos**: subtipos (significa herencia) no son compatibles con todos los tipos de bases de datos relacionales.
- **Identidad** : Como modelo de objetos, modelo relacional no identidad, a la vez que exponer por escrito la igualdad.
- **Asociaciones** : modelos relacionales no puede determinar relaciones múltiples mientras se mira a un objeto modelo de dominio.
- **Navegación de datos**: Los datos de navegación entre objetos en un objeto red es diferente en ambos modelos.

¿Qué es JPA?

Java Persistence API es un conjunto de clases y métodos que persistentemente almacenar la gran cantidad de datos a una base de datos que es proporcionada por Oracle Corporation.

¿Dónde usar JPA?

Para reducir la carga de escribir códigos relacionales para gestión de objetos, un programador sigue el "Proveedor" marco JPA, que permite la fácil interacción con instancia de la base de datos. Aquí el marco necesario se realiza a través de JPA.



JPA Historia

Las versiones anteriores de EJB, define persistencia capa combinada con capa de lógica empresarial utilizando javax.ejb.EntityBean Interfaz.

- Al mismo tiempo, introduce EJB 3.0 , la persistencia de capa se separó y se especifica en JPA 1.0 (Java Persistence API). Las especificaciones de esta API se libera junto con las especificaciones de JAVA EE5 11 de mayo de 2006, utilizando JSR 220.
- JPA 2.0 fue lanzado con las especificaciones de JAVA EE6 en 10 de diciembre de 2009 como parte del proceso de Comunidad Java JSR 317.
- JPA 2.1 fue lanzado con la especificación de JAVA EE7 el 22 de abril de 2013 utilizando JSR 338.

JPA Proveedores

APP es una API open source, por lo tanto distintos proveedores como Oracle, Redhat, Eclipse, etc. proporcionar nuevos productos mediante la adición de la persistencia JPA sabor en ellas. Algunos de estos productos incluyen:

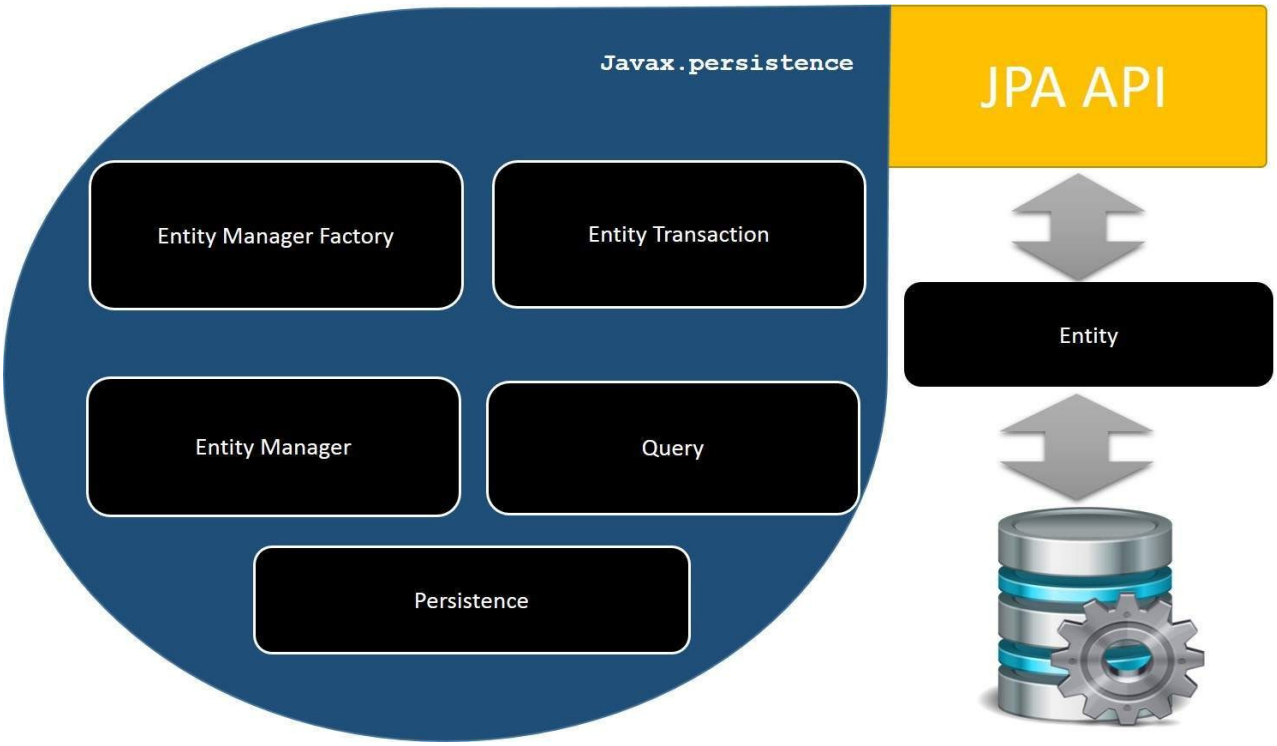
Hibernate, Eclipselink, Toplink, Muelle Datos JPA, etc.

JPA - Arquitectura

Java Persistence API es una fuente para almacenar las entidades empresariales entidades relacionales. Muestra cómo definir un Plain Oriented Java Object (POJO) como una entidad y la forma de gestionar las relaciones con las entidades.

Nivel de clase Arquitectura

La siguiente imagen muestra el nivel de clase arquitectura de JPA. Muestra las clases principales y las interfaces de JPA.



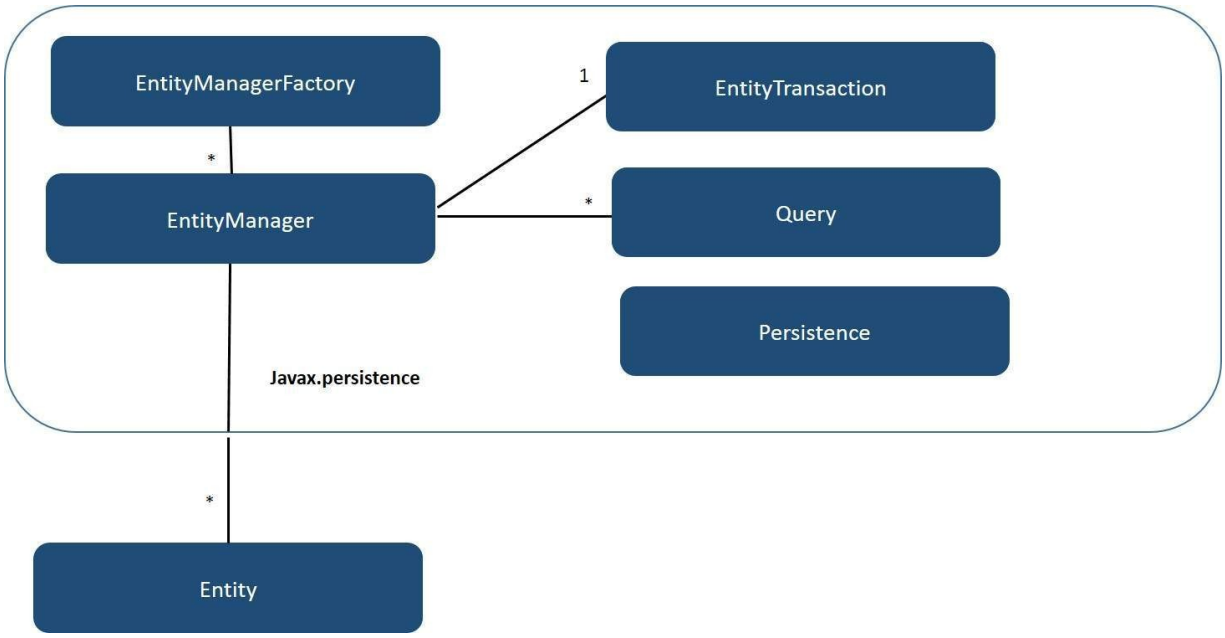
La siguiente tabla describe cada una de las unidades mostradas en la arquitectura.

Unidades	Descripción
EntityManagerFactory	Esta es una clase de fábrica de EntityManager. Crea y gestiona múltiples instancias EntityManager.
EntityManager.	Es una interfaz, que gestiona la persistencia de objetos. Funciona como instancia de consulta.
Entidad	Las entidades son los objetos de persistencia, tiendas como registros en la base de datos.
EntityTransaction	Tiene una relación de uno a uno con EntityManager. Para cada método EntityManager, se mantienen las operaciones de EntityTransaction clase.
Persistencia	Esta clase contiene métodos estáticos para obtener EntityManagerFactory.
Consulta	Esta interfaz es implementada por cada proveedor JPA relacional para obtener objetos que cumplan los criterios.

Por encima de las clases y las interfaces se utilizan para almacenar entidades en una base de datos como un registro. Ayudan a los programadores por reducir sus esfuerzos por escribir los códigos para almacenar datos en una base de datos, de modo que puedan concentrarse en actividades más importantes, tales como escribir los códigos para la cartografía de la clases con tablas de la base de datos.

Las relaciones entre las clases JPA.

En la arquitectura, las relaciones entre las clases e interfaces pertenecen a la clase javax.persistence paquete. El siguiente diagrama muestra la relación entre ellos.



- La relación entre EntityManagerFactory EntityManager y es de **uno a varios**. Se trata de una clase de fábrica a instancias EntityManager.
- La relación entre método EntityManager y EntityTransaction es **uno a uno**. EntityManager para cada operación, hay un ejemplo EntityTransaction.
- La relación entre EntityManager y consulta es de **uno a varios**. Número de consultas puede ejecutar mediante una instancia EntityManager.
- La relación entre Entidad y EntityManager es **uno de muchos**. Un EntityManager instancia puede administrar varias entidades.

JPA - Componentes ORM

La mayoría de las aplicaciones que utilizan bases de datos relacionales para almacenar datos. Recientemente, muchos proveedores de base de datos de objeto de reducir su carga de mantenimiento de datos. Esto significa Object Relational Mapping de base de datos o las tecnologías están al cuidado de almacenar, recuperar, actualizar y mantener los datos. La parte principal de este objeto es tecnología relacional orm mapeo.xml files. Como xml no necesitan compilación, podemos realizar cambios fácilmente a varias fuentes de datos con menos administración.

Object Relational Mapping

Object Relational Mapping (ORM) informa brevemente sobre lo que es ORM y cómo funciona. ORM es una capacidad de programación para convertir los datos de tipo de objeto de tipo relacional y viceversa.

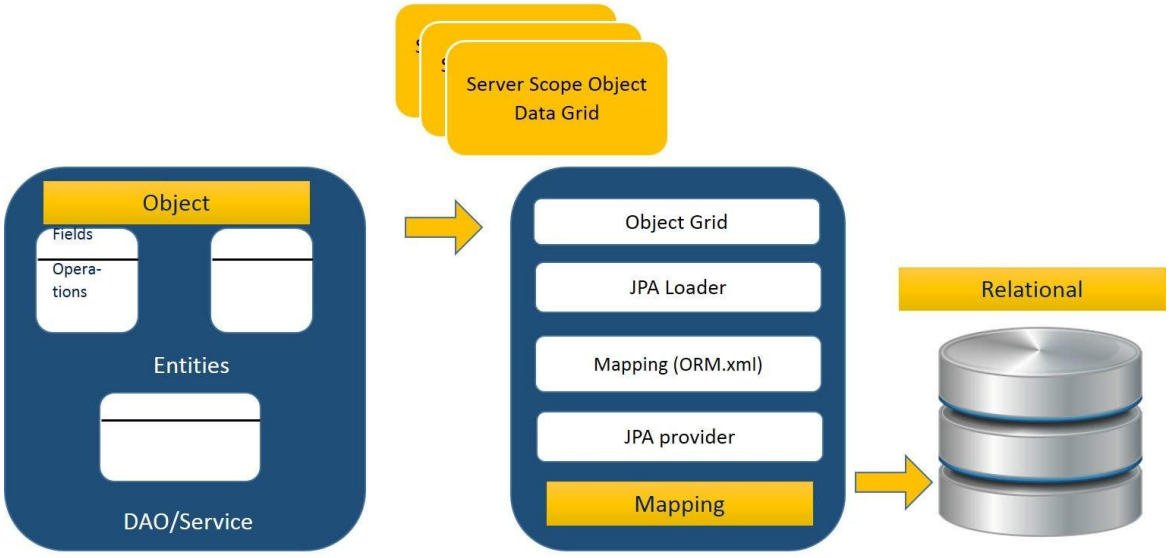
La principal característica de ORM es de cartografía o enlazar un objeto a sus datos en la base de datos. Si bien la cartografía, tenemos que tener en cuenta los datos, el tipo de datos, y sus relaciones con la entidad o entidades en cualquier otra tabla.

Funciones avanzadas

- **Persistencia idiomáticas**: le permite escribir la persistencia clases utilizando las clases orientadas a objetos.
- **Alto rendimiento** : tiene muchos buscar técnicas técnicas de bloqueo y esperanzador.
- **Fiable**: es muy estable y utilizado por muchos programadores profesionales.

Arquitectura ORM

El ORM arquitectura es similar al siguiente.



Object Relational Mapping

La arquitectura se explica cómo datos del objeto se almacenan en bases de datos relacionales en tres fases.

Fase1

La primera fase, denominada **fase de datos del objeto**, contiene las clases POJO, interfaces y clases. Es el principal componente de empresa de capa, que tiene lógica de negocios operaciones y atributos.

Por ejemplo, vamos a tener una base de datos de empleados como esquema.

- POJO empleado clase contiene atributos como ID, nombre, salario, y designación. También contiene métodos setter y getter de esos atributos.
- Empleado DAO/clases de servicio contienen métodos de servicio tales como crear empleado, encontrar los empleados y eliminar empleado.

Fase 2

La segunda fase, denominada **fase de mapeo o persistencia** JPA proveedor, contiene, archivo de mapas (ORM.xml), JPA Cargadora y rejilla de objeto.

- **JPA Provider** : es el producto de proveedor que contiene el JPA sabor (javax.persistence). Por ejemplo Eclipselink, Toplink, hibernación, etc.
- **Archivo de asignación** : El archivo de asignación (ORM.xml) contiene configuración de la asignación entre los datos de un POJO clase y los datos en una base de datos relacional.
- **JPA Cargador** cargador : La APP funciona como una memoria caché. Puede cargar los datos relacionales. Funciona como una copia de base de datos para interactuar con las clases de servicio de datos POJO POJO (atributos de clase).
- **Rejilla de Objeto** : es una ubicación temporal que puede almacenar una copia de los datos relacionales, como una memoria caché. Todas las consultas en la base de datos se efectuará, primero en los datos del objeto grid. Sólo después de que se ha comprometido, que afecta a la base de datos principal.

Fase 3

La tercera fase es la **fase de datos relacionales**. Contiene los datos relacionales que lógicamente está conectado al componente comercial. Como se ha indicado anteriormente, sólo cuando el componente comercial se compromete los datos, que se almacenan en la base de datos físicamente. Hasta entonces, los datos modificados se almacenan en una memoria caché como un formato de cuadrícula. El proceso de obtención de los datos es idéntica a la de almacenar los datos.

El mecanismo de la interacción mediante programación por encima de tres fases se denomina **asignación objeto-relacional**.

Mapping.xml

La asignación de archivo.xml es el de instruir a la JPA proveedor mapa las clases de entidad con las tablas de la base de datos.

Tomemos un ejemplo del Empleado entidad que contiene cuatro atributos. El POJO entidad clase de empleados **Employee.java** es la siguiente:

```
public class Employee
{
   private int eid;
   private String ename;
   private double salary;
   private String deg;
   public Employee(int eid, String ename, double salary, String deg)
   {
      super( );
      this.eid = eid;
      this.ename = ename;
      this.salary = salary;
      this.deg = deg;
   }

   public Employee( )
   {
      super();
   }

   public int getEid( )
```

```
   {
      return eid;
   }
   public void setEid(int eid)
   {
      this.eid = eid;
   }
   public String getEname( )
   {
      return ename;
   }
   public void setEname(String ename)
   {
      this.ename = ename;
   }

   public double getSalary( )
   {
      return salary;
   }
   public void setSalary(double salary)
   {
      this.salary = salary;
   }

   public String getDeg( )
   {
      return deg;
   }
   public void setDeg(String deg)
   {
      this.deg = deg;
   }
}
```

El código anterior es la entidad Employee clase POJO. Contiene cuatro atributos b>eid, **ename**, **salary**, and **deg**. Considerar estos atributos como los campos de tabla en una tabla y **eid** como clave principal de la tabla. Ahora tenemos que tener en cuenta en el diseño del archivo de asignación para hibernar. Nombre del archivo de mapas **mapping.xml** es el siguiente:

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

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
   http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
   version="1.0">

   <description> XML Mapping file</description>

   <entity class="Employee">
      <table name="EMPLOYEETABLE"/>

      <attributes>
         <id name="eid">
            <generated-value strategy="TABLE"/>
         </id>

         <basic name="ename">
            <column name="EMP_NAME" length="100"/>
```

```
        </basic>

        <basic name="salary">
        </basic>

        <basic name="deg">
        </basic>
      </attributes>

   </entity>
</entity-mappings>
```

La secuencia de comandos anterior se utiliza para la cartografía de la clase de entidad con la tabla de la base de datos. En este archivo

- **<entity-mappings>** etiqueta define la definición de esquema para permitir etiquetas de entidad en archivo xml.
- **<description>** : etiqueta ofrece una descripción acerca de la aplicación.
- **<entity>** : etiqueta define la clase de entidad que desea convertir en una tabla en una base de datos. Clase de atributo POJO define el nombre de la clase de entidad.
- **<table>** : etiqueta define el nombre de la tabla. Si desea tener nombres idénticos tanto para la clase, así como la tabla y, a continuación, esta etiqueta no es necesario.
- **<attributes>** : etiqueta define los atributos (campos de una tabla).
- **<id>** : etiqueta define la clave principal de la tabla. El **<genera valor de etiqueta>** define cómo asignar el valor de la clave principal como **automática**, **manual** o de **secuencia**.
- **<basic>** : etiqueta se utiliza para definir los atributos de la tabla.
- **<column-name>** : etiqueta se usa para definir tabla definidas por el usuario nombres de campo en la tabla.

Las anotaciones

Por lo general se utilizan archivos xml para configurar los componentes específicos, o asignación de dos diferentes especificaciones de los componentes. En nuestro caso, tenemos que mantener archivos xml por separado en un marco. Eso significa que al escribir un archivo de asignación xml, necesitamos comparar los atributos de clase POJO con las etiquetas de la entidad en el archivo mapping.xml.

Aquí está la solución. En la definición de la clase, podemos escribir la configuración con anotaciones. Las anotaciones se utilizan para las clases, propiedades y métodos. Las anotaciones comienzan con " @" el símbolo. Las anotaciones son declarados antes de una clase, propiedad o método. Todas las anotaciones de JPA se definen en el **javax.persistence** paquete

Aquí lista de anotaciones utilizadas en nuestros ejemplos se dan a continuación.

Anotación	Descripción
@Entidad	Declara la clase como una entidad o una tabla.
@Tabla	Declara nombre de la tabla.

@Basic	Especifica no campos de restricción explícita.
@Embedded	Especifica las propiedades de la clase o de una entidad cuyo valor es una instancia de una clase se puede incrustar.
@Id	Especifica la propiedad, el uso de la identidad (la clave principal de una tabla de la clase.
@GeneratedValue	Especifica el modo en que la identidad se puede inicializar atributo como automática, manual o valor tomado de la tabla de secuencias.
@Transitorios	Especifica la propiedad que no es constante, es decir, el valor nunca se almacena en la base de datos.
@Columna	Especifica el atributo de columna para la propiedad persistence.
@SequenceGenerator	Especifica el valor de la propiedad que se especifica en la anotación @GeneratedValue. Crea una secuencia.
@TableGenerator	Especifica el generador de valor para la propiedad especificada en la anotación @GeneratedValue. Crea una tabla de generación de valor.
@AccessType	Este tipo de anotación se utiliza para establecer el tipo de acceso. Si establece el valor de @métodos Accesstype() y formattype() CAMPO), luego se produce acceso Campo sabio. Si establece el valor de @métodos Accesstype() y formattype() PROPIEDAD), a continuación, el acceso se produce bienes.
@JoinColumn	Especifica la entidad asociación o entidad colección. Esto se utiliza en muchos-a-uno y uno-a-muchas asociaciones.
@UniqueConstraint	Especifica los campos y las restricciones unique para la primaria o la secundaria.
@ColumnResult	Hace referencia al nombre de una columna de la consulta SQL que utiliza cláusula select.
@ManyToMany	Define una relación many-to-many entre el unir tablas.
@ManyToOne	Define una relación de many-to-one entre el unir tablas.
@OneToMany	Define una relación one-to-many entre los unir tablas.
@OneToOne	Define una relación one-to-one entre los unir tablas.
@NamedQueries	Especifica la lista de consultas con nombre.
@NamedQuery	Especifica una consulta con nombre estático.

Estándar Java Bean

La clase Java encapsula los valores de instancia y sus comportamientos en una sola unidad llamada objeto. Java Bean es un almacenamiento temporal y componentes reutilizables o de un objeto. Se trata de una clase serializable que tiene un constructor predeterminado y métodos get y set para inicializar los atributos de la instancia individual.

Bean Convenios

- Frijol contiene su constructor predeterminado o un archivo que contiene serializa la instancia. Por lo tanto, un frijol puede crear una instancia de otro grano.
- Las propiedades de un frijol pueden ser segregadas en propiedades booleanas o no booleano.
- Propiedad booleanos no contiene **getter** y **setter** métodos.

- Propiedad booleana contienen **setter** y **es** método.
- **Getter** método de cualquier propiedad debe comenzar con pequeñas Letras **get** (java method convention) y continuó con un nombre de campo que comienza con mayúscula. Por ejemplo, el nombre del campo es **salario** por tanto el método getter de este campo es **getSalary ()**.
- **Setter** método de cualquier propiedad debe comenzar con pequeñas Letras **set** (java method convention), continuó con un nombre de campo que comienza con letra mayúscula y el **argument value** para establecer en campo. Por ejemplo, el nombre del campo es **salario** por tanto el método setter de este campo es **setSalary (double sal)**.
- Para propiedad booleana, **is** método para comprobar si es verdadero o falso. Por ejemplo la propiedad booleana **vacío**, el **es** método de este campo es **isEmpty ()**.

JPA - Instalación

Este capítulo le lleva a través del proceso de creación de JPA en Windows y sistemas basados en Linux. JPA puede ser fácilmente instalado e integrada con su actual entorno Java siguiendo unos sencillos pasos sin ningún procedimiento de configuración compleja. Administración de usuarios se requiere mientras que instalación.

Requisitos del sistema

JDK	Java SE 2 JDK 1.5 or above
Memoria	1 GB RAM (se recomienda)
Espacio en disco	No hay requisito mínimo
Versión del sistema operativo	Windows XP or above, Linux

Procedamos ahora con los pasos para instalar app.

Paso 1: Verifique la instalación de Java

En primer lugar, necesitas tener Java Software Development Kit (SDK) instalado en su sistema. Para comprobar esto, ejecutar cualquiera de los dos comandos siguientes dependiendo de la plataforma que está trabajando en.

Si la instalación de Java se ha hecho correctamente, entonces se mostrará la versión actual y la especificación de tu instalación de Java. Un ejemplo de salida se da en la siguiente tabla.

Plataforma	Comando	Muestra salida
Windows	Consola de comandos abierta y tipo: **\>java –version**	Java version "1.7.0_60" Java (TM) SE Run Time Environment (build 1.7.0_60-b19) Java Hotspot (TM) 64-bit Server VM (build 24.60-b09,mixed mode)
Linux	Abra el comando terminal y	java version "1.7.0_25"

escriba: **$java –version**	Open JDK Runtime Environment (rhel-2.3.10.4.el6_4-x86_64) Open JDK 64-Bit Server VM (build 23.7-b01, mixed mode)

- Suponemos que los lectores de este tutorial tienen Java SDK versión 1.7.0_60 instalado en su sistema..
- In case you do not have Java SDK, download its current version from http://www.oracle.com/technetwork/java/javase/downloads/index.html y tenerlo instalado.

Step 2: configurar su entorno Java

Establece la variable de entorno JAVA_HOME para señalar la ubicación del directorio base donde Java está instalado en su máquina. Por ejemplo

Plataforma	Descripción
Windows	Set JAVA_HOME to C:\ProgramFiles\java\jdk1.7.0_60
Linux	Export JAVA_HOME=/usr/local/java-current

Anexar la ruta completa de la ubicación del compilador de Java a la ruta del sistema.

Plataforma	Descripción
Windows	Anexar la cadena "C:\Program Files\Java\jdk1.7.0_60\bin" al final de la variable PATH.
Linux	Export PATH=$PATH:$JAVA_HOME/bin/

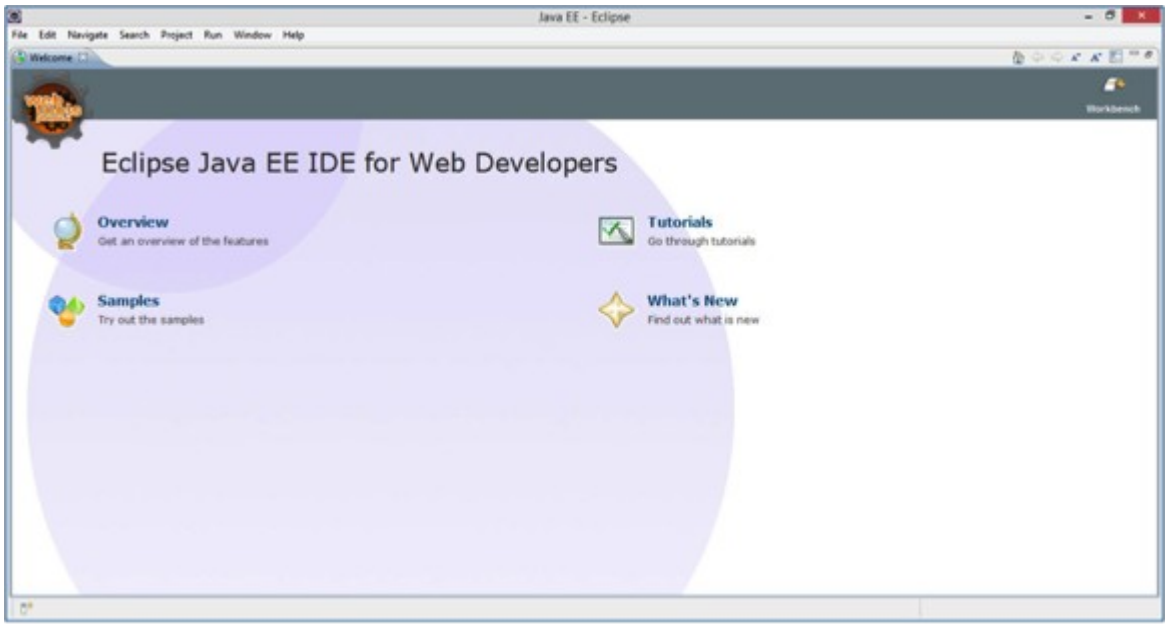
Ejecute el comando **java -version** desde el sistema del símbolo como se explicó anteriormente.

Step3: Installing JPA

You can go through the JPA installation by using any of the JPA Providers from this tutorial, e.g., Eclipselink, Hibernate. Let us follow the JPA installation using Eclipselink. For JPA programming, we require to follow the specific folder framework, therefore it is better to use IDE.

Descargar formulario de Eclipse IDE siguiente enlace https://www.eclipse.org/downloads/ Elegir el EclipseIDE para el desarrollador de JavaEE que es **Eclipse indigo**.

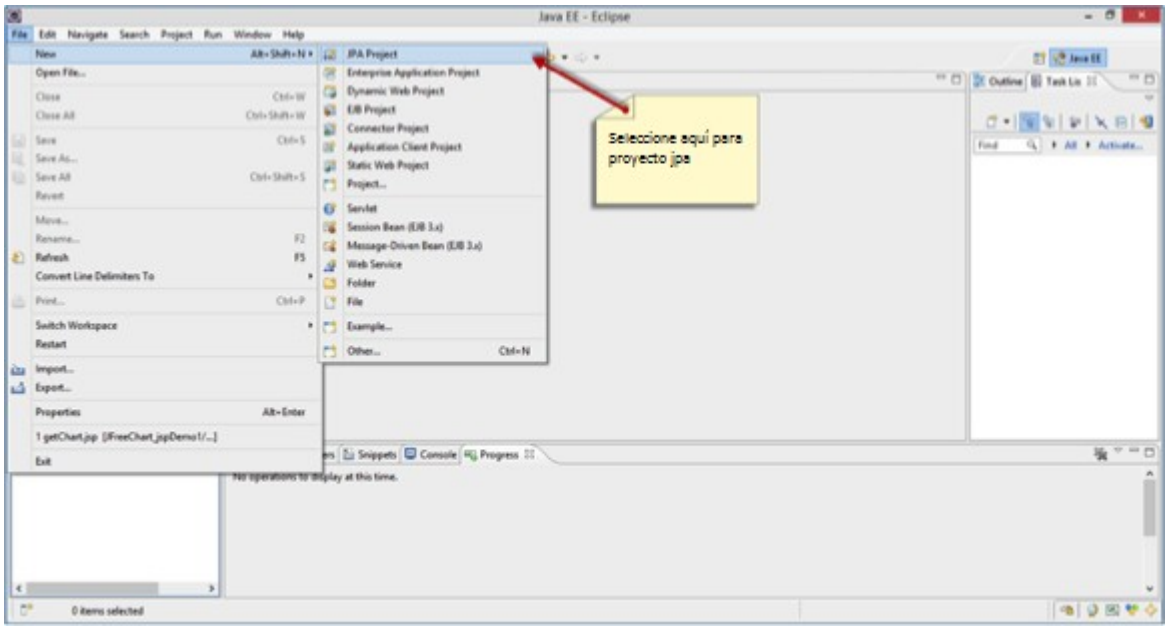
Descomprima el archivo zip de Eclipse en la unidad C. Abierto Eclipse IDE.



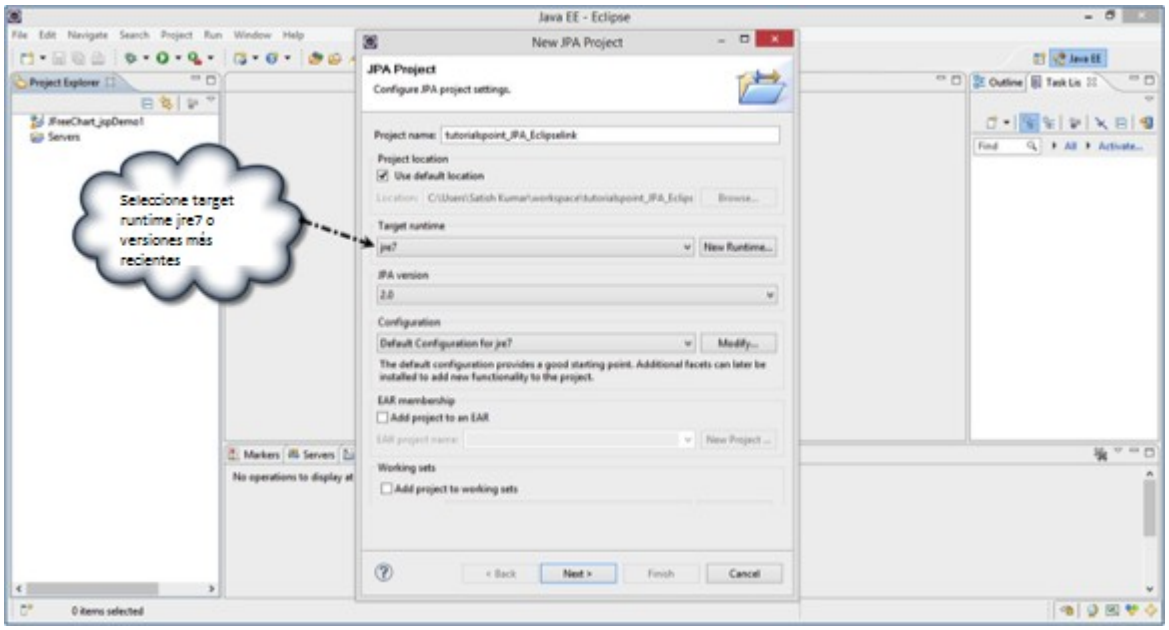
Instalación de JPA con Eclipselink

EclipseLink es una biblioteca, por tanto, no podemos agregarlo directamente a IDE de Eclipse. Para instalar app utilizando Eclipselink debes seguir los pasos indicados a continuación.

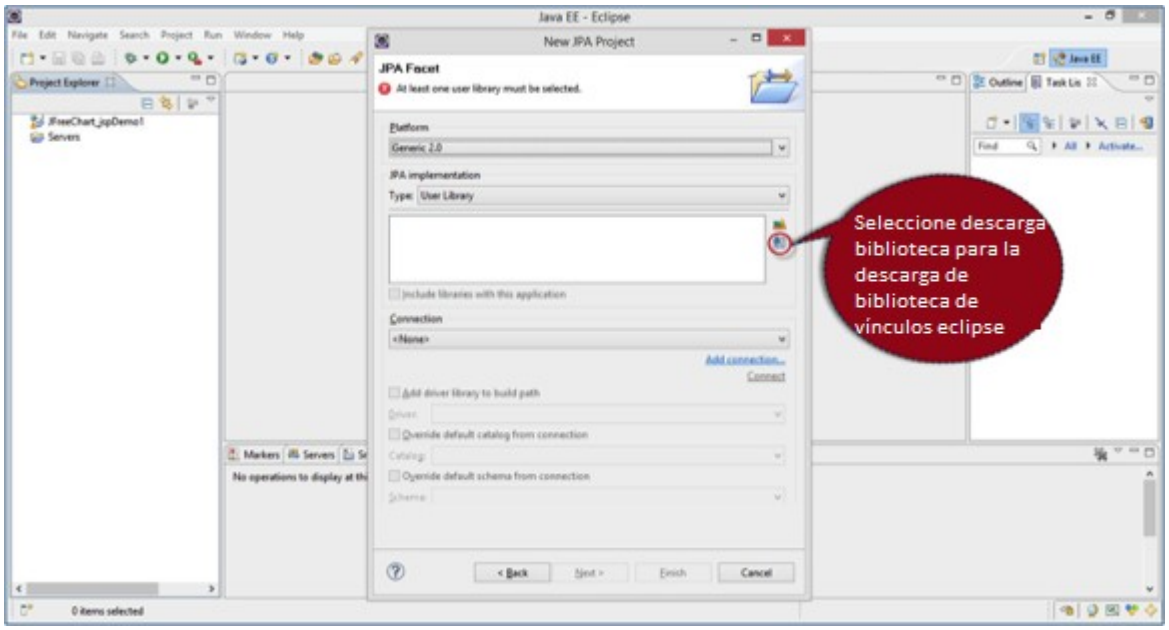
- Crear un nuevo proyecto JPA seleccionando **archivo-> nueva-> proyecto JPA** en el IDE de Eclipse como sigue:



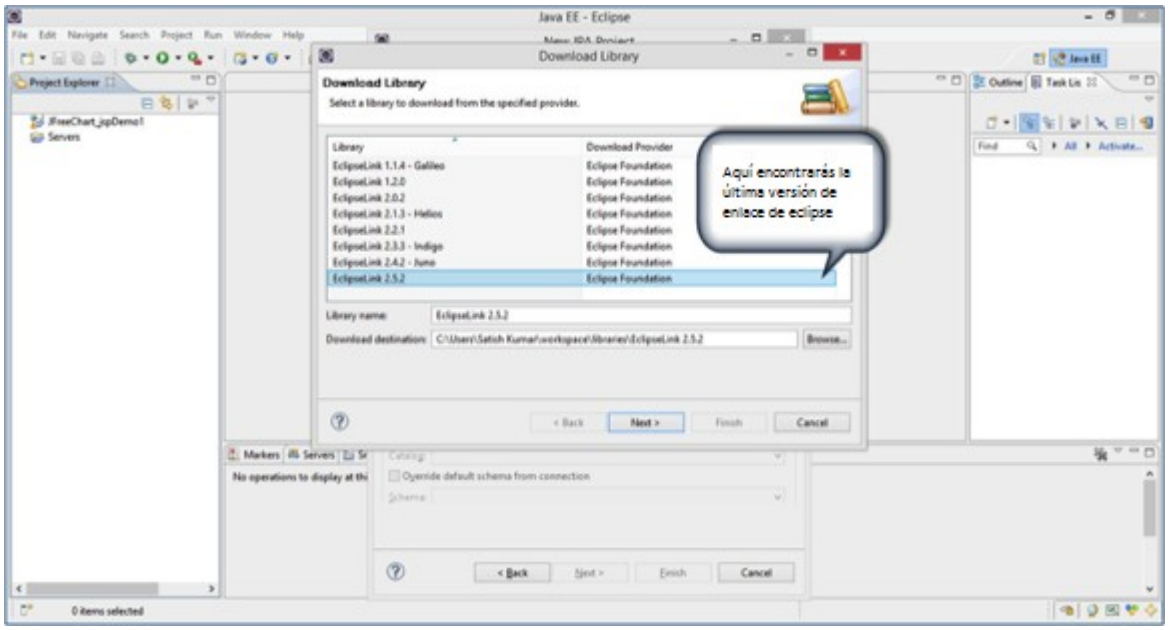
- Usted recibirá un cuadro de diálogo nombre **Nuevo proyecto JPA**. Introduzca el nombre de proyecto **tutorialspoint_JPA_Eclipselink**, Compruebe la versión de **jre** y haga clic en siguiente:



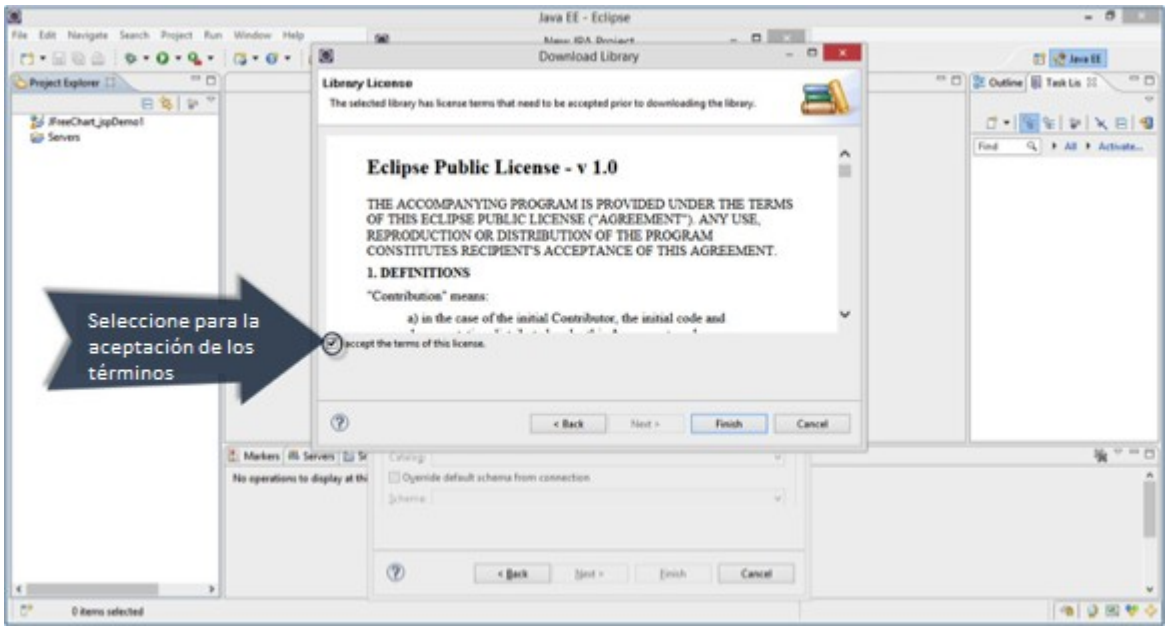
- Haga clic en descargar biblioteca (si no tienes la biblioteca) en la sección de la biblioteca de usuario.



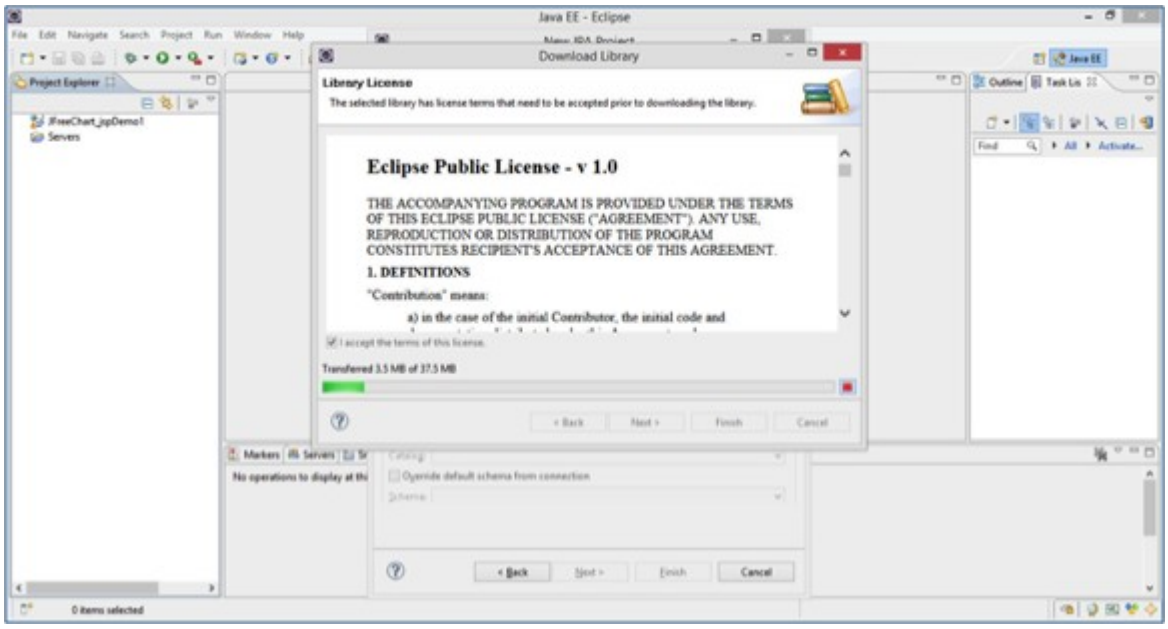
- Seleccione la versión más reciente de Eclipselink biblioteca en el cuadro de diálogo Descargar biblioteca y haga clic en next siguiente:



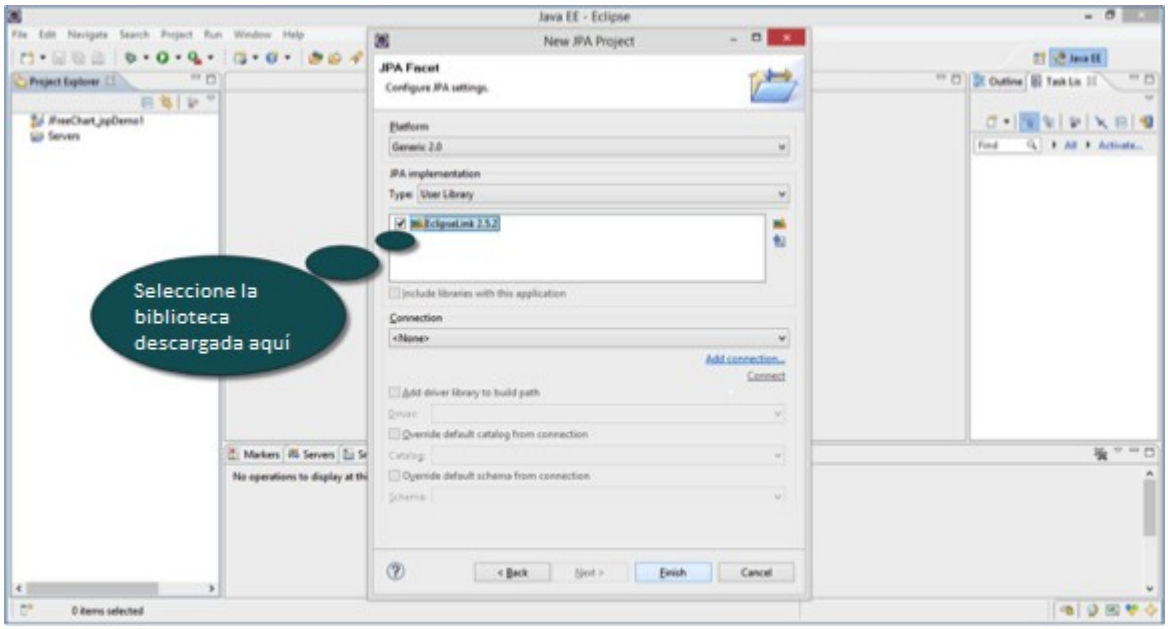
- Aceptar los términos de licencia y haga clic en finalizar para descargar biblioteca.



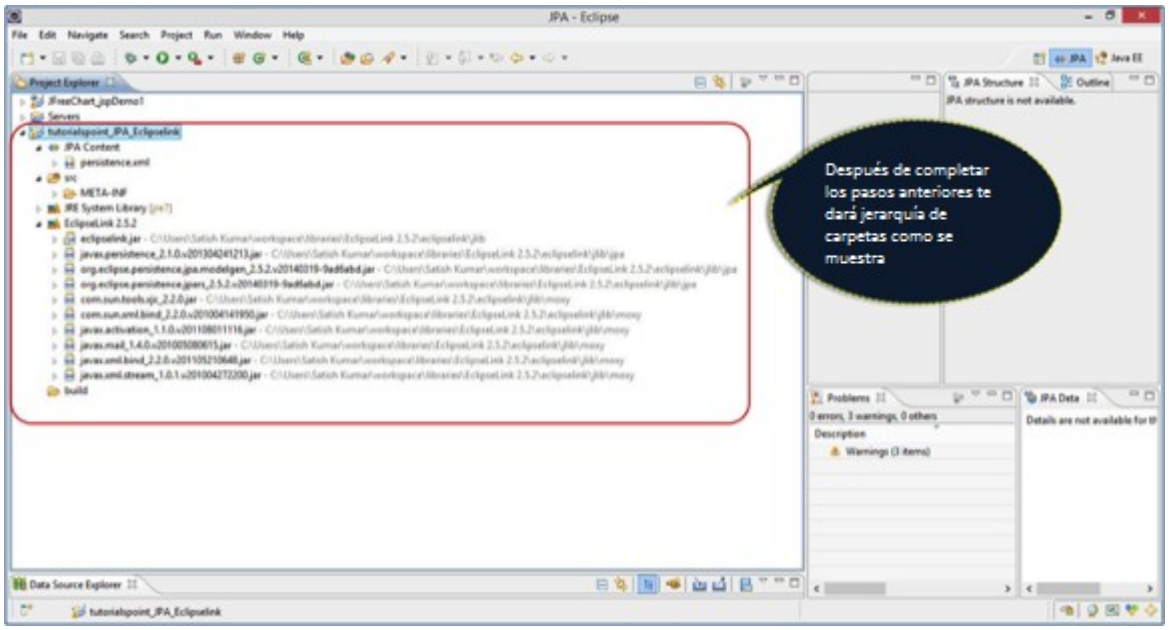
- 6. Descarga comienza como se muestra en la siguiente captura de pantalla.



- Después de descargar, seleccione la biblioteca descargada en el apartado de biblioteca y haga clic en finalizar.



- Finalmente tienes el archivo de proyecto en el **Explorador de paquete** en Eclipse IDE. Extraer todos los archivos, usted conseguirá la carpeta y la jerarquía de archivos de la siguiente manera

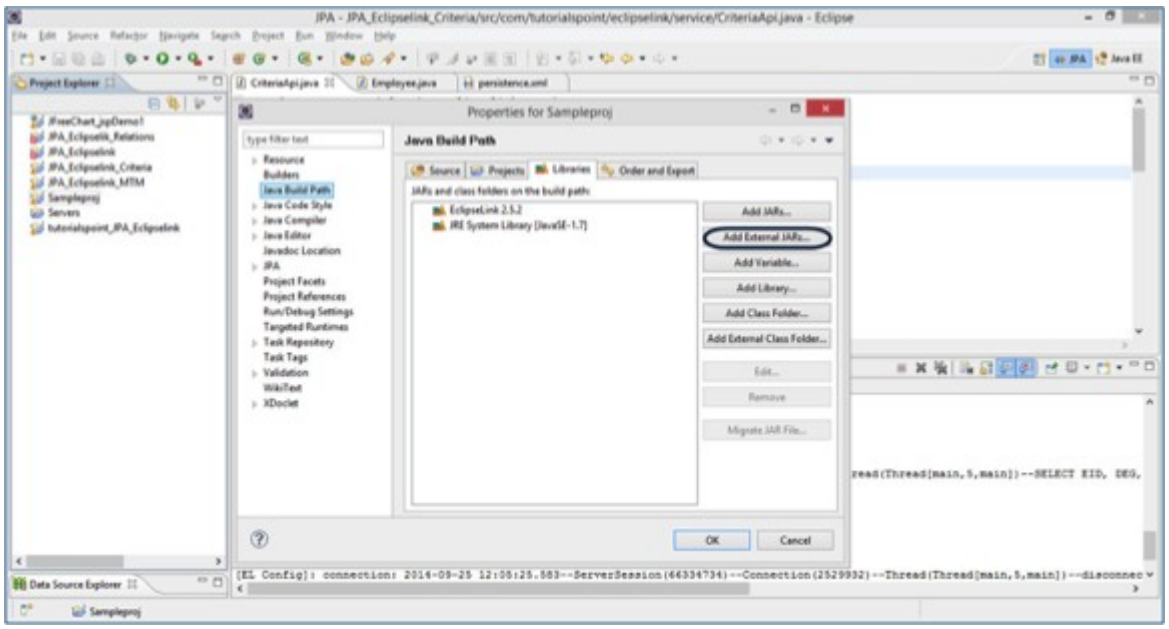


Conector de MySQL añadiendo al proyecto

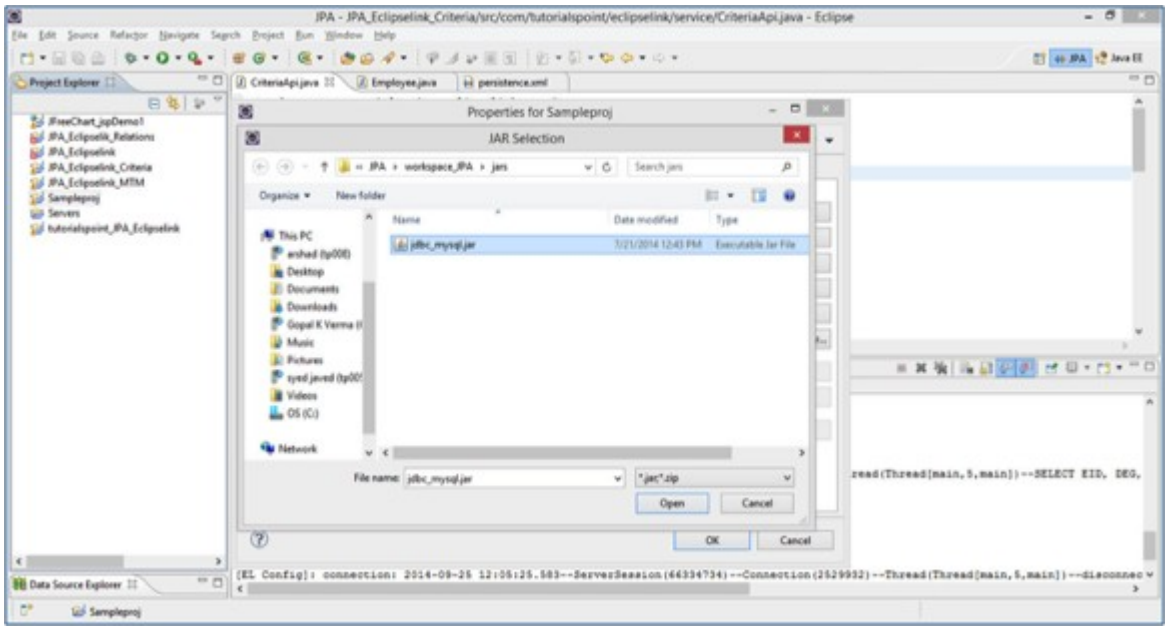
Cualquier ejemplo que discutimos aquí requiere conectividad de base de datos. Consideremos la base de datos MySQL para las operaciones de base de datos. Requiere frasco mysql-conector para interactuar con un programa Java.

Seguir los pasos para configurar el tarro de la base de datos en su proyecto.

- Ir a proyecto propiedades -> Java Build Path por haga clic derecho sobre él. Usted recibirá un cuadro de diálogo como se muestra en la siguiente captura de pantalla. Haga clic en Add External Jars.



- Ir a la ubicación de la jarra en la memoria del sistema, seleccione el archivo y haga clic en abrir.



- Haga clic en aceptar en el cuadro de diálogo Propiedades. Usted conseguirá la jarra MySQL-conector en su proyecto. Ahora eres capaz de base de datos de operaciones utilizando MySQL.

JPA - Administradores de la Entidad

Este capítulo utiliza un sencillo ejemplo para mostrar cómo funciona JPA. Consideremos Gestión empleado como un ejemplo. Supongamos que el empleado Gestión crea, actualiza y elimina los registros de un empleado. Como ya se ha mencionado, estamos utilizando bases de datos MySQL para operaciones de base de datos.

Los módulos principales para este ejemplo son los siguientes:

- **Model or POJO**

  Employee.java

- **Persistence**

  Persistence.xml

- **Service**

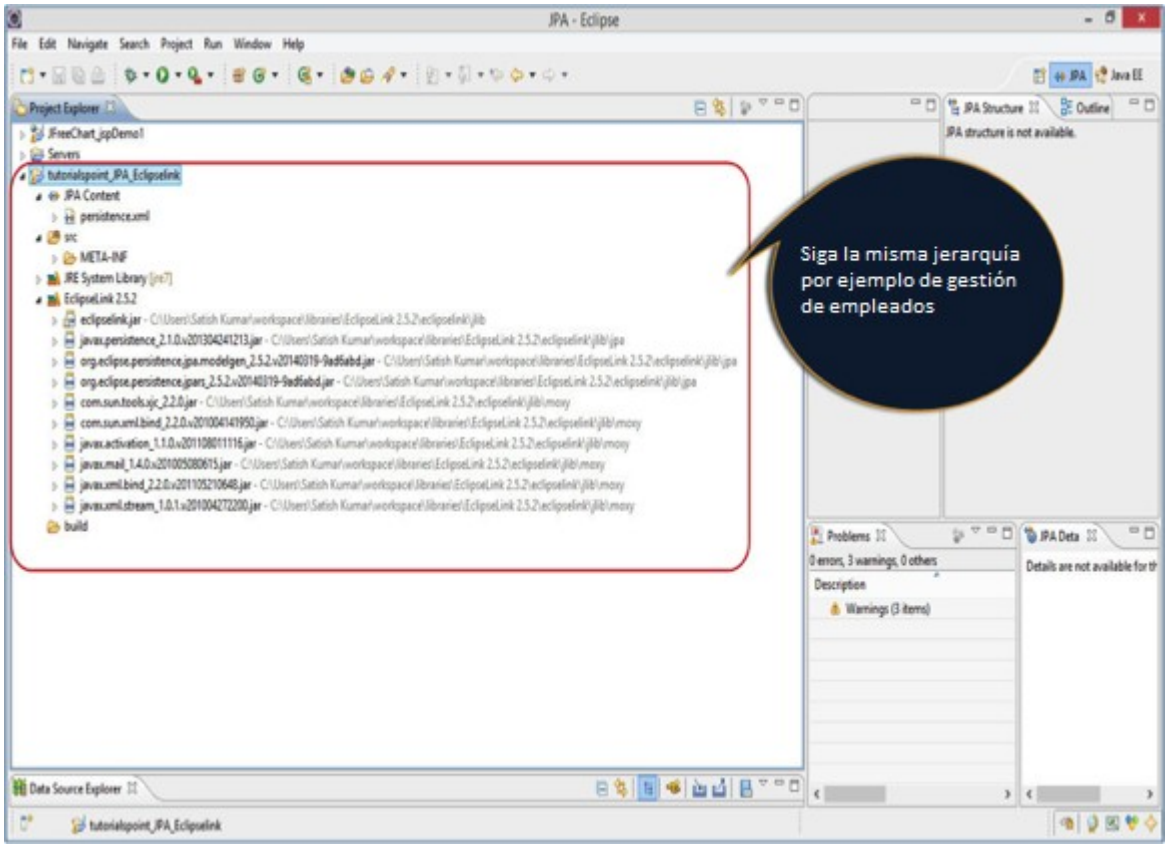
  CreatingEmployee.java

  UpdatingEmployee.java

  FindingEmployee.java

  DeletingEmployee.java

Tomemos la jerarquía del paquete que hemos utilizado en la instalación de JPA con Eclipselink . Seguimiento de la jerarquía de este ejemplo como se muestra a continuación:



Crear entidades

Las entidades no son sino los frijoles o modelos. En este ejemplo, usaremos **empleado** como entidad. **eid**, **ename**, **salario**, y **deg** son los atributos de esta entidad. Contiene un constructor predeterminado, así como los métodos setter y getter de esos atributos.

En el se muestra jerarquía, crear un paquete denominado '**com.tutorialspoint.eclipselink.entity'**, en '**src'** (Fuente). Crear una clase denominada **Employee.java** bajo determinado paquete de la siguiente manera:

```
package com.tutorialspoint.eclipselink.entity;

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

@Entity
@Table
public class Employee
{
   @Id
   @GeneratedValue(strategy= GenerationType.AUTO)
   private int eid;
   private String ename;
   private double salary;
   private String deg;
   public Employee(int eid, String ename, double salary, String deg)
   {
```

```
        super( );
        this.eid = eid;
        this.ename = ename;
        this.salary = salary;
        this.deg = deg;
    }

    public Employee( )
    {
        super();
    }

    public int getEid( )
    {
        return eid;
    }
    public void setEid(int eid)
    {
        this.eid = eid;
    }
    public String getEname( )
    {
        return ename;
    }
    public void setEname(String ename)
    {
        this.ename = ename;
    }

    public double getSalary( )
    {
        return salary;
    }
    public void setSalary(double salary)
    {
        this.salary = salary;
    }

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

En el código anterior, hemos utilizado anotación @Entidad POJO para hacer esta clase de entidad.

Antes de pasar al módulo siguiente tenemos que crear base de datos para entidad relacional, que registrará la base de datos en **persistence.xml** archivo. Abierto MySQL workbench y escriba la siguiente consulta.

```
create database jpadb
use jpadb
```

Persistence.xml

Este módulo tiene un papel crucial en el concepto de JPA. En este archivo xml que se registrará la base de datos y especificar la clase de entidad.

Lo anterior se muestra en la jerarquía del paquete, persistence.xml en JPA Contenido del paquete es la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="Eclipselink_JPA"
      transaction-type="RESOURCE_LOCAL">

      <class>com.tutorialspoint.eclipselink.entity.Employee</class>

      <properties>

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

      </properties>

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

En el xml, **<persistence-unit>** etiqueta se define con un nombre específico de persistencia JPA. La **<class>** etiqueta define la clase de entidad con nombre del paquete. La **<properties>** etiqueta define todas las propiedades, y **<property>** etiqueta define cada propiedad como registro de base de datos, especificación URL, nombre de usuario y contraseña. Estas son las propiedades Eclipselink. Este archivo configurará la base de datos.

Operaciones de persistencia

Las operaciones de persistencia se utilizan para interactuar con una base de datos y son **carga** y **tienda** operaciones. En un componente de negocio, todas las operaciones de persistencia caen bajo clases de servicio.

En el anterior se muestra la jerarquía de paquete, crear un paquete denominado ‘**com.tutorialspoint.eclipselink.service’**, en ‘**src’** (fuente) paquete. Todas las clases de servicio nombradas CreateEmloyee.java, UpdateEmployee.java, FindEmployee.java, y DeleteEmployee.java. viene en el paquete determinado como sigue:

Crear empleado

El segmento de código siguiente muestra cómo crear una clase empleado llamada **CreateEmployee.java**.

```
package com.tutorialspoint.eclipselink.service;

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

import com.tutorialspoint.eclipselink.entity.Employee;

public class CreateEmployee
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );

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

      entitymanager.persist( employee );
      entitymanager.getTransaction( ).commit( );

      entitymanager.close( );
      emfactory.close( );
   }
}
```

En el código anterior el **createEntityManagerFactory ()** crea una unidad de persistencia proporcionando el mismo nombre único que proporcionamos para unidad de persistencia en persistent.xml archivo. El **entitymanagerfactory** objeto creará el **entitymanger** instancia mediante el uso de **createEntityManager()** método. El **entitymanager** objeto crea **entitytransaction** instancia para la gestión de transacciones. Mediante el uso de **entitymanager** objeto, podemos persistimos entidades en la base de datos.

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

Para el resultado, abra el MySQL workbench y escriba las siguientes consultas.

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

La tabla de base de datos efectuado llamada **empleado** se mostrará en un formato tabular de la siguiente manera:

Eid	Ename	Salary	Deg
1201	Gopal	40000	Technical Manager

Empleado de actualización

Para actualizar los registros de un empleado, tenemos que recuperar la forma existente de registros la base de datos, realizar cambios y finalmente lo comprometen a la base de datos. La clase denominada **UpdateEmployee.java** se muestra como sigue:

```
package com.tutorialspoint.eclipselink.service;

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

import com.tutorialspoint.eclipselink.entity.Employee;

public class UpdateEmployee
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory = Persistence.
      createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );
      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();
   }
}
```

Después de la compilación y ejecución del programa anterior usted recibirá notificaciones de biblioteca Eclipselink sobre el panel de la consola de eclipse IDE.

Para el resultado, abra el MySQL workbench y escriba las siguientes consultas.

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

La tabla de base de datos efectuado llamada **empleado** se mostrará en un formato tabular de la siguiente manera:

Eid	Ename	Salary	Deg
1201	Gopal	46000	Technical Manager

El salario del empleado, 1201 se actualiza a 46000.

Encontrar empleados

Para encontrar los registros de un empleado, tenemos que recuperar los datos existentes desde la base de datos y mostrarlo. En esta operación, EntityTransaction no se aplica al recuperar un registro.

La clase denominada **FindEmployee.java** como sigue.

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.Employee;

public class FindEmployee
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager();
      Employee employee = entitymanager.find( Employee.class, 1201 );

      System.out.println("employee ID = "+employee.getEid( ));
      System.out.println("employee NAME = "+employee.getEname( ));
      System.out.println("employee SALARY = "+employee.getSalary( ));
      System.out.println("employee DESIGNATION = "+employee.getDeg( ));
   }
}
```

Después de compilar y ejecutar el programa anterior, usted recibirá la siguiente salida de la biblioteca Eclipselink sobre el panel de la consola de eclipse IDE.

```
employee ID = 1201
employee NAME = Gopal
employee SALARY = 46000.0
employee DESIGNATION = Technical Manager
```

Eliminar empleados

Para borrar los registros de un empleado, primero encontraremos los registros existentes y luego borrarlo. Aquí EntityTransaction desempeña un papel importante.

La clase denominada **DeleteEmployee.java** como sigue:

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.Employee;

public class DeleteEmployee
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
```

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

      Employee employee=entitymanager.find( Employee.class, 1201 );
      entitymanager.remove( employee );
      entitymanager.getTransaction( ).commit( );
      entitymanager.close( );
      emfactory.close( );
   }
}
```

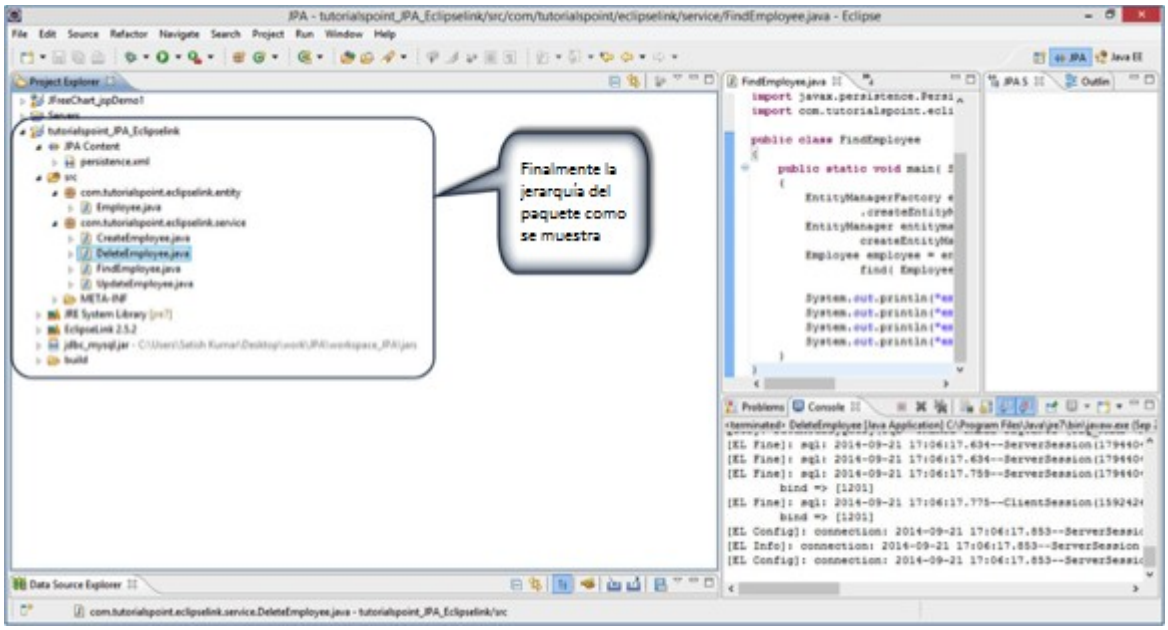
Después de la compilación y ejecución del programa anterior usted recibirá notificaciones de biblioteca Eclipselink sobre el panel de la consola de eclipse IDE.

Para el resultado, abra el MySQL workbench y escriba las siguientes consultas.

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

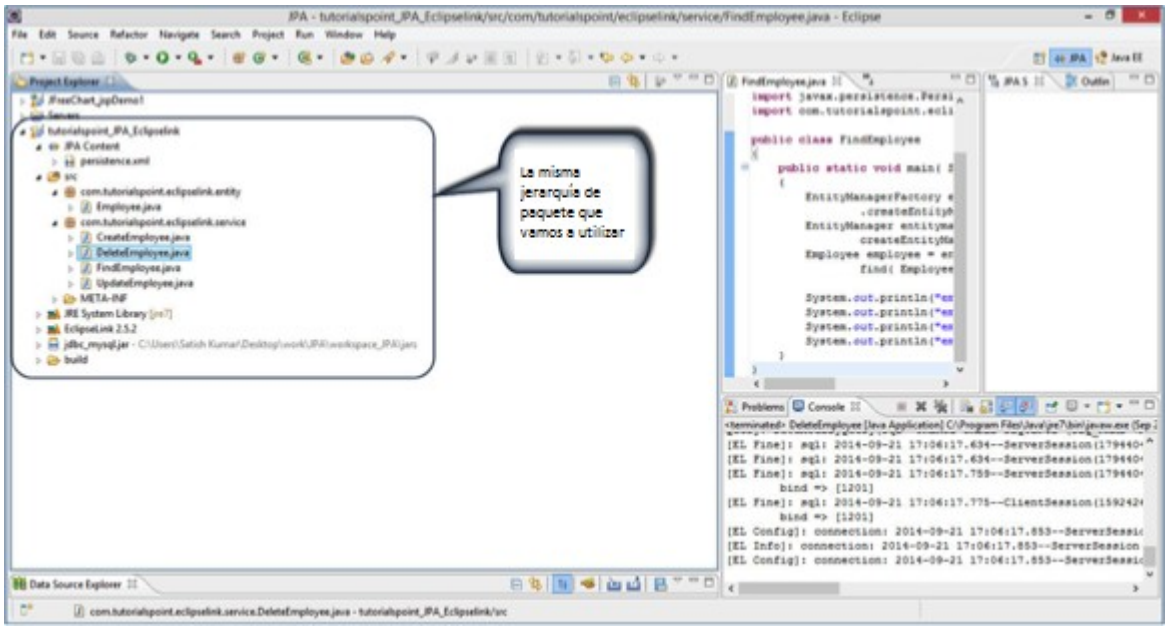
La base de datos efectuado llamado **empleado** tendrá registros nulos.

Después de la terminación de todos los módulos en este ejemplo, la jerarquía de archivos y paquete Mira como sigue:



JPA - JPQL

Este capítulo describe sobre JPQL y cómo funciona con unidades de persistencia. En este capítulo, los ejemplos dados sigan la misma jerarquía del paquete, que se utilizó en el capítulo anterior.



Lenguaje de consulta de persistencia de Java

JPQL está parado para el lenguaje de consulta de persistencia de Java. Se utiliza para crear consultas contra entidades para almacenar en una base de datos relacional. JPQL está desarrollado en base a la sintaxis SQL. Pero no afectará directamente a la base de datos.

JPQL puede recuperar datos mediante la cláusula SELECT, puede hacer a granel actualizaciones con cláusula UPDATE y DELETE cláusula.

Estructura de consulta

Sintaxis JPQL es muy similar a la sintaxis de SQL. Tener SQL como sintaxis es una ventaja porque SQL es simple y siendo ampliamente utilizado. SQL trabaja directamente contra la base de datos relacional tablas, registros y campos, mientras que JPQL trabaja con Java clases e instancias.

Por ejemplo, una consulta JPQL puede recuperar una entidad objeto en lugar de campo conjunto de resultados de una base de datos, al igual que con SQL. El JPQL consulta estructura como sigue.

```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

La estructura de JPQL borrar y consultas de actualización son las siguientes.

```
DELETE FROM ... [WHERE ...]
```

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

Funciones escalares y agregadas

Las funciones escalares devuelven valores resultantes basados en los valores de entrada. Funciones de agregado de devuelvan los valores resultantes mediante el cálculo de los valores de entrada.

Utilizaremos el mismo ejemplo de gestión de empleados como en el capítulo anterior. Aquí nos dirigiremos a través de las clases de servicio utilizando funciones escalares y agregadas de JPQL.

Nos dejaron asumen que la tabla **jpadb.employee** contiene los siguientes registros.

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

Crear una clase denominada **ScalarandAggregateFunctions.java** bajo **com.tutorialspoint.eclipselink.service** paquete de la siguiente manera.

```
package com.tutorialspoint.eclipselink.service;

import java.util.List;

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

public class ScalarandAggregateFunctions
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager();

      //Scalar function
      Query query = entitymanager.createQuery("Select UPPER(e.ename) from
Employee e");
      List<String> list=query.getResultList();

      for(String e:list)
      {
         System.out.println("Employee NAME :"+e);
      }

      //Aggregate function
      Query query1 = entitymanager.createQuery("Select MAX(e.salary) from
Employee e");
      Double result = (Double) query1.getSingleResult();
      System.out.println("Max Employee Salary :" + result);
   }
}
```

Después de la compilación y ejecución del programa anterior obtendrá la siguiente salida en el panel de consola del IDE de Eclipse.

```
Employee NAME :GOPAL
Employee NAME :MANISHA
Employee NAME :MASTHANVALI
```

```
Employee NAME :SATISH
Employee NAME :KRISHNA
Employee NAME :KIRAN
Max Employee Salary :40000.0
```

Between, And, Like palabras clave

Between (Entre), **And (Y)**, y **Like (Como)** son las palabras clave principales de JPQL. Estas palabras clave se utilizan después de **Donde cláusula** en una consulta.

Crear una clase denominada **BetweenAndLikeFunctions.java** bajo **com.tutorialspoint.eclipselink.service** paquete de la siguiente manera:

```
package com.tutorialspoint.eclipselink.service;

import java.util.List;

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

import com.tutorialspoint.eclipselink.entity.Employee;

public class BetweenAndLikeFunctions
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager();

      //Between
      Query query = entitymanager.createQuery( "Select e " + "from Employee e "
+ "where e.salary " + "Between 30000 and 40000" );
      List<Employee> list=(List<Employee>)query.getResultList( );

      for( Employee e:list )
      {
         System.out.print("Employee ID :" + e.getEid( ));
         System.out.println("\t Employee salary :" + e.getSalary( ));
      }

      //Like
      Query query1 = entitymanager.createQuery("Select e " + "from Employee e "
+ "where e.ename LIKE 'M%'");
      List<Employee> list1=(List<Employee>)query1.getResultList( );

      for( Employee e:list1 )
      {
         System.out.print("Employee ID :"+e.getEid( ));
         System.out.println("\t Employee name :"+e.getEname( ));
      }
   }
}
```

Después de compilar y ejecutar el programa anterior, obtendrá la siguiente salida en el panel de la consola de Eclipse IDE.

```
Employee ID :1201        Employee salary :40000.0
Employee ID :1202        Employee salary :40000.0
Employee ID :1203        Employee salary :40000.0
Employee ID :1204        Employee salary :30000.0
Employee ID :1205        Employee salary :30000.0
Employee ID :1206        Employee salary :35000.0

Employee ID :1202        Employee name :Manisha
Employee ID :1203        Employee name :Masthanvali
```

Ordenar

Para ordenar los registros en JPQL, utilizamos la cláusula ORDER BY. El uso de esta cláusula es igual que en SQL, pero se trata de entidades. El ejemplo siguiente muestra cómo utilizar la cláusula ORDER BY.

Crear una clase **Ordering.java** bajo **com.tutorialspoint.eclipselink.service** paquete de la siguiente manera:

```
package com.tutorialspoint.eclipselink.service;

import java.util.List;

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

import com.tutorialspoint.eclipselink.entity.Employee;

public class Ordering
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager();

      //Between
      Query query = entitymanager.createQuery( "Select e " + "from Employee e "
+ "ORDER BY e.ename ASC" );
      List<Employee> list=(List<Employee>)query.getResultList( );

      for( Employee e:list )
      {
         System.out.print("Employee ID :" + e.getEid( ));
         System.out.println("\t Employee Name :" + e.getEname( ));
      }
   }
}
```

compilar y ejecutar el programa anterior producirá la siguiente salida en el panel de la consola de Eclipse IDE.

```
Employee ID :1201        Employee Name :Gopal
Employee ID :1206        Employee Name :Kiran
Employee ID :1205        Employee Name :Krishna
Employee ID :1202        Employee Name :Manisha
Employee ID :1203        Employee Name :Masthanvali
```

```
Employee ID :1204        Employee Name :Satish
```

Llamado consultas

Una anotación @NamedQuery se define como una consulta con una cadena de consulta predefinida que es inmutable. En contraste con consultas dinámicas, llamadas consultas pueden mejorar la organización del código al separar las cadenas de consulta JPQL de POJO. También pasa los parámetros de consulta en lugar de incrustar los literales dinámicamente en la cadena de consulta y por lo tanto produce más eficientes consultas.

En primer lugar, añadir anotación @NamedQuery a la clase de entidad empleado llamada **Employee.java** bajo **com.tutorialspoint.eclipselink.entity** paquete de la siguiente manera:

```
package com.tutorialspoint.eclipselink.entity;

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

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

public class Employee
{
   @Id
   @GeneratedValue(strategy= GenerationType.AUTO)
   private int eid;
   private String ename;
   private double salary;
   private String deg;

   public Employee(int eid, String ename, double salary, String deg)
   {
      super( );
      this.eid = eid;
      this.ename = ename;
      this.salary = salary;
      this.deg = deg;
   }

   public Employee( )
   {
      super();
   }

   public int getEid( )
   {
      return eid;
   }
   public void setEid(int eid)
   {
      this.eid = eid;
   }
```

```
   public String getEname( )
   {
      return ename;
   }
   public void setEname(String ename)
   {
      this.ename = ename;
   }

   public double getSalary( )
   {
      return salary;
   }
   public void setSalary(double salary)
   {
      this.salary = salary;
   }

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

Crear una clase denominada **NamedQueries.java** bajo **com.tutorialspoint.eclipselink.service** paquete de la siguiente manera:

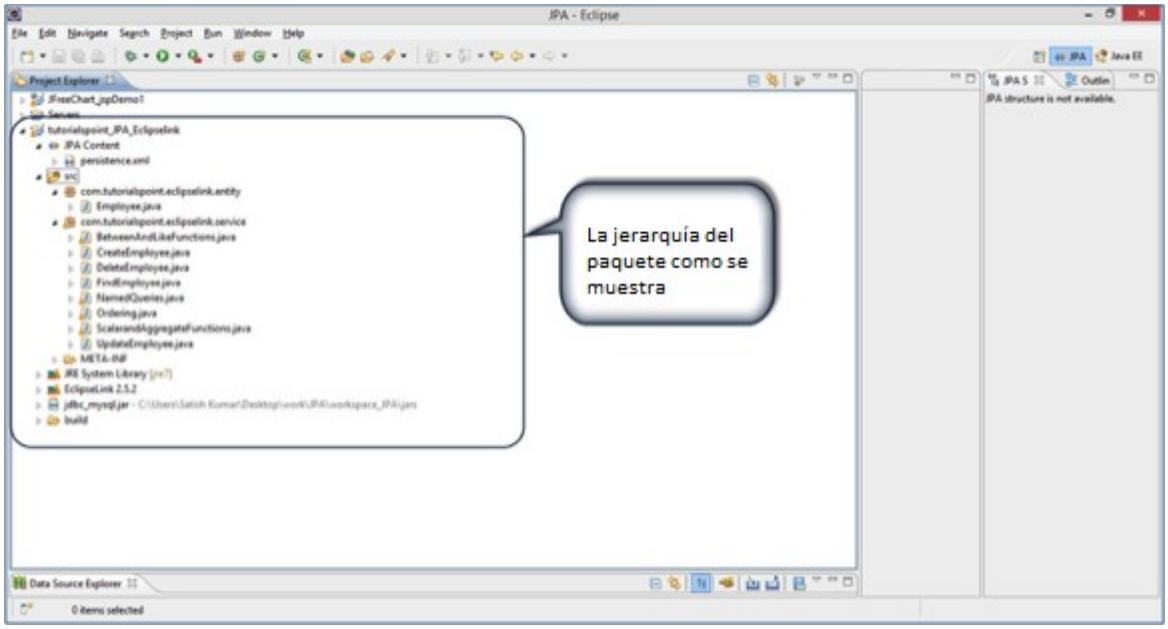
```
package com.tutorialspoint.eclipselink.service;

import java.util.List;

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

import com.tutorialspoint.eclipselink.entity.Employee;

public class NamedQueries
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager();

      Query query = entitymanager.createNamedQuery("find employee by id");
      query.setParameter("id", 1204);

      List<Employee> list = query.getResultList( );
```

```
      for( Employee e:list )
      {
         System.out.print("Employee ID :" + e.getEid( ));
         System.out.println("\t Employee Name :" + e.getEname( ));
      }
   }
}
```

Después de compilar y ejecutar el programa anterior obtendrá la siguiente salida en el panel de la consola de Eclipse IDE.

```
Employee ID :1204        Employee Name :Satish
```

Después de agregar todas las clases sobre la jerarquía del paquete Mira como sigue:



Ansiosos y perezoso a buscar

El concepto más importante de JPA es hacer una copia duplicada de la base de datos en la memoria caché. Mientras se tramita con una base de datos, la JPA primero crea un duplicado de los datos y sólo cuando se ha comprometido con una entidad Director, los cambios se realizan en la base de datos.

Allí es dos maneras de obtener los registros de la base de datos.

Fetch ansioso

En buscar ansiosos, relacionados con objetos secundarios se cargan automáticamente al ir a buscar un registro concreto.

Lazy fetch

En recuperar el perezoso, objetos relacionados no se cargan automáticamente a menos que usted solicite específicamente para ellos. En primer lugar, comprueba la disponibilidad de objetos

relacionados y notifica. Más tarde, si se llama a cualquiera del método getter de esa entidad, luego recupera todos los registros.

Fetch perezoso es posible cuando se intenta buscar los registros por primera vez. De esa manera, una copia del registro todo ya está almacenada en la memoria caché. Rendimiento, es preferible fetch perezoso.

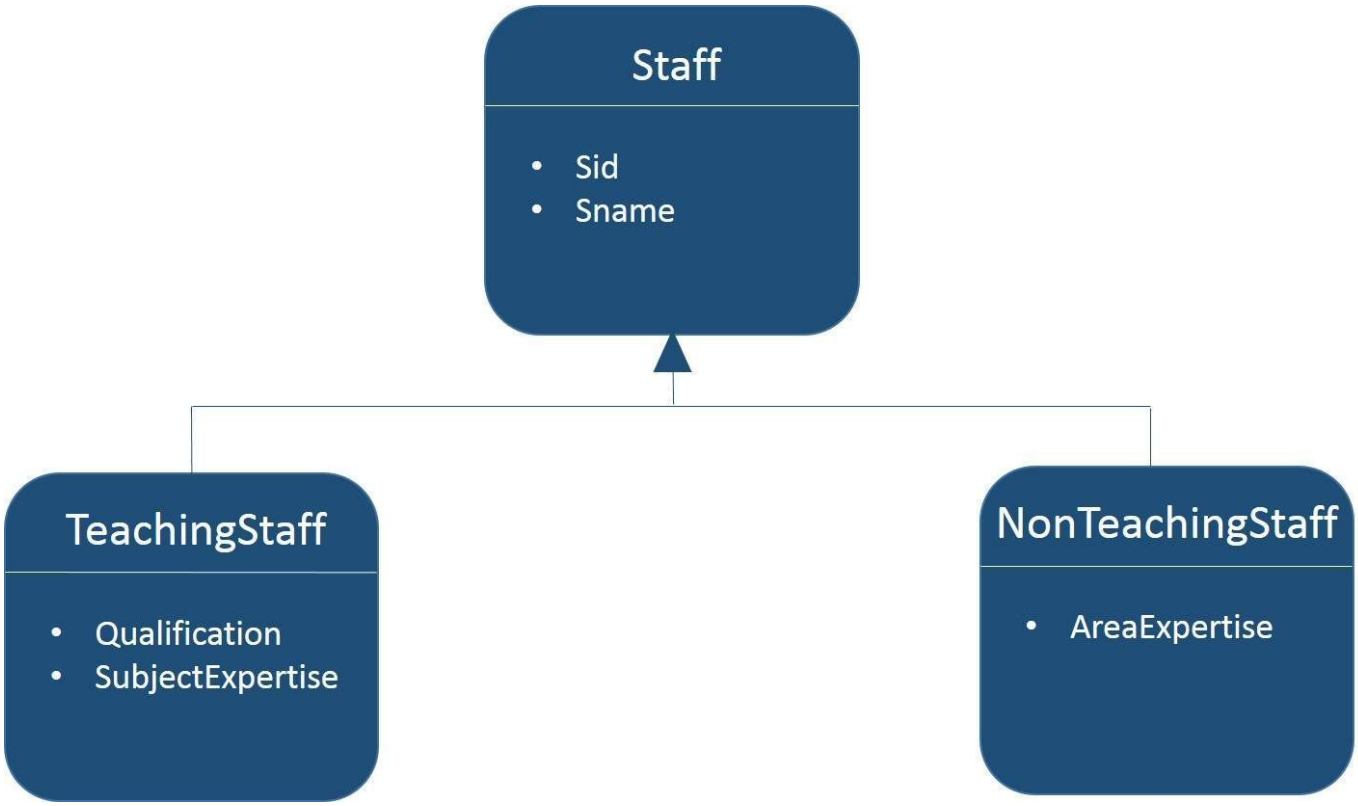
JPA - Asignaciones de Avanzada

JPA es una biblioteca que se lanza con las especificaciones de Java. Por lo tanto, soporta los conceptos orientados a objetos para la persistencia de la entidad. Hasta ahora, hemos terminado con los conceptos básicos de mapeo relacional de objetos. Este capítulo le lleva a través de las asignaciones de avanzada entre objetos y entidades relacionales.

Estrategias de herencia

La herencia es el concepto de la base de cualquier lenguaje orientado a objetos, por lo tanto, podemos utilizar las relaciones de herencia o estrategias entre las entidades. JPA admiten tres tipos de estrategias de herencia: SINGLE_TABLE, JOINED_TABLE y TABLE_PER_CONCRETE_CLASS.

Nos permite considerar un ejemplo. El siguiente diagrama muestra tres clases, es decir personal, TeachingStaff y NonTeachingStaff y sus relaciones.



En el diagrama anterior, el personal es una entidad, mientras que TeachingStaff y NonTeachingStaff son las entidades secundarias del personal. Vamos a utilizar el ejemplo anterior para demostrar los tres tres estrategias de la herencia de.

Estrategia de mesa única

Mesa única estrategia toma todos los campos de las clases (clases tanto super y sub) y mapa abajo en una sola tabla conocida como estrategia SINGLE_TABLE. Aquí el valor discriminador desempeña un papel clave en diferenciar los valores de las tres entidades en una tabla.

Nos permite considerar el ejemplo anterior. TeachingStaff y NonTeachingStaff son las subclases de personal. Según el concepto de herencia, una sub clase hereda las propiedades de su super-clase. Por lo tanto sid y sname son los campos que pertenecen a ambos TeachingStaff y NonTeachingStaff. Crear un proyecto JPA. Todos los módulos de este proyecto son los siguientes:

Creación de entidades

Crear un paquete denominado **'com.tutorialspoint.eclipselink.entity'** bajo **'src'** paquete. Crear una nueva clase de java **Staff.java** bajo el nombre dado paquete. La clase de entidad personal se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import java.io.Serializable;

import javax.persistence.DiscriminatorColumn;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table
@Inheritance( strategy = InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn( name="type" )

public class Staff implements Serializable
{
   @Id
   @GeneratedValue( strategy = GenerationType.AUTO )
   private int sid;
   private String sname;

   public Staff( int sid, String sname )
   {
      super( );
      this.sid = sid;
      this.sname = sname;
   }

   public Staff( )
   {
      super( );
   }
```

```
   public int getSid( )
   {
      return sid;
   }

   public void setSid( int sid )
   {
      this.sid = sid;
   }

   public String getSname( )
   {
      return sname;
   }

   public void setSname( String sname )
   {
      this.sname = sname;
   }
}
```

En el código anterior **@DescriminatorColumn** specifies the field name **(tipo)** y sus valores muestran el restante (Teaching and NonTeachingStaff) campos.

Crear una subclase (clase) a la clase de personal nombrado **TeachingStaff.java** bajo el **com.tutorialspoint.eclipselink.entity** paquete. La clase de entidad TeachingStaff se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue( value="TS" )

public class TeachingStaff extends Staff
{
   private String qualification;
   private String subjectexpertise;

   public TeachingStaff( int sid, String sname, String qualification,String
subjectexpertise )
   {
      super( sid, sname );
      this.qualification = qualification;
      this.subjectexpertise = subjectexpertise;
   }

   public TeachingStaff( )
   {
      super( );
   }

   public String getQualification( )
   {
      return qualification;
   }
```

```
   public void setQualification( String qualification )
   {
      this.qualification = qualification;
   }

   public String getSubjectexpertise( )
   {
      return subjectexpertise;
   }

   public void setSubjectexpertise( String subjectexpertise )
   {
      this.subjectexpertise = subjectexpertise;
   }
}
```

Crear una subclase (clase) a la clase de personal nombrado **NonTeachingStaff.java** bajo el **com.tutorialspoint.eclipselink.entity** paquete. La clase de entidad NonTeachingStaff se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue( value = "NS" )

public class NonTeachingStaff extends Staff
{
   private String areaexpertise;

   public NonTeachingStaff( int sid, String sname, String areaexpertise )
   {
      super( sid, sname );
      this.areaexpertise = areaexpertise;
   }

   public NonTeachingStaff( )
   {
      super( );
   }

   public String getAreaexpertise( )
   {
      return areaexpertise;
   }

   public void setAreaexpertise( String areaexpertise )
   {
      this.areaexpertise = areaexpertise;
   }
}
```

Persistence.xml

Persistence.xml contiene la información de configuración de base de datos y la información de registro de clases de entidad. El archivo xml se muestra como sigue:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="Eclipselink_JPA" transaction-type="RESOURCE_LOCAL">

      <class>com.tutorialspoint.eclipselink.entity.Staff</class>
      <class>com.tutorialspoint.eclipselink.entity.NonTeachingStaff</class>
      <class>com.tutorialspoint.eclipselink.entity.TeachingStaff</class>

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

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

Clase de servicio

Clases de servicio son la parte de la implementación del componente empresarial. Crear un paquete bajo '**src**' paquete denominado '**com.tutorialspoint.eclipselink.service**'.

Crear una clase denominada **SaveClient.java** bajo el paquete dado para almacenar campos clase personal, TeachingStaff y NonTeachingStaff. La clase SaveClient se muestra como sigue:

```
package com.tutorialspoint.eclipselink.service;

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

import com.tutorialspoint.eclipselink.entity.NonTeachingStaff;
import com.tutorialspoint.eclipselink.entity.TeachingStaff;

public class SaveClient
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );

      //Teaching staff entity
      TeachingStaff ts1=new TeachingStaff(1,"Gopal","MSc MEd","Maths");
      TeachingStaff ts2=new TeachingStaff(2, "Manisha", "BSc BEd", "English");

      //Non-Teaching Staff entity
      NonTeachingStaff nts1=new NonTeachingStaff(3, "Satish", "Accounts");
      NonTeachingStaff nts2=new NonTeachingStaff(4, "Krishna", "Office Admin");

      //storing all entities
```

```
      entitymanager.persist(ts1);
      entitymanager.persist(ts2);
      entitymanager.persist(nts1);
      entitymanager.persist(nts2);

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

Después de compilar y ejecutar el programa anterior usted recibirá notificaciones en el panel de la consola de Eclipse IDE. Comprobar MySQL workbench para la salida. La salida en formato tabular se muestra como sigue:

Sid	Type	Sname	Areaexpertise	Qualification	Subjectexpertise
1	TS	Gopal		MSC MED	Maths
2	TS	Manisha		BSC BED	English
3	NS	Satish	Accounts		
4	NS	Krishna	Office Admin		

Finalmente usted conseguirá una sola tabla que contiene el campo de todas las clases de tres con una columna de discriminador denominada **tipo** (campo).

Unificadas mesa estrategia

Se unió a mesa estrategia es compartir la columna que se hace referencia que contiene valores únicos para unirse a la mesa y hacer transacciones fáciles. Consideremos el ejemplo de arriba.

Crear un proyecto JPA. Abajo se muestran todos los módulos de proyecto.

Creación de entidades

Crear un paquete denominado '**com.tutorialspoint.eclipselink.entity'** bajo '**src'** paquete. Crear una nueva clase de java llamada **Staff.java** bajo dado paquete. La clase de entidad personal se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table
@Inheritance( strategy = InheritanceType.JOINED )

public class Staff implements Serializable
{
   @Id
```

```
   @GeneratedValue( strategy = GenerationType.AUTO )

   private int sid;
   private String sname;

   public Staff( int sid, String sname )
   {
      super( );
      this.sid = sid;
      this.sname = sname;
   }
   public Staff( )
   {
      super( );
   }
   public int getSid( )
   {
      return sid;
   }
   public void setSid( int sid )
   {
      this.sid = sid;
   }
   public String getSname( )
   {
      return sname;
   }
   public void setSname( String sname )
   {
      this.sname = sname;
   }
}
```

Crear una subclase (clase) a la clase de personal nombrado **TeachingStaff.java** bajo el **com.tutorialspoint.eclipselink.entity** paquete. La clase de entidad TeachingStaff se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@PrimaryKeyJoinColumn(referencedColumnName="sid")

public class TeachingStaff extends Staff
{
   private String qualification;
   private String subjectexpertise;

   public TeachingStaff( int sid, String sname, String qualification,String
subjectexpertise )
   {
      super( sid, sname );
      this.qualification = qualification;
      this.subjectexpertise = subjectexpertise;
   }

   public TeachingStaff( )
   {
```

```
      super( );
   }

   public String getQualification( )
   {
      return qualification;
   }

   public void setQualification( String qualification )
   {
      this.qualification = qualification;
   }

   public String getSubjectexpertise( )
   {
      return subjectexpertise;
   }

   public void setSubjectexpertise( String subjectexpertise )
   {
      this.subjectexpertise = subjectexpertise;
   }
}
```

Crear una subclase (clase) a la clase de personal nombrado **NonTeachingStaff.java** bajo el **com.tutorialspoint.eclipselink.entity** paquete. La clase de entidad NonTeachingStaff se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@PrimaryKeyJoinColumn(referencedColumnName="sid")

public class NonTeachingStaff extends Staff
{
   private String areaexpertise;

   public NonTeachingStaff( int sid, String sname, String areaexpertise )
   {
      super( sid, sname );
      this.areaexpertise = areaexpertise;
   }

   public NonTeachingStaff( )
   {
      super( );
   }

   public String getAreaexpertise( )
   {
      return areaexpertise;
   }

   public void setAreaexpertise( String areaexpertise )
   {
      this.areaexpertise = areaexpertise;
   }
```

```
}
```

Persistence.xml

Persistence.xml archivo contiene la información de configuración de la base de datos y la información de registro de clases de entidad. El archivo xml se muestra como sigue:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="Eclipselink_JPA" transaction-type="RESOURCE_LOCAL">

   <class>com.tutorialspoint.eclipselink.entity.Staff</class>
   <class>com.tutorialspoint.eclipselink.entity.NonTeachingStaff</class>
   <class>com.tutorialspoint.eclipselink.entity.TeachingStaff</class>

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

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

Clase de servicio

Clases de servicio son la parte de la implementación del componente empresarial. Crear un paquete debajo del paquete **'src'** llamado **'com.tutorialspoint.eclipselink.service'**.

Crear una clase denominada **SaveClient.java** bajo el paquete dado para almacenar campos de clase personal, TeachingStaff y NonTeachingStaff. A continuación se muestra la clase SaveClient como sigue:

```
package com.tutorialspoint.eclipselink.service;

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

import com.tutorialspoint.eclipselink.entity.NonTeachingStaff;
import com.tutorialspoint.eclipselink.entity.TeachingStaff;

public class SaveClient
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );
```

```
        //Teaching staff entity
        TeachingStaff ts1=new TeachingStaff(1,"Gopal","MSc MEd","Maths");
        TeachingStaff ts2=new TeachingStaff(2, "Manisha", "BSc BEd", "English");

        //Non-Teaching Staff entity
        NonTeachingStaff nts1=new NonTeachingStaff(3, "Satish", "Accounts");
        NonTeachingStaff nts2=new NonTeachingStaff(4, "Krishna", "Office Admin");

        //storing all entities
        entitymanager.persist(ts1);
        entitymanager.persist(ts2);
        entitymanager.persist(nts1);
        entitymanager.persist(nts2);

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

Después de compilar y ejecutar el programa anterior usted recibirá notificaciones en el panel de la consola de Eclipse IDE. Para la salida, verifique MySQL workbench.

Aquí se crean las tres tablas y el resultado de la tabla **personal** se muestra en un formato tabular.

Sid	Dtype	Sname
1	TeachingStaff	Gopal
2	TeachingStaff	Manisha
3	NonTeachingStaff	Satish
4	NonTeachingStaff	Krishna

El resultado de la tabla **TeachingStaff** se muestra como sigue:

Sid	Qualification	Subjectexpertise
1	MSC MED	Maths
2	BSC BED	English

En la tabla anterior sid es la clave foránea (tabla de referencia campo forma personal) el resultado de **NonTeachingStaff** mesa se visualiza de la siguiente manera:

Sid	Areaexpertise
3	Accounts
4	Office Admin

Finalmente, las tres tablas se crean utilizando sus respectivos campos y el campo de SID es compartido por todas las tres tablas. En el cuadro de personal, SID es la clave principal. En las restantes dos tablas (TeachingStaff y NonTeachingStaff), SID es la clave foránea.

Mesa por estrategia de clase

Tabla por estrategia de clase es crear una tabla para cada entidad sub. Se creará la tabla personal, pero contendrá valores null. Los valores del campo de la tabla del personal deben ser compartidos por las tablas de los TeachingStaff y NonTeachingStaff.

Vamos a considerar el mismo ejemplo que el anterior.

Creación de entidades

Crear un paquete denominado '**com.tutorialspoint.eclipselink.entity'** bajo '**src'** paquete. Crear una nueva clase de java llamada **Staff.java** bajo dado paquete. La clase de entidad personal se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table
@Inheritance( strategy = InheritanceType.TABLE_PER_CLASS )

public class Staff implements Serializable
{
   @Id
   @GeneratedValue( strategy = GenerationType.AUTO )

   private int sid;
   private String sname;

   public Staff( int sid, String sname )
   {
      super( );
      this.sid = sid;
      this.sname = sname;
   }
   public Staff( )
   {
      super( );
   }
   public int getSid( )
   {
      return sid;
   }
   public void setSid( int sid )
   {
      this.sid = sid;
   }
   public String getSname( )
   {
      return sname;
   }
   public void setSname( String sname )
   {
      this.sname = sname;
   }
}
```

Crear una subclase (clase) a la clase de personal nombrado **TeachingStaff.java** bajo el **com.tutorialspoint.eclipselink.entity** paquete. La clase de entidad TeachingStaff se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
public class TeachingStaff extends Staff
{
   private String qualification;
   private String subjectexpertise;

   public TeachingStaff( int sid, String sname, String qualification,String
subjectexpertise )
   {
      super( sid, sname );
      this.qualification = qualification;
      this.subjectexpertise = subjectexpertise;
   }

   public TeachingStaff( )
   {
      super( );
   }

   public String getQualification( )
   {
      return qualification;
   }
   public void setQualification( String qualification )
   {
      this.qualification = qualification;
   }

   public String getSubjectexpertise( )
   {
      return subjectexpertise;
   }

   public void setSubjectexpertise( String subjectexpertise )
   {
      this.subjectexpertise = subjectexpertise;
   }
}
```

Crear una subclase (clase) a la clase de personal nombra **NonTeachingStaff.java** bajo el **com.tutorialspoint.eclipselink.entity** paquete. La clase de entidad NonTeachingStaff se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
public class NonTeachingStaff extends Staff
```

```
{
   private String areaexpertise;

   public NonTeachingStaff( int sid, String sname, String areaexpertise )
   {
      super( sid, sname );
      this.areaexpertise = areaexpertise;
   }

   public NonTeachingStaff( )
   {
      super( );
   }

   public String getAreaexpertise( )
   {
      return areaexpertise;
   }

   public void setAreaexpertise( String areaexpertise )
   {
      this.areaexpertise = areaexpertise;
   }
}
```

Persistence.xml

Persistence.xml archivo contiene la información de configuración de información de bases de datos y el registro de las clases de entidad. El archivo xml se muestra como sigue:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="Eclipselink_JPA" transaction-type="RESOURCE_LOCAL">

   <class>com.tutorialspoint.eclipselink.entity.Staff</class>
   <class>com.tutorialspoint.eclipselink.entity.NonTeachingStaff</class>
   <class>com.tutorialspoint.eclipselink.entity.TeachingStaff</class>

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

Clase de servicio

Clases de servicio son la parte de la implementación del componente empresarial. Crear un paquete bajo ‘**src**’ paquete denominado ‘**com.tutorialspoint.eclipselink.service**’.

Crear una clase denominada **SaveClient.java** bajo el paquete dado para almacenar campos clase personal, TeachingStaff y NonTeachingStaff. La clase SaveClient se muestra como sigue:

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.NonTeachingStaff;
import com.tutorialspoint.eclipselink.entity.TeachingStaff;

public class SaveClient
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );

      //Teaching staff entity
      TeachingStaff ts1=new TeachingStaff(1,"Gopal","MSc MEd","Maths");
      TeachingStaff ts2=new TeachingStaff(2, "Manisha", "BSc BEd", "English");

      //Non-Teaching Staff entity
      NonTeachingStaff nts1=new NonTeachingStaff(3, "Satish", "Accounts");
      NonTeachingStaff nts2=new NonTeachingStaff(4, "Krishna", "Office Admin");

      //storing all entities
      entitymanager.persist(ts1);
      entitymanager.persist(ts2);
      entitymanager.persist(nts1);
      entitymanager.persist(nts2);

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

Después de compilar y ejecutar el programa anterior, usted recibirá notificaciones en el panel de la consola de Eclipse IDE. Para la salida, compruebe MySQL workbench.

Aquí se crean las tres tablas y el **Personal** tabla contiene registros nulos.

El resultado de **TeachingStaff** se visualiza de la siguiente manera:

Sid	Qualification	Sname	Subjectexpertise
1	MSC MED	Gopal	Maths
2	BSC BED	Manisha	English

La tabla TeachingStaff contiene campos personal y de las entidades TeachingStaff.

El resultado de **NonTeachingStaff** se visualiza de la siguiente manera:

Sid	Areaexpertise	Sname
3	Accounts	Satish
4	Office Admin	Krishna

La tabla NonTeachingStaff contiene campos de los funcionarios y entidades NonTeachingStaff.

JPA - Relaciones de la Entidad

Este capítulo le lleva a través de las relaciones entre las entidades. Generalmente las relaciones son más efectivas entre las tablas en la base de datos. Aquí las clases de entidad son tratadas como tablas relacionales (concepto de JPA), por lo tanto, las relaciones entre las clases de entidad son los siguientes:

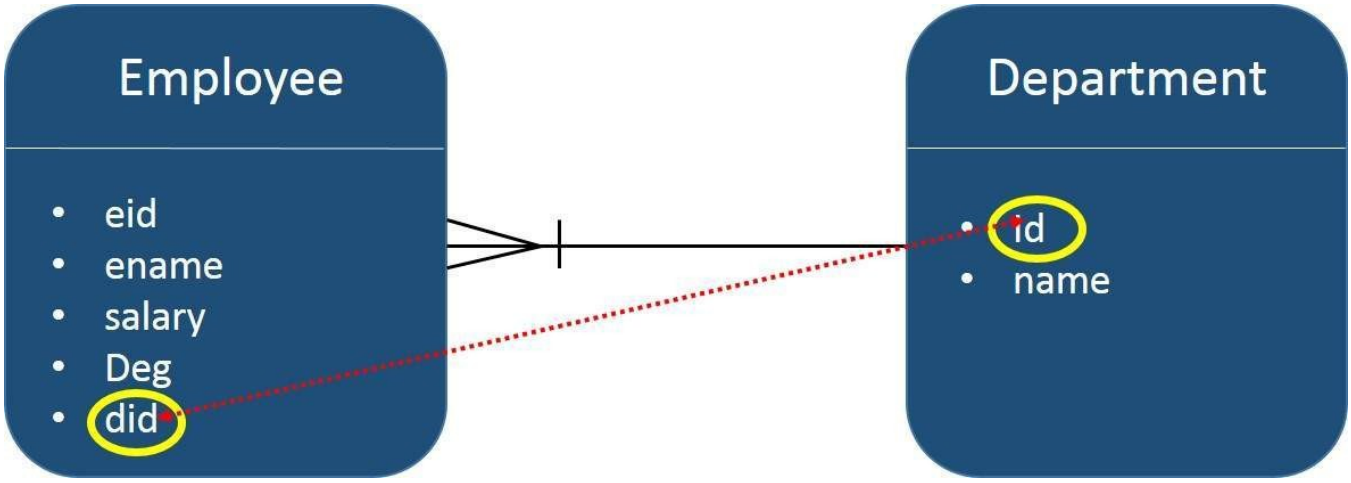
- @ManyToOne Relation
- @OneToMany Relation
- @OneToOne Relation
- @ManyToMany Relation

@ManyToOne Relation

Existe relación Many-To-One entre las entidades donde se hace referencia a una entidad (columna o conjunto de columnas) con valores únicos que contienen de otra entidad (columna o conjunto de columnas). En bases de datos relacionales, estas relaciones se aplican mediante el uso de clave primaria clave externa entre las tablas.

Nos permite considerar un ejemplo de una relación entre entidades empleado y departamento. De manera unidireccional, es decir, de empleado al Departamento, Many-To-One relación es aplicable. Eso significa que cada registro de empleado contiene un id de departamento, que debe ser una clave principal en la tabla Department. Aquí en la tabla Employee, Departamento id es la clave foránea.

El siguiente diagrama muestra el Many-To-One relación entre las dos tablas.



Crear un proyecto JPA en eclipse que IDE llamado **JPA_Eclipselink_MTO**. Todos los módulos de este proyecto se discuten más abajo.

Creación de Entidades

Siga el anterior dado diagrama para la creación de entidades. Crear un paquete denominado '**com.tutorialspoin.eclipselink.entity'** bajo '**src'** paquete. Crear una clase denominada **Department.java** bajo dado paquete. La clase entidad departamento se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

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

@Entity
public class Department
{
   @Id
   @GeneratedValue( strategy=GenerationType.AUTO )
   private int id;
   private String name;

   public int getId()
   {
      return id;
   }

   public void setId(int id)
   {
      this.id = id;
   }

   public String getName( )
   {
      return name;
   }

   public void setName( String deptName )
   {
      this.name = deptName;
   }
}
```

Crear la segunda entidad en esta relación - clase de entidad empleado llamado **Employee.java** bajo '**com.tutorialspoint.eclipselink.entity'** paquete. La clase de entidad empleado se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Employee
{
   @Id
   @GeneratedValue( strategy= GenerationType.AUTO )
```

```
private int eid;
private String ename;
private double salary;
private String deg;

@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;
}

public Employee( )
{
   super();
}

public int getEid( )
{
   return eid;
}
public void setEid(int eid)
{
   this.eid = eid;
}

public String getEname( )
{
   return ename;
}
public void setEname(String ename)
{
   this.ename = ename;
}

public double getSalary( )
{
   return salary;
}
public void setSalary(double salary)
{
   this.salary = salary;
}

public String getDeg( )
{
   return deg;
}
public void setDeg(String deg)
{
   this.deg = deg;
}

public Department getDepartment()
{
   return department;
```

```
   }

   public void setDepartment(Department department)
   {
      this.department = department;
   }
}
```

Persistence.xml

Persistence.xml archivo es necesario para configurar la base de datos y el registro de las clases de entidad.

Persitence.xml será creado por el eclipse IDE durante la creación de un proyecto de JPA. Los detalles de configuración son las especificaciones del usuario. El archivo persistence.xml se muestra como sigue:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

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

      <class>com.tutorialspoint.eclipselink.entity.Employee</class>
      <class>com.tutorialspoint.eclipselink.entity.Department</class>

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

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

Clases de servicio

Este módulo contiene las clases de servicio, que implementa la parte relacional con la inicialización de atributo. Crear un paquete bajo ‘**src’** paquete denominado ‘**com.tutorialspoint.eclipselink.service’**. La clase DAO **ManyToOne.java** se crea bajo dado paquete. La clase DAO se muestra como sigue:

```
package com.tutorialspointeclipselink.service;

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

import com.tutorialspoint.eclipselink.entity.Department;
```

```
import com.tutorialspoint.eclipselink.entity.Employee;

public class ManyToOne
{
   public static void main( String[ ] args )
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );

      //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.setSalary(45000.0);
      employee1.setDeg("Technical Writer");
      employee1.setDepartment(department);

      //Create Employee2 Entity
      Employee employee2 = new Employee();
      employee2.setEname("Krishna");
      employee2.setSalary(45000.0);
      employee2.setDeg("Technical Writer");
      employee2.setDepartment(department);

      //Create Employee3 Entity
      Employee employee3 = new Employee();
      employee3.setEname("Masthanvali");
      employee3.setSalary(50000.0);
      employee3.setDeg("Technical Writer");
      employee3.setDepartment(department);

      //Store Employees
      entitymanager.persist(employee1);
      entitymanager.persist(employee2);
      entitymanager.persist(employee3);

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

Después de compilar y ejecutar el programa anterior, usted recibirá notificaciones en el panel de la consola de Eclipse IDE. Para la salida, verifique MySQL workbench. En este ejemplo, se crean dos tablas.

Pasar la siguiente consulta en interfaz MySQL y el resultado de **Departamento** mesa se mostrará como sigue:

```
Select * from department
```

ID	Nombre
101	Desarrollo

Pase a la siguiente consulta en interfaz MySQL y se visualizará el resultado de la tabla **Empleado** de la siguiente manera.

```
Select * from employee
```

Eid	Deg	Ename	Salary	Department_Id
102	Technical Writer	Satish	45000	101
103	Technical Writer	Krishna	45000	101
104	Technical Writer	Masthanwali	50000	101

En la tabla anterior Department_Id es la clave foránea (campo de referencia) de la tabla Department.

@OneToMany Relación

En esta relación, cada fila de una entidad se hace referencia a los muchos registros secundarios en otra entidad. Lo importante es que los registros secundarios no pueden tener varios padres. En una relación uno a varios entre la tabla A y B de la tabla, cada fila en la tabla A puede ser vinculado a una o varias filas en la tabla B.

Consideremos el ejemplo de arriba. Supongamos que empleado y mesas del Departamento en el ejemplo anterior se conectan de manera unidireccional inversa, entonces la relación se convierte en uno de varios relación. Crear un proyecto JPA en eclipse IDE llamado **JPA_Eclipselink_OTM**. Todos los módulos de este proyecto se discuten más abajo.

Creación de Entidades

Siga el anterior dado diagrama para la creación de entidades. Crear un paquete denominado '**com.tutorialspoin.eclipselink.entity'** bajo '**src'** paquete. Crear una clase denominada **Department.java** bajo dado paquete. La clase entidad departamento se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Department
{
   @Id
   @GeneratedValue( strategy=GenerationType.AUTO )
   private int id;
   private String name;

   @OneToMany( targetEntity=Employee.class )
   private List employeelist;
```

```
   public int getId()
   {
     return id;
   }

   public void setId(int id)
   {
      this.id = id;
   }

   public String getName( )
   {
      return name;
   }

   public void setName( String deptName )
   {
      this.name = deptName;
   }

   public List getEmployeelist()
   {
      return employeelist;
   }

   public void setEmployeelist(List employeelist)
   {
      this.employeelist = employeelist;
   }
}
```

Crear la segunda entidad en esta relación-clase de entidad empleado, denominada **Employee.java** bajo '**com.tutorialspoint.eclipselink.entity'** paquete. La clase de entidad empleado se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

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

@Entity
public class Employee
{
   @Id
   @GeneratedValue( strategy= GenerationType.AUTO )
   private int eid;
   private String ename;
   private double salary;
   private String deg;

   public Employee(int eid, String ename, double salary, String deg)
   {
      super( );
      this.eid = eid;
      this.ename = ename;
      this.salary = salary;
      this.deg = deg;
   }
```

```
   public Employee( )
   {
      super();
   }

   public int getEid( )
   {
      return eid;
   }
   public void setEid(int eid)
   {
      this.eid = eid;
   }

   public String getEname( )
   {
      return ename;
   }
   public void setEname(String ename)
   {
      this.ename = ename;
   }

   public double getSalary( )
   {
      return salary;
   }
   public void setSalary(double salary)
   {
      this.salary = salary;
   }

   public String getDeg( )
   {
      return deg;
   }
   public void setDeg(String deg)
   {
      this.deg = deg;
   }
}
```

Persistence.xml

The persistence.xml archivo es la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

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

      <class>com.tutorialspoint.eclipselink.entity.Employee</class>
      <class>com.tutorialspoint.eclipselink.entity.Department</class>

      <properties>
```

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

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

Clases de servicio

Este módulo contiene las clases de servicio, que implementa la parte relacional con la inicialización de atributo. Crear un paquete bajo '**src**' paquete denominado '**com.tutorialspoint.eclipselink.service**'. La clase DAO llamada **OneToMany.java** se crea bajo dado paquete. La clase DAO se muestra como sigue:

```
package com.tutorialspointeclipselink.service;

import java.util.List;
import java.util.ArrayList;

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

import com.tutorialspoint.eclipselink.entity.Department;
import com.tutorialspoint.eclipselink.entity.Employee;

public class OneToMany
{
   public static void main(String[] args)
   {
      EntityManagerFactory emfactory = Persistence.
      createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );

      //Create Employee1 Entity
      Employee employee1 = new Employee();
      employee1.setEname("Satish");
      employee1.setSalary(45000.0);
      employee1.setDeg("Technical Writer");

      //Create Employee2 Entity
      Employee employee2 = new Employee();
      employee2.setEname("Krishna");
      employee2.setSalary(45000.0);
      employee2.setDeg("Technical Writer");

      //Create Employee3 Entity
      Employee employee3 = new Employee();
      employee3.setEname("Masthanvali");
      employee3.setSalary(50000.0);
      employee3.setDeg("Technical Writer");
```

```
        //Store Employee
        entitymanager.persist(employee1);
        entitymanager.persist(employee2);
        entitymanager.persist(employee3);

        //Create Employeelist
        List<Employee> emplist = new ArrayList();
        emplist.add(employee1);
        emplist.add(employee2);
        emplist.add(employee3);

        //Create Department Entity
        Department department = new Department();
        department.setName("Development");
        department.setEmployeelist(emplist);

        //Store Department
        entitymanager.persist(department);

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

Después de la compilación y ejecución del programa anterior usted recibirá notificaciones en el panel de la consola de Eclipse IDE. Para la salida de comprobar MySQL workbench de la siguiente manera.

En este proyecto se crean las tres tablas. Pase la siguiente consulta en interfaz MySQL y el resultado de la tabla department_employee se mostrará como sigue:

```
Select * from department_Id;
```

Department_ID	Employee_Eid
254	251
254	252
254	253

En la tabla anterior, **department_id** y **employee_id** son las claves externas (campos de referencia) de departamento y empleado tablas.

Pase la siguiente consulta en interfaz MySQL y el resultado de la tabla department se mostrará en un formato tabular de la siguiente manera.

```
Select * from department;
```

ID	Nombre
254	Desarrollo

Pase a la siguiente consulta en interfaz MySQL y se visualizará el resultado de la tabla employee de la siguiente manera:

```
Select * from employee;
```

Eid	Deg	Ename	Salary

251	Technical Writer	Satish	45000
252	Technical Writer	Krishna	45000
253	Technical Writer	Masthanwali	50000

@OneToOne Relation

En una relación uno-a-uno, un elemento puede vincularse al único otro elemento. Significa que cada fila de una entidad se refiere a una y sólo una fila de otra entidad.

Nos permite considerar el ejemplo anterior. **Empleado** y **Departamento** de manera unidireccional inversa, la relación es relación uno-a-uno. Significa que cada empleado pertenece al Departamento de sola. Crear un proyecto JPA en eclipse IDE llamado **JPA_Eclipselink_OTO**. Todos los módulos de este proyecto se examinan a continuación.

Creación de entidades

Siga el anterior dado diagrama para la creación de entidades. Crear un paquete denominado '**com.tutorialspoin.eclipselink.entity'** bajo '**src'** paquete. Crear una clase denominada **Department.java** bajo dado el paquete. La clase entidad departamento se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

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

@Entity
public class Department
{
    @Id
    @GeneratedValue( strategy=GenerationType.AUTO )
    private int id;
    private String name;

    public int getId()
    {
    return id;
    }

    public void setId(int id)
    {
    this.id = id;
    }

    public String getName( )
    {
    return name;
    }

    public void setName( String deptName )
    {
    this.name = deptName;
    }
}
```

Crear la segunda entidad en esta relación-clase de entidad Employee, denominada **Employee.java** en paquete **'com.tutorialspoint.eclipselink.entity'**. La clase de entidad empleado se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class Employee
{
   @Id
   @GeneratedValue( strategy= GenerationType.AUTO )
   private int eid;
   private String ename;
   private double salary;
   private String deg;

   @OneToOne
   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;
   }

   public Employee( )
   {
      super();
   }

   public int getEid( )
   {
      return eid;
   }
   public void setEid(int eid)
   {
      this.eid = eid;
   }

   public String getEname( )
   {
      return ename;
   }
   public void setEname(String ename)
   {
      this.ename = ename;
   }

   public double getSalary( )
   {
      return salary;
```

```
   }
      public void setSalary(double salary)
   {
      this.salary = salary;
   }

   public String getDeg( )
   {
      return deg;
   }
      public void setDeg(String deg)
   {
      this.deg = deg;
   }

   public Department getDepartment()
   {
      return department;
   }

   public void setDepartment(Department department)
   {
      this.department = department;
   }
}
```

Persistence.xml

Persistence.xml archivo de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

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

      <class>com.tutorialspoint.eclipselink.entity.Employee</class>
      <class>com.tutorialspoint.eclipselink.entity.Department</class>

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

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

Clases de servicio

Crear un paquete debajo del paquete '**src**' llamado '**com.tutorialspoint.eclipselink.service**'. La clase DAO llamada **OneToOne.java** se crea bajo el paquete determinado. La clase DAO se muestra como sigue:

```
package com.tutorialspointeclipselink.service;

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

import com.tutorialspoint.eclipselink.entity.Department;
import com.tutorialspoint.eclipselink.entity.Employee;

public class OneToOne
{
   public static void main(String[] args)
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );

      //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.setSalary(45000.0);
      employee.setDeg("Technical Writer");
      employee.setDepartment(department);

      //Store Employee
      entitymanager.persist(employee);

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

Después de la compilación y ejecución del programa anterior usted recibirá notificaciones en el panel de la consola de Eclipse IDE. Para la salida, compruebe los siguientes MySQL workbench.

En el ejemplo anterior, se crean dos tablas. Pase la siguiente consulta en interfaz MySQL y se visualizará el resultado de la tabla department de la siguiente manera:

```
Select * from department
```

ID	nombre
301	desarrollo

Pase la siguiente consulta en interfaz MySQL y se visualizará el resultado de la tabla **empleado** de la siguiente manera:

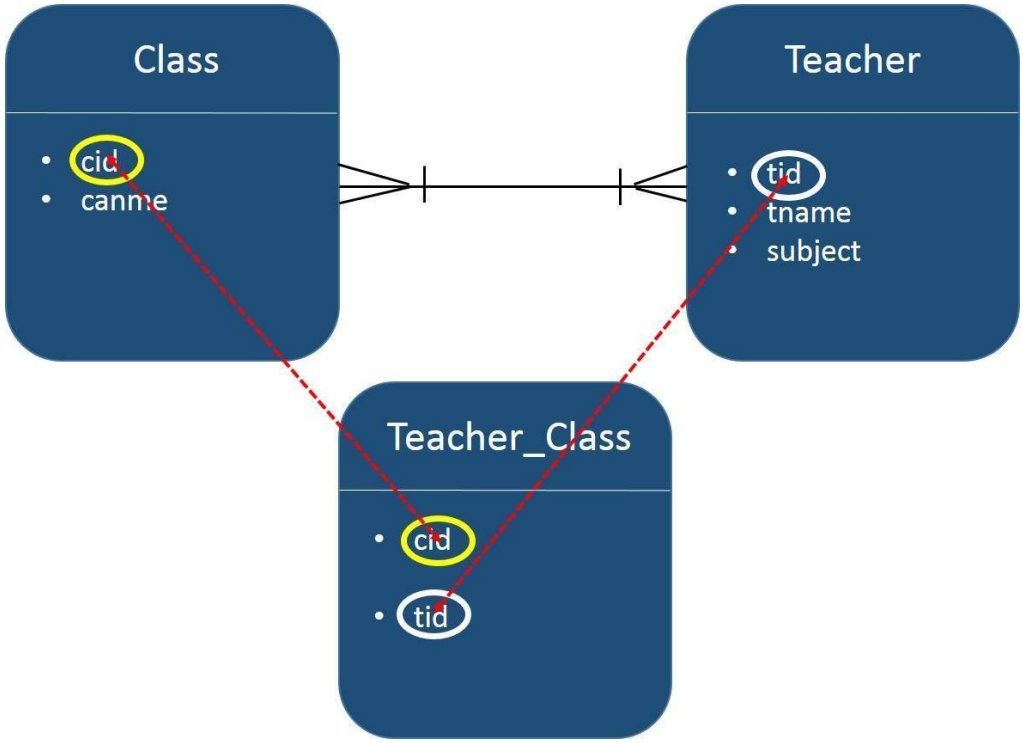
```
Select * from employee
```

Eid	Deg	Ename	Salary	Department_id
302	Technical Writer	Satish	45000	301

@ManyToMany Relation

Relación de varios a varios es donde una o más filas de una entidad se asocian a más de una fila en otra entidad.

Vamos a considerar un ejemplo de una relación entre dos entidades: **clase** y **profesor**. De manera bidireccional, tanto clase como profesor tienen muchos a uno relación. Que significa que cada registro de clase es referido por el profesor (ids de profesor), que debe ser primaria llaves en la mesa del profesor y almacenados en la tabla Teacher_Class y viceversa. Aquí, la tabla Teachers_Class contiene los campos claves extranjeros. Crear un proyecto JPA en eclipse IDE llamado **JPA_Eclipselink_MTM**. Todos los módulos de este proyecto se examinan a continuación.



Creación de entidades

Crear entidades siguiendo el esquema que se muestra en el diagrama de arriba. Crear un paquete denominado **'com.tutorialspoin.eclipselink.entity'** bajo **'src'** paquete. Crear una clase denominada **Clas.java** bajo dado el paquete. La clase entidad departamento se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import java.util.Set;

import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Clas
{
   @Id
   @GeneratedValue( strategy = GenerationType.AUTO )
   private int cid;
   private String cname;

   @ManyToMany(targetEntity=Teacher.class)
   private Set teacherSet;

   public Clas()
   {
      super();
   }

   public Clas(int cid, String cname, Set teacherSet)
   {
      super();
      this.cid = cid;
      this.cname = cname;
      this.teacherSet = teacherSet;
   }

   public int getCid()
   {
      return cid;
   }

   public void setCid(int cid)
   {
      this.cid = cid;
   }

   public String getCname()
   {
      return cname;
   }

   public void setCname(String cname)
   {
      this.cname = cname;
   }

   public Set getTeacherSet()
   {
      return teacherSet;
   }

   public void setTeacherSet(Set teacherSet)
   {
      this.teacherSet = teacherSet;
   }
}
```

Crear la segunda entidad en esta relación-clase de entidad Employee, denominada **Teacher.java** en paquete **'com.tutorialspoint.eclipselink.entity'**. La clase de entidad empleado se muestra como sigue:

```
package com.tutorialspoint.eclipselink.entity;

import java.util.Set;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Teacher
{
   @Id
   @GeneratedValue( strategy = GenerationType.AUTO )
   private int tid;
   private String tname;
   private String subject;

   @ManyToMany(targetEntity=Clas.class)
   private Set clasSet;

   public Teacher()
   {
      super();
   }

   public Teacher(int tid, String tname, String subject, Set clasSet)
   {
      super();
      this.tid = tid;
      this.tname = tname;
      this.subject = subject;
      this.clasSet = clasSet;
   }

   public int getTid()
   {
      return tid;
   }

   public void setTid(int tid)
   {
      this.tid = tid;
   }

   public String getTname()
   {
      return tname;
   }

   public void setTname(String tname)
   {
      this.tname = tname;
   }
```

```
   public String getSubject()
   {
      return subject;
   }

   public void setSubject(String subject)
   {
      this.subject = subject;
   }

   public Set getClasSet()
   {
      return clasSet;
   }

   public void setClasSet(Set clasSet)
   {
      this.clasSet = clasSet;
   }
}
```

Persistence.xml

Persistence.xml archivo de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

   <persistence-unit name="Eclipselink_JPA" transaction-type="RESOURCE_LOCAL">
      <class>com.tutorialspoint.eclipselink.entity.Employee</class>
      <class>com.tutorialspoint.eclipselink.entity.Department</class>

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

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

Clases de servicio

Crear un paquete debajo del paquete **'src'** llamado **'com.tutorialspoint.eclipselink.service'**. La clase DAO llamada **ManyToMany.java** se crea bajo dado paquete. La clase DAO se muestra como sigue:

```
package com.tutorialspoint.eclipselink.service;

import java.util.HashSet;
```

```
import java.util.Set;

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

import com.tutorialspoint.eclipselink.entity.Clas;
import com.tutorialspoint.eclipselink.entity.Teacher;

public class ManyToMany
{
   public static void main(String[] args)
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );
      entitymanager.getTransaction( ).begin( );

      //Create Clas Entity
      Clas clas1=new Clas(0,"1st",null);
      Clas clas2=new Clas(0,"2nd",null);
      Clas clas3=new Clas(0,"3rd",null);

      //Store Clas
      entitymanager.persist(clas1);
      entitymanager.persist(clas2);
      entitymanager.persist(clas3);

      //Create Clas Set1
      Set<Clas> classSet1 = new HashSet();
      classSet1.add(clas1);
      classSet1.add(clas2);
      classSet1.add(clas3);

      //Create Clas Set2
      Set<Clas> classSet2 = new HashSet();
      classSet2.add(clas3);
      classSet2.add(clas1);
      classSet2.add(clas2);

      //Create Clas Set3
      Set<Clas> classSet3 = new HashSet();
      classSet3.add(clas2);
      classSet3.add(clas3);
      classSet3.add(clas1);

      //Create Teacher Entity
      Teacher teacher1 = new Teacher(0, "Satish", "Java", classSet1);
      Teacher teacher2 = new Teacher(0, "Krishna", "Adv Java", classSet2);
      Teacher teacher3 = new Teacher(0, "Masthanvali", "DB2", classSet3);

      //Store Teacher
      entitymanager.persist(teacher1);
      entitymanager.persist(teacher2);
      entitymanager.persist(teacher3);

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

En este proyecto de ejemplo, se crean las tres tablas. Pase la siguiente consulta en interfaz MySQL y el resultado de la tabla teacher_clas se mostrará como sigue:

```
Select * form teacher_clas
```

Teacher_tid	Classet_cid
354	351
355	351
356	351
354	352
355	352
356	352
354	353
355	353
356	353

En la tabla anterior **teacher_tid** es la clave externa de la tabla maestra, y **classet_cid** es la clave externa de la tabla de la clase. Por lo tanto diferentes maestros se asignan a diferentes clase

Pase la siguiente consulta en interfaz MySQL y el resultado de la mesa del profesor se mostrará como sigue:

```
Select * from teacher
```

Tid	Subject	Tname
354	Java	Satish
355	Adv Java	Krishna
356	DB2	Masthanvali

Pase la siguiente consulta en interfaz MySQL y el resultado de la **clase** tabla aparecerá de la siguiente manera:

```
Select * from clas
```

Cid	Cname
351	1st
352	2nd
353	3rd

JPA - API Criterios

Criterios predefinidos es una API que se utiliza para definir las consultas a las entidades. Es una forma alternativa de definir un JPQL consulta. Estas consultas son del tipo de seguro, portátil y fácil de modificar cambiando la sintaxis. Similar a JPQL., sigue un esquema abstracto (fácil de editar esquema) y objetos incrustados. La API de metadatos se mezcla con criterios de API modelo entidad persistente criterios para las consultas.

La principal ventaja de los Criterios API es que los errores se pueden detectar antes durante el tiempo de compilación. Basado en la cadena JPA consultas y JPQL criterios consultas basadas en rendimiento y eficiencia.

Historia del API criteria

Los criterios se incluye en todas las versiones de JPA. por lo tanto, cada paso de los criterios es notificado de las especificaciones de JPA.

- En JPA 2.0 , los criterios API de consulta, la normalización de las consultas se han desarrollado.
- En JPA 2.1 , actualización Criterios y eliminar (actualización masiva y eliminar) están incluidos.

Criterios Estructura de consulta

Los criterios y el JPQL están estrechamente relacionados y se les permite diseñar utilizando los operadores similares en las consultas. El siguiente **paquete javax.persistence.criterios** para el diseño de una consulta. La estructura de consulta significa la sintaxis criterios consulta.

Los siguientes criterios simples consulta devuelve todas las instancias de la clase de entidad en el origen de datos.

```
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();
```

La consulta muestra los pasos básicos para crear un conjunto de criterios.

- **EntityManager** ejemplo se utiliza para crear un objeto CriteriaBuilder.
- **CriteriaQuery** ejemplo se utiliza para crear un objeto de la consulta. Esta consulta atributos del objeto será modificado con los detalles de la consulta.
- **CriteriaQuery.form** se llama al método set la consulta.
- **CriteriaQuery.select** está llamado a establecer el resultado tipo de lista.
- **TypedQuery<T>** se utiliza para preparar una consulta para la ejecución y especificando el tipo de resultado de la consulta.
- **getResultList** method on the TypedQuery<T> para ejecutar una consulta. Esta consulta devuelve una colección de entidades, el resultado se almacena en una lista.

Ejemplo de API criterios.

Consideremos el ejemplo de base de datos de empleados. Supongamos que el jpadb.tabla de empleados contiene los siguientes registros:

```
Eid   Ename          Salary   Deg
```

```
401   Gopal          40000    Technical Manager
402   Manisha        40000    Proof reader
403   Masthanvali    35000    Technical Writer
404   Satish         30000    Technical writer
405   Krishna        30000    Technical Writer
406   Kiran          35000    Proof reader
```

Crear un proyecto de JPA en el eclipse que IDE llamado **JPA_Eclipselink_Criteria**. Todos los módulos de este proyecto se describen a continuación:

Crear entidades

Crear un paquete denominado **com.tutorialspoint.eclipselink.entity** en **"src'**

Crear una clase denominada **Empleado.java** en paquete. La clase entidad empleado se muestra de la siguiente manera:

```
package com.tutorialspoint.eclipselink.entity;

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

@Entity
public class Employee
{
   @Id
   @GeneratedValue(strategy= GenerationType.AUTO)
   private int eid;
   private String ename;
   private double salary;
   private String deg;

   public Employee(int eid, String ename, double salary, String deg)
   {
      super( );
      this.eid = eid;
      this.ename = ename;
      this.salary = salary;
      this.deg = deg;
   }

   public Employee( )
   {
      super();
   }

   public int getEid( )
   {
      return eid;
   }
   public void setEid(int eid)
   {
      this.eid = eid;
   }

   public String getEname( )
   {
      return ename;
```

```
   }
   public void setEname(String ename)
   {
      this.ename = ename;
   }

   public double getSalary( )
   {
      return salary;
   }
   public void setSalary(double salary)
   {
      this.salary = salary;
   }

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

Persistence.xml

Persistence.xml Archivo es la siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

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

      <class>com.tutorialspoint.eclipselink.entity.Employee</class>

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

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

Clases de servicio

Este módulo contiene las clases de servicio, que implementa los Criterios parte de la consulta mediante la API de metadatos inicialización. Crear un paquete denominado '**com.tutorialspoint.eclipselink.service'**. La clase denominada **CriteriaAPI.java** se crea en paquete. La clase DAO se muestra de la siguiente manera:

```
package com.tutorialspoint.eclipselink.service;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;

import com.tutorialspoint.eclipselink.entity.Employee;

public class CriteriaApi
{
   public static void main(String[] args)
   {
      EntityManagerFactory emfactory =
Persistence.createEntityManagerFactory( "Eclipselink_JPA" );
      EntityManager entitymanager = emfactory.createEntityManager( );

      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 =criteriaQuery.select(from);
      TypedQuery<Object> typedQuery = entitymanager.createQuery(select);
      List<Object> resultlist= typedQuery.getResultList();

      for(Object o:resultlist)
      {
         Employee e = (Employee);
         System.out.println("EID : " + e.getEid() + " Ename : " + e.getEname());
      }

      //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();

      for(Object o:resultlist1)
      {
         Employee e = (Employee);
         System.out.println("EID : " + e.getEid() + " Ename : " + e.getEname());
      }

      entitymanager.close( );
```

```
      emfactory.close( );
   }
}
```

Después de compilar y ejecutar el programa anterior, obtendrá el siguiente mensaje de salida en el panel de la consola de Eclipse IDE.

```
Select All records
EID : 401 Ename : Gopal
EID : 402 Ename : Manisha
EID : 403 Ename : Masthanvali
EID : 404 Ename : Satish
EID : 405 Ename : Krishna
EID : 406 Ename : Kiran
Select All records by follow Ordering
EID : 401 Ename : Gopal
EID : 406 Ename : Kiran
EID : 405 Ename : Krishna
EID : 402 Ename : Manisha
EID : 403 Ename : Masthanvali
EID : 404 Ename : Satish
```