**Java Persistence API – JPA**

**Persistencia / Almacenamiento permanente**

Debido a esta situación, aparecieron los motores de persistencia, cuya función es traducir entre los dos formatos de datos: de registros a objetos y de objetos a registros. Persistir objetos Java en una base de datos relacional implica serializar un árbol de objetos Java en una base de datos de estructura tabular y viceversa. Esencial es la necesidad de mapear objetos Java para optimizar velocidad y eficiencia de la base de datos

**¿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.

Object-Relational Mapping

Técnica usada para hacer el paso del modelo de objetos (Data Access Object) y el modelo relacional (JDBC-Java Database Connectivity)

**Unidades de persistencia**

La unidad de persistencia define un conjunto de todas las entidades (clases) que son gestionadas por la instancia del EntityManager en una aplicación. Este conjunto de clases de entidad representa los datos contenidos en una única BBDD.

Las unidades de persistencia se definen en el fichero de configuración persistence.xml

**La persistencia puede tener 4 estados diferentes:**

**Transient:** Un objeto recién creado que no ha sido enlazado con el gestor de persistencia.

**Persistent:** Un objeto enlazado con la sesión (Todos los cambios serán persistentes).

**Detached:** Un objeto persistente que sigue en memoria después de que termina la sesión: existe en java y en la BDD.

**Removed**: Un objeto marcado para ser eliminado de la BBDD: existe en java y se borrará de la BDD al terminar la sesión.

**Frameworks de persistencia**

* Hibernate,
* Toplink
* JDO.

**JPA - Entity Manager**

Antes que nada, tenemos que tener bien en claro dos temas muy importantes que son:

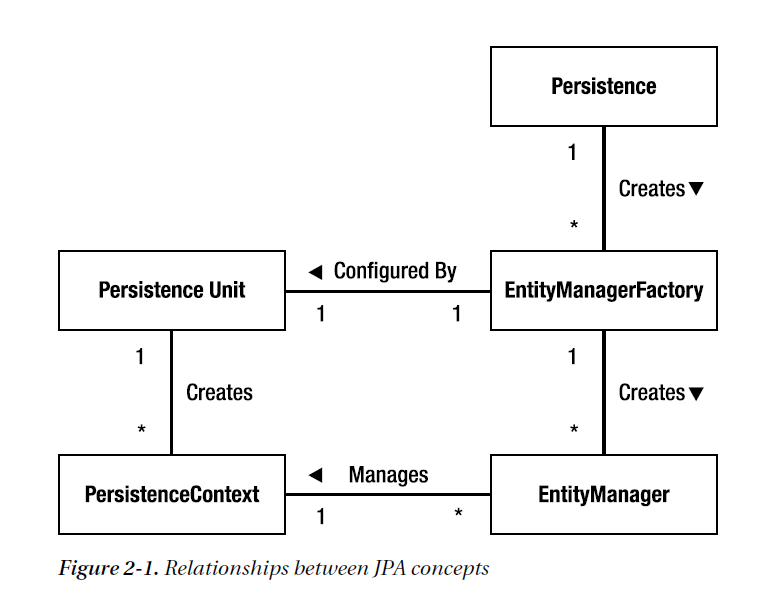
Application-managed entity manager

Container-managed entity manager

Para las situaciones en donde no requerimos los servicios ofrecidos por un EJB3 Container, pero querremos utilizar el modelo de persistencia de JPA, el API JPA provee el Application-managed entity manager. En estos casos las aplicaciones son denominadas aplicaciones standalone. Estas aplicaciones que corren fuera de un EJB3 container (porque no lo necesitan) usaran el resource-local transaction provisto por el entity manager, lo que permite que la aplicación será la encargada de manejar el ciclo de vida del entity manager. Para las situaciones en donde sí se requiera de los servicios de un EJB3 container (como por ejemplo JBoss ó WebLogic application server), el API JPA provee el Container-managed entity manager.

Siempre que una transacción sea iniciada, un nuevo contexto de persistencia (persistence context) es creado. Esto es así tanto para el Application-managed entity manager como también para el Container-managed entity manager: Para el caso del Application-managed entity manager (cuando no usamos un application server), la aplicación es la encargada de abrir y cerrar la transacción. Para el caso del Container managed entity manager (cuando utilizamos un ejb container), por defecto, la transacción es iniciada cuando se invoque desde el cliente al EJB (a un stateless session bean, ahora bien, para los stateful session bean el comportamiento de las transacciones es distinto). La transacción termina cuando finaliza la ejecución del método del session bean

Conceptos JPA



|  |  |
| --- | --- |
| **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. |
| Entity | 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. |
| Persistence | Esta clase contiene métodos estáticos para obtener EntityManagerFactory. |
| Query | Esta interfaz es implementada por cada proveedor JPA relacional para obtener objetos que cumplan los criterios. |

**Interfaces JPA**

Los 4 tipos de interfaces de las que se compone JPA son:

javax.persistence.Persistence: Contiene métodos estáticos de ayuda para obtener una instancia de Entity Manager Factory de una forma independiente al vendedor de la implementación de JPA. Una clase de inicialización que va proporcionar un método estático para la creación de una Entity Manager Factory.

javax.persistence.EntityManagerFactory: La clase javax.persistence.Entity.Manager.Factory nos ayuda a crear objetos de EntityManager utilizando el patrón de diseño del Factory. Este objeto en tiempo de ejecución representa una unidad de persistencia particular. Generalmente va a ser manejado como un singleton y proporciona métodos para la creación de instancias EntityManager.

javax.persistence.EntityManagerFactory: La clase javax.persistence.Entity es una anotación Java que se coloca a nivel de clases Java serializables y que cada objeto de una de estas clases anotadas representa un registro de una base de datos.

javax.persistence.EntityManager: Es la interfaz principal de JPA utilizada para la persistencia de las aplicaciones. Cada Entity Manager puede realizar operaciones CRUD (Create, Read, Update, Delete) sobre un conjunto de objetos persistentes. Es un objeto único, no compartido que representa una unidad de trabajo particular para el acceso a datos. Proporciona métodos para gestionar el ciclo de vida de las instancias entidad y para crear instancias Query.

javax.persistence.Query: La interface javax.persistence.Query está implementada por cada vendedor de JPA para encontrar objetos persistentes manejando cierto criterio de búsqueda. JPA estandariza el soporte para consultas utilizando Java Persistence Query Language (JPQL) y Structured Query Language (SQL). Podemos obtener una instancia de Query desde una instancia de un Entity Manager.

javax.persistence.EntityTransaction: Cada instancia de Entity Manager tiene una relación de uno a uno con una instancia de javax.persistence.EntityTransaction, permite operaciones sobre datos persistentes de manera que agrupados formen una unidad de trabajo transaccional, en el que todo el grupo sincroniza su estado de persistencia en la base de datos o todos fallan en el intento, en caso de fallo, la base de datos quedará con su estado original. Maneja el concepto de todos o ninguno para mantener la integridad de los datos.

**Entidad:**

Una entidad esencialmente es un nombre, un grupo o un estado asociado a una unidad simple.

**Persistencia:**

Es una característica de las entidades, se refiere al hecho de que el estado de una entidad puede ser representado en un almacenamiento de datos y este estado podría ser accedido tiempo después.

Campos de persistencia permanente: Si la entidad utiliza campos de persistencia permanente, los accesos se realizan en tiempo de ejecución. Aquellos campos que no tienen anotaciones del tipo javax.persistence.Transient o no han sido marcados como Java transitorio serán persistentes para el almacenamiento de datos.

Las anotaciones de mapeo objeto/relación deben aplicarse a los atributos de la instancia.

Propiedades de persistencia permanente: Si la entidad utiliza propiedades de persistencia permanente, la entidad debe seguir el método de los convenios de componentes JavaBeans. Las propiedades de JavaBean usan métodos getters y setters en cuyo nombre va incluido el atributo de la clase al cual hacen referencia. Si el atributo es booleano podrá utilizarse isProperty en lugar de getProperty.

Características de las entidades: Persistentes, Identidad, Transaccional, Granularidad

**Entity Metadata**

Para tener un estado persistente cada entidad tiene asociado metadata que la describe, se puede hacer por medio de un archivo de configuración XML o con el uso de anotaciones

**Metadata por medio de Anotaciones**

Es una característica del lenguaje que permite estructurar y tipar los metadatos adjuntos al código fuente.

**Metadata por medio de XML**

Es el mecanismo de adicionar funcionalidad a las entidades usando descriptores XML es más complejo.

**Relaciones múltiples de la entidad**

Hay cuatro tipos de relaciones: uno a uno, uno a muchos, muchos a uno, y muchos a muchos.

**Uno a uno:** Cada entidad se relaciona con una sola instancia de otra entidad. Las relaciones uno a uno utilizan anotaciones de la persistencia de java "OneToOne".

**Uno a muchos:** Una entidad, puede estar relacionada con varias instancias de otras entidades. Las relaciones uno a muchos utilizan anotaciones de la persistencia de java "OneToMany" en los campos o propiedades persistentes.

**Muchos a uno:** Múltiples instancias de una entidad pueden estar relacionadas con una sola instancia de otra entidad. Esta multiplicidad es lo contrario a la relación uno a muchos. Las relaciones muchos a uno utilizan anotaciones de la persistencia de java "ManyToOne" en los campos o propiedades persistentes.

**Muchos a muchos:** En este caso varias instancias de una entidad pueden relacionarse con múltiples instancias de otras entidades. Este tipo de relación utiliza anotaciones de la persistencia de java "ManyToMany" en los campos o propiedades persistentes.

**Manejo de Excepciones**

El siguiente es el orden de las excepciones que pueden arrojar errores en operaciones y manejo de jpa, cada excepción debe ser controlada o propagada según sea el caso.

java.lang.Object

java.lang.Throwable

java.lang.Exception

java.lang.RuntimeException

javax.persistence.PersistenceException

javax.persistence.EntityExistsException

javax.persistence.EntityNotFoundException

javax.persistence.LockTimeoutException

javax.persistence.NonUniqueResultException

javax.persistence.NoResultException

javax.persistence.OptimisticLockException

javax.persistence.PessimisticLockException

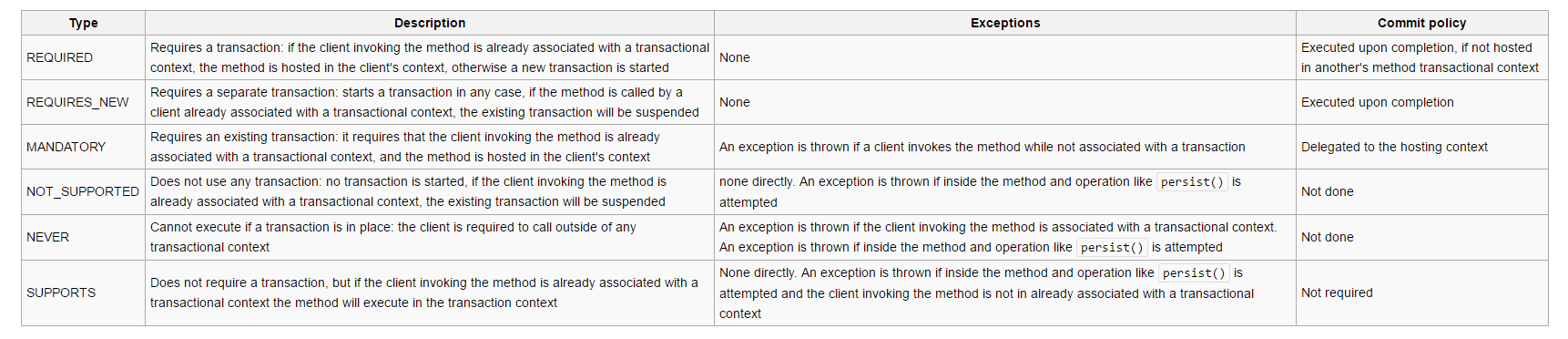
javax.persistence.QueryTimeoutException

javax.persistence.RollbackException

javax.persistence.TransactionRequiredException

**Transacciones**

Una transacción es un conjunto de operaciones que, o bien fracasar o tienen éxito como una unidad. Las transacciones son una parte fundamental de la persistencia. Una transacción de base de datos consiste en un conjunto de operaciones de SQL DML (Data Manipulation Language) que se comprometen o se deshace como una sola unidad. Una transacción de nivel de objeto es una en la que un conjunto de cambios realizados en un conjunto de objetos está persistido con la base de datos como una sola unidad. JPA proporciona dos mecanismos para las transacciones. Cuando se utiliza en Java EE JPA proporciona integración con JTA (Java Transaction API). JPA también proporciona su propia implementación EntityTransaction para Java SE y para su uso en un modo no gestionados en Java EE. Las transacciones en JPA son siempre a nivel del objeto, esto significa que todos los cambios realizados a todos los objetos persistentes en el contexto de persistencia son parte de la transacción.



Ver más en: <https://en.wikibooks.org/wiki/Java_Persistence/Transactions>

**Consultas - Java Persistence Query Language JPQL**

Consultas simples

* Obtener la entidad completa:

SELECT e FROM Employee e;

* Obtener un atributo de la entidad

SELECT e.name FROM Employee e;

* Obtener otra entidad relacionada a la entidad principal -

SELECT e.department FROM Employee e

* Filtros de búsqueda en la consulta

SELECT e

FROM Employee e

WHERE e.department.name = 'NA42' AND

e.address.state IN ('NY','CA')

* Operaciones básicas para filtrar datos

IN, LIKE, y BETWEEN

* Proyección de resultados: obtener solo una porción o un subconjunto de propiedades desde la entidad

SELECT e.name, e.salary

FROM Employee e

* Uniones entre entidades – JOINS

Ejemplo de una consulta navegando por la asociación de dos entidades:

SELECT p.number

FROM Employee e, Phone p

WHERE e = p.employee AND

e.department.name = 'NA42' AND

p.type = 'Cell'

Los JOIN se pueden poner en el FROM de la consulta, de esta forma el JOIN se puede expresar en términos de la asociación en sí, y el motor de consultas puede suplir automáticamente el criterio de unión necesario para generar la sentencia SQL

SELECT p.number

FROM Employee e JOIN e.phones p

WHERE e.department.name = 'NA42' AND

p.type = 'Cell'

* Las consultas de agregado
* AVG, COUNT, MIN, MAX, y SUM
* GROUP BY que puede tener un filtro HAVING

SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)

FROM Department d JOIN d.employees e

GROUP BY d

HAVING COUNT(e) >= 5

* Parámetros en las consultas
* Enlace posicional: Aquí los parámetros son marcados usando el signo de interrogación seguido por un parámetro numérico, cuando la consulta es ejecutada se debe indicar el número del parámetro del elemento a ser reemplazado

SELECT e

FROM Employee e

WHERE e.department = ?1 AND

e.salary > ?2

* Enlace por medio del nombre del parámetro usando dos puntos seguido del nombre del parámetro

SELECT e

FROM Employee e

WHERE e.department = :dept AND

e.salary > :base

* Definición de consultas
* Consultas dinámicas

@Stateless

public class QueryServiceBean implements QueryService {

@PersistenceContext(unitName="DynamicQueries")

EntityManager em;

public long queryEmpSalary(String deptName, String empName) {

String query = "SELECT e.salary " +

"FROM Employee e " +

"WHERE e.department.name = '" + deptName +

"' AND " +

" e.name = '" + empName + "'";

return em.createQuery(query, Long.class).getSingleResult();

}

}

@Stateless

public class QueryServiceBean implements QueryService {

private static final String QUERY =

"SELECT e.salary " +

"FROM Employee e " +

"WHERE e.department.name = :deptName AND " +

" e.name = :empName ";

@PersistenceContext(unitName="DynamicQueries")

EntityManager em;

public long queryEmpSalary(String deptName, String empName) {

return em.createQuery(QUERY, Long.class)

.setParameter("deptName", deptName)

.setParameter("empName", empName)

.getSingleResult();

}

}

* Definiendo Named Query

Named Query con una herramienta que permite definir y mejorar el rendimiento de la aplicación, las consultas nombradas normalmente se colocan en la clase de entidad que más se corresponde directamente al resultado de la consulta. Ya que la consulta está definida por una anotación no puede ser cambiada en tiempo de ejecución.

@NamedQuery(name="findSalaryForNameAndDepartment",

query="SELECT e.salary " +

"FROM Employee e " +

"WHERE e.department.name = :deptName AND " +

" e.name = :empName")

@NamedQueries({

@NamedQuery(name="Employee.findAll",

query="SELECT e FROM Employee e"),

@NamedQuery(name="Employee.findByPrimaryKey",

query="SELECT e FROM Employee e WHERE e.id = :id"),

@NamedQuery(name="Employee.findByName",

query="SELECT e FROM Employee e WHERE e.name = :name")

})

@Stateless

public class EmployeeServiceBean implements EmployeeService {

@PersistenceContext(unitName="EmployeeService")

EntityManager em;

public Employee findEmployeeByName(String name) {

return em.createNamedQuery("Employee.findByName", Employee.class)

.setParameter("name", name)

.getSingleResult();

}

// ...

}

* Tipos de parámetros

Hay tres variantes para el paso de parámetros tanto para los que son con nombre y los que son posicionales.

* El primer argumento es siempre el nombre o número de parámetros.
* El segundo argumento es el objeto a ser unido al parámetro llamado.
* Parámetros de tipo Date y Calendar también requieren un tercer argumento que

especifica si el tipo aprobado para JDBC es un valor java.sql.Date, java.sql.Time, o java.sql.Timestamp

@NamedQuery(name="findEmployeesAboveSal",

query="SELECT e " +

"FROM Employee e " +

"WHERE e.department = :dept AND " +

" e.salary > :sal")

@Stateless

public class EmployeeServiceBean implements EmployeeService {

@PersistenceContext(unitName="EmployeeService")

EntityManager em;

public List<Employee> findEmployeesAboveSal(Department dept, long minSal) {

return em.createNamedQuery("findEmployeesAboveSal",

Employee.class)

.setParameter("dept", dept)

.setParameter("sal", minSal)

.getResultList();

}

// ...

}

@Stateless

public class EmployeeServiceBean implements EmployeeService {

@PersistenceContext(unitName="EmployeeService")

EntityManager em;

public List<Employee> findEmployeesHiredDuringPeriod(Date start, Date end) {

return em.createQuery("SELECT e " +

"FROM Employee e " +

"WHERE e.startDate BETWEEN ?1 AND ?2",

Employee.class)

.setParameter(1, start, TemporalType.DATE)

.setParameter(2, end, TemporalType.DATE)

.getResultList();

}

// ...

}

* Ejecutando consultas

Existen 3 maneras para ejecutar consultas

* getSingleResult(): Obtiene un solo resultado, para su uso se deben tener en cuenta el manejo de excepciones ya que si no encuentra un resultado único la ejecución de la consulta lanza un NonUniqueResultException y cuando no se encuentran resultados se lanza NoResultException, dichas excepciones se deberán controlar.
* getResultList(): Obtiene un conjunto de resultados, si la consulta no retorna resultados se obtiene una colección vacia (List)
* executeUpdate() : Usado para invocar actualizaciones y borrados en bloque

public void displayProjectEmployees(String projectName) {

List<Employee> result = em.createQuery(

"SELECT e " +

"FROM Project p JOIN p.employees e "+

"WHERE p.name = ?1 " +

"ORDER BY e.name",

Employee.class)

.setParameter(1, projectName)

.getResultList();

int count = 0;

for (Employee e : result) {

System.out.println(++count + ": " + e.getName() + ", " +

e.getDepartment().getName());

}

}

public class ProjectManagerBean implements ProjectManager {

@PersistenceContext(unitName="EmployeeService", type=PersistenceContextType.EXTENDED)

EntityManager em;

TypedQuery<Employee> unassignedQuery;

@PostConstruct

public void init() {

unassignedQuery =

em.createQuery("SELECT e " +

"FROM Employee e " +

"WHERE e.projects IS EMPTY",

Employee.class);

}

public List<Employee> findEmployeesWithoutProjects() {

return unassignedQuery.getResultList();

}

// ...

}

* Manipulando el resultado de las consultas

El tipo de resultado de una consulta se determina por las expresiones que figuran en la cláusula SELECT de la consulta.

• Tipos básicos, como String, los tipos primitivos y los tipos JDBC

• Los tipos de entidad

• Una serie de objetos

• Los tipos definidos por el usuario creados a partir de una expresión constructor

Cada vez que se devuelve una instancia de la entidad, se convierte en una entidad manejada por el contexto de persistencia activa. La única excepción a esta regla es el uso de la entidad con ámbito de transacción (transaction-scoped)

Optimizando consultas de solo lectura

Con el tipo de transacción TransactionAttributeType.NOT\_SUPPORTED se puede lograr que cualquier transacción activa pueda ser suspendido, obligando que los resultados de la consulta sean desprendido del contexto (detached) y se permite optimización que el contexto no crezca

@Stateless

public class QueryServiceBean implements QueryService {

@PersistenceContext(unitName="EmployeeService")

EntityManager em;

@TransactionAttribute(TransactionAttributeType.NOT\_SUPPORTED)

public List<Department> findAllDepartmentsDetached() {

return em.createQuery("SELECT d FROM Department d",

Department.class)

.getResultList();

}

// ...

}

* Uso de Arrays para obtener resultados

public void displayProjectEmployees(String projectName) {

List result = em.createQuery(

"SELECT e.name, e.department.name " +

"FROM Project p JOIN p.employees e " +

"WHERE p.name = ?1 " +

"ORDER BY e.name")

.setParameter(1, projectName)

.getResultList();

int count = 0;

for (Iterator i = result.iterator(); i.hasNext();) {

Object[] values = (Object[]) i.next();

System.out.println(++count + ": " +

values[0] + ", " + values[1]);

}

}

* Definiendo una clase para usar una expresión con el constructor

public class EmpMenu {

private String employeeName;

private String departmentName;

public EmpMenu(String employeeName, String departmentName) {

this.employeeName = employeeName;

this.departmentName = departmentName;

}

public String getEmployeeName() { return employeeName; }

public String getDepartmentName() { return departmentName; }

}

public void displayProjectEmployees(String projectName) {

List<EmpMenu> result =

em.createQuery("SELECT NEW example.EmpMenu(" +

"e.name, e.department.name) " +

"FROM Project p JOIN p.employees e " +

"WHERE p.name = ?1 " +

"ORDER BY e.name",

EmpMenu.class)

.setParameter(1, projectName)

.getResultList();

int count = 0;

for (EmpMenu menu : result) {

System.out.println(++count + ": " +

menu.getEmployeeName() + ", " +

menu.getDepartmentName());

}

}

Paginando Consultas: Es una estrategia para la manipulación de resultados de un gran conjunto de atributos que puede retornar una consulta

public class ResultPagerBean implements ResultPager {

@PersistenceContext(unitName="QueryPaging")

private EntityManager em;

private String reportQueryName;

private long currentPage;

private long maxResults;

private long pageSize;

public long getPageSize() {

return pageSize;

}

public long getMaxPages() {

return maxResults / pageSize;

}

public void init(long pageSize, String countQueryName,

String reportQueryName) {

this.pageSize = pageSize;

this.reportQueryName = reportQueryName;

maxResults = em.createNamedQuery(countQueryName, Long.class)

.getSingleResult();

currentPage = 0;

}

public List getCurrentResults() {

return em.createNamedQuery(reportQueryName)

.setFirstResult(currentPage \* pageSize)

.setMaxResults(pageSize)

.getResultList();

}

public void next() {

currentPage++;

}

public void previous() {

currentPage--;

if (currentPage < 0) {

currentPage = 0;

}

}

public long getCurrentPage() {

return currentPage;

}

public void setCurrentPage(long currentPage) {

this.currentPage = currentPage;

}

@Remove

public void finished() {}

}

* Usando Update y Delete en bloque

@Stateless

public class EmployeeServiceBean implements EmployeeService {

@PersistenceContext(unitName="BulkQueries")

EntityManager em;

@TransactionAttribute(TransactionAttributeType.REQUIRES\_NEW)

public void assignManager(Department dept, Employee manager) {

em.createQuery("UPDATE Employee e " +

"SET e.manager = ?1 " +

"WHERE e.department = ?2")

.setParameter(1, manager)

.setParameter(2, dept)

.executeUpdate();

}

}

@Stateless

public class ProjectServiceBean implements ProjectService {

@PersistenceContext(unitName="BulkQueries")

EntityManager em;

@TransactionAttribute(TransactionAttributeType.REQUIRES\_NEW)

public void removeEmptyProjects() {

em.createQuery("DELETE FROM Project p " +

"WHERE p.employees IS EMPTY")

.executeUpdate();

}

}

* Las consultas de selección

SELECT <select\_expression>

FROM <from\_clause>

[WHERE <conditional\_expression>]

[ORDER BY <order\_by\_clause>]

Joins

* Son consultas que dan la posibilidad de obtener resultados desde diversas entidades

Los Joins ocurren siempre que cualquiera de las siguientes condiciones se cumplen en una consulta de selección:

1. Dos o más variables son listadas en la cláusula FROM y aparecen en la cláusula SELECT.

2. El operador JOIN se utiliza para extender una variable de identificación mediante una expresión.

3. Una expresión en cualquier lugar de la consulta se desplaza a través de un campo de asociación, a ella misma o una entidad diferente.

4. Una o más condiciones compraran atributos de diferentes variables

La semántica de los Joins es la misma que la usada en SQL

* Inner Join: es cuando se retorna los objetos desde ambas entidades que satisfacen todas las condicione de unión
* Outer join: es el conjunto de objetos desde ambos tipos de entidades que satisfacen la condición de unión adicionando el conjunto de objetos desde un tipo de entidad que no hace la unión con las condiciones de unión de la otra entidad

La usencia de las condiciones de unión en el retorno de la consulta podría producir un producto cartesiano.

* JOIN Operator and Collection Association Fields

Sintaxis: [INNER] JOIN <path\_expression> [AS] <identifier>.

Ejemplo: SELECT p FROM Employee e JOIN e.phones p

Ejemplo SQL: SELECT p.id, p.phone\_num, p.type, p.emp\_id

FROM emp e, phone p

WHERE e.id = p.emp\_id

* JOIN Operator and Single-Valued Association Fields

Ejemplo: SELECT d FROM Employee e JOIN e.department d

* Join Conditions in the WHERE Clause

Ejemplo:

SELECT DISTINCT d

FROM Department d, Employee e

WHERE **d = e.department**

SELECT d, m

FROM Department d, Employee m

WHERE **d = m.department AND m.directs** **IS NOT EMPTY**

* **Multiple Joins:** El uso de las uniones se puede realizar en cascada si es requerido

Ejemplo:

SELECT DISTINCT p

FROM Department d JOIN d.employees e JOIN e.projects p

* **Map Joins: Una expresión que navega a través de un valor tipo colección es un caso especial, en donde se tiene una llave y un conjunto de valores, se puede usar KEY y VALUE como identificadores de las partes del mapa**

SELECT e.name, p

FROM Employee e JOIN e.phones p

SELECT e.name, VALUE(p) FROM Employee e JOIN e.phones p

SELECT e.name, KEY(p), VALUE(p)

FROM Employee e JOIN e.phones p

WHERE KEY(p) IN ('Work', 'Cell')

**Outer Joins**

el outer join de Employee a Department sobre la relación entre empleados y departamento retorna todos los empleados y el departamento al cual está asignado cada empleado, pero el departamento es retornado solo si este está disponible.

Sintaxis: LEFT [OUTER] JOIN <path\_expression> [AS] <identifier>.

Ejemplo: SELECT e, d FROM Employee e LEFT JOIN e.department d

Ejemplo SQL: SELECT e.id, e.name, e.salary, e.manager\_id, e.dept\_id, e.address\_id,

d.id, d.name

FROM emp e, dept d

WHERE e.dept\_id = d.id (+)

Fetch Joins

Son Joins que están destinadas a ayudar a los diseñadores de aplicaciones a optimizar su base de datos y preparar resultado de consultas separadas o detachment. Esto permite consultas para especificar una o más relaciones que deben ser navegadas y pre-buscados por el motor de búsqueda de modo que no sean cargadas más tarde en tiempo de ejecución.

SELECT e FROM Employee e JOIN FETCH e.address

Input Parameters

* SELECT e FROM Employee e WHERE e.salary > **?**1
* SELECT e FROM Employee e WHERE e.salary > **:**sal

Expresiones básicas en el FROM

1. Operador de navegacion (.)

2. Unario +/–

3. Multiplicación (\*) and división (/)

4. Adición (+) y resta (–)

5. Operadores de comparación : =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT]

IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]

6. Operadores logicos (AND, OR, NOT)

Expresión Between

SELECT e

FROM Employee e

WHERE e.salary BETWEEN 40000 AND 45000

Expresión LIKE

SELECT d

FROM Department d

WHERE d.name LIKE '\_\_Eng%'

Subconsultas - Subqueries

SELECT e

FROM Employee e

WHERE e.salary = (SELECT MAX(emp.salary)

FROM Employee emp)

SELECT e

FROM Employee e

WHERE EXISTS (SELECT 1

FROM Phone p

WHERE p.employee = e AND p.type = 'Cell')

Expresión IN

SELECT e

FROM Employee e

WHERE e.address.state IN ('NY', 'CA')

Expresiones de Colecciones

SELECT e

FROM Employee e

WHERE e.directs IS NOT EMPTY

SELECT m

FROM Employee m

WHERE (SELECT COUNT(e)

FROM Employee e

WHERE e.manager = m) > 0

SELECT e

FROM Employee e

WHERE e MEMBER OF e.directs

Expresión CASE

CASE {WHEN <cond\_expr> THEN <scalar\_expr>}+ ELSE <scalar\_expr> END

SELECT p.name,

CASE WHEN TYPE(p) = DesignProject THEN 'Development'

WHEN TYPE(p) = QualityProject THEN 'QA'

ELSE 'Non-Development'

END

FROM Project p

WHERE p.employees IS NOT EMPTY

Clausula ORDER BY

SELECT e, d

FROM Employee e JOIN e.department d

ORDER BY d.name, e.name DESC

Consultas de agregado

SELECT <select\_expression>

FROM <from\_clause>

[WHERE <conditional\_expression>]

[GROUP BY <group\_by\_clause>]

[HAVING <conditional\_expression>]

[ORDER BY <order\_by\_clause>]

SELECT d.name, AVG(e.salary)

FROM Department d JOIN d.employees e

WHERE e.directs IS EMPTY

GROUP BY d.name

HAVING AVG(e.salary) > 50000

Consultas Update

UPDATE <entity name> [[AS] <identification variable>]

SET <update\_statement> {, <update\_statement>}\*

[WHERE <conditional\_expression>]

UPDATE Employee e

SET e.salary = 60000

WHERE e.salary = 55000

Consultas Delete

DELETE FROM <entity name> [[AS] <identification variable>]

[WHERE <condition>]

DELETE FROM Employee e

WHERE e.department IS NULL

**Anotaciones**

|  |  |
| --- | --- |
| **Anotación** | **Descripción** |
| @Entityº | 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. |

**EJERCICIOS PRÁCTICOS:**

* **Creando entidades**

public class Employee {

private int id;

private String name;

private long salary;

public Employee() {}

public Employee(int id) { this.id = id; }

public int getId() { return id; }

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

public String getName() { return name; }

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

public long getSalary() { return salary; }

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

}

@Entity

public class Employee {

@Id private int id;

private String name;

private long salary;

public Employee() {}

public Employee(int id) { this.id = id; }

public int getId() { return id; }

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

public String getName() { return name; }

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

public long getSalary() { return salary; }

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

}

* **Definir la unidad de persistencia (configurar)**

EmployeeService

* **Obtener EMF**

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

* **Obtener EM desde el EMF**

EntityManager em = emf.createEntityManager();

* **Operaciones de persistencia**

Employee emp = new Employee(158);

em.persist(emp);

Employee emp = em.find(Employee.class, 158);

em.remove(emp);

En el método remove si se envía un nulo se obtiene un error java.lang.IllegalArgumentException

Employee emp = em.find(Employee.class, 158);

emp.setSalary(emp.getSalary() + 1000);

**Arquetipo maven + jpa**

mvn archetype:generate -DgroupId=com.javawebtutor -DartifactId=JPAMavenExample -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false - See more at: http://www.javawebtutor.com/articles/jpa/jpa-example-using-maven.php#sthash.OnYDeww3.dpuf

**Referencias**

Apress - Pro JPA 2 Mastering the Java Persistence API (November 2009) (ATTiCA)

<http://www.javawebtutor.com/articles/jpa/jpa-example-using-maven.php>

Olivas, C. (2016). *JPA - Java, Entidades, Persistencia y managers - conceptos básicos.* [online] Lab.inf.uc3m.es. Available at: http://www.lab.inf.uc3m.es/~a0080802/RAI/jpa.html [Accessed 22 Sep. 2016].